Parallel Evaluation Strategies for Lazy Data Structures in Haskell

Prabhat Totoo

Thesis submitted for the degree of Doctor of Philosophy in the School of Mathematical and Computer Sciences

May 2016
Edinburgh

Parallel Evaluation Strategies for Lazy Data Structures in Haskell
Prabhat Totoo
May 2016

Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Examination committee
Kevin Hammond, University of St Andrews
Josef Svenningsson, Chalmers University of Technology
Greg Michaelson, Heriot-Watt University

Supervisors
Hans-Wolfgang Loidl, Heriot-Watt University
Phil Trinder, University of Glasgow
Sven-Bodo Scholz, Heriot-Watt University
Murray Cole, University of Edinburgh

Abstract

Conventional parallel programming is complex and error prone. To improve programmer productivity, we need to raise the level of abstraction with a higher-level programming model that hides many parallel coordination aspects. Evaluation strategies use non-strictness to separate the coordination and computation aspects of a Glasgow parallel Haskell (GpH) program. This allows the specification of high level parallel programs, eliminating the low-level complexity of synchronisation and communication associated with parallel programming.

This thesis employs a data-structure-driven approach for parallelism derived through generic parallel traversal and evaluation of sub-components of data structures. We focus on evaluation strategies over list, tree and graph data structures, allowing re-use across applications with minimal changes to the sequential algorithm.

In particular, we develop novel evaluation strategies for tree data structures, using core functional programming techniques for coordination control, achieving more flexible parallelism. We use non-strictness to control parallelism more flexibly. We apply the notion of fuel as a resource that dictates parallelism generation, in particular, the bi-directional flow of fuel, implemented using a circular program definition, in a tree structure as a novel way of controlling parallel evaluation. This is the first use of circular programming in evaluation strategies and is complemented by a lazy function for bounding the size of sub-trees.

We extend these control mechanisms to graph structures and demonstrate performance improvements on several parallel graph traversals. We combine circularity for control for improved performance of strategies with circularity for computation using circular data structures. In particular, we develop a hybrid traversal strategy for graphs, exploiting breadth-first order for exposing parallelism initially, and then proceeding with a depth-first order to minimise overhead associated with a full parallel breadth-first traversal.

The efficiency of the tree strategies is evaluated on a benchmark program, and two non-trivial case studies: a Barnes-Hut algorithm for the n-body problem and sparse matrix multiplication, both using quad-trees. We also evaluate a graph search algorithm implemented using the various traversal strategies.

We demonstrate improved performance on a server-class multicore machine with up to 48 cores, with the advanced fuel splitting mechanisms proving to be more flexible in throttling parallelism. To guide the behaviour of the strategies, we develop heuristics-based parameter selection to select their specific control parameters.

In memory of my dad, and to my mum.

Acknowledgements

The last four and a half years have been a unique journey. It was marked by moments of excitement when things worked well, and the desire to discover and do more, and periods of frustration when things did not seem to go quite well. Even though completing the PhD was always my number one priority, it was easy to lose focus at times with other priorities and challenges in one's life. The thought of whether I would ever finish the PhD was always at the back of my head. Now that I have completed the dissertation, I am elated. I could not have succeeded without the invaluable support of many.

The person who deserves the most credit is my primary supervisor, Hans-Wolfgang Loidl. I am grateful to him for giving me the opportunity to do research under his supervision. He has spent much time and effort following my experiments closely from the beginning and was always available when needed. I thank him for his guidance, advice and, mostly, for his patience with me, especially when I did not seem to know what I was doing and was painfully trying to explain it to him. In the tedious writing phase, his meticulous reviews and suggestions were valuable. Thanks also go to my other supervisors: Phil Trinder, for his critical comments on my writing, and Sven-Bodo Scholz and Murray Cole, for useful advice and discussions.

I thank members of my examination committee, Kevin Hammond, Josef Svenningsson and Greg Michaelson, for their constructive comments to improve my thesis.

I thank fellow PhD students, friends and members of the Dependable Systems Group at the department, in particular, Evgenij Belikov, Konstantina Panagiotopoulou, Pantazis Deligiannis, and Rob Stewart, for providing a friendly environment to work and to share ideas on research and in general. I value the inspiring discussions and lively debates we have had especially over pints at the pub.

I am indebted to the Scottish Informatics and Computer Science Alliance (SICSA) for sponsoring me through a PhD studentship for 3.5 years, and the EU FP7 ORIGIN project to offer me support under an 8 months research assistantship.

I am forever grateful to my wonderful family back home whom I could only see on two occasions during the PhD. They have been my source of inspiration and encouragement. Finally, words cannot express the gratitude I owe to my wife who has been by my side all along this journey.

Prabhat Totoo
Edinburgh, May 2016.

ACADEMIC REGISTRY
Research Thesis Submission

Name*:*	PRABHAT TOTOO		
School/PGI:	MACS		
Version: *(i.e. First, Resubmission, Final)*	Final	Degree Sought (Award **and** Subject area)	PhD in Computer Science

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

1) the thesis embodies the results of my own work and has been composed by myself
2) where appropriate, I have made acknowledgement of the work of others and have made reference to work carried out in collaboration with other persons
3) the thesis is the correct version of the thesis for submission and is the same version as any electronic versions submitted*.
4) my thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying and be available via the Institutional Repository, subject to such conditions as the Librarian may require
5) I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.

* *Please note that it is the responsibility of the candidate to ensure that the correct version of the thesis is submitted.*

Signature of Candidate*:*		Date:	02.05.2016

Submission

Submitted By *(name in capitals):*	
Signature of Individual Submitting:	
Date Submitted:	

For Completion in the Student Service Centre (SSC)

Received in the SSC by *(name in capitals):*			
Method of Submission *(Handed in to SSC; posted through internal/external mail):*			
*E-thesis Submitted (**mandatory for final theses**)*			
Signature:		Date:	

Contents

List of Tables

List of Figures

List of Abbreviations and Acronyms

CAS	Compare-And-Swap atomic instruction.
CPU	Central Processing Unit.
CUDA	NVIDIA's Compute Unified Device Architecture API.
DPH	Data Parallel Haskell.
EdenSkel	Eden Skeleton library.
EdenTV	Eden Trace Viewer profiling tool.
GC	Garbage Collection.
GHC	Glasgow Haskell Compiler.
GHC-Eden	Parallel Haskell Eden compiler.
GHC-SMP	Glasgow Haskell Compiler for Symmetric Multi-Processor.
GPGPU	General-Purpose Graphics Processing Unit.
GpH	Glasgow parallel Haskell.
MIMD	Multiple Instruction, Multiple Data architecture.
MPI	Message Passing Interface communication library.
MUT	Mutation time.
NUMA	Non-Uniform Memory Access.
PE	Processing Element.
PGAS	Partitioned Global Address Space programming model.

PLINQ	Microsoft's Parallel Language-Integrated Query.
RT	Running time.
RTS	Run-Time System.
SIMD	Single Instruction, Multiple Data architecture.
SoC	System-on-Chip.
SP	Speed up.
TBB	Intel's Threading Building Blocks C++ template library.
TPL	Microsoft .NET Task Parallel Library.
TSO	Thread State Object.
UMA	Uniform Memory Access.
WHNF	Weak Head Normal Form evaluation degree.

List of Publications

Part of the work presented in this thesis is derived from the author's contribution to the following papers. The author's main contribution in (Belikov et al., 2013) is the systematic classification of parallel programming models, including the table and figure, which is presented in the background research in this thesis in Chapter 2. The background research covers materials from (Totoo et al., 2012) in parallelism support in modern functional languages. Chapter 3 draws from work published in (Totoo and Loidl, 2014a) and Chapter 4 is a revised version of materials published in (Totoo and Loidl, 2014b).

1. Totoo, P., Deligiannis, P., and Loidl, H.-W. (2012). *Haskell vs. F# vs. Scala: A high-level language features and parallelism support comparison.* In Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC 12, pp. 49-60, New York, NY, USA. ACM.
   DOI: 10.1145/2364474.2364483

2. Belikov, E., Deligiannis, P., Totoo, P., Aljabri, M., and Loidl, H.-W. (2013). *A survey of high-level parallel programming models.* Technical report, Heriot-Watt University, Edinburgh.
   Tech.Rep. No.: HW-MACS-TR-0103

3. Totoo, P. and Loidl, H.-W. (2014a). *Parallel Haskell implementations of the N-body problem.* Concurrency Computation: Practice and Experience, Vol. 26(4), pp. 987-1019.
   DOI: 10.1002/cpe.3087

4. Totoo, P. and Loidl, H.-W. (2014b). *Lazy data-oriented evaluation strategies.* In Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC 14, pp. 63-74, New York, NY, USA. ACM.
   DOI: 10.1145/2636228.2636234

Prabhat Totoo

Chapter 1

Introduction

Hardware is increasingly parallel and efficient. Efficient parallel programming is needed to exploit its computational power. However, parallel programming has the deserved reputation of being difficult. Harnessing the enormous computational power of parallel hardware remains a challenging task. Based on the sequential von Neumann machine, mainstream parallel programming technologies have inherent issues: they require the programmer to handle many aspects of low-level parallel management such as thread synchronisation, data access and exchange. Consequently, it becomes difficult to ensure correct behaviour of parallel programs, performance scalability and portability.

Writing a parallel program entails specifying the *computation*, i.e. a correct, efficient algorithm, and the *coordination*, i.e. arranging the computations to achieve good parallel behaviour. Specifying full coordination details is a significant burden on the programmer. Therefore, choosing the right programming model with the right level of abstraction and degree of control is a crucial step for productivity and performance, respectively.

High-level parallel programming models simplify parallel programming by offering powerful abstractions and by hiding low-level details of parallel coordination (Loidl et al., 2003). Most aspects of parallel coordination are encoded in the underlying system, and, often, when using a high-level model, the programmer only has to identify potential parallelism. In particular, functional programming represents significant advantages for parallel computation (Hammond and Michaelson, 2000).

Functional languages are based on the theoretical and mathematical computation model of function abstraction and application, lambda calculus, provide referential transparency, support higher-order functions, and, important to parallel programming, the absence of data races due to side-effect-free functions. The key advantage

derived is deterministic parallelism: the parallel program has the same (simple) semantics as the sequential. Functional languages provide a high degree of abstractions and expressiveness, enabling the parallel programmer to only specify *what* value the program should compute instead of *how* to compute it. Managing parallelism is all about *how* and therefore largely hidden from the programmer. Furthermore, functional programming is gaining widespread adoption with languages such as F#, Scala and Erlang (Syme et al., 2007; Odersky et al., 2006; Armstrong et al., 1995) building on the strength of functional programming to facilitate both control and data parallelism. Haskell (Hudak et al., 1992) is an advanced purely-functional language that offers a very high-level approach to specifying parallelism.

A large number of parallel programs fall under the data-parallel category where parallelism is derived from data decomposition. Data, usually represented in arrays or lists, are distributed and processed simultaneously by different processing units. Parallel data structures are important components of data-parallel algorithm and programming. A parallel data structure is specifically designed to take advantage of parallel evaluation of independent components, often in an implicit way, and scales with both data size and processing capabilities.

1.1 Thesis Statement

Central to this research are *laziness* and *strategies*.

Laziness A rather unusual combination of concepts is that of laziness and parallel evaluation. Essentially, laziness entails delaying computation, whereas parallel evaluation seeks to execute as much as possible to have expressions reduced to values as soon as possible. Laziness can be useful in the administration of parallelism. And this is a main component of study in this thesis.

Evaluation Strategies

Algorithm + Strategy = Parallelism (Trinder et al., 1998).

Evaluation strategies (Trinder et al., 1998a) provide a high-level way of specifying the dynamic behaviour of a Haskell program. Evaluation strategies use laziness to separate the computation from the coordination aspects of parallelism. As higher-order functions, strategies can be composed, passed as arguments, and be defined for most control and data structures. Additionally, its high-level coordination notation means that different parallelisations require little effort.

This thesis investigates the use of laziness in the implementation of evaluation strategies that operate on data structures to improve parallel performance. In particular, it looks at the high-level specification of data-parallelism over Haskell's built in list type, and custom-defined tree and graph types, using functional programming and lazy evaluation techniques. The suitability of lazy programming techniques for parallel computation has not been studied in similar level of detail, and this is a main focus of the thesis.

The main type of parallelism covered is data-parallelism, enabled through parallel evaluation strategies that operate on data structures that are carefully represented for efficient parallel processing. The thesis covers approaches and methods to defining such strategies. We take advantage of laziness as a language feature, the ability to define circular programs that derives from it, and other functional programming techniques for efficient parallelism control in the implementation of strategies for tree and graph data-structures that can be re-used in a range of different applications.

A key thesis hypothesis is laziness can be exploited to make parallel programs run faster. Parallel performance can be improved and parallel programming can be simplified by enabling parallel evaluation of data structures, supported with lazy evaluation techniques. Using data-structure-driven parallelism through the careful choice of data representation, and parallel operations, programs should benefit from parallelism for free or with minimal code changes. Parallel data structures are represented in a way that favours independent evaluation of sub-components, for example, tree-based representation is preferred over sequence, and with implicitly parallel operations. Parallel coordination is expressed at a very high level of abstraction using Haskell evaluation strategies.

The efficiency of our method is validated through experimental evidence presented in meticulous evaluation on constructed test program as well as non-trivial applications (Section 4.12). We are primarily concerned with execution time, but also usage of space through careful optimisation to limit space.

The thesis advocates the use of high-level parallel programming models in modern functional languages that abstract most of the complexities of parallel coordination, thus enabling specifying parallelism in a minimally intrusive way. Experimental evidence supports our claim that high-level programming techniques are suitable to gain reasonable performance on medium-scale multi-core machines (desktops 8-16 cores) and larger-scale server-class many-core machines with up to 48 cores.

Following the same design philosophy of algorithmic skeletons, the focus on data structures will provide easy-to-use parallelism by using the right data structure. The parallel structure inherently implements advanced parallelism control mechanisms

e.g. through automatic or heuristic-assisted granularity control.

In particular, this research seeks to answer the following main research questions:

- Non-strictness and parallel evaluation: What is the conflict between laziness and parallel evaluation? Can we take advantage of laziness to arrange parallel computation, in particular, for the parallel manipulation of data structures? What are the issues of laziness as the default strategy, especially with regard to parallel data structure implementation?

  Laziness and parallel computation seem contradictory. Indeed, any parallel execution requires a certain degree of strictness to kick-start. What is the right default of evaluation mechanism? Can we still build on non-strict semantics by default and benefit from parallelism, in particular for infinite data structures and circular programs which depend on non-strictness.

- The efficiency of tree-based parallelism: Is the underlying representation of a data structure important for parallel processing? Does a tree-based representation enable more efficient and effective parallelisation strategies?

  In particular, we seek to substantiate the benefits of using recursive tree-based representation over a linear representation, in particular, through implementation of advanced parallelism control mechanisms over recursively defined tree-based data structures. This representation can be adopted by inherently sequential data structures e.g. linked-list through an append-tree representation which lends itself more easily to parallelisation. We seek to minimise the overhead of representation change, and investigate if it could be recovered through performance gain from parallelisation.

- Auto-parallelisation and parallelism for free: What aspects and degree of control should be left to the programmer?

  Certain parameters to control parallel execution can be programmer-specified or fully automated. Is it better to allow the library to determine the right amount of parallelism generation, for example, by implementing strategies that can auto-tune through heuristic-based parameter selection?

  How much automatic parallelisation can we achieve? The use of data structures with inherent parallel operations is expected to require minimal change to the sequential algorithm.

More widely, we seek to answer the following:

- What are the benefits of using functional programming for parallel computation? Does the high-level of abstraction facilitate the specificaton of parallelism

in the language, in particular, whether it is sufficiently powerful for efficient data-oriented parallelism? What are the limitations of expressiveness?

A higher level of abstraction often comes with decreased level of flexibility and control, both of which are needed for specific tunings. We seek to answer whether data structures expressed in a strongly typed system can be parallelised using high-level language constructs, and what are the limitations presented.

1.2 Contributions

The main research contributions of this thesis are:

- novel evaluation strategies for controlling parallelism;
- implementation of the novel strategies in Haskell;
- experimental evaluation of properties of the strategies on appropriate exemplars on large scale many core platforms.

The detailed contributions are presented as follows:

1. *Parallel Haskells programmability and performance evaluation based on list and tree processing* (Totoo and Loidl, 2014a) (presented in Chapter 3). We present a comparative study of high-level parallel programming models in Haskell, covering parallel extensions (GpH, Eden) and libraries (Par monad), by implementing two highly tuned n-body problem algorithms: a naive allpair version using list and a realistic Barnes-Hut algorithm using list and tree.
2. *Lazy data-oriented evaluation strategies for tree-based data structures* (Totoo and Loidl, 2014b) (presented in Chapter 4). We implement a number of flexible parallelism control mechanisms defined as data-oriented evaluation strategies for tree-like data structures. In particular, we use the concept of fuel as a more flexible mechanism to throttle parallelism. To specify the administration of the parallel execution, we add annotations to the data structure and perform lazy size computation using natural numbers to avoid full data structure traversal. We achieve flexible control through the application of lazy evaluation techniques, including the ability to have circular programs definition, for parallel coordination, and use heuristic-based methods for auto-tuning strategies.

3. *Parallel graph traversal strategies* (presented in Chapter 5). We implement graph strategies using similar control mechanisms to those used for tree data structures. A number of sequential and parallel traversal strategies are developed, further analysing the interplay between laziness and parallelism to deal with cyclic graph instances. We develop a hybrid traversal strategy for graphs, adapting the concept of iterative deepening from artificial intelligence, to have improved parallelism generation in a breadth-first order initially, and then proceed depth-first to minimise overhead associated with a full parallel breadth-first traversal.

1.3 Thesis Structure

The thesis is structured in the following chapters.

Chapter 2 provides a literature review on parallel hardware, programming models and higher-level approaches to parallelism. It covers a range of parallel programming models and attempts to classify various models based on language properties and level of abstraction and coordination provided. The chapter motivates parallel functional programming, and emphasises Haskell as the main implementation language. The chapter also provides a brief history of laziness.

Chapter 3 covers Haskell evaluation strategies in more detail and highlights methods for developing parallel list strategies. The chapter evaluates this model against two other parallel Haskells – Par monad and Eden – both on shared-memory machines. It provides a detailed performance analysis and evaluation of the three parallel frameworks in Haskell based on the implementation and parallelisation of two algorithms for the n-body problem.

Chapter 4 extends the methods for developing data-oriented evaluation strategies that further exploit lazy evaluation techniques and implement flexible parallelism control mechanisms for parallel evaluation of tree data structure sub-components. It covers the use of heuristics for automatic parameter selection and describes three applications that internally use trees to test and evaluate the strategies.

Chapter 5 carries over techniques used in the previous chapter and applies them to the more complex graph data structure which uses a tree-based representation. It covers implementation of a number of sequential and parallel traversals on graphs. These traversals are tested as graph search algorithms over different types of graph including cyclic which requires lazy constructs for its representation.

Chapter 6 draws conclusions from our experiments and results, and highlights on-

going work to further improve performance without sacrificing key properties such as separation of concerns, expressiveness, and compositionality.

Chapter 2

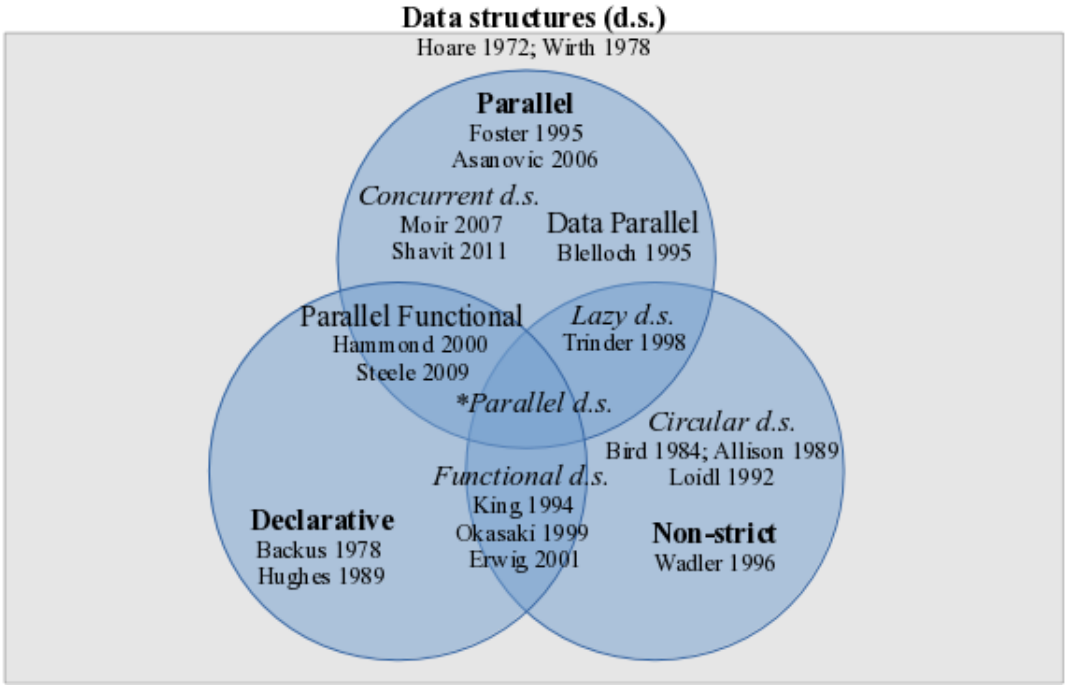
Background

This chapter provides a literature review on parallel hardware trends and associated parallel programming models used in both mainstream software industry and research. It presents a classification of programming models based on language properties and grouped into different classes of languages. In particular, the chapter highlights the problems with traditional technologies with mainstream parallel programming, and it emphasises higher-level approaches to parallel programming, through parallel patterns and parallel functional programming, where both abstract over the low-level complexities from application programmers. It describes Haskell as the main language used in this research for its built-in concurrency and parallelism support. Lastly, the chapter provides a brief history of laziness and covers trends in data structures for parallel programming.

2.1 Research Overview

The research in this thesis is within the broad area of parallel programming, centred on three specific topics: high-level declarative style parallel programming, specifically, parallel functional programming; data-parallelism; and non-strict evaluation in the context of arranging parallel computation. Figure 2.1 identifies key references from literature of the principle research concepts.

- *Parallel Programming.* At the centre of this research is parallel programming which aims at reducing the execution time by arranging computation and evaluation in parallel to exploit multiple cores. A programming model that allows us to expand to network of multi-cores with little to no change to an existing algorithm is highly desired. Our emphasis is on high-level parallel programming models.



Parallel processing of lazy data structures in a functional setting.

Figure 2.1: Main research concepts

- *Parallel Declarative Programming.* Conventional parallel programming is highly complex and error prone. To improve programmer's productivity, we need to raise the level of abstraction with a higher-level programming model that hides most aspects of managing parallel execution in the system. Our research takes a high-level approach to parallelism. This is often achieved in a declarative style of programming, in particular, parallel functional programming.

- *Non-Strict Evaluation.* Non-strictness represents an interesting language property, which can be exploited to make programs, and in particular, data structures, more efficient if used correctly. Laziness allows us to write elegant code that would be otherwise impractical in a strict language. However, laziness may seem incompatible with parallel evaluation. Through careful implementation, laziness can be exploited in conjunction with parallel evaluation to improve performance.

- *Data-Oriented Parallelism* Data structures designed with parallelism in mind can significantly improve performance with minimal to no code change to the sequential algorithm. Our data-structure-driven approach encourages a data-centric approach where parallelism is derived through generic parallel traversal and evaluation of components of data structures. We seek performance gain from the use of efficiently designed, represented, and implemented data structures that favour parallel evaluation.

2.2 Parallel Hardware

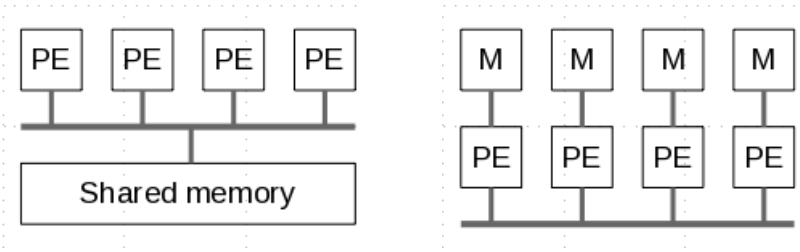
Parallel machines are ubiquitous with desktops, laptops and even mobile phones containing two or more cores. The trend is heading towards tens or hundreds of processing units in commodity hardware (Asanovic et al., 2006). Performance gain cannot be expected by upgrading to a newer processor as it was previously possible by exploiting higher clock frequency, which has now stalled. Instead, application programs need to be re-written for parallel processing to take advantage of multiple processing units. Parallel programming is the dominant technology to harness increasingly parallel hardware potential. Computational problems that were previously unfeasible for serial processing can now be solved using parallel processing. It is, however, harder than sequential programming, as it requires the programmer to think in parallel from the offset.

The parallel hardware landscape is changing faster than software technology especially since around 2000 with GPGPUs (Owens et al., 2007) and 2005 with multi-cores becoming mainstream. Initiated by the physical limits of the number of transistors that can be embedded in a single processor, the default is now to have multiple compute cores in a chip. Soon, the multi-core machines will be superseded by *many-core*, for example, the Intel MIC-based architecture Xeon Phi co-processors (Chrysos, 2012), and the lower cost and open source alternative Parallella platform (Adapteva, 2015), have 60 and 64 cores, respectively. General-purpose GPUs are now the hardware of choice for many data-intensive and data-parallel applications. FPGAs (Sadrozinski and Wu, 2010) offer even higher performance, at higher programming cost, and are used in niche areas.

Flynn's taxonomy classifies computer architectures based on the number of instruction and data streams (Flynn, 1972). Data-parallel systems, which includes GPUs and accelerators, are classified as *single instruction, multiple data* (SIMD) machines. Most general-purpose parallel systems fall under the *multiple instruction, multiple data* (MIMD) category. In this architecture type, the processing units work on separate instruction and data streams, thus encompassing a wider range of parallel applications. MIMD machines are usually classified based on the assumptions they make about the underlying memory architecture – *shared* or *distributed*.

2.2.1 Shared Memory

In shared memory architectures, all processors operate independently but share the same physical memory. Changes in a memory location effected by one processor are visible to all other processors. Shared memory architectures are further divided into



(a) shared (b) distributed

Figure 2.2: Shared and distributed memory architectures

two classes based on the memory access time (Tanenbaum and Austin, 2012, ch.8): Uniform Memory Access (UMA) – which mainly represents Symmetric Multiprocessor (SMP) machines with identical processors and equal access times to memory; and Non-Uniform Memory Access (NUMA) – in which the access time to all memory locations is not the same for all processors. NUMA is becoming the norm with increasing core numbers. NUMA machines are usually made up of two or more SMPs grouped together, and making the memory of each SMP directly accessible by the others. For both classes, cache coherency can be achieved at hardware level, by making sure update made by one processor is seen by all others. Again this is a source of overhead increasing with core numbers and it is unclear whether future generations of many-cores will support full cache coherence. The main drawback of shared memory architecture is scalability: increasing the number of processors leads to high memory contention causing bottleneck.

2.2.2 Distributed Memory

In distributed memory architectures, there is no global view of all memories. Each processor has its own memory attached to it and is connected via a network to other processors. A processor has quick access to its local memory but requires explicit communication, thus high access times, for data residing on another processor. This hardware model is, however, scalable since memory traffic is not an issue with increasing number of processors. Irrespective of the physical memory layout, languages sometimes implement a virtual shared memory view. This helps to abstract the physical distribution of memory.

Recent architectural trend indicates a move to increasingly heterogeneous hardware (Belikov et al., 2013). It is common for mobile devices to come with system-on-a-chip (SoC) processors (Furber, 2000; Risset, 2011) integrating general-purpose CPU with other compute cores e.g. GPU and DSP. Programming these increasingly complex hardware is a difficult task. It requires the programmer to not

only think about the computation but also the coordination aspects of parallelism. Architecture-independent programming models that abstract over underlying hardware are key.

2.3 Parallel Programming and Patterns

Concurrent and *parallel* programming are often used to mean the same thing. It is important to first clarify our use of the term parallel programming, both in general and in the context of parallel data structures. Concurrency is the property of a system in which independent computations can run simultaneously. On the other hand, parallelism is meant to execute computations in parallel for the main purpose of improving performance through the efficient use of available resources on separate CPUs.

In a concurrent system, a moderate number of threads or processes cooperate and communicate with each other to synchronise their actions. This is often through shared variables or by message passing. In such a system, performance is not key. Proper synchronisation ensures correctness and consistency by avoiding race conditions and deadlocks. Some parallel systems are built on this model to make programs run faster: independent threads can execute on different processing elements (PEs), thus leading to reduced execution time for a particular system. As the hardware has moved from single core to multicore, and now manycore, this revolution requires parallel programming that can use several processing units at the same time is becoming increasingly important to exploit the available resources. Traditional programming models originated from the high-performance computing community using low-level languages such as C and Fortran with language extensions or libraries to enable parallel execution. For example, C+OpenMP (OpenMP, 2012) and C+MPI (ANL, 2012) are widely used on shared-memory single-node machine and networks of nodes, respectively. These models offer detailed control of parallel execution to the programmer and as such leave many opportunities for errors in the parallel program. Debugging such programs is hard with often non-deterministic outcome of parallel runs. Other thread-based models are also non-deterministic and require careful programming to avoid data races. Deterministic models often do not expose the explicit notion of a thread or task to the programmer and handle most parallel coordination implicitly in the system.

Designing parallel program from scratch is not easy especially in the absence of a proper methodology. Unstructured parallel programming not only leads programmers to "re-invent the wheel", but it also does not allow the underlying run-time system to exploit any pre-defined structure in the computation for optimisation

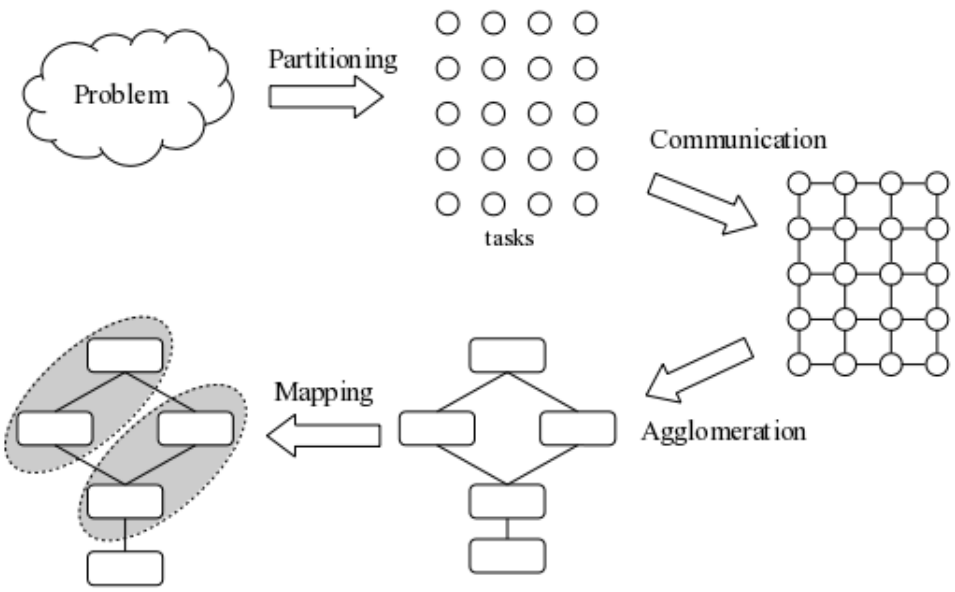


Figure 2.3: Foster's parallel algorithm design methodology.

opportunities. To help with the difficulties of parallel programming, a number of best practices or standards have emerged in the last decades. For example, Foster's design methodology (Foster, 1995) emphasises a model which is based on tasks connected by channels, hence well-suited for distributed memory, and a methodical four-stage approach to designing parallel algorithms (depicted in Figure 2.3). In brief, the four stages are:

1. *Partitioning.* Identify parallel tasks through domain (data) or functional decomposition. A key issue in this phase is ensuring comparable task sizes so each processor has equal workload to facilitate load balancing.

2. *Communication.* Identify channels of communication between tasks. The aim here is to limit communication to a few neighbours, and local communication is preferred over global communication.

3. *Agglomeration.* Combine tasks to improve performance or reduce overheads by improving locality and granularity.

4. *Mapping.* Allocate tasks to physical processors using static and dynamic methods for regular and irregular computation, respectively. This step depends on architecture details such as number of processors.

Design patterns (Gamma et al., 1995) encourages structured coding and code re-use by specifying generalised solutions to recurring problems. A pattern language for parallel programming, inspired by design patterns, is described by (Mattson et al., 2004). Design patterns have been hugely successful to help programmer to "think OOP", and now to "think parallel" for parallel programming. Parallel programs

exhibit common patterns that can be abstracted. Parallel patterns abstract certain details and delegate them to the system, while offering an easy-to-use interface to the programmer.

2.4 A Survey of Parallel Programming Models

A parallel programming model abstracts hardware and memory architectures, and aims to improve productivity and implementation efficiency. Parallel programming models are diverse (see Asanovic et al. (2006); Silva and Buyya (1999); Skillicorn (1995); Diaz et al. (2012); Van Roy (2009)) and there is no clear-cut way of classifying them. We identify the following key language properties of a parallel programming model, that is, the main classification for the scope of this thesis, and provide examples from different classes shown in Table 2.1:

2.4.1 Language Properties

The properties (listed across the columns on Table 2.1) highlight key characteristics of a language (Belikov et al., 2013).

- *Coordination* What is the level of abstraction provided by the language?
- *Parallelism Types* Does the language support task or data parallel applications, or both?
- *Memory programming model* How do threads communicate with each other?
- *Parallel programs behaviour* Does parallel execution result in deterministic or non-deterministic behaviour?
- *Embedding techniques* How is parallelism introduced into the code?

Coordination Abstraction

Coordination abstraction refers to the degree of explicit control required by the programmer to manage parallelism and access to shared resources. A high level of abstraction leads to higher productivity and reduced risk of introducing errors in the parallel program at the potential cost of decreased performance (Spetka et al., 2008). This is often the case with implicitly parallel language, without any explicit control of parallelism. Other, semi-explicit, languages, only expose potential parallelism, while

more explicit and low-level languages have constructs for generation and handling of explicit threads.

Very Low-Level: Models at a very low-level leave all aspects of parallel coordination to the programmer. Any model requiring knowledge of underlying hardware and architecture is essentially very low-level. GPU programming is put at this level, as the model is often vendor-specific. For instance, CUDA targets only NVIDIA GPUs (NVIDIA, 2008). Other models for heterogeneous CPU and GPU systems are OpenCL (Stone et al., 2010) and OpenACC (Wolfe, 2013), and not discussed in more detail due to their very low level abstraction.

Low-Level: Such models expose most coordination issues such as problem decomposition, communication, and synchronisation to the programmer (Diaz et al., 2012). These issues are orthogonal to the algorithmic problem, require additional effort, and thus reduce productivity. Dealing with these is notoriously difficult, which constitutes the challenge of parallel programming. Low-level models offer extensive tuning opportunities for expert programmers at the cost of significant effort. In Table 2.1 we classify Java Threads (Lea, 1999) and MPI as low-level languages as thread management and interaction is fairly explicit.

Mid-Level: Models in this category hide some of the coordination issues from the programmer, in particular thread and memory management, as well as mapping of work units to threads. Mid-level models attempt to strike a balance between the performance benefits through available tuning opportunities and the productivity advantage through the increased level of abstraction. A mid-level language example is OpenMP (2012) in which the programmer uses directives to identify parallel regions and the compiler generates the threaded code. The Task Parallel Library (Leijen et al., 2009) can also be classified as mid-level. Although it provides task abstraction and the run-time system automatically maps tasks that represent distinct units of work to worker threads, the programmer is still responsible for synchronisation and splitting work into tasks.

High-Level: These models abstract over most of the coordination, often leaving only advisory identification of parallelism to the programmer. Usually built on top of basic parallel constructs, these models provide a more structured way of describing parallelism through the use of abstractions that encapsulate common patterns of computation and coordination. As we move to a higher-level model, parallel coordination becomes more implicit. High-level models, e.g. those that implement algorithmic skeletons (more detail and examples in Section 2.5.1), can offer an architecture-independent interface while providing multiple parallel backends to retain performance across different architectures. The programmer needs merely to select suitable skeletons and get parallelism for free. High-level models

offer the most powerful abstractions whilst substantially complicating the efficient implementation of the underlying language or library.

Types of Parallelism

There two basic types of parallelism, *data* and *task*, corresponding to the domain and functional decomposition techniques, respectively, of a computational problem (Foster, 1995), i.e., independent data and task processing. Some models are specialised for either data or task parallel applications, but many support both types.

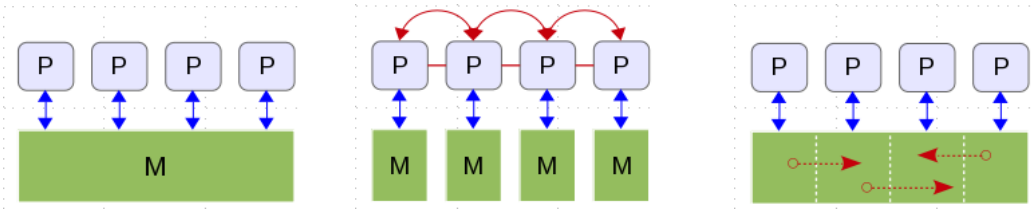
Data parallel: This involves breaking down a large data set into smaller chunks that are processed in parallel by applying the same function to each chunk. The outputs from each processor are aggregated to produce the final result. Many problems, including embarassingly parallel ones, fit into data parallel category. Data-parallel specialised hardware such as GPUs are extensively used for these problems. There are several possible ways of distributing data across processors, including static (allocated at the start of computation) and dynamic (happens during execution to improve load balancing). Data-parallel also encapsulates parallel data structures designed with performance in mind. They have efficient underlying representation suited for parallel processing and implicitly implements parallel operations on the structures.

Task parallel: This programming model exploits the fact that a problem can be structured based on inter-dependencies among separate tasks. Parallel execution is achieved by running a separate thread for each independent task. The granularity of tasks can be varied in this model, and more advanced load balancing and dynamic scheduling is required. Often, data parallelism is implemented on top of a task parallel framework.

Memory Programming Model

The memory model of programming describes how threads interact and synchronise with each other in memory. The two main ways are through *shared access* and *message passing* (Shan et al., 2003):

Shared-access: Multiple threads share a single address space. In this model, locks are used as semaphores to control access to the shared memory. This ensures that shared data is not manipulated by more than one thread at a time thus ensuring no race conditions. This coordination is required for synchronisation in thread-based parallel programs. Some languages provide higher-level constructs e.g. barriers to



(a) shared memory model (b) distributed memory model (c) PGAS model

P: process/thread/task
M: memory address space
(blue arrow): direct memory access
(red arrow): message passing

Figure 2.4: Memory programming model

avoid the issues such as deadlocks. These models work well on medium-scale parallel machines. However, as the number of processors increases, the synchronisation overheads grow higher with increased contentions.

Message-passing: Disjoint address space over all compute nodes. This programming model adopts explicit message communication between threads, thus removing many of the issues associated with sharing variables. Synchronisation is realised through sending and receiving of messages. There can still be race conditions if message passing is asynchronous. This model is well-suited for distributed systems. Message-passing is used in actor-based programming model, e.g. in Scala Actor (Haller and Odersky, 2009) and Erlang's process communication (Armstrong et al., 1995), to achieve high scalability on very high number of processing nodes.

The programming model usually, but not necessarily, reflects the underlying memory architecture. For instance, on a SMP, shared-memory programming is common, but it is also common to adopt a message-passing model, possibly with an increased overhead. Similarly, distributed systems may abstract over distributed memory as a single virtual memory space, as is the case in the Partitioned Global Address Space (PGAS) model (PGAS, 2015). PGAS supports a shared namespace, like shared memory. In this model, tasks can access any visible variable, local or remote. Local variables are cheaper to access than remote ones. PGAS retains many of the downsides of shared memory. Programming model using a hybrid approach uses shared-access on local nodes, and message-passing across nodes over the network. Hybrid programming model is also suited for heterogeneous computing.

Parallel Programs Behaviour

A deterministic model of parallelism guarantees that the parallel execution yields the same results as the sequential execution of a program. This is not always the

case with languages that implement parallelism through concurrency. For instance, programming models such as Java Threads depend on basic concurrency constructs to implement parallelism and are non-deterministic, requiring the programmer to explicitly manage the synchronisation among threads to ensure correct program behaviour.

By contrast, invoking the parallel version of a query on a PLINQ (Campbell et al., 2010) object and the `par` combinator in GpH (Trinder et al., 1998b), are both deterministic. Deterministic models can be achieved by design or implemented by building on top of non-deterministic constructs and providing a deterministic library, e.g the parallel Par monad library for Haskell (Marlow et al., 2011). The main advantage of deterministic programming models are that they abstract over low-level thread management issues such as synchronisation, hence prevent the appearance of race conditions and deadlocks.

Language Embedding

There is a range of different technologies to embed support for parallel programming into a host language. Starting from a fresh language design, first-class primitives for parallelism are the most obvious choice, maximising the flexibility and allowing to use standard language concepts. When extending an existing language, new parallel features can be provided as pre-processor or compiler directives, or built on top of available low-level concurrency primitives. Often libraries are used to provide similar features in a less invasive way (Marlow et al., 2011; Maier and Trinder, 2012). However, this approach is restricted to the optimisations available in the host language. Alternatively, a separate coordination language can be used to specify parallel execution and communication (Gelernter and Carriero, 1992), separating the concerns of computation and coordination. Sufficiently high-level languages enable seamless embedding of the coordination language in the host language, as exemplified by Evaluation Strategies (Marlow et al., 2010) for GpH.

2.4.2 Classes of Programming Models

Now we group different programming models based on the emerging clusters. Figure 2.5 depicts models across the dimensions of computation, i.e. the algorithmic solution, and coordination, i.e. the management of parallelism. Below we discuss each group drawing specific examples from Table 2.1.

	Language/Extension	Reference	Coordination Abstraction	Parallelism Type	Memory Model	Deterministic	Embedding
Par.Impe.	MPI/PVM	(ANL, 2012; Sunderam, 1990)	low	task, data	msg pass (expl.)	no	library
	OpenMP	(OpenMP, 2012)	mid	data, task	shared (expl.)	no	compiler directives
	Cilk	(Cilk, 1998)	mid	task, data	shared	no	C extension
	TBB	(IntelTBB, 2012)	mid	data, task	shared	no	C++ library
Par. OO	Java Threads/Pthreads	(Lea, 1999; Nichols et al., 1996)	low	task, data	shared	no	library
	Fork/Join framework	(Lea, 2000)	mid	task, data	shared	no	library
	TPL	(Leijen et al., 2009)	mid	task, data	shared	no	.NET library
	Concurrent Collections	(Orale, 2015)	high	data	shared	yes	library
Data Par.	ArBB	(Newburn et al., 2011)	mid	data	shared	yes	C++ library
	SAC	(Scholz, 2003)	high	data	shared	yes	new language
	HPF	(Richardson, 1996)	high	data	msg pass	no	Fortran extension
	DPH	(Chakravarty et al., 2007)	high	data	shared	yes	Haskell extension/lib
	PLINQ	(Campbell et al., 2010)	high	data	shared	yes	.NET library
PGAS	CAF	(Numrich and Reid, 2005)	mid	data, task	PGAS	no	Fortran extension
	UPC	(UPC Consortium, 2005)	mid	task, data	PGAS	no	C extension
	Chapel/Fortress/X10	(Weiland, 2007)	high	data, task	PGAS	no	new language
Par. Decl.	CnC	(Newton et al., 2011)	mid	task, data	shared/msg pass	yes	library
	Parallel Haskells	(Berthold et al., 2009)	high	task, data	shared/msg pass	yes/no	extensions/libraries
	Erlang	(Armstrong et al., 1995)	high	task, data	msg pass (expl.)	no	new language
	Manticore	(Fluet et al., 2010)	high	task, data	msg pass	yes/no	SML extension
GPGPU	OpenCL, CUDA	(Stone et al., 2010; NVIDIA, 2008)	low	data	hierar. mem	no	par comp./seq kernel
	Renderscript	(Google Inc., 2011)	mid	data	hierar. mem	no	C-extension/lib
	C++AMP	(Gregory and Miller, 2012)	mid	data	hierar. mem	no	C++ extension/lib
	Offload	(Cooper et al., 2010)	mid	data, task	hierar. mem	no	C++ extension/lib
	SkePU	(Enmyren and Kessler, 2010)	high	data, task	hierar. mem	yes	library
Skel.	Hadoop MapReduce	(Dean and Ghemawat, 2008)	high	data	msg pass, shared	yes	library
	P3L	(Bacci et al., 1995)	high	task, data	implicit	yes	new language

Table 2.1: Parallel Programming Models, Languages and Systems

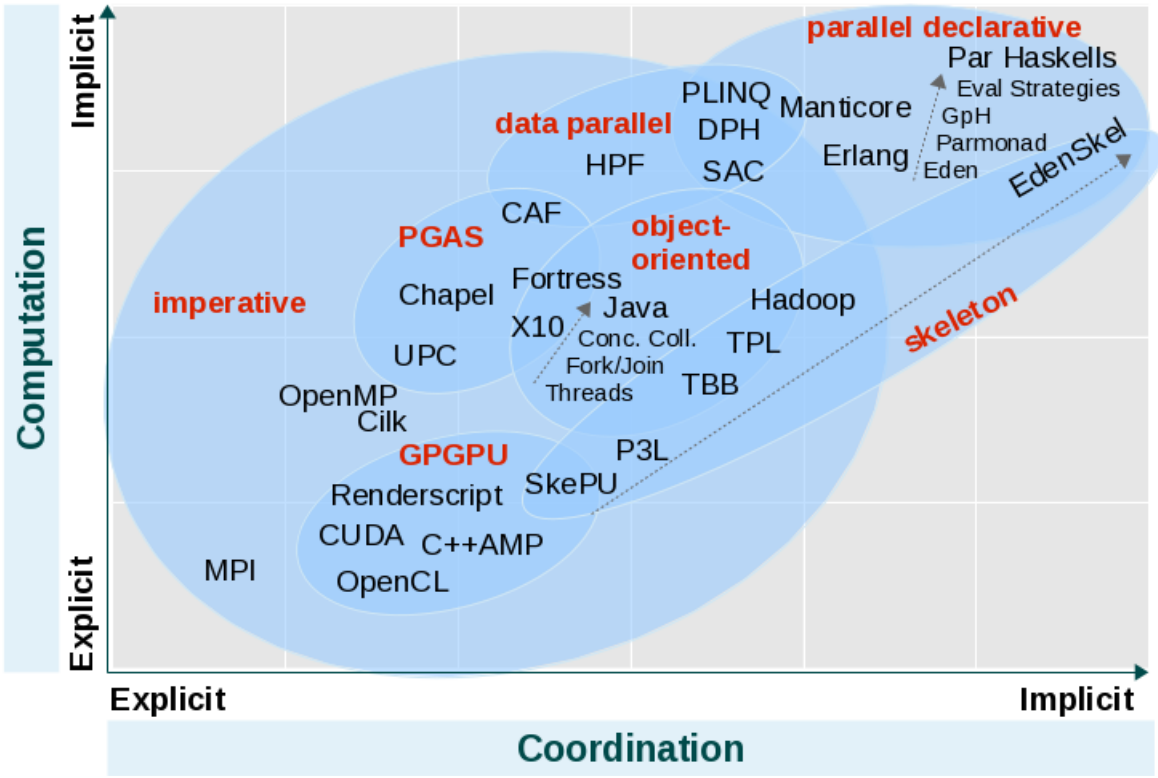


Figure 2.5: Parallel Programming Models – An Overview

Parallel Imperative Imperative languages are based on the concepts of state, side-effects, variable manipulation, pointers, iteration, and program counter to control execution and are rather low-level closely matching the uniprocessor architecture (Van Roy, 2009). Although low-level of abstraction and explicit control enable manual optimisation and may result in high performance on a single architecture, this approach prevents many automatic optimisations which may result in poor performance across other architectures and reduces portability as well as programmer productivity. Nevertheless, these languages are heavily used in the industry (TIOBE Software, 2013) and are likely to remain at the core of system-level software, at least in the near future.

Parallel Object-Oriented Starting from the lowest level, threads are used in object-oriented languages like in Java to run jobs in parallel. The programmer is exposed to the management of threads. Software frameworks such as Fork/Join (Lea, 2000) or, at a higher level, TPL (Leijen et al., 2009) abstract over threads and represent independent units of work as tasks with less management involved with manually creating threads. This is left to the RTS which manages a fixed or dynamic pool of threads and automatically maps tasks to running threads. Even more implicit are libraries of concurrent collections which have efficient parallel implementations of operations on common data structures, e.g. arrays and hash tables. These collections hide concurrent access through implicit synchronisation.

Data Parallel and Array Languages Some languages support only data parallelism via constructs such as parallel for loop and parallel arrays. This fits a large group of applications where parallelism is identified by domain decomposition. Data parallel languages often provide a sequential model of computation and most coordination aspects are almost completely implicit. However, this model is restrictive and unless the application exhibits data-parallelism, it cannot be used. Most languages in this category efficiently handle regular data-parallelism. DPH (Jones et al., 2008) is an instance of languages well-suited for irregular data-parallelism using flattening transformation and distributing equal workload to processing units (Blelloch, 1995).

In Table 2.1, GPGPU is grouped under a separate class, though GPU programming is essentially data-parallel. Data parallel models often exploit GPUs for fine-grained data-parallel computations. For instance, ArBB (Newburn et al., 2011) and SAC (Grelck and Scholz, 2006) can generate vector instructions. GPGPUs and accelerators are increasingly used for massively data parallel computation. The programming models are very low (as mentioned earlier) and often vendor-specific. However, there are libraries (see Section 2.5.1) that provide a high level API and supports different backends for heterogeneous computing by building on top of OpenCL and CUDA for GPU, and OpenMP for CPU.

PGAS The Partitioned Global Address Space (PGAS) abstraction, akin to a tunable virtual-shared memory view, attempts to unify programming by hiding communication and by providing a shared-memory view (Figure 2.4 (c)) on potentially physically distributed memory and is becoming increasingly popular in HPC (PGAS, 2015). Extensions to established languages include Unified Parallel C (UPC) (UPC Consortium, 2005) and Co-Array Fortran (CAF) (Numrich and Reid, 1998) and new developments include X10 (Charles et al., 2005), Chapel (Chamberlain et al., 2007), and Fortress (Allen et al., 2008). Although the level of abstraction is raised, the difficulty of arranging shared-memory accesses re-appears. Moreover, the explicit specification of blocking factors may yield undesirable distributions of shared data that may lead to performance degradation if data locality is impaired, making performance prediction difficult unless the programmer is intimately familiar with the architecture of the underlying target platform.

Parallel Declarative Declarative languages are based on the concepts of immutability, single assignment, isolated side-effects, higher-order functions, recursion and pattern matching, among others. Due to sophisticated compilation techniques and run-time optimisations available because of their foundation in lambda calculus, declarative programs can deliver competitive performance (Mainland et al., 2013). An early functional language with performance competitive to Fortran was

SISAL (Skedzielewski, 1991; Cann, 1992). Moreover, the performance losses are often offset by productivity gains of the declarative approach that encourages writing portable and high quality code. Most importantly, declarative languages better fit modern parallel architectures, since they allow more flexible coordination of parallel execution and avoid over-specifying evaluation order. For example, Manticore (Fluet et al., 2007) supports multi-level parallelism.

Skeletons Algorithmic skeletons abstract commonly used patterns of parallel computation, communication and interaction (Cole, 1991; González-Vélez and Leyton, 2010). Skeletons hide coordination details in possibly multiple underlying implementations that map to different target architectures, and provide a common interface to the application programmers who can focus on the computational solution to a problem. To the programmer, most parallelism is implicit. Skeletons can be composed for both data and task parallel. e.g. parallel map, task farm, d&c (more detail in Section 2.5.1).

2.5 Higher-Level Approaches to Parallelism

A parallel program must specify the *computation*, i.e. algorithm + data structures, and the *coordination*, e.g. partition of computation into sub-computations, tasks placement, communication and synchronisation between them. This makes writing and debugging parallel programs hard and even harder in an imperative language with shared state. Imperative languages are based on serial computers which follow the Von Neumann design. A parallel imperative language requires the programmer to handle both the computation and coordination aspects of a parallel program.

Imperative programming languages often distract the parallel programmer from the main computational core of a problem and the coordination aspects of parallel computation becoming the overwhelming part of the program. Issues such as thread creation, placement, communication and synchronisation are often handled explicitly. By dealing with all these aspects manually, it becomes difficult to ensure the correct parallel behaviour of a parallel program and verification is made difficult. Moving away from low-level details allow the programmer to be more productive by concentrating on the computational problem. The trend in languages is to incorporate higher-level constructs to hide details to some degree.

2.5.1 Algorithmic Skeletons

An increasingly important area of high-level abstractions for parallelism are algorithmic skeletons (Cole, 1991): higher-order functions with pre-defined parallel computation structures. Because they can hide all complexities of the efficient, possibly hardware-dependent, handling of parallelism in a library, it is being picked up as technology of choice in mainstream languages without built-in high-level parallelism support. Prominent examples are Google's MapReduce (Dean and Ghemawat, 2008) implementation, on large-scale, distributed architectures, Intels Task Building Block library (Reinders, 2007), and the Task Parallel Library (Campbell et al., 2010).

Other notable systems for skeleton-based parallelism are: eSkel (Benoit et al., 2005), built in C and run on top of MPI; Muesli (Ciechanowicz et al., 2010), a C++ template library implementing higher order functions, currying, and polymorphic types, and support both data and task parallel skeletons; Skandium (Leyton and Piquer, 2010), a Java library that supports the use of nested skeletons on shared-memory architectures; and P3L (Bacci et al., 1995), a skeleton based coordination language. The SkelCL (Steuwer et al., 2011) skeleton library builds on top of OpenCL and provides high-level abstractions to facilitate programming heterogeneous CPU/GPU systems. Similarly, SkePU is a C++ template library that provides multi-backend skeletons for heterogeneous architectures (Enmyren and Kessler, 2010). The library provides implementations in CUDA and OpenCL for execution on GPU systems, and in OpenMP to exploit multicore processors.

An up-to-date survey on algorithmic skeletons is given in (González-Vélez and Leyton, 2010). It also provides a taxonomy for algorithmic skeleton constructs based on their functionality and categorise the applications into data-parallel, task-parallel and resolution skeletons. In the following, we highlight some of the most common skeletons.

1. *Map/Reduce* is probably the most popular data-parallel skeleton which has its origin in functional languages. The skeleton applies the same function simultaneously to all data elements of a list, optionally, builds up the result by combining intermediate results from workers in a reduce phase. Map/reduce is a combination of task farm (map) and D&C (reduce) (see below). The skeleton has been widely popularised by Google's MapReduce implementation on large-scale, distributed architectures.

2. *Task Farm* is a task-oriented pattern where the master process distributes tasks to worker processes which in turn can distribute to other workers. The master splits input once in n chunks in static task farm; whereas continually assigns

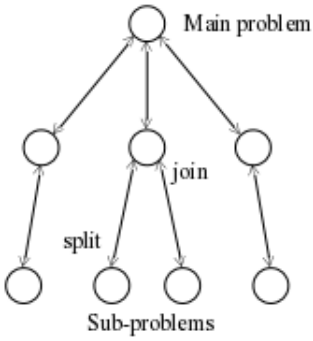


Figure 2.6: D&C call tree structure

input to free workers in a dynamic task farm. The master is a bottleneck in this skeleton.

3. *Pipeline.* This skeleton enables staged computation where different tasks can be run simultaneously on different pipe stages. Ideally, a pipeline needs a large number of input items. A lazy language can overlay stages.

4. *Divide and Conquer.* D&C is a commonly occurring recursive skeleton that creates a call tree structure. A problem is divided in n sub-problems each of which are solved independently as a separate task, and finally, the results are conquered, i.e. combined (*split* and *join* in Figure 2.6).

2.5.2 Parallel Declarative Programming

The proliferation of parallel hardware poses a significant challenge to the way these machines are programmed. Traditionally, the coordination of the parallel execution is controlled in every detail to achieve near-optimal performance. The prevalent programming models are fairly low-level. Issues such as avoiding race conditions, ensuring good load balance and minimising communication cost, needed to be addressed by the expert programmer through explicit language or library constructs. This is only feasible in a setting where a lot of person-effort can be spent on the parallelisation of a single application. Thus, this traditional view of parallelism, *"supercomputing parallelism"*, represents a niche market in the overall domain of computer science, and often confined to experts in the high-performance computing community.

With multi-core hardware now dominating the architecture landscape, a new view of parallelism, *"desktop parallelism"*, is of increasing importance. Here the goal is to achieve some speedup from general-purpose, compute-intensive application, that has been developed and maintained by domain experts, rather than experts in parallel programming. Programming models that can be picked up by mainstream application programmers are thus necessary. Rapid development becomes more

important than raw performance. Many programming models seeking to deal with these difficulties have been developed and the general trend is to move towards higher-level approaches where system programmers hide much of the complexities in the implementation and thus requiring less effort from the application programmer's side to write parallel programs.

With a declarative style of programming, functional languages (Backus, 1978; Hughes, 1989) offer a high level of abstraction. They allow the programmer to focus on *what* instead of *how* to compute. Coupled with high-order functions and advanced type systems with polymorphic types, functional languages offer powerful abstraction mechanisms. One of the main promises of functional programming is seen to be their use for parallel computation. In a parallel setting, there are several reasons why functional programming is suitable (Hammond and Michaelson, 2000). Foremost, because functional languages are not defined in terms of operations on a hidden global state, they avoid unnecessary sequentialisation and provide ample latent parallelism that can be exploited by the compiler and runtime-system. This property makes them an attractive platform for exploiting common-place parallel hardware without imposing new concepts of explicit threads with explicit communication onto every parallel application.

A key property of functional languages is referential transparency (Sondergaard and Sestoft, 1989), which ensures that functions always return the same values given the same parameters. This enables one to replace functions with the values without changing the behaviour of the program. Pure functions disallow side-effects, i.e. without changing the state of the world, by performing an action that changes the global states of the program. Functional languages also discourage mutable states. Thereby, the computations of expressions cannot interfere, permitting any order of evaluation, in particular also a parallel one. By avoiding a hidden, mutable state, it is easier to recognise which operations can be safely parallelised, and which not. Function evaluation can be naturally done in parallel. The degree of parallelism is only limited by the data dependencies in the program. This comes in contrast with expressing parallelism in object-oriented and imperative languages, where specifying parallelism is not only often intrusive but also very challenging.

Functional language implementations, such as GHC (Marlow and Jones, 2012), offers ample opportunities for program transformation through compile-time optimisations. Functional features are being increasingly adopted in mainstream languages. For instance, Java 8, C# 3.0 and C++11 (Goetz, 2013; Hejlsberg et al., 2003; Jarvi and Freeman, 2010) have support for lambda expressions, making such concept more accessible to mainstream programmers. Additionally, skeletons are naturally expressed as higher-order functions in functional languages. For instance, evaluation strategies are analogous to skeletons and cleanly separate computation from

coordination aspects.

2.5.3 Parallel Functional Languages

An influential, early system for parallel functional programming was Mul-T (Kranz et al., 1989) using Lisp. It introduced the concept of futures as handles for a data-structure, that might be evaluated in parallel and on which other threads should synchronise. Importantly for performance, this system introduced lazy task creation (Mohr et al., 1991) as a technique, where one task can automatically subsume the computation of another task, thus increasing the granularity of the parallelism. Both, the language- and the system-level contributions have been picked up in recent implementations of parallel functional languages.

One prominent example of this approach is the Manticore system (Fluet et al., 2007) using a parallel variant of ML that includes futures and constructs for data parallelism. It allows to specify parallelism on several levels in a large-scale applications, typically using explicit synchronisation and coordination on the top level (Reppy et al., 2009) and combining it with implicit, automatically managed, fine-grained threads on lower levels (Fluet et al., 2010).

Another earlier ML extension is PolyML (Matthews, 1986; Matthews and Wenzel, 2010), which also supports futures, light-weight threads and implicit scheduling in its implementation. Reflecting its main usage as an implementation language for automated theorem provers, such as Isabelle, it has been used to parallelise its core operations. Parallel SML (Hammond, 1990) is another example from the ML family. Caliban (Kelly, 1989) is a declarative annotation language for controlling the partitioning and placement of the evaluation of expressions in a distributed functional language.

SAC (Scholz, 2003) is a data-parallel functional language. Its syntax is based on C, but its semantics is based on single assignment, and therefore referentially transparent. It mainly targets numerical applications and achieves excellent speedups on the NAS benchmark suite.

New generation functional languages

A new generation of programming languages, such as F# and Scala, often take a multi-paradigm approach, embedding the advantages of functional languages into a mainstream, object-oriented language. They use existing, highly-optimised VM technology, .NET and JVM (Lindholm and Yellin, 1999) respectively, to combine

the ease of expressing parallelism with efficient sequential execution.

F# (Syme et al., 2007) combines the features of a strict, higher-order, impurely functional language of an ML-style, with features of mainstream object-oriented languages. F# supports parallelism through the Task Parallel Library (TPL) (Leijen et al., 2009; Campbell et al., 2010). TPL handles many of the low-level details such as partitioning of work, scheduling of threads on the threadpool and scaling the degree of concurrency dynamically to exploit all available processors in the most efficient way.

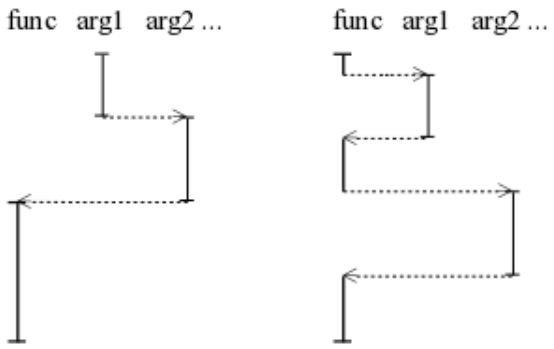
Another statically typed, strict, and multi-paradigm programming language combining functional and object-oriented features is Scala (Odersky et al., 2006). The language allows the expression of common programming patterns in a concise, elegant and type-safe manner. A main focus of Scala is to deliver high-level constructs and abstractions for concurrent programming, emphasising large-scale distribution, scalability and fault-tolerance. Towards this goal it provides a number of programming frameworks, most notably Scala Actors (Haller and Odersky, 2009) for concurrency and Scala Parallel Collections (Prokopec et al., 2011) for implicit parallelism, built on top of the Java Fork/Join framework (Lea, 2000).

Haskell has strong built-in support for concurrency and parallelism. It offers diverse extensions and libraries (Totoo and Loidl, 2014a) to exploit parallelism on multi-cores (GpH), GPUs (Accelerate) and distributed-memory architectures (Eden, Cloud Haskell, HdpH). Additionally, Meta-par (Foltzer et al., 2012) aims to unify parallel heterogeneous programming using these models. An in-depth comparison of the high-level language features and parallelism support in the modern functional languages, F#, Scala and Haskell, with further references to other systems, is provided in (Totoo et al., 2012).

In subsequent sections, we focus on Haskell and three specific parallel programming models which we use in subsequent chapters.

2.6 A Brief History of Laziness

A programming language definition usually specifies an evaluation strategy which refers to when arguments to a function get evaluated (Schmidt, 1986). The most common strategy used in mainstream languages, including C, C++, C# and Java, is *strict evaluation* which uses call-by-value and call-by-reference notions of parameter passing to ensure function application evaluates the argument first and then the function body.



Strict (left): function body gets evaluated after arguments;
Non-strict (right): arguments evaluated if/when needed.

Figure 2.7: Strict vs. non-strict evaluation of a function call

In *non-strict evaluation*, the arguments to a function are only evaluated if they are used inside the function body. For instance, the call-by-name function calling mechanism does not evaluate arguments before a function call. A similar approach, call-by-need, also memoizes evaluated arguments which are then used in subsequent functional calls. Figure 2.7 depicts the two strategies for evaluating function arguments. More generally, non-strictness is often used to refer to the language property that allows expressions to return a value even if some sub-expressions are left unevaluated. However, non-strict evaluation is often referred to as lazy evaluation; though here the latter is referred to the implementation strategy for call-by-need semantics. Haskell is the most prominent example of a lazy language, but other languages e.g. R also uses call-by-need, and many other languages use some form of laziness.

A History of Being Lazy Laziness refers to the operational behaviour, i.e. a particular implementation of non-strictness, though they are often used interchangeably. Lazy evaluation delays evaluation of an expression until its value is needed. Many functional languages are based on lambda calculus, a model to express computation based on function definition, function application and recursion (Turing, 1937). Lambda-calculus beta-reduction captures function application and encodes laziness through successive reduction steps applied to reduce complex expression into simple expression (Wadsworth, 1971). For instance, applying a value (e.g. 1) to the function

$$(\lambda x \to x + y)$$

will reduce it to its next reducible form

$$(\lambda x \to x + y)(1) \implies 1 + y$$

by replacing all occurrence of x with the given value.

In the variable-free theory of functions, combinatory logic (Curry and Feys, 1967), combinator reduction is used for lazy evaluation in a similar way. Turner developed efficient methods using combinators for lazy evaluation using graph reduction: he changed SASL from strict to fully lazy based on combinators, and developed fully lazy Miranda (Turner, 1979, 1986). These methods are widely adopted in lazy functional languages implementation. A modern and perhaps the best known example of language implementing laziness as default is Haskell (Hudak et al., 1999). Though the language specification does not specify a particular evaluation order, the defacto implementation, GHC (The GHC Team, 2015), uses graph reduction as an efficient lazy evaluation implementation. A history of functional languages implementation from strict to lazy is provided in (Turner, 2013).

Other early references to lazy evaluation include (Henderson and Morris, 1976) for proposing to delay evaluation of parameters and list data structures in McCarthy's strict LISP (McCarthy, 1962) without performing more evaluation steps than needed; and (Friedman and Wise, 1976) for proposing *cons should not evaluate its arguments*. In the latter scheme, the structure building function (*cons*) is non-strict so that evaluation of its arguments is suspended until needed by the strict elementary functions, for example, by a print routine.

Lazy Features in Strict Languages Though most mainstream languages are strict by default, they support a number of constructs that provides some degree of laziness. Example of these lazy constructs that do not force unnecessary evaluation in a generally strict language are:

- `if..else..` conditional cases, used to specify a block of code to be executed only if the condition is true.
- `&&` and `||` boolean expressions, also known as short-circuit evaluation. For example, `False && <expr>` and `True || <expr>` do not require the evaluation of the second expression, as the results (`False` and `True`, respectively) depend on the first only.
- `lazy` keyword e.g. in SML, to annotate lazy computations. This overrides the default of strictly evaluating expressions e.g. arguments in a function call.
- lazy data structures, as discussed below.

2.6.1 Full vs Data Structure Laziness

Language strictness or non-strictness through lazy evaluation is a delicate design choice. Many languages adopt strictness as the default but with strong support for lazy data structures and other language construct to selectively mark lazy expressions, for example, Standard ML. Other mainstream languages, e.g. C, C# and Java, implement laziness constructs through concepts such as closures and delegate. Regardless of the language default, lazy data structures have useful properties. For instance, they allow the definition of *infinite* and *circular* program and data structures, which we use in our mechanisms to control parallelism. Many scripting languages support lazy iterators to cope with large data structures.

Circular programming (Allison, 1989) applies transformation techniques to a program to avoid multiple traversals of a data structure. These techniques have been used to improve (sequential) performance by reducing the number of traversals over linked-lists, trees and queues. A circular program can be more efficient in terms of space than a traditional program by avoiding the creation of intermediary data structures. An example of such a program is the Haskell Prelude function nub, which is used to remove duplicates from a list. Circular programs require the combination of lazy evaluation and recursion. A related program transformation technique, particularly applied to programs in non-strict functional languages, is deforestation to eliminate intermediate tree data structures in an algorithm (Wadler, 1990).

Both evaluation orders have their advantages and disadvantages. Strict evaluation facilitates reasoning about program behaviour and execution. With lazy evaluation it is difficult to predict when expressions are evaluated but programs can be more modular. The language strictness property has implications for the design and analysis of data structures.

2.6.2 Parallelism and Laziness

The *strictness* property of a language is crucial as it determines the order of evaluation which directly influences parallelism. Strict languages require arguments to a function to be evaluated before the function call. This strict property allows for arguments to be evaluated in parallel but enforces sequential evaluation between arguments and the function body. The challenge is to *restrict parallelism* to avoid over-evaluation. In non-strict languages, arguments are evaluated only if and when needed in the function body. Consequently, analysis is required to determine which expressions can be evaluated in parallel – to *detect parallelism*. Non-strictness and parallel processing often seen incompatible since parallelism requires certain degree

of strictness for evaluation to proceed. However, parallelism has been implemented successfully in the lazy functional language Haskell.

2.7 Parallel Haskells

Haskell (Hudak et al., 1992) is a statically typed, lazy and purely functional language. Haskell is strongly typed but type definitions are rarely needed as it uses type inference to deduce types automatically. Its advanced type system also supports algebraic data types and polymorphic functions. Type classes are used as interfaces with default implementations. Instances of a specific type class group types together e.g. the `Num` type groups `int`, `double` and other numerical types which have similar operations e.g. `+`, `-`, etc. One of its most distinctive features is its lazy semantics, which means that expressions are only ever evaluated when they are demanded. This demand-driven evaluation strategy makes it possible to have infinite data structures and circular programs in Haskell.

Monads are used to model different computational contexts (Wadler, 1995). Originally they were introduced in Haskell as a way to perform IO by enforcing an execution order. A monad defines two basic operators: `>>=` (bind) and `return`. The bind operator is used to combine monadic values, or create a computational flow, and the `return` operator inject a normal value into a monad. It provides an efficient way of separating pure from effectful computation.

```
class  Monad m  where
    (>>=)            :: m a -> (a -> m b) -> m b
    return           :: a -> m a
```

For instance, we can now chain computation in the following way:

```
putStrLn "Enter␣x:␣" >>= \_ ->
getLine >>= \x ->
putStrLn ("x=" ++ x)
```

And using the `do` notation:

```
do
  putStrLn "Enter␣x:␣"
  x <- getLine
  putStrLn ("x=" ++ x)
```

The main feature of Haskell are summarised below:

- *Purity.* Functions are pure by default and do not alter a state.
- *Lazy.* Demand-driven evaluation.

- *Type system.* Static and strongly type with advanced support for algebraic data types, type classes, polymorphic type.
- *HOF.* Higher-order functions raise abstraction level – functions that can return function as result and passed as arguments.
- *Monad.* Used to structure program and separate pure from impure functions.
- *Concurrency and parallelism.* Strong built-in support and many extensions.

The key advantage of Haskell for parallel computation is referential transparency (Sondergaard and Sestoft, 1989) which guarantees that evaluation can happen in any order. This implies that the amount of inherent parallelism in a Haskell program is large such that each sub-computation can happen in parallel. However, this leads to far too fine-grained parallelism and an approach that allows the programmer to specify which computation is worthwhile to be evaluated in parallel is desirable.

The lazy semantics of Haskell has implications for the parallel programming models that can be supported by the language. Unconstrained lazy evaluation is essentially sequential which contradicts how parallel evaluation should proceed. Some degree of eager evaluation is essential in order to arrange computations in parallel. The programmer also needs to specify the evaluation degree of expressions, such that just enough of an expression is evaluated in order to enable other expressions to continue evaluation in parallel.

Concurrent Haskell Haskell supports concurrency by providing a set of primitives to spawn explicit IO threads that are executed in an interleaved fashion (Jones et al., 1996) and to synchronise between threads. The `forkIO` primitive is used to spawn a concurrent thread. Haskell threads have very low overheads. Haskell-level threads are mapped onto the system threads, usually one per processor core. The `MVar` type, which can be either empty or hold a value, allows for synchronisation and communication between threads. Concurrent execution of threads in this model is non-deterministic and can result in race conditions and deadlocks if proper synchronisation is not implemented.

Synchronisation Primitives: GHC offers a range of low to high level set of synchronisation primitives for implementing shared-state concurrency abstractions (Sulzmann et al., 2009).

`IORef+atomicModifyIORef` At the lowest level, this synchronisation method uses a mutable variable `IORef` with the operations `newIORef`, `readIORef` and `writeIORef`,

and an atomic read-modify-write operation which modifies the contents of an `IORef` atomically. `atomicModifyIORef` is useful for using `IORef` in a safe way in a multi-threaded program by preventing race conditions.

```
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
atomicModifyIORef :: IORef a -> (a -> (a, b)) -> IO b
```

`MVar` At an intermediate level, the `MVar` primitive, introduced as a synchronising variable in Concurrent Haskell is used for communication between concurrent threads. An `MVar` may be either empty or full. The `takeMVar` returns the value if the `MVar` is full or blocks if it is empty, and `putMVar` puts a value in the `MVar` if it is empty or blocks otherwise. Two important properties of `takeMVar` and `putMVar` are: they are both single-wakeup, that is, if multiple threads block in either operation, only one thread will be woken up when the `MVar` becomes full (for take) or empty (for put); and threads are woken up in FIFO order thus providing fairness properties.

```
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
```

STM: Software Transactional Memory STM (Jones, 2007; Harris et al., 2008) is a modular composable concurrency abstraction and is the highest-level of the three primitives. The programmer marks a section of code to run atomically, that is, in isolation with respect to other threads, and the runtime system deals with acquiring and releasing of locks or other kind of concurrency control system to get atomicity. The main benefit of STM is that transactions are composable. Using an optimistic concurrency, consistency check is performed at the end of transactions, retrying a transaction from the start if a conflict has occurred. STM introduces the `TVar` transaction variable. A set of `TVar`s makes a transaction which yields a result, with each transaction performed atomically.

We are interested in pure parallelism to speed up programs, and will not deal with concurrency and low-level issues associated with it. However, we will shortly see how the Par monad builds on top of concurrent primitives to deliver a new deterministic model for parallel computation. In the following sections, we cover GpH, the Par monad and Eden in more detail, which are the parallel programming models used as basis of implementation in the next chapter.

2.7.1 GpH: Glasgow parallel Haskell

GpH extends Haskell with two basic primitives to enable semi-explicit parallelism: `par` for parallel composition and `pseq` for sequential composition (Trinder et al., 1998a; Marlow et al., 2009, 2010). The `par` primitive *sparks* its first argument, i.e. it records it to potentially be evaluated in parallel. The `pseq` primitive evaluates its first argument to WHNF (Weak Head Normal Form) before continuing with the evaluation of its second argument and thus enforces sequential ordering. Both primitives return their second argument as the result.

Listing 2.1: GpH basic primitives.

```
-- parallel composition
par :: a -> b -> b
-- sequential composition
pseq :: a -> b -> b
```

Listing 2.2: GpH Primitives: factorial example.

```
fact m n
  | m == n = m
  | otherwise = left `par` right `pseq` (left * right)
      where
        mid   = (m + n) `div` 2
        left  = fact m mid
        right = fact (mid + 1) n
```

GpH's runtime system uses lazy task creation, by representing potential parallelism as "sparks" that can move freely and cheaply between processors and work represented by one spark can be subsumed by a running thread if no additional parallelism is required. Another key runtime-system design goal is to support light-weight threads, thus reducing the overhead for creating parallelism and encouraging a programming style that generates a massive amount of parallelism, giving the runtime-system the flexibility to arrange the parallelism in a way most suitable to the underlying hardware. Haskell/GHC excels at light-weight threads, as shown by the thread-ring benchmark of the Computer Language Shootout (Fulgham, 2012). Filaments (Lowenthal et al., 1996) and Cilk (Frigo et al., 1998), now integrated in Intel's Cilk Plus compiler, are other examples of runtime-systems for light-weight threads.

Sparks are added to a spark pool and are taken up for execution by lightweight Haskell threads which in turn are mapped down to the underlying OS threads. Creating sparks using `par` is cheap, amounting to adding a pointer to a spark pool, and thousands of them can be created. Converted sparks represent parallelism extracted from the algorithm, incurring the usual thread creation overhead.

The use of the two GpH primitives directly to write parallel programs may often obscure the code with no clear separation between the algorithm and parallelism.

Introducing Evaluation Strategies Evaluation strategies (Trinder et al., 1998a) provide further abstraction over this level of programming which allows the separation between the computation and coordination. It is the preferred way of introducing parallelism in GpH and the main programming model used in technical chapters for data-oriented parallelism. Evaluation strategies are built around the `Eval` monad, which is used to encapsulate a particular evaluation behaviour. It is explained in more detail in Section 3.1.

2.7.2 Par Monad

The Par monad is a parallel programming model implemented entirely as a Haskell library (Marlow et al., 2011). Programming in the Par monad looks a lot like programming in Concurrent Haskell but it preserves determinism and is side-effect-free. Par is a new type declared as a monad. `IVar`s are used for communication, an implementation of the I-Structures, a concept from the pH and Id languages (Nikhil and Arvind, 2001). The basic operations use an explicit approach to specify parallel computations. Parallelism is introduced using `fork` which creates a parallel asynchronous task. Tasks are scheduled using an optimised parallel scheduler among threads. The computation in the monad is extracted using `runPar`. The communication constructs include the following functions:

- `new` to create a new `IVar`.
- `put` to place some value in the `IVar`. It is executed once per `IVar` otherwise an error occurs.
- `get` to retrieve the value from the `IVar`. It is a blocking operation and waits until something is put into the `IVar`.

The derived `spawn` function hides the explicit `put` and `get` operations and therefore ensures that each `IVar` created is only ever put into once. This raises the level of abstraction provided by this model.

```
runPar :: Par a -> a
fork :: Par () -> Par ()
spawn :: NFData a => Par a -> Par (IVar a)

-- communication
data IVar a
new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
```

The library currently offers a limited number of higher-level function abstractions; the most obvious being a parallel map implementation, `parMap`. More abstractions can be built on top of the basic constructs.

2.7.3 Eden

Eden extends Haskell by providing constructs for parallel *process definition*, which abstracts a function that takes an argument and produces a value into a process with input and output that correspond to the function argument and result respectively; and *process instantiation*, which evaluates the process in parallel (Loogen et al., 2005).

```
-- process definition
process::(Trans a,Trans b) => (a->b) -> Process a b
-- process instantiation
(#)::(Trans a,Trans b) => Process a b -> a -> b
```

Building on these coordination constructs, a parallel function application operator and eager process creation function are derived. `spawn` is only denotationally specified, ignoring demand control.

```
-- parallel function application
($#)::(Trans a,Trans b) => (a->b) -> a -> b
f $# x = process f # x

-- eager process creation
spawn::(Trans a,Trans b) => [Process a b] -> [a] -> [b]
spawn = zipWith (#)
```

The parallel runtime system distributes the processes to the available processors. Since Eden has been designed for a distributed-memory model, processes communicate messages to each other to provide input and retrieve output. Synchronisation and coordination are handled implicitly by the runtime system. The programmer does not need to worry about low-level send and receive between parallel processes, and only uses process abstraction or skeletons built on top of the basic construct. Eden processes produce output eagerly with the argument to the process being evaluated locally in the parent process before sending. Lists are handled as streams and are sent element-by-element. This can cause significant overhead and techniques to avoid element-wise streaming are used.

EdenSkel: Eden provides a rich set of higher-order functions that abstract common parallel patterns in its skeleton library EdenSkel. For instance, an initial implementation of parallel map uses `spawn` to eagerly instantiate a list of process abstractions. The `parMap` skeleton creates a process for each list element and this

often results in far too many processes in comparison to the number of processing elements available.

Farm process: The farm process skeleton adapts the number of processes to the number of available processing elements (given by `noPe`). The input list is grouped into *noPe* sublists and a process is created for each sublist instead of each individual element. Implementing the farm process, the `parMapFarm` skeleton provides a simpler interface and a familiar name to the programmer who specifies `unshuffle` as the distribution function and `shuffle` as the combination function. `parMapFarm` creates $(noPe+1)$ processes in total with *noPe* farm processes and one main process which means that one machine will be allocated two processes. A slight variation of this skeleton, implemented as `parMapFarmMinus`, creates $noPe - 1$ processes so that each processor gets exactly one process.

Chunking input stream: The farm process reduces the number of processes but does not have any effect on the messages exchanged between the processes. Each element of the list is sent as a separate message by default. To improve process communication, the number of messages is reduced using a chunking policy. `parMapFarmChunk` is defined as a new skeleton which decomposes the input list into chunks of a specified size e.g. 1000 then creates the farm processes and distributes the chunks to them. This reduces communication overhead.

```
1 -- parallel map definition in Eden
2 parMap f = spawn (repeat (process f))
3
4 -- using farm process
5 parMapFarm f = shuffle . (parMap (map f)) . (unshuffle noPe)
6
7 -- and chunking input stream
8 parMapFarmChunk f xs = concat (parMapFarm (map f) (chunk size xs))
```

Offline processes: Parallel map implemented using offline processes modifies the communication behaviour where the unevaluated data, when typically smaller than the evaluated result, is sent to be evaluated lazily by the new process instead of being evaluated by the parent process. This reduces the combined effort of the main process having to completely evaluate all input to the farm processes.

```
1 -- x is strictly reduced and sent to child process
2 f $# x
3
4 -- parameter passing: input serialised and sent to remote PE
5 (\() -> f x) $# ()
```

By default, the function application is strict in its argument. In the offline version, tasks are evaluated inside the workers, which reduces the communication overhead

since unevaluated data is often smaller than evaluated data. This is done by passing the unit type as argument to the function application as shown above.

2.7.4 Other Parallel Haskells

Other models for parallel computation in Haskell include Data Parallel Haskell (DPH). DPH (Chakravarty et al., 2007) implements the nested data parallelism programming model into GHC. DPH is influenced by the NESL language (Blelloch, 1995) and targets multicores for irregular parallel computations and irregular data structures. DPH adds parallel arrays with parallel operations. The syntax is similar to using lists, however, unlike lists, parallel arrays are strict. Demanding an element will force the evaluation of all elements in parallel. This model is the most implicit one among the other models discussed here. Another example is Cloud Haskell (Epstein et al., 2011), which is a distributed programming model building on the same principle as the highly successful Erlang programming language. A graph-based programming model for Haskell, Intel Concurrent Collections (CnC) (Newton et al., 2011) offers pure and deterministic computation as evaluation strategies. The model allows more control over performance, since the programmer can be explicit about granularity and the structure of parallel computation.

Model	construct	type	pure	deterministic	embedding	
Conc. Prims	`forkIO/OS`		no	no	explicit	↑
CnC	`forkStep`	dataflow	yes	yes	explicit	low
Par monad	`fork`		yes	yes	explicit	
Eden/EdenSkels	`process,#`		yes	yes	semi-explicit	
GPH/EvalStrat	`par,pseq`		yes	yes	semi-explicit	high
DPH	`[:e:]` par array	data	yes	yes	implicit	↓

Table 2.2: Summary of Parallel Haskell Programming Models

Table 2.2 summarises the parallel programming models in the Haskell ecosystem. (Loidl et al., 2003) and (Trinder et al., 2002) gives a detailed performance and programming comparison and classification of parallel and distributed Haskell languages.

2.8 Data Structures in Parallel Programming

"Decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often strongly depend on the structure of the underlying data" (Hoare, 1972).

A data structure organises data in a particular way in memory so that it can be used efficiently. Data structures are building blocks of a program. The performance of a program depends on a correct, efficient *algorithm* and on the right choice of underlying *data structures* (Wirth, 1978). It is, therefore, important to take special consideration for data structures in a parallel environment.

Algorithms + Data Structures = Programs (Wirth, 1978).

A data structure implements an abstract data type (ADT) which only defines the behaviour of the data type. The ADT hides internal data representation and allows operations through a well-defined interface. The structure of the underlying data is abstracted from the user. The programmer then makes a decision to what underlying representation is best for performance.

A data structure is constructed from basic data objects. Common data structures used in early and modern languages are arrays and records. Prominent in imperative languages, arrays support constant access and update times using indexes. Arrays are usually used in a *destructive* fashion, i.e. they are mutable, and are homogeneous structures with all elements having the same type of data. In contrast, records allow different types in an ordered group of elements. Arrays and records are static data structures in most languages. Their sizes are determined at initialisation. Recursive or inductive data structures such as trees or linked lists are composed of individual data components linked to each other, e.g. using pointers in a low-level language like C. They can therefore grow to arbitrary size.

In functional languages, data types are abstractions of data structures e.g. a list type implements a list data structure. In typed functional languages, e.g. Haskell, there is a separation of type constructor, that is, the definition of a data type; and data constructor, that is, an instance of the type. Basic algebraic data types and more advanced type system features, such as, GADTs, type families, dependent types (Jones et al., 2004; Chakravarty et al., 2005; Barthe and Coquand, 2002), allow the high-level specification of data structures and their operations in the language. Most functional languages have strong support for lists and records, also known as tuples. These are defined using the language primitive data types. Arrays are unconventional in such languages. Implementations are often pure, that is, they do not allow direct updates but instead preserves previous versions of the data structure, making it a persistent object (Kaplan, 2001).

2.8.1 Design Issues and Considerations

An important challenge in ensuring scalable performance in parallel programming is the design of efficient data structures. *Concurrent data structures* have been around since uniprocessor machines to enable multi-threaded execution, ensure program correctness and responsiveness. They are intended to allow threadsafe access to shared data structures by multiple threads. They use explicit synchronisation techniques to restrict data access to only one executing thread at a time to avoid race conditions and data inconsistencies. Major progress has been made in understanding key issues and developing techniques, both at hardware level (e.g. compare-and-swap (CAS) instruction) and software level (e.g. higher-level synchronisation constructs), in both blocking and non-blocking concurrent data structures research (Moir and Shavit, 2007).

The need for parallel data structures emerges as important in the context of parallel programming where performance is key.

Parallel data structures are implemented as libraries in a particular programming language which take advantage of parallel processing. Parallelism may be derived freely from using inherently parallel data structures. This form of data-structure-driven parallelism fits the data-parallel category but it deserves more attention due to the ease of enabling parallelism from its use. The use of parallel data structures also comes with minimal code changes to the sequential algorithm. All operations are inherently parallel.

Data structures not designed for parallel execution create bottlenecks by limiting the upper bound of performance gain, and (Shavit, 2011; Moir and Shavit, 2007) emphasise on the need for a different approach in the design and use of data structures in the age of multicore machines. Rather than focusing on data structures as the main components that drive parallelism in a data-parallel application, they argue that serial data structures create bottleneck and do not allow programs to scale (thus limiting speedup) because of unnecessary sequentialisation introduced. The discussion is mostly in an imperative context and covers data structure design issues such as ensuring correctness, and techniques such as low-level synchronisation through locks, semaphores, transactional memory at software level (Silberschatz et al., 2012, ch.5), CAS instructions (Dice et al., 2013), and hardware support to facilitate these. From a design perspective, the wider message is to encourage data structures that are *parallel by design*. For this, a substantiative "relaxation process" of the properties of data structures that are used in a parallel environment is needed. For instance, an efficient queue implementation should help the system scale under high loads. Therefore, its implementation does not require strict queue semantics:

a k-FIFO queue where elements may be dequeued out-of-order up to k-1 would be well-suited, providing a high degree of scalability (Kirsch et al., 2013).

2.8.2 Parallel Operations vs Representations

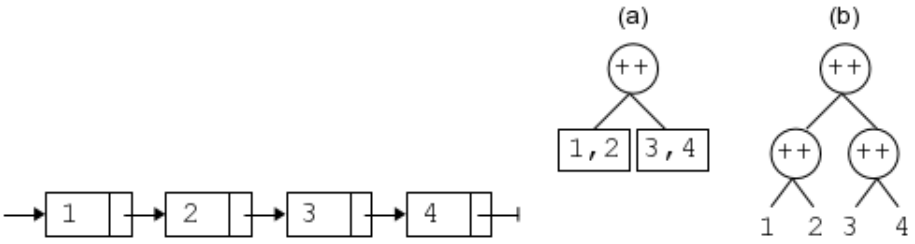
A data structure implementation comprises a concrete representation of data and a set of operations on the data. The set of operations offered by an abstract type, and their complexities, is often influenced by the choice of representation. This representation has a direct impact on performance, as well as parallel scalability.

Motivating tree-based representation for parallel processing. An important challenge in keeping pace with the hardware revolution is to design data structures that are parallel, both in their representations and their operations. Parallel by design encompasses both these two aspects. However, attempts to parallelise data structures have been largely targeted towards parallelising their operations only, leaving the underlying, often sequential, representation unchanged. Starting with an inherently sequential representation is fundamentally wrong (Steele, 2009).

Get rid of cons! (Steele, 2009).

Data structures that do not force a sequential evaluation mechanism on the algorithms are highly desirable. Effective parallelism uses trees instead of linear representations. This encourages the design of high-level data-oriented algorithms, specifying parallelism in a minimally intrusive way. Enabling data-structure driven parallelism removes the unnecessary sequentialisation usually imposed by wrong data structure representation, and thus maximises the benefit of parallelisation.

Figure 4.1 depicts the use of an alternative representation for a linear/sequential representation using append-trees, representing a list comprised of leaf values found in a depth-first traversal of the tree. This design moves away from an inherently sequential representation, but potentially maintains the same interface. Trees are hierarchical structures making it easy to apply divide-and-conquer algorithm and thus represent a good match for parallelisation. The representation can also be adapted for various target architectures, for example, one that suits distributed memory system. Manticore uses ropes (Boehm et al., 1995), a balanced binary-tree representation of sequences, as the underlying representation of parallel arrays. In (Totoo, 2011), we cover random-access lists as the internal tree-based representation for list in Haskell.



(a) Sequential representation (b) Parallel representation

Figure 2.8: Alternative append-tree representations of linked-list.

2.8.3 Imperative vs Functional

In an imperative setting, data structures are mutable, i.e. their components can be modified by assigning new values. Therefore, in a concurrent or parallel environment, it becomes crucial to ensure *safety*, i.e. no unexpected bad behaviour happens; and *liveness*, i.e. the correct behaviour, properties when operating on data structures. This entails using various techniques to synchronise shared access to data components.

Mainstream languages, such as Java and C#, provide implementations of common data structures in standard libraries known as collections. The use of the standard collections with multi-threaded applications makes it difficult to ensure correct program behaviour. Previously, the programmer needed to implement code with explicitly synchronised access to data structures manually. Concurrent collections were introduced to eliminate this issue. The emphasis is on thread-safe access to solve shared access problems by implementing built-in synchronisation in the data structures. Additionally, they avoid the bottleneck of sequential data structures by allowing simultaneous access. Moderate performance gains are possible on a parallel machine. However, they are not designed nor intended for scalable parallel performance.

The Java Concurrent Collections include a number of concurrent counterparts of sequential collections such as map, list and queue which are thread-safe and optimised for concurrent access. The library implements atomic and mutative operations that implicitly acquire and release locks. The equivalent library for C# and F# implemented on top of .NET, called Coordination Data Structures (Toub, 2010), provides concurrent collection classes that can be used with multi-threaded application code, and also in conjunction with other parallel programming infrastructure provided by the platform, such as TPL and PLINQ.

The benefits of functional programming are widely known (Backus, 1978; Hughes, 1989) and covered earlier in this chapter. Functional languages encourage the design

of pure data structures that do not operate on changed states. Functional programming does not specify a particular evaluation order, consequently only part of the data structure that is being processed needs to be evaluated at a time. With advances in compiler technology and optimisations, functional languages are not too far from their imperative counterparts in terms of performance, for example, GHC execution time is comparable with Java for some benchmarks from the Computer Language Benchmarks [1].

Purely Functional Data Structures The main body of work done in the area of functional data structures is by Okasaki in his PhD thesis (Okasaki, 1996) in which he described a number of efficient and elegant definitions of purely functional data structures in Standard ML. Functional data structures have the persistent property meaning that they are immutable. An update does not destroy the previous version of the data structure, but rather creates a new version, thus enabling a kind of automatic versioning of the structure. The absence of destructive updates has immediate property beneficial to parallel execution. It allows the definition of parallel operations on data structures, with ease of partitioning and executing parallel sub-computations, without running into deadlocks. In this thesis, we look at parallel implementations as a way to further improve application performance using a set of parallel evaluation strategies over purely functional data structures in Haskell.

2.9 Summary

This chapter has provided an overview of general hardware trends and a classification and critical evaluation of parallel programming models. It highlights the issues with low-level parallel programming, discusses key abstractions, and systematically classifies language models based on a number of language properties and different classes of languages. It motivates the use of higher-level programming models, in particular, parallel functional programming. The chapter also provides a brief history of laziness, and covers the Haskell programming language and parallel programming support. Finally, a section on data structures for parallel programming gives an overview of trends, design issues and considerations.

[1] The Computer Language Benchmarks: http://benchmarksgame.alioth.debian.org/

Chapter 3

Parallel List and Tree Processing

The purpose of this chapter is two fold. First it expands on data-oriented programming in GpH using evaluation strategies as the main programming model. Secondly, it serves as an evaluation of GpH, specifically through the use of the higher-level evaluation strategies library, against two other parallel Haskell programming models: Par monad and Eden. In the chapter, we highlight general parallelisation guideline to write parallel algorithms in Haskell. We implement two versions of the n-body problem where parallelism is derived from parallel list and tree processing. We review the steps to implement data-oriented parallel list strategies. The chapter emphasises the compositionality and high abstraction level of evaluation strategies among other benefits, and motivates its use in the subsequent chapter for defining more advanced strategies on trees and graphs.

3.1 Evaluation Strategies

This section expands on the evaluation strategies programming model as introduced in the background (Section 2.7.1). The evaluation strategies library is an additional level of abstraction provided on top of the GpH `par` and `pseq` primitives. It allows modular writing of parallel code by separating the algorithm from the parallelism. A strategy can be substituted by a different one if a different parallel behaviour is desired without having to rewrite the algorithm.

A strategy is a function that is executed for coordination effects. In the original formulation, it was defined to return a unit type `()` since its only purpose was to specify the dynamic behaviour. The new formulation (Marlow et al., 2010) is an improvement on the original. It fixes a number of issues, including a space management issue, and it introduces an evaluation-order monad, `Eval`, which allows a set of evaluations to be specified in a compositional way.

Listing 3.1: Strategy function type

```
-- original formulation
type Strategy a = a -> ()

-- new formulation
type Strategy a = a -> Eval a
data Eval a = Done a

runEval :: Eval a -> a
runEval (Done x) = x
```

Listing 3.1 shows the strategy type definition. A strategy is a function in the Eval monad that takes a value and return the same value within the Eval context which encapsulates the evaluation behaviour of the value.

Applying Strategies A strategy can be applied in the following way with the `using` function.

```
using :: a -> Strategy a -> a
x `using` strat = runEval (strat x)
```

Simple Strategies The basic strategies `rpar` and `rseq` are defined directly in terms of their primitives. The `using` function applies a strategy to an expression. Since all parallelism, evaluation order and evaluation degree are specified within the `Eval` monad, an explicit `runEval` is used at the point where it is applied to a concrete expression. Using a monad helps to separate the purely functional aspects of the execution from the behavioural aspects of the execution. It also allows the programmer to use rich sets of libraries and abstractions available for monads in Haskell as we will see in later implementation sections.

The `Eval` monad is used to specify evaluation order. It is chosen to be a strict identity monad (more on this in Section 5.5.2 – Defining a lazier version of the Eval monad) to allow parallel evaluation. The following strategy combinators defined:

```
-- no evaluation
r0 :: Strategy a
r0 x = return x

-- spark x
rpar :: Strategy a
rpar x = x `par` return x

-- evaluate x to WHNF
rseq :: Strategy a
rseq x = x `pseq` return x

-- fully evaluate x (normal form)
rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x
```

Both `r0` and `rseq` control the evaluation degree of an expression: no evaluation or part evaluation (i.e. only up to the top-level constructor). To completely reduce an expression, that is, to normal form (NF), we use the `rdeepseq` strategy which is implemented in terms of the `rnf` function (reduce to normal form). Any type that can be fully reduced in Haskell needs to be an instance of the `NFData` typeclass and to implement the `rnf` function. For example, the different evaluation degrees of a list `[1,2,3]` can be as follows:

```
-- r0: no evaluation
-- (referred to as a thunk in Haskell)
xs = <unevaluated value>

-- part evaluation applying rseq
-- (up to top-level construct i.e. spine of the list)
xs `using` rseq
xs = _:_:_:_

-- full evaluation applying rdeepseq
-- (no remaining redex -- reducible expression)
xs `using` rdeepseq
xs = 1:2:3:[]
```

Apart from `rseq`, anything between `r0` and `rdeepseq` evaluation degree is possible. For example, we may define a strategy that evaluates the first N elements of the list, or every other elements; the possible evaluation degrees depend on the data types.

A simple definition of a strategy to evaluate a tuple in parallel is given in Listing 3.2.

Listing 3.2: Strategy to evaluate elements of a tuple in parallel

```
parTuple :: Strategy (a,b)

parTuple :: Strategy (a,b)
parTuple (a,b) = do
  a' <- rpar a
  b' <- rseq b
  return (a',b')

-- apply strategy
pair=(x,y)
pair `using` parTuple
```

Refer to the direct use of the GpH primitives in Listing 2.2. In contrast, the example in Listing 3.3 highlights the modularity of evaluation strategies, i.e. the separation between the parallel specification and the algorithm.

Listing 3.3: Evaluation Strategies: factorial example.

```
fact m n
  | m == n = m
  | otherwise = (left * right) `using` strategy
      where
        mid   = (m + n) `div` 2
        left  = fact m mid
        right = fact (mid + 1) n
```

```
8         strategy result = do
9                             rpar left
10                            rseq right
11                            return result
```

Combining Strategies Strategies can be composed just like functions using the `dot` strategy combinator e.g. `(rpar ‘dot‘ rdeepseq)` sparks parallel evaluation of its argument (specified by `rpar`) and completely evaluates it to normal form (specified by `rdeepseq`). In this example `rdeepseq` is used to specify the evaluation degree. The expressive power of evaluation strategies comes from the ability to compose them, as above, to separate the specification of parallelism from evaluation degree and other parallelism optimisations such as clustering, as we will see later, and the possibility to nest strategies, by providing other strategies as arguments, exploiting higher-order functions in the language.

Data-oriented Strategies Map is a higher-order function which takes a function f and a list as parameters and applies f to each element of the list. Implementing a parallel map is a major source of data-oriented parallelism over list data structures.

```
1 -- parallel map definition
2 parMap strat f xs = map f xs ‘using‘ parList strat
3
4 xs = parMap rdeepseq f list
```

In the version above, `parMap` is the parallel version of the `map` function which applies a computation `f` to each element of the container e.g. a list in this case. `parMap` exposes the maximal parallelism, creating a spark for each item to be evaluated in parallel. It is implemented in terms of `parList`, the definition of which is covered in the next section. In subsequent sections, we also discuss several techniques for improving parallel performance by generating fewer, more coarse-grained threads.

3.1.1 Parallel List Strategies

A list is defined as a recursive data type which consists of a *head* as its first node and a *tail* representing the rest of the list.

list = head + tail

This can be captured using the following type definition in Haskell:

data List a = Cons a (List a) | Nil

The list *1,2,3,4* is encoded by:

Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))

This algebraic sum data type allows a list to be constructed using the type constructor *List*, thus making list a new type, and the data constructors *Cons* and *Nil* which inhabit the *data* or *value* domain. This new type implements a singly-linked list in Haskell, with each node holding an implicit reference to the next node in the sequence. This definition also allows circular list where the last element points back to the head element instead of an empty list, as well as infinite list.

Listing 3.4: Built-in Haskell list definition.

```
data [a]  = []  | a : [a]

-- instances: xs and ys are equivalent
xs=1:2:3:4:[]
ys=[1,2,3,4]

-- infinite list
zs=[1..]

-- circular list
ws=1:2:3:ws
```

Definition 3.1 (Linked list)**.** *A linked list is a data structure in which the objects are arranged in a linear order. (Cormen et al., 2009)*

In Haskell built-in notation, *Cons* and *Nil* are represented using syntactic sugar form of `:` and `[]` (empty list). Using the recursive definition of a list, a new element can be added to the front of another list, or to an empty one. A list can be built of any type, including primitive (e.g. list of integers *List Int*) and custom types. However, lists are homogeneous, i.e. all elements need to be of the same type for a given list. With polymorphism, it is possible to define a list of any type (e.g. *List a*), and write generic functions, such as length, to operate on it. GHC allows the list element type to be left unspecified. The type can be inferred, in most cases, by the system. This inference is usually the most generic one. For instance, in Listing 3.4, the inferred type will be *Integer* (unbounded) rather than *Int* (32-bit).

Parallel list processing using Evaluation Strategies. Strategies can be defined to evaluate list in a number of ways. Below we highlight some useful strategies defined over a list as depicted in Figure 3.1.

(a) Strategy 1: *Element-wise* Each list element is evaluated in parallel.

(b) Strategy 2: *Chunk* The list is recursively split into chunks or blocks of an appropriate size s and then each chunk is processed in parallel instead of each individual element. This method uses the list size as input to the split function,

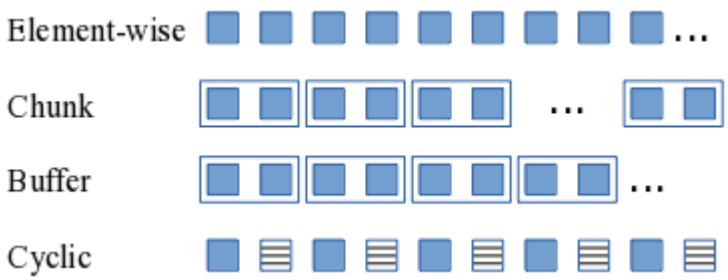


Figure 3.1: List parallel processing

thus requires parallel evaluation to be strict at the constructor level. Thus, it will only work on finite list.

(c) Strategy 3: *Buffer* A lazier variant of the previous method where the list is processed as a stream, but with every s elements evaluated in parallel. It preserves the lazy and infinite properties of the list.

Other typical way of list processing, e.g. using cyclic distribution, which is assigning alternative list element to specific thread for processing, is not easily implemented using this model of programming. This method requires explicit control of computation-to-task in the parallel programming model which we will not consider.

Implementing data-oriented list strategies in GpH The following provides a step-wise approach to defining strategies over a data type, the built-in list in this case, in GpH. We define the two list strategies that we apply in the initial and tuned parallel versions of the two algorithms we cover later.

1. Strategy type

Since we are defining a strategy over a list, the strategy type is

```
parList :: Strategy [a]
```

i.e. if we replace the type synonym from Listing 3.1, `parlist` takes a list as argument and returns a list in the Eval monad.

```
parList :: [a] -> Eval [a]
```

2. Strategy definition

First, we define a strategy that will evaluate each element in parallel (element-wise). This involves specifying two pattern match conditions: 1) what to do if list is empty; 2) and, what to do for non-empty list i.e. do something to the head, and to the rest of the list. The following is a basic `parList` implementation in which we mark the

head element x for parallel evaluation, using `rpar` and recursively do the same with the remaining elements.

```
parList :: [a] -> Eval [a]
parList [] = return []
parList (x:xs) = do
                   x'  <- rpar x
                   xs' <- parList xs
                   return (x':xs')
```

3. Problem? Specify an evaluation degree as well.

In the previous definition, we are only specifying to spark each element to evaluate in parallel, but we are not specifying a degree. So, Haskell being lazy, the elements will not be evaluated until needed – unless we specify a degree of parallel evaluation. This can be done using `rseq`.

Hardcode evaluation degree in the strategy definition:

```
parList :: [a] -> Eval [a]
parList [] = return []
parList (x:xs) = do
                   x'  <- (rpar `dot` rseq) x
                   xs' <- parList xs
                   return (x':xs')
```

Or, we can parameterise it:

```
parList :: Strategy a -> Strategy [a]
parList strat [] = return []
parList strat (x:xs) = do
                   x' <- (rpar `dot` strat) x
                   xs' <- parList xs
                   return (x':xs')
```

Note we switched back to using the Strategy type synonym again.

Now we can pass other evaluation degrees, e.g. `parList rdeepseq`. Note that the element type `a` must be first made an instance of the NFData typeclass by implementing the `rnf` function so we know what it means to completely evaluate its value. All built-in types in Haskell e.g. integer, char, boolean have a default rnf implementation. If we are to apply `rdeepseq` to a list type, we would implement rnf for list as follows:

```
instance NFData a => NFData [a] where
    rnf [] = ()
    rnf (x:xs) = rnf x `seq` rnf xs
```

This is needed if we want parallel evaluation to normal form of nested list, as performed in `parListChunk` below.

4. Better style, and being concise. Applicative style (McBride and Paterson, 2008)

is a short hand of doing the above using the monadic `do` notation, through two combinators: `<$>` which is an infix synonym for `fmap`, and `<*>` which is used to sequence computations and combine their results.

Applicative syntax:

```
parList :: Strategy a -> Strategy [a]
parList strat [] = pure []
parList strat (x:xs) = pure (:) <$> strat x <*> parList xs
```

4. Generalise: fmap/foldr/traverse

The `parList` strategy is essentially a map function which applies an evaluation strategy to each element in the list. So if the type is an instance of the `Functor` (class used for types that can be mapped over), `Foldable` (data structures that can be folded) and `Traversable` (data structures that can be traversed from left to right) Haskell typeclasses by implementing the `fmap`, `foldr` and `traverse` functions[1], respectively, we have automatic generalisation, that is, the `parList` strategy can be defined in terms of `traverse`.

```
parList = traverse
```

These functions are already implemented for the built-in list type in Haskell. In later chapter, the strategies defined for trees have been generalised in the same way.

5. Tuning. Defining the chunk strategy.

Changing the parallel evaluation behaviour of a list simply requires a different strategy definition. For e.g. `parListChunk` applies a strategy to sublists of length n rather than to individual list elements. Note the second argument specifies an evaluation degree which applies to a list, i.e. of type `[a]`, not `a`, thus requiring the previously `rnf` implementation if we want to pass `rdeepseq` to this strategy.

```
parListChunk :: Int -> Strategy [a] -> Strategy [a]
parListChunk n strat [] = return []
parListChunk n strat xs = do
                          ys' <- (rpar `dot` strat) ys
                          zs' <- (rpar `dot` strat) zs
                          return (ys' ++ zs')
                            where
                              (ys,zs) = splitAt n xs
```

In this chapter, we use `parList`, `parListChunk` and `parCluster` (which we cover later) strategies in our algorithms. We provide further tuning techniques in the relevant sub-sections as we cover the implementation.

[1] Hackage: `fmap` – Map a function f to each element of a structure. `foldr` – Right-associative fold of a structure. `traverse` – Map each element of a structure to an action, evaluate these actions from left to right, and collect the results.

3.2 Application: The N-body Problem

As an example application, we implement two algorithms for solving the n-body problem following the general parallelisation methodology highlighted in Section 3.2.1. In the process, we incrementally optimise the parallel solutions in three parallel Haskell programming models. Both algorithms use list and tree data structures in their core implementation. Parallelism is derived through parallel processing of these structures.

3.2.1 Implementation Approach

A set of guidelines for engineering large parallel functional programs is discussed in detail in (Loidl and Trinder, 1997).We follow these main steps to write our parallel Haskell programs in the three programming models. In particular, we expand on two sub-steps (1.1. and 4.1.) aimed at further improving performance in a lazy language and on which we elaborate and provide detailed analysis in our implementation:

1. *Sequential implementation.* Start with a correct initial implementation of the inherently-parallel algorithm;

    1.1. *Sequential optimisation.* Optimise the sequential algorithm e.g. improve heap consumption by identifying and fixing any space leaks; use tail recursive functions; and add strictness where needed to avoid unnecessary delayed computation or indirection;

2. *Time profile.* Identify "big eaters", i.e. parts of the program that take up a large percentage of running time;

3. *Top-level parallelisation.* Parallelise top-level data independent operations e.g. map functions representing data-oriented parallelism, and independent tasks representing task-oriented parallelism, using high-level constructs provided in the parallel programming model;

4. *Parallel execution.* Run parallel programs on a multi-core machine or a cluster to obtain initial results, and debug parallel performance e.g. poor load balance;

    4.1. *Parallel tuning (and scaling).* Advanced and more explicit tuning based on the programming model to improve parallel performance. This step also looks at scalability both in terms of increasing problem size and processing units.

Starting with a sequential implementation may seem counter-intuitive to the idea of "think in parallel". It may be true in an imperative setting, where a major re-write of sequential code is often necessary for parallel execution. However, in a functional setting, we can start with an existing sequential implementation of a problem and use parallel constructs that exploits execution of pure functions in parallel.

3.2.2 Problem Description

The n-body problem (Leimanis and Minorsky, 1958) involves predicting the motion of a system of N bodies which interact with each other according to the laws of physics. The problem and several variations are defined in many areas of science. Depending on the domain, the bodies can refer to elements at the microscopic level, for example, interactions between molecules in biological systems, or between particles in particle mechanics. At the macroscopic level, it is traditionally used to model and simulate the orbit of celestial bodies such as planets, stars and galaxies, which is affected by gravitational force.

The n-body simulation is a computationally-intensive task which proceeds over several time steps. For each body with mass m in the system, the gravitational force F exerted on it with respect to the other bodies is calculated. This is used to update the velocity and position of the bodies in each iteration. A number of methods exists to solve the problem. We look at a direct and an tree-based methods. The direct approach is a simple naive method, referred to all-pairs or body-to-body (Aarseth, 2003), and is used for simulation consisting of up to a few thousand bodies. The tree-based method is an approximate but realistic method using the Barnes-Hut algorithm (Barnes and Hut, 1986; Pfalzner and Gibbon, 1996) and is well suited for large system of bodies.

Method 1: All-Pairs

The all-pairs algorithm is a direct method in which the forces between each pair of bodies are calculated in a traditional brute-force technique. Given the number of bodies in the system is N, all body-to-body comparisons require a time complexity of $O(N^2)$ to complete, making the algorithm suitable for only a small system of bodies.

The algorithm proceeds over a number of time steps represented by a time change ($\Delta t = 0.001$). For each body, with an initial position (x, y), velocity (v_x, v_y) and mass m (in 2D space), the algorithm:

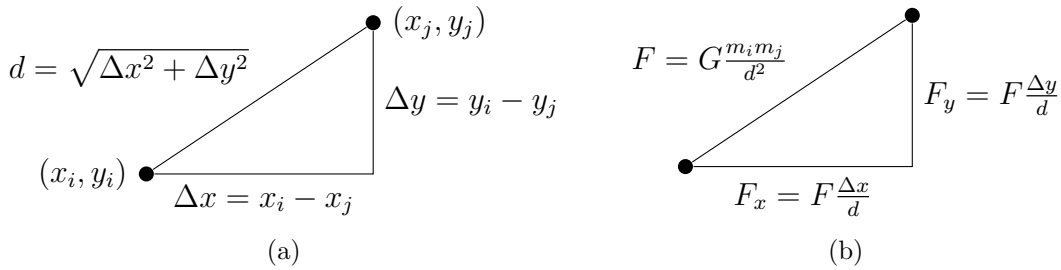


Figure 3.2: All-pairs force calculation (2D)

1) calculates the force F as depicted in Figure 3.2

The pairwise force F calculation is the gravitational constant G times the product of masses m_i and m_j for bodies i and j, respectively, divided by the distance d between the two bodies.

$$F = G\frac{m_i m_j}{d^2} \tag{3.1}$$

where the distance d is found from the change in x and y (i.e. Δx and Δy):

$$(\Delta x, \Delta y) = (x_i - x_j, y_i - y_j) \tag{3.2}$$

$$d = \sqrt{\Delta x^2 + \Delta y^2} \tag{3.3}$$

2) computes the acceleration (a_x, a_y) due to each body combining the net force induced by the other bodies

$$(a_x, a_y) = (\frac{F_x}{m}, \frac{F_y}{m}) \tag{3.4}$$

$$(F_x, F_y) = (F\frac{\Delta x}{d}, F\frac{\Delta y}{d}) \tag{3.5}$$

where (F_x, F_y) are the x- and y- components of force.

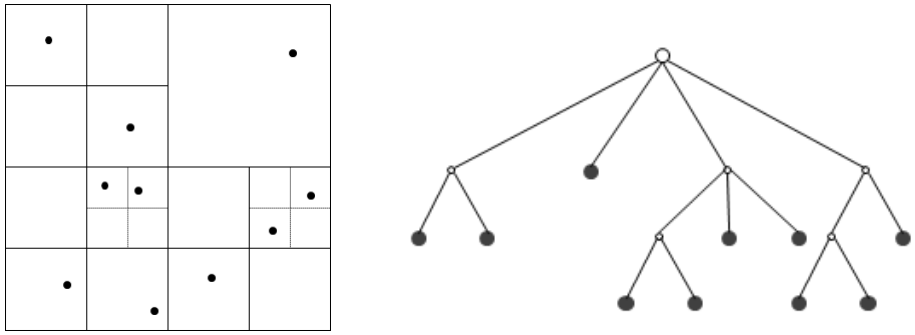
3) updates the velocity (v_x, v_y) to

$$(v_x + \Delta t a_x, v_y + \Delta t a_y) \tag{3.6}$$

4) finally, updates the position x and y

Method 2: Barnes-Hut Algorithm

The computational cost of the body-to-body comparison increases quadratically when the interaction between each pair of bodies need to be computed in a large system. Tree-based algorithms avoid this problem by grouping bodies that are far



(a) Physical location of points in 2D space

(b) Tree representation

Figure 3.3: 2D Barnes-Hut algorithm: The region containing the bodies is divided recursively into smaller sub-regions (bounding boxes) which can then be represented in a tree data structure.

away from the current body under consideration into a region (i.e. a whole branch in the tree rather than the leaf nodes which represent the bodies – see Figure 3.3). The error arising from combining bodies into one point mass is negligible. The algorithm builds a tree, a quad-tree for two-dimentional problem, and an octree for three-dimentional space.

The Barnes-Hut algorithm is a widely used approximation method for the n-body problem. It is particularly useful for systems consisting of a large number of bodies, where an approximate but efficient solution is more feasible than the naive approach. The Barnes-Hut algorithm proceeds in two stages:

1) Tree construction: recursively divide the region containing the bodies into sub-regions (four for 2D or eight for 3D space) and represent bodies as nodes in a tree. This division into sub-regions continues until each sub-region contains zero or one body. The centre of mass and total mass of each region are calculated and stored in the inner nodes.

The total mass (M) and centre of mass (R) of bodies in a region are given by:

$$M = \sum_{i=1}^{N} m_i \tag{3.7}$$

$$R = \frac{1}{M} \sum_{i=1}^{N} m_i \times r_i \tag{3.8}$$

where N is the number of bodies, m_i represents the mass and r_i the coordinate of body i.

2) Tree traversal: the tree is traversed and the forces between all bodies are com-

Version	Algorithm	Description
S1	AP	Sequential All-Pairs algorithm
S2	BH	Sequential Barnes-Hut algorithm
P1.1	AP	Parallel version in GpH–Evaluation Strategies
P1.2	BH	
P2.1	AP	Parallel version in Par monad
P2.2	BH	
P3.1	AP	Parallel version in Eden
P3.2	BH	

AP=All-Pairs; BH=Barnes-Hut

Table 3.1: Sequential and parallel algorithm versions

puted, either directly, or by approximating bodies that are too far away using a region. This is determined by a distance threshold, using the region centre of mass.

Depending on how dispersed the bodies in the system are, the tree construction phase will usually lead to an imbalanced tree structure, which makes achieving good parallel performance difficult. The implementation in this chapter uses a random input generation which distributes the bodies uniformly in space and thus generates a fairly balanced tree. In Chapter 4, we use different input distributions to test more advanced strategies that are designed specifically to deal with such irregular input through more advanced parallelism control mechanisms.

In the implementation sections, we number the sequential and parallel versions as shown in Table 3.1.

3.3 Sequential Implementation

The n-body problem has been implemented across a wide variety of programming languages covering many paradigms. Using a functional language like Haskell raises the level of abstraction as it makes it easier to express the problem, for example, through the use of higher-order functions. The three parallel Haskell variants covered in the previous chapter are used for implementing parallel versions of the two algorithms. The main focus remains on GpH for implementing custom data structure evaluation strategies, and we then use Par monad and Eden to contrast the implementation.

This section describes the two sequential algorithms and their step-by-step implementation:

- Firstly, the simple all-pairs (naive all-to-all) algorithm which uses lists.

- Secondly, the more realistic Barnes-Hut algorithm to be used on large number of bodies on quad/oct-trees.

A number of sequential optimisations are applied to the initial implementations to address memory issues that are inherent in a lazy language such as space leak arising from holding a reference for too long. As a general rule, we avoid premature optimisations that could hinder parallelisation. The effect of each optimisation is reported in terms of runtime and heap usage improvement. The experimental setup for these results and an overall evaluation of performance across all models are given in Sections 3.4 and 3.5. The runtimes reported exclude the generation of input and show the main part of the program that performs the core computation. We consider the problem in three dimensional space, hence use oct-trees for the Barnes-Hut implementation.

Body representation A body is represented by its position (x, y, z), velocity (vx, vy, vz), and mass (m). Our data structure design uses two algebraic data types defined to represent a body and the acceleration (Listing 3.5), and collects all bodies in a list container.

Listing 3.5: N-body data types definition

```
data Body   -- body type def
  = Body {  x::Double,  y::Double,  z::Double, -- position
            vx::Double, vy::Double, vz::Double, -- velocity
             m::Double }                        -- mass

data Accel  -- acceleration type def
  = Accel { ax::Double, ay::Double, az::Double }
```

3.3.1 S1: All-Pairs

The pairwise force calculation for this algorithm is relatively simple to implement in an imperative language using a double nested loop with inplace update. In pure Haskell, the absence of destructive update requires an approach that uses functional recursion to implement iteration in the program to simulate a loop construct. Alternatively, a nested map function can also be employed to get the same compute structure. In both cases, new data structures are generated. To achieve destructive updates, a monadic style of programming and the use of updatable data structures are needed. The use of mutable data structures to solve the problem is presented in (Totoo et al., 2012) in F#. Here the focus is on a pure solution.

The main function `doSteps` that encodes the iteration in Listing 3.6 internally uses two map functions, corresponding to a double nested loop in an imperative lan-

guage. The top-level map function applies the composite function `(updatePos . updateVel)` to each body in the list `bs`. The main computation happens in the `updateVel` function, which has another map function to calculate the accelerations against all the bodies. The fold function updates the accelerations giving the updated velocity. The code below shows the computation performed in the main iteration.

Listing 3.6: All-Pairs main iteration

```
dt = 0.001  -- time change for each step
eps = 0.01  -- epsilon value

doSteps::Int->[Body]->[Body]
doSteps 0 bs = bs
doSteps s bs = doSteps (s-1) new_bs
  where
    new_bs = map (updatePos . updateVel) bs

    updatePos (Body x y z vx vy vz m) =
        Body (x+dt*vx) (y+dt*vy) (z+dt*vz) vx vy vz m

    updateVel b = foldl deductChange b accels
      where
        accels = map (calcAccel b) bs

        deductChange (Body x y z vx vy vz m) (Accel ax ay az) =
            Body x y z (vx-ax) (vy-ay) (vz-az) m

        calcAccel b_i b_j =
            Accel (dx*jm*mag) (dy*jm*mag) (dz*jm*mag)
          where
            Body ix iy iz _ _ _ _  = b_i
            Body jx jy jz _ _ _ jm = b_j

            mag = dt / (d * r)
            r   = sqrt d
            d   = dx*dx + dy*dy + dz*dz + eps
            dx = ix - jx
            dy = iy - jy
            dz = iz - jz
```

3.3.2 S2: Barnes-Hut

The Barnes-Hut implementation is based on the 2D version of the algorithm described in (Chakravarty et al., 2001) and (Jones et al., 2008). The program consists of two main steps which are broken down into smaller functions. In addition to the `Body` and `Accel` types used in the all-pairs version, three new data types central to the data design are defined to represent:

1. `BHTree` the Barnes-Hut tree structure;
2. `Bbox` the bounding box representing a region in 3D space, and

3. `Centroid` the centroid of a region i.e. centre of mass and total mass.

The `BHTree` data type implements a rose tree where each node can have an arbitrary number of sub-trees. In our case of modelling a 3D space, this will not be more than 8 children per node, thus an oct-tree. The node of the tree consists of the size of a region (`size`), the centre of mass (`centerx`, `centery`, `centerz`), total mass (`totalmass`) and sub-trees (`subTrees`).

Listing 3.7: Additional data types for Barnes-Hut

```
data BHTree
  = BHT { size::Double, centerx::Double, centery::Double,
        centerz::Double, totalmass::Double, subTrees::[BHTree] }

data Bbox
  = Bbox { minx::Double, miny::Double, minz::Double,
           maxx::Double, maxy::Double, maxz::Double }

data Centroid
  = Centroid { cx::Double, cy::Double, cz::Double, cm::Double }
```

The algorithm proceeds in two main phases:

a) Tree construction.

This phase constructs an oct-tree from the list of bodies, passed as argument to the `buildTree` function in Listing 3.8. The bounding box representing the lower and upper coordinates of the region containing all the points is determined (`findBounds`) and the size of the region is calculated. The centre of mass (`cx`, `cy`, `cz`) and total mass (`cm`) are calculated and stored at the root node of the tree to represent the whole space.

The bounding box is used to subdivide the bodies into 8 smaller regions (`splitPoints`) and then the centre of mass and total mass of the bodies contained in each region are computed in the same way and stored in the children nodes. The process continues until a region has no body in it — `buildTree` is essentially a recursive function. The actual bodies are not stored in the tree structure as in some implementation since the centre and total mass are calculated in the tree construction phase.

Listing 3.8: Barnes-Hut tree building

```
doSteps 0 bs = bs
doSteps s bs = doSteps (s-1) new_bs
  where
    bbox = findBounds bs
    tree = buildTree (bbox,bs)
    new_bs = map (updatePos . updateVel) bs

-- build the Barnes-Hut tree
buildTree :: (Bbox,[Body])->BHTree
buildTree (bb,bs) = BHT size cx cy cz cm subTrees
  where
```

```
    subTrees =  if bs <= 1 then []
                else map buildTree (splitPoints bb bs)
    Centroid cx cy cz cm = calcCentroid bs
    size = calcBoxSize bs

findBounds::[Body]->Bbox
-- split bodies into subregions
splitPoints::Bbox->[Body]->[(Bbox,[Body])]
-- calculate the centroid of points
calcCentroid::[Body]->Centroid
-- size of the region
calcBoxSize::Bbox->Double
```

b) Force calculation.

In this phase, the acceleration due to each body is computed by traversing the tree, shown in the `calcAccel` function below.

The traversal along any path is stopped as soon as a node is too far away to make a significant contribution to the overall force (`isFar`). This is determined by dividing the total size s of the region by the distance d between the body and the node centre of mass coordinates. If the ratio $\frac{s}{d}$ is less than a certain threshold t, where $0 < t < 1$, then the centroid is used as an approximation. Setting t to zero degenerates to a brute force version, while increasing the value improves performance at the expense of losing precision.

The `accel` function for the Barnes-Hut algorithm differs from the one in the all-pairs version in that instead of calculating the acceleration between two bodies, it uses the centroid (geometric centre) of a region and a body. The `updateVel` function deducts the net acceleration due to each body.

Listing 3.9: Barnes-Hut acceleration calculation

```
updatePos (Body x y z vx vy vz m) = ... -- same as allpairs

updateVel b@(Body x y z vx vy vz m) =
    Body x y z (vx-ax) (vy-ay) (vz-az) m
  where
    Accel ax ay az = calcAccel b tree

calcAccel::Body->BHTree->Accel
calcAccel b tree@(BHT _ _ _ _ subtrees)
    | null subtrees = accel tree b
    | isFar tree b  = accel tree b
    | otherwise     =
          foldl addAccel (Accel 0 0 0) (map (calcAccel b) subtrees)
      where
        addAccel (Accel ax1 ay1 az1) (Accel ax2 ay2 az2) =
            Accel (ax1+ax2) (ay1+ay2) (az1+az2)

accel::BHTree->Body->Accel
isFar::BHTree->Body->Bool
```

3.3.3 Sequential Tuning

A number of sequential optimisation techniques, targeting stack and heap consumption, are applied to improve the runtime.

Optimisations: The following general optimisations apply to both algorithms:

Reducing stack consumption: The naive implementation of the algorithm suffers from an excessive stack consumption when a large number of bodies and iterations are used. Space profiling helps to understand the memory usage of each algorithm and to find space leaks. To fix such leaks and to improve general performance of the sequential algorithm, the following steps are taken:

- *Tail recursion:*

  By making functions tail recursive by using accumulating parameters helps to reduce space consumption. This is a well known technique with functional languages[2].

- *Strictness:*

  More specifically to a lazily evaluated language such as Haskell, strictness annotations can be added, where delaying evaluation is not necessary thus avoid unnecessary thunking of computations[3]. This can be achieved in a number of ways, for instance, by using the `seq` primitive, the strict function application (`$!`) or strictness annotation (`!`) provided by the BangPatterns[4] extension.

- *Algebraic types with strict fields:*

  Initially, type synonyms are used to represent position, velocity and mass as triple tuple, e.g. `type Pos = (Double,Double,Double)`. Through the use of more advanced data types provided in Haskell e.g. algebraic sum and product types with strict data fields, space leaks can be avoided and thereby improving performance considerably with high input sizes as we will see shortly.

Reducing heap consumption: While the GHC compiler performs numerous automatic optimisations, more opportunities can be exposed by specific code changes, in particular in fusing function applications where necessary. This removes any intermediate data structures that could potentially decrease performance.

[2]Tail recursion: https://wiki.haskell.org/Tail_recursion
[3]A thunk is a value that is yet to be evaluated in Haskell. https://wiki.haskell.org/Thunk
[4]Haskell Wiki – BangPatterns extension https://prime.haskell.org/wiki/BangPatterns

```haskell
-- A trivial example
map updatePos (map updateVel bs)

-- rewritten using function composition
map (updatePos . updateVel) bs
```

The composed version using a single `map` is easier to read, and, additionally, does not depend on compiler optimisation. The use of `foldr` in conjunction with list comprehension (`foldr/build`) also eliminates the intermediate lists produced by build and consumed by foldr. This is used in the Barnes-Hut tuning specifically.

Quantifying sequential tuning: Table 3.2 shows the runtimes, maximum residency and percentage of heap allocation change from previous version of each step of tuning the sequential all-pairs and Barnes-Hut algorithms.

All-Pairs

- *Version 1:* the initial version of the all-pairs program uses type synonyms and tuples to represent position, velocity, mass and acceleration. Type synonyms are not new data types but are used mainly for code clarity. For example, it is easy to read that a position consists of 3 doubles, thus representing it using a tuple.

```haskell
type Pos   = (Double,Double,Double)
type Vel   = (Double,Double,Double)
type Mass  = Double
type Accel = (Double,Double,Double)
```

- *Version 2:* change type synonyms to data types. This causes the initial runtime to go up by 60%. Using type synonym is usually more efficient as it incurs one less indirection than a data type. However, data types are more powerful and can be used with further optimisation as we will see in the following versions. The usage of data types makes it necessary to derive appropriate typeclasses, not shown here, e.g. `Eq` if we need to be able to compare them.

```haskell
data Pos   = Pos Double Double Double
data Vel   = Vel Double Double Double
data Mass  = Mass Double
data Accel = Accel Double Double Double
```

- *Version 3:* add strictness to avoid unnecessary thunking of computation. For example, the return type of the `accel` function below is `Accel`. By default the data fields of `Accel` are evaluated lazily, explaining why Version 2 uses a lot of heap. These values are needed anyway. By making them strict, they are computed eagerly. A clearer way to achieve this is by using the bang patterns

extension strictness annotation (!) instead of the most explicit `seq`. As the results show, this step accounts for the main reduction in heap and time by 78% and 96% respectively.

```
-- data fields evaluated lazily
accel bodyi bodyj = Accel (dx*jm*mag) (dy*jm*mag) (dz*jm*mag)
-- add strictness annotation (!)
accel bodyi bodyj = Accel ax ay az
  where
    !ax = (dx*jm*mag)
    !ay = (dy*jm*mag)
    !az = (dz*jm*mag)
```

- *Version 4:* use strict data fields. Making the data fields strict removes the need for the previous strictness annotation added inside the function. Additionally, the use of the `UNPACK` pragma indicates to the compiler to unpack the contents of the constructor field into the constructor itself, removing a level of indirection. This is compiler-specific and not a language issue.

```
data Pos = Pos {-# UNPACK #-} !Double {-# UNPACK #-} !Double {-
   # UNPACK #-} !Double
```

- *Version 5:* use appropriate higher order functions from the standard libraries e.g. use `foldl'`, which is tail recursive and does not accumulate intermediate results, instead of `foldl`. This ensures fixed heap usage and avoids the accumulation of huge thunk of computations. Though left fold is more space- and probably time-efficient, `foldr` is used in conjunction with `build` (the foldr/build rule) to eliminate intermediate lists produced by build and consumed by foldr.

- *Final version:* use single `Body` data type. This removes the need to deal with many different data types, makes the program more compact, and as a result, reduces the runtime by 10%. The maximum residency is reduced similarly.

```
data Body
      = Body
        { x  :: {-# UNPACK #-} !Double  -- pos of x
        , y  :: {-# UNPACK #-} !Double  -- pos of y
        , z  :: {-# UNPACK #-} !Double  -- pos of z
        , vx :: {-# UNPACK #-} !Double  -- vel of x
        , vy :: {-# UNPACK #-} !Double  -- vel of y
        , vz :: {-# UNPACK #-} !Double  -- vel of z
        , m  :: {-# UNPACK #-} !Double } -- mass
```

Barnes-Hut In addition to the optimisations applied to all-pairs, the main tuning for Barnes-Hut sequential performance is the use of `foldr/build`.

- *Version 1:* The first Barnes-Hut version includes all all-pairs optimisations detailed earlier. This gives a good initial runtime.

Version	Runtime (s)	Max Resi (KB)	Heap Alloc (%)
All-Pairs (5000 bodies)			
allpairs1	39.47	1976	–
allpairs2	63.24	3533	+30.9
allpairs3	2.54	1273	-77.9
allpairs4	2.15	726	-28.9
allpairs5	2.14	726	-0.0
allpairs-final	1.94	69	-0.0
Barnes-Hut (80000 bodies)			
bh1	41.90	28	–
bh-final	33.37	27	-88.6

Table 3.2: Sequential tuning

- *Version 2:* Use `foldr/build` in the `calcAccel` function which eliminates intermediate lists produced by `build` and consumed by `foldr`.

```
-- before
foldl' addAccel (Accel 0 0 0) [calcAccel st b | st <- subtrees]
-- after
foldr addAccel (Accel 0 0 0) [calcAccel st b | st <- subtrees]
```

Compiler optimisation: The sequential runtimes up to now are based on fully optimised code and on an input size of 5000 bodies for the all-pairs program. GHC's automatic optimisation already manages to improve time and heap performance significantly. Table 3.3 shows that the runtime is very high when compiler optimisation (`-O0`) is disabled. We use 2000 bodies here so the base runtime (without optimisation) is within one minute and show the difference between level 1 and 2 optimisation. The table shows that GHC's aggressive optimisation machinery manages to automatically improve performance of the final version by a factor of 13.0, relative to the unoptimised version, but does not find many more sources of optimisation when going to full optimisation. Algorithmic optimisation, represented by all the sequential tuning steps as discussed in this section, gives a performance gain of a factor of 12.0 (combined with full optimisations). Both taken together account for a sequential speedup of 155.7.

		allpairs1			allpairs-final		
	Optimisation	RT (sec)		Max Resi (KB)	RT (sec)		Max Resi (KB)
`-O0`	(disable optimisations)	48.26	(–)	1667000	18.58	(–)	71
`-O1`	(standard optimisations)	3.76	(92%↓)	1094	0.31	(98%↓)	69
`-O2`	(full optimisations)	3.72	(1%↓)	1117	0.31	(–)	69

Input: 2000 bodies; RT=Runtime

Table 3.3: Effect of compiler optimisation (All-Pairs)

Baseline comparison: We use the n-body benchmark from the Computer Language Benchmarks Game website [5] as baseline to compare with our all-pairs implementations. We are particularly interested in the Haskell and the C versions, both implementing the same algorithm as our Haskell version. At the time of publication (Totoo and Loidl, 2014a), the baseline Haskell version, which is run against 50 million iterations and 5 bodies as input, is slower than the C versions by a factor of 1.2.

We adapt the Haskell and C versions from the Benchmarks Game with our input generation function and run them against a large number of bodies as we use for our all-pairs program (16000 bodies and 1 iteration). This allows for a head-to-head comparison between two Haskell versions and a C version. We note that our all-pairs Haskell program runs at 20.14s compared to 12.55s for baseline Haskell version, making it slower by a factor of 1.6. Compared to the C version which runs at 5.70s, our all-pairs Haskell program is 3.5 time slower. However, it is worth pointing out that the Benchmarks Game version, though implementing the same algorithm, is highly optimised by expert programmers in Haskell and the GHC compiler, and makes use of inplace update operations organised through monadic code and unsafe operations such as `unsafePerformIO`. The disadvantage of this approach is that it introduces sequentialisation in the code as part of the optimisation and therefore loses many potential for parallelism.

As a baseline comparison, we observe a sequential overhead of a factor of 3.5 against the C version from the Computer Language Benchmarks Game. By remaining faithful to a purely functional programming model, our implementation provides opportunities for parallelism, that do not exist in the lower-level implementations and have to be refactored in a time-consuming and error prone parallelisation methodology. By exploiting high-level parallelism, we can compensate for the sequential overhead, using a fairly small number of processors, and achieve high scalability of our code, through the usage of more massively parallel hardware. In particular, the final parallel program will not be tied to one particular class of architectures, nor to a certain number of processors.

3.4 Parallel Implementation

Through the use of high-level constructs and functions implementing common skeletons in each programming model, the sequential algorithm is expected to require only a few changes to obtain an initial parallel version of both algorithms. The GpH im-

[5] The Computer Language Benchmarks Game (Fulgham, 2012)
http://benchmarksgame.alioth.debian.org/

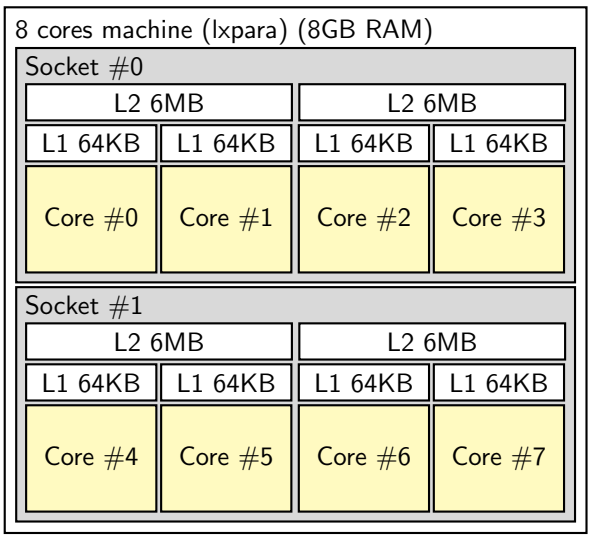


Figure 3.4: Desktop-class 8 cores machine topology map.

plementations use Evaluation Strategies and not the primitives. We present results from the initial implementation and show, through a set of parallel performance tuning steps, how runtime and speedup are improved. The experimental setup for these results is given below. A detailed performance evaluation is covered in Section 3.5.

Experimental Setup: The machine used for measurements has the following hardware specifications: an Intel Xeon CPU E5410 2.33 GHz processor with 8 cores, 8GB RAM and 12MB L2 cache. Figure 3.4 depicts the topology map of the machine which runs a 64-bit CentOS 5.8 Linux distribution with kernel version 2.6.18-308.11.1.el5. The GHC compiler version 7.0.1 is used with the `parallel` and `monad-par` packages for GpH–Evaluation Strategies and Par monad respectively. The Eden extension to the GHC compiler version 6.12.3 (GHC-6.12.3-Eden) is used for Eden. At the time of experiment, both the GHC and GHC-Eden compilers have newer but not yet stable versions. Any comparison between GpH/Par monad and Eden is therefore on the basis of speedup.

Input Set: Unless explicitly stated otherwise in relevant sections and tables, the input size for all-pairs measurements is 16000 bodies; and for the Barnes-Hut measurements is 80000 bodies as it is a more efficient algorithm and able to cope with high number of bodies. The input ensure the runtimes of one iteration are within a minute for both algorithms. Larger input sizes are used for scale-up tests in the evaluation section and smaller sizes for incremental sequential or parallel optimisation phases to highlight key statistics. The tables and graphs show absolute speedups. The runtimes are the mean running times obtained from 3 runs.

Time profiling: Identifying the main source of parallelism is the first step in the parallel implementation. Time profiling points out the "big eaters," that is, functions that take the largest percentage of the total time. Listing 3.10 shows the time and allocation profiling for both algorithms.

Listing 3.10: Time and Allocation Profiling Report

```
                                                  individual     inherited
COST CENTRE          MODULE  entries  %time %alloc  %time %alloc

-- All-Pairs (2000 bodies, 1 iteration)

doSteps              Main          1    0.0    0.0  100.0   99.8
 updatePos           Main       2000    0.0    0.0    0.0    0.0
 updateVel           Main       2000    3.1    1.9   99.9   99.8
  accel              Main    4000000   23.5   37.2   94.5   93.3
  deductChange       Main    4000000    2.4    4.5    2.4    4.5

-- Barnes-Hut (8000 bodies, 1 iteration)

doSteps              Main          1    0.0    0.0   99.9   99.8
 findBounds          Main          1    0.0    0.0    0.0    0.0
 buildTree           Main      11804    0.0    0.0    0.2    0.2
  splitPoints        Main       3804    0.0    0.0    0.1    0.2
  calcCentroid       Main      11804    0.0    0.0    0.0    0.0
 updatePos           Main       8000    0.0    0.0    0.0    0.1
 updateVel           Main       8000    0.0    0.0   99.7   99.5
  calcAccel          Main   14893157    4.8    7.1   99.7   99.5
   accel             Main   12193706   12.9   16.1   65.1   63.3
   isFar             Main   10247211    7.1    9.2   29.7   29.1
```

In both algorithms, the top-level `doSteps` function performs the iterations and inherits the largest percentage of time. The main source of parallelism arises from the update velocity function `updateVel` which is used as the function argument of a map operation in both all-pairs and Barnes-Hut. It accounts for almost 100% of the inherited overall time from other function calls. As it is used in a map, it presents data-oriented parallelism.

```
1 new_bs = map (updatePos . updateVel) bs
```

While the tree construction phase can naturally be done in parallel, the time profile indicates that `buildTree` accounts for less than 1% of the time in the Barnes-Hut algorithm. Parallelising this part of the program may not cause any significant improvement but could, on the other hand, create overheads. However, with the same number of bodies as used for all-pairs, the time percentage spent in `buildTree` reaches approximately 12%. This is explained by less computation involved in the acceleration calculation phase and thus better distribution of the time between the two phases. Additionally, depending on the distance threshold which defines when a body is considered to be too far away, the time spent in `updateVel` can vary significantly. For instance, if the distance threshold is high (closer to 1), the traversal

is very fast, as a result of little computation involved. This is not very good for parallelism as the cost of creating parallelism may be higher than the actual computation. A small threshold value, e.g. t=0.1, results in a slower runtime, and t=0 degenerates to pair-wise comparison. Ideally, we use t=0.25 which gives a reasonable approximation, accuracy and speed.

Both the all-pairs and Barnes-Hut algorithms are structured such that the function to update the velocity and position of each body can be applied to the list of all bodies in a map operation, thus exposing data-oriented parallelism. All of the three models provide at least a basic parallel map for data-parallel operations.

3.4.1 P1: GpH–Evaluation Strategies

P1.1: All-Pairs

Initial parallelism is obtained by replacing `map` with `parMap` which is implemented in terms of `parList` (see Section 3.1.1) such that the velocity for each body is computed in parallel.

Listing 3.11: GpH–Evaluation Strategies parallel map

```
1 new_bs = parMap rdeepseq (updatePos . updateVel) bs
2
3 -- equivalent to
4 new_bs = map (updatePos . updateVel) bs `using` parList rdeepseq
```

The function composition, which is used as the map first argument, can be turned into a pipeline using the parallel `.||` combinator. This arranges the composition between the two functions to proceed in parallel. The result of `updateVel` is evaluated in parallel with the application of the first function. However, given that the `updatePos` function does negligible computation as opposed to `updateVel`, this is not a useful source of parallelism and therefore not considered any further. It demonstrates, however, that this programming model makes it easy to compose different sources of parallelism, and to prototype alternative parallelisations, without having to restructure the existing code in a fundamental way.

Listing 3.12: Global statistics of a parallel run on 2 cores

```
./allpairs 16000 1 +RTS -N2 -s
56026.00329381344
54897.906546913
time taken: 16.76s
  31,145,652,016 bytes allocated nonein the heap
      27,366,360 bytes copied during GC
       2,999,520 bytes nonemaximum residency (5 sample(s))
         517,760 bytes nonemaximum slop
              10 MB total memory nonein use (0 MB lost due to
                 fragmentation)

  Generation 0: 44953 collections, 44952 parallel,  2.70s,  1.22s
     elapsed
  Generation 1:     5 collections,     5 parallel,  0.05s,  0.03s
     elapsed

  Parallel GC work balance: 1.12 (3256443 / 2904329, ideal 2)

                        MUT time (elapsed)       GC time  (elapsed)
  Task  0 (worker) :    4.90s    ( 15.63s)       1.51s    (  0.82s)
  Task  1 (worker) :    6.21s    ( 15.63s)       0.52s    (  0.08s)
  Task  2 (bound)  :    9.96s    ( 15.63s)       0.72s    (  0.36s)
  Task  3 (worker) :    0.00s    ( 15.63s)       0.00s    (  0.00s)

  SPARKS: 16000 (8192 converted, 0 pruned)

  INIT  time    0.00s  (  0.01s elapsed)
  MUT   time   21.07s  ( 15.63s elapsed)
  GC    time    2.75s  (  1.25s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time   23.83s  ( 16.89s elapsed)

  %GC time      11.6%  (7.4% elapsed)

  Alloc rate    1,477,931,568 bytes per MUT second

  Productivity  88.4% noneof total user, 124.8% noneof total elapsed
```

The performance results from this naive version are disappointing. In the best case, we observe a speedup of 1.4 on 4 processors and a slow-down on 8 processors. The reason for this poor performance is too fine granularity: considerable overhead associated with generating a thread for every list element, potentially 16000 in total. While the generation of sparks is cheap – it amounts to adding a pointer to a queue – the generation of a thread requires the allocation and initialisation of a thread state object (TSO), which among other data contains the stack used by the thread. In this case, the computation performed by one thread, namely updating the velocity and position of one body, is too small in comparison with the overhead for TSO initialisation and for scheduling the available threads. The statistics in Listing 3.12 summarise the execution on 2 cores. In total, 16000 sparks are created, one for each list element, and of these 8192 are converted into threads. The remaining sparks are overflowed (shown in newer GHC report) due to the spark pool size limit which

is set at 8k by default. New sparks are discarded when the pool is full. Since the nature of the parallelism is data-parallel, no work can be subsumed by a sibling-thread, and thus lazy task creation is not effective in automatically increasing thread granularities.

To tune parallel performance, we control the number of sparks created by grouping elements into larger units called chunks. Instead of creating a spark for each element in the list, the list is broken down into chunks and a spark is created for each chunk, thus significantly reducing the thread creation overhead. The number of chunks is determined by the number of available processors. Having too few chunks may result in some processors not getting enough work while too many chunks create excessive overhead.

As often in data-parallel programs, a careful balance between low thread management overhead and massive parallelism is crucial. There is no dynamic parallelism here, i.e. all parallelism is generated at the beginning. Thus a low, fixed number of chunks is likely to be the best choice for performance. Each processor does not necessarily get the same number of chunks. Using more chunks retains more flexibility for the runtime system, because a faster or more lightly loaded processor can pick-up new work after having finished its initial work allocation.

In the following, we survey three ways to introduce chunking (or clustering) into the algorithm. The language-level differences between these approaches are discussed in more detail in (Marlow et al., 2010).

1) Explicit Chunking The most obvious way of performing chunking, is to explicitly apply functions performing chunking before and de-chunking after the data-parallel core of the application (see below). Explicit chunking is also used in the Par monad version, and its performance discussed in Section 3.4.2.

Listing 3.13: Explicit chunking

```
1 s = 1000 -- chunk size
2 f = (updatePos . updateVel)
3 new_bs = concat (map (map f) (chunk s bs) `using` parList rdeepseq)
```

Used directly in the application code, this technique obfuscates the computational core of the application, and introduces an intermediate data structure that is only needed to increase thread granularity.

2) Strategic Chunking Another skeleton-based approach to introduce chunking is to modify the definition of the strategy and encode chunking in addition to the specification of parallelism inside this skeleton (as we introduced in Section 3.1.1).

Thus, we change `parList` to `parListChunk`, which takes an additional argument, specifying the chunk size. The `parListChunk` strategy applies the given strategy, in this case `rdeepseq`, to each chunk. This achieves a clean separation of computation and coordination, leaving the core code unchanged, and hiding the intermediate data structure in a custom strategy. However, this strategy is now fixed to one parallel pattern and one way of chunking.

Listing 3.14: Strategic chunking

```
s = (length bs) `quot` (numCapabilities * 4) -- 4 chunks/PE
f = (updatePos . updateVel)
new_bs = map f bs `using` parListChunk s rdeepseq
```

The chunk size s is determined using a simple calculation:

$$s = \frac{N}{(PE \times chunksPerPE)} \tag{3.9}$$

where s is the chunk size, N is the input list size, PE is the number of cores, and $chunksPerPE$ is the number of chunks to be allocated to each PE.

While generating exactly one chunk per processor might intuitively seem to be the best choice, it is also the least flexible one, because it deprives the runtime system from distributing parallelism in the case where one processor suffers from a high external load. Therefore, a small number of chunks greater than one is usually a good choice. In this case, the right balance is to have approximately four chunks per processor. Table 3.4 shows the number of bodies processed by each processor for an all-pairs experiment involving 16000 bodies and for possible number of chunks allocated to each PE equals to 1, 2 and 4.

	chunks per PE		
no. PE	1	2	4
1	16000	8000	4000
2	8000	4000	2000
4	4000	2000	1000
8	2000	1000	500

Table 3.4: No. of bodies in each chunk (chunk size).

Table 3.5 shows the runtime and speedup results for different number of chunks per processor and on up to 8 processors using `parListChunk` strategy, motivating our choice of a chunk size leading to 4 chunks per PE in the code in Listing 3.14. The table shows that on 2 cores below performance deteriorates by up to 12% for larger chunk sizes (highlighted – 8000 vs 2000).

	nochunk		1chunk/PE		2chunks/PE		4chunks/PE	
no. PE	RT (s)	SP	RT (s)	SP	RT (s)	SP	RT (s)	SP
Seq.	20.04	1.00	20.05	1.00	20.03	1.00	20.06	1.00
1	20.64	0.97	27.80	0.72	24.07	0.83	22.10	0.91
2	*16.76*	*1.20*	**12.73**	*1.58*	*11.98*	*1.67*	**11.33**	*1.77*
4	13.89	1.44	6.42	3.12	6.15	3.26	5.97	3.36
8	15.64	1.28	3.40	5.90	3.35	5.98	3.29	6.10

Algorithm: All-Pairs; Input: 16k bodies, 1 iteration

Table 3.5: GpH–Evaluation Strategies runtimes and speedups (All-Pairs).

3) Implicit Clustering A more compositional way to introduce chunking is to delegate the task to an instance of a new `Cluster` class, with functions for performing clustering and declustering (or flattening) as introduced in (Loidl et al., 2001). We can use available abstractions of performing an operation on each element of a cluster (`lift`) and of flattening the resulting data structure (`decluster`). Thus, to define an instance of this class the programmer only needs to define `cluster` in such a way that the specified proof obligation is fulfilled e.g. an instance for lists as given below requires us only to define `cluster`.

Listing 3.15: `Cluster` typeclass definition

```
class (Traversable c, Monoid a) => Cluster a c where
  cluster    :: Int -> a -> c a
  decluster  :: c a -> a
  lift       :: (a -> b) -> c a -> c b

  lift = fmap       -- c is a Functor, via Traversable
  decluster = fold  -- c is Foldable,  via Traversable
  -- we require: decluster . cluster n == id

instance Cluster [a] [] where
  cluster = chunk
```

Based on this class definition, we can then separately define an `evalCluster` strategy, which uses these functions before and after applying its argument strategy to each cluster, thus separating the definition of parallelism from any form of clustering.

Listing 3.16: Implicit clustering

```
evalCluster::Cluster c => Int->Strategy a->Strategy a
evalCluster n s x = return (decluster (cluster n x `using` cs))
  where cs = evalTraversable s :: Strategy c
```

Using this approach, we can add clustering to the basic data-parallel strategy, without changing the original strategy at all — replace `evalList (rpar `dot` rdeepseq)`, which is the definition of `parList`, by `evalCluster s (rpar `dot` rdeepseq)`. In short, the compositionality of this style of programming allows us to specify a parallel strategy combined with a clustering strategy. This provides more flexibility in aggregating collections in ways that cannot be expressed using only

strategies.

In summary, the code below shows the application of the three clustering techniques:

Listing 3.17: Clustering techniques summary

```
-- explicit clustering
concat (map (map f) (chunk s bs) `using` parList rdeepseq)
-- strategic clustering
map f xs `using` parListChunk s rdeepseq
-- combining parallel and clustering strategies
map f xs `using` evalCluster s (rpar `dot` rdeepseq)
```

	nochunk		parListChunk		evalCluster	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	20.04	1.00	20.06	1.00	20.02	1.00
1	20.64	0.97	22.10	0.91	19.71	1.02
2	16.76	1.20	11.33	1.77	10.93	1.83
4	13.89	1.44	5.97	3.36	5.83	3.43
8	15.64	1.28	3.29	6.10	3.28	6.10

Algorithm: All-Pairs; Input: 16k bodies, 1 iteration

Table 3.6: GpH–Evaluation Strategies runtimes and speedups (All-Pairs; different chunking strategies)

Table 3.6 summarises runtime and speedup when using no chunk and chunking through `parListChunk` and `evalCluster`, always with four chunks per PE. Most notably, the implicit `evalCluster` version achieves the same performance as the strategic `parListChunk`. Thus, using this more compositional version, which makes it easy to introduce and modify clustering strategies separately from specifying parallelism over the data structure, does not incur a significant performance penalty.

P1.2: Barnes-Hut

Sequential profiling of the Barnes-Hut algorithm identifies the same `updateVel` function as the most compute-intensive. As the call count for this function shows, this is due to the iterative use in the top level map. Therefore, the Barnes-Hut algorithm is parallelised in the same, data-parallel way as the all-pairs version. Since an abundance of fine-grained parallelism is also a problem in this version, we use the same form of chunking to tune the parallel performance.

In this version a natural parallel phase is `buildTree`, where sub-trees can be constructed in parallel. But as the profiling report showed earlier, it does not account for a significant percentage of the overall time, and therefore the benefits from parallelising this stage are limited. However, it is cheap to mark the stage as parallel computation in GpH, and whether to take the spark for parallel execution is up to the runtime system.

Tree-specific optimisation An important generic optimisation that is applied in the `buildTree` function is thresholding. By adding an explicit argument to the function that represents the current level of the tree, the generation of parallelism can be restricted to just the top levels. This makes sure there are not too many parallel threads for the tree construction otherwise it would cause overheads with large number of bodies which would require a big tree structure. This approach is static and in the next chapter we define and apply more advanced tree traversal strategies for the same algorithm.

	Top-level map only		Parallel buildTree	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	33.31	1.00	33.31	1.00
1	35.77	0.93	35.77	0.93
2	21.50	1.55	21.61	1.54
4	11.00	3.03	10.77	3.09
8	6.77	4.92	6.11	5.45

Input: 80k bodies, 1 iteration

Table 3.7: GpH–Evaluation Strategies runtimes and speedups (Barnes-Hut)

Table 3.7 shows that parallel `buildTree` has no big impact on performance on up to 4 cores and a slight improvement on 8 cores. More importantly, it does not cause additional overhead. The main observation though is that the algorithm does not achieve as good speedup as the all-pairs algorithm. This is expected as all parallel tasks in the all-pairs algorithm have the same amount of computation to perform, i.e. the parallelism is regular, whereas the acceleration calculation steps in the Barnes-Hut algorithm vary for each body depending on its location. Some bodies require traversing deeper inside the tree to calculate the net acceleration, while for some, it may not require to do so. Quantifying the irregularity of the computation involved in Barnes-Hut, random generation of 80000 bodies gives an unbalanced tree with minimum tree depth 6 and maximum depth 9. In the next chapter, we look at different input distributions for the same algorithm.

3.4.2 P2: Par Monad

P2.1: All-Pairs

We use the pre-defined parallel map provided in the Par monad library to add parallelism the same way as we did in GpH. In contrast to GpH, the parallel computation happens in a monad and therefore the result has to be extracted using `runPar`.

Listing 3.18: Par monad parallel map

```
1 new_bs = runPar $ parMap (updatePos . updateVel) bs
```

The runtime and speedup given in Table 3.8 shows that the Par monad achieves slightly better performance than GpH. Notably, the results without chunking show a speedup of 6.17 on 8 cores. The reason for this efficient behaviour is the work-inlining scheduler, which distributes the potential parallel tasks to a number of implicitly created threads and then executes the task within the existing thread. This dramatically reduces the thread creation overhead, at the expense of less flexibility in how to distribute the tasks in the first place. This model is well suited for homogeneous, multi-core architectures and no explicit chunking is needed to improve parallel performance. However, as shown in Table 3.9 the use of chunking reduces the maximum residency by 50% from 4822MB to 2417MB for a parallel run on 8 cores, because fewer tasks will be active at any point, thus reducing the amount of live data needed by these tasks.

	nochunk		1chunk/PE		2chunks/PE		4chunks/PE	
no. PE	RT (s)	SP	RT (s)	SP	RT (s)	SP	RT (s)	SP
Seq.	20.30	1.00	20.06	1.00	20.03	1.00	20.04	1.00
1	20.48	0.99	20.16	1.00	20.08	1.00	20.19	0.99
2	10.96	1.85	10.91	1.84	10.98	1.82	10.94	1.83
4	5.93	3.42	5.85	3.43	5.82	3.44	5.78	3.47
8	3.29	6.17	3.24	6.19	3.22	6.22	3.25	6.17

Algorithm: All-Pairs; Input: 16k bodies, 1 iteration

Table 3.8: Par monad runtimes and speedups (All-Pairs).

Par. run on 8 cores	nochunk	chunking
copied during GC (MB)	31527	16171
max residency (MB)	4822	2417

Algorithm: All-Pairs; Input: 16k bodies, 1 iteration

Table 3.9: Par monad statistics: GC and max. residency.

Table 3.8 also shows that varying the number of chunks causes negligible change to the runtime and speedup. However, to maintain low memory residency and to facilitate scalability beyond the number of cores available for these measurements, a chunking policy is preferred.

The Par monad, however, does not come with a pre-defined parallel map function with chunking. So, we use explicit chunking, as described earlier. In the following code extract, s is the chunk size and is used in the same way as in GpH to produce an appropriate number of chunks to match the number of cores.

Listing 3.19: Par monad explicit chunking

```
1 new_bs = parMapChunk (updatePos . updateVel) s bs
2
3 parMapChunk f n xs = concat (runPar $ parMap (map f) (chunk n xs))
```

P2.2: Barnes-Hut

For the Barnes-Hut algorithm, we note that chunking causes a noticeable improvement in the speedup from 5.29 to 6.50 on 8 cores, unlike the all-pairs version (see Table 3.8). A chunking strategy is more important due to the fact that a large number of bodies are used in this algorithm, thereby making the memory usage overhead significant.

The large number of bodies used in Barnes-Hut makes the heap usage significantly higher compared to the all-pairs algorithm. Without chunking, the maximum residency is 83MB and productivity is at 63%. With chunking, residency is 55MB and improved productivity by 10%. This reduced percentage of garbage collection time has an immediate impact on the performance of the parallel program.

	Top-level map only		Parallel buildTree	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	33.39	1.00	33.65	1.00
1	34.49	0.97	33.96	0.99
2	17.72	1.88	17.79	1.89
4	9.21	3.63	8.97	3.75
8	5.91	5.65	5.18	6.50

Input: 80k bodies, 1 iteration

Table 3.10: Par monad runtimes and speedups (Barnes-Hut)

Executing the `buildTree` stage in parallel using Par monad shows a noteworthy improvement on 8 cores from the GpH version (Table 3.7 and 3.10), though this stage does not represent a large part of the overall computation for this algorithm.

3.4.3 P3: Eden

P3.1: All-Pairs

As with the previous two models, we only need a parallel map implementation to add data-oriented parallelism to the algorithm. Eden offers several skeletal approaches and, in particular, has several implementations of parallel map as described earlier. The default `parMap` implementation creates a process for each list element causing far too much overheads in terms of number of processes instantiated and messages communicated (16001 and 64000, respectively). Table 3.11 shows the number of processes, threads, conversations and messages overheads of each skeleton. Observing number of processes and communications is motivation for picking a different strategy, and with it numbers drop significantly and speedup improves.

	processes	threads	conversations	messages
parMap	16001	32001	64000	64000
parMapFarm	9	17	48	32048
parMapFarm w/ chunking	9	17	48	80
offlineFarm w/ chunking	9	17	40	56

Table 3.11: Eden skeleton overheads - par. run on 8 cores

The `parMapFarm` farm process skeleton creates the same number of processes as the number of available processing elements. But the message overheads remain. Each list element is communicated as a single message which generates 32048 messages. While this is a high number, performance is considerably improved compared to the naive parallel map and good speedup is achieved. This confirms that process creation overheads is far more important than the number of messages.

With further parallel tuning to reduce message overheads, we use chunking to break the stream into chunks of size 1000 items which are then sent as one message. This enables the process to do more computation at one time rather than having to send and receive messages in between. The chunking reduces the total number of messages communicated in the `parMapFarm` version from 32048 to just 80 messages. As a result of this, the runtime and speedup are improved as shown in Table 3.12. The offline farm process, where process input is evaluated by the child process instead of the parent process, causes a small performance improvement compared to the farm process. Sending the process input to child processes to be evaluated is intended to reduce the combined time the parent process has to spend on reducing all input. However, in our algorithm, the input to a farm process is a `Body` type with strict fields. So there is not much reduction happening after sending it to the child processes.

	parMap		parMapFarm		parMapFarm w/ chunking		offlineFarm w/ chunking	
no. PE	Runtime (s)	SP	Runtime (s)	SP	Runtime (s)	SP	Runtime (s)	SP
Seq.	22.11	1.00	22.13	1.00	22.13	1.00	22.13	1.00
1	362.38	0.06	23.67	0.93	22.91	0.97	23.02	0.96
2	294.99	0.07	11.91	1.86	11.57	1.91	11.53	1.92
4	259.19	0.09	6.08	3.64	5.82	3.80	5.80	3.82
8	245.72	0.09	3.41	6.49	3.09	7.16	3.04	7.28

Input: 16k bodies, 1 iteration; SP=Speedup

Table 3.12: Eden runtimes and speedups (All-Pairs)

P3.2: Barnes-Hut

Although the Eden all-pairs implementation has given the best performance so far compared to the other two models, the performance for the Barnes-Hut algorithm using Eden is not as good as the other models. The speedup is roughly the same compared to the other two models on 1 to 4 cores but then there is no further speedup up to 8 cores. The best speedup is achieved using the offline farm process with chunking as given in Table 3.13. This is partially due to the high maximum residency caused by all PEs, in turn caused by the large number of bodies used. The topology map in Figure 3.4 shows that the machine used is organised in 2 sockets with 4 cores each. The process creation and communication model of Eden proves to be less efficient on this particular architecture; communication involved is across sockets not NUMA regions. Furthermore, this indicates that spark-oriented parallelism, as in GpH, and parallel tasks, as in the Par monad, deal better with dynamic and irregular parallelism. We further evaluate the performance against the other implementations in the next section (Sec 3.5).

	offlineFarm w/ chunking			
	Top-level map only		Parallel buildTree	
no. PE	Runtime (s)	Speedup	Runtime (s)	Speedup
Seq.	35.34	1.00	35.32	1.00
1	33.11	1.07	33.89	1.04
2	18.41	1.92	18.73	1.89
4	10.62	3.33	11.24	3.14
8	10.37	3.41	10.71	3.30

Input: 80k bodies, 1 iteration

Table 3.13: Eden runtimes and speedups (Barnes-Hut)

3.5 Performance Evaluation

3.5.1 Tuning

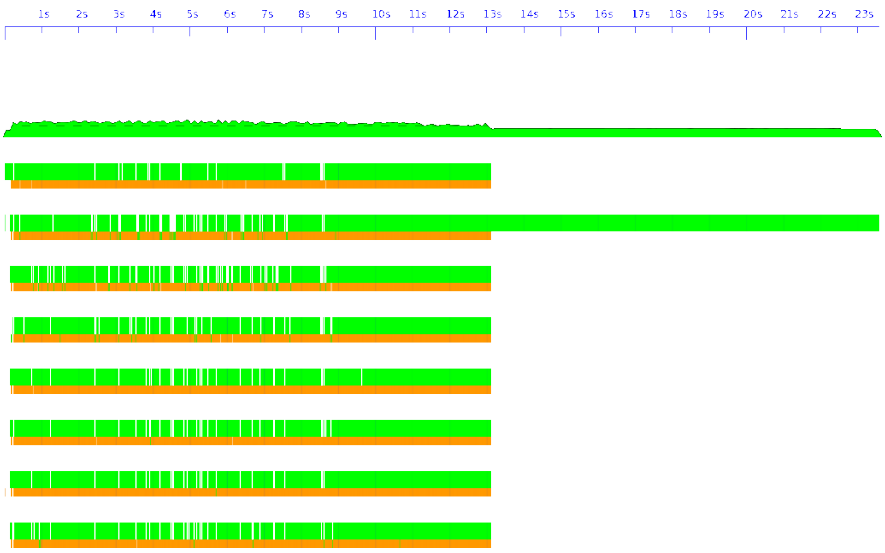
While an initial parallel version was produced with only a one-line program change, GpH required some parallel performance tuning, in particular by using chunks to generate a bounded number of threads of suitable granularity. Selecting a good chunk size required a series of experiments, establishing four threads per processor to be the best balance between massive parallelism and coarse thread granularity. The irregular nature of the parallelism in the Barnes-Hut version, compared to the all-pairs version, diminishes the achieved speedup, but also demonstrates that the runtime system effectively manages and distributes the available parallelism, without requiring further application code changes from the programmer.

We compare these results with the Par monad version which uses a highly tuned, data-parallel map skeleton, and thus can efficiently handle a large number of parallel tasks with an initial version, eliminating the need for explicit chunking. However, chunking does improve the maximum residency and therefore the scalability of the application.

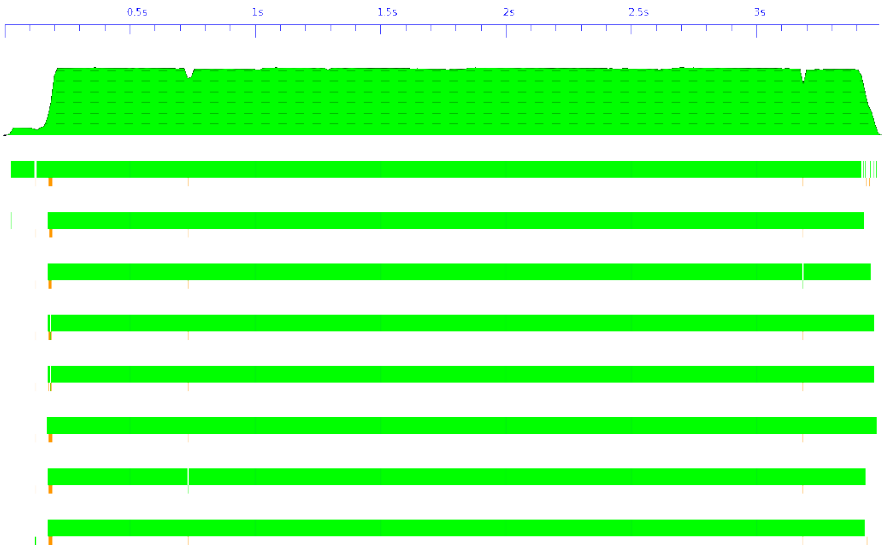
Eden provides a rich set of skeletons and parallelisation amounts to selecting the most suitable skeleton for the main worker function. Ample literature on the advantages and disadvantages of different skeletons helps in making the best decision for a specific application and architecture. For fine-tuning the parallel performance, however, an understanding of the process creation and message passing is required to minimise the amount of communication in this distributed-memory model.

For any high-level language model, good tool support is crucial in order to understand the concrete dynamic behaviour of a particular algorithm and to tune its performance. Threadscope helps to visualise the work distribution among the available processors for GpH–Evaluation Strategies and Par monad. Figure 3.5 shows the work distribution of running the all-pairs program on 8 cores before and after parallel tuning using strategic chunking. The top horizontal bar on the graph shows the overall activity, measured in terms of the total number of active processors at each time in the execution. The other bars shows the activity of each processor with green, orange and red representing running, garbage collection and idle time, respectively. The number of sparks is given in the runtime statistics for GpH. For Par monad, however, the exact number of threads created is not given.

Similarly, the Eden Trace Viewer (EdenTV) gives more detailed information about processes, their placement, conversations and messages between processes. Figure 3.6 compares the use of the naive parallel map (`parMap`) against the farm process implementation using stream chunking (`parMapFarmChunk`). It shows the overheads of too many processes, and consequently messages, being generated in the former. The overheads are eliminated in the tuned version. Each line represents the activity on one processor with green and blue representing “running” and “waiting for data”, respectively. While the first trace shows frequent changes between running and waiting states, reflecting the element-by-element transfer of the input data from the master to the workers, the second trace shows much better utilisation as uninterrupted activity once the entire block of input data has been received by a worker. The master remains idle, while the workers produce their results. A related set of skeletons allows a dual usage of the master process as worker in such a case, and can be used to improve performance further.



(a) nochunk

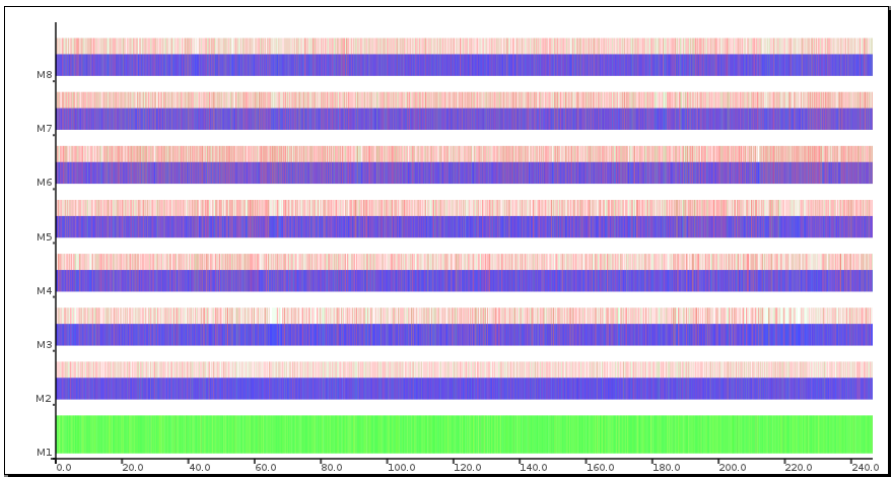


(b) 4chunksPerPE

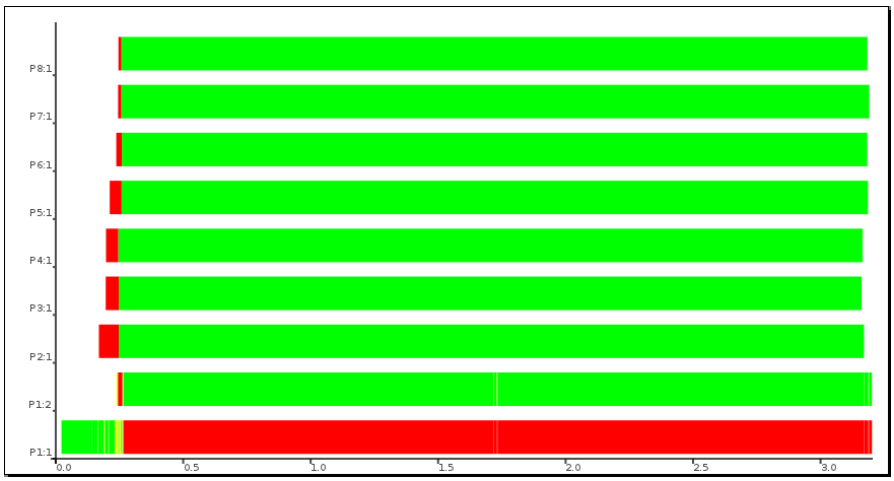
Figure 3.5: Threadscope work distribution (Par. run on 8 cores)

3.5.2 Speedup

The head-to-head comparison of speedups for the all-pairs versions in Figure 3.7(a) show that, despite a higher variability, the Eden implementation performs best, even though it was designed for distributed-memory architectures. This indicates that message passing can work well on shared-memory architectures. Using a highly tuned skeleton that avoids synchronisation bottlenecks on high-latency, distributed-memory systems, is beneficial even on a single multi-core. The support for light-weight parallelism in all three runtime systems helps to reduce the overhead that has to be paid for exposing parallelism. The GpH–Evaluation Strategies version is potentially more flexible and adaptive, through its dynamic, spark-based load distribution policy. This is beneficial in particular in heterogeneous environments, with dynamically changing external load. On an otherwise idle machine as used



(a) parMap

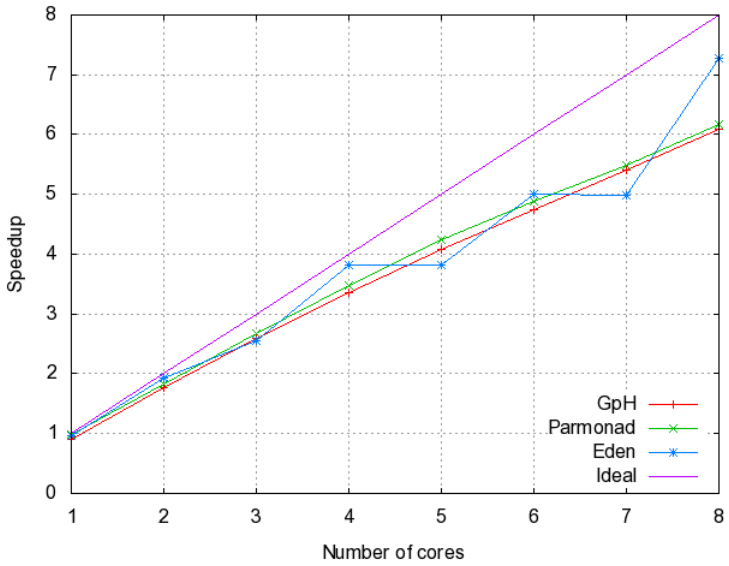


(b) parMapFarmChunk

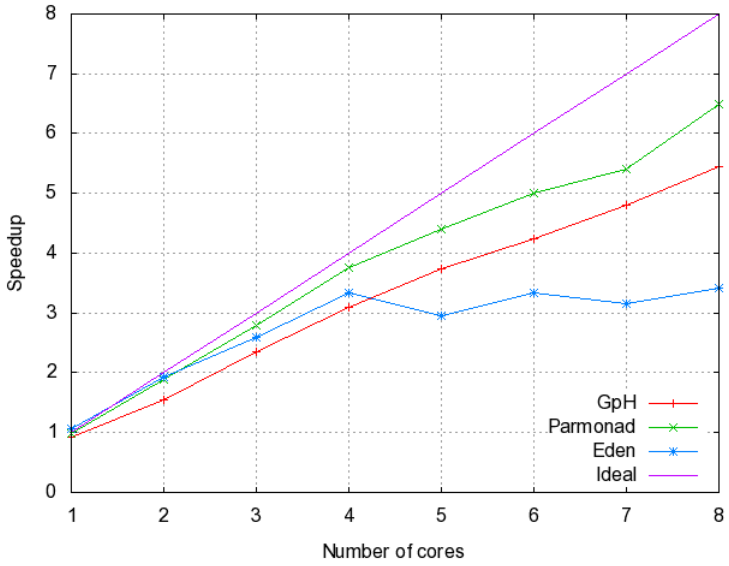
Figure 3.6: EdenTV using different map skeleton (Par. run on 8 PEs)

for these measurements, however, these benefits cannot be capitalised on, while the overhead still has to be paid for. The Par monad version performs very well with an initial, unoptimised version, but does not exceed the performance of the other systems in its final version. In this case, the overhead of encoding scheduling and other dynamic machinery in the application, rather than the runtime system, is higher compared to the other two systems.

The speedup results for the Barnes-Hut algorithm in Figure 3.7(b) show a significantly different picture. The dynamic behaviour of the Barnes-Hut algorithm differs from that of the all-pairs version, in that the parallel threads vary significantly in their granularities. The amount of work is significantly higher when calculating the impact of a densely populated cube in the oct-tree representation. In contrast, the parallelism in the all-pairs version is regular, with parallel tasks taking approximately the same amount of time to execute on different processors. The irregular parallelism in the Barnes-Hut version is more challenging to manage efficiently. The underlying runtime system of GpH and the application-level implementation



(a) All-Pairs



(b) Barnes-Hut

Figure 3.7: All-Pairs and Barnes-Hut speedup graph (1-8 cores)

of scheduling for Par monad, are designed to be very flexible and dynamic in their management of parallelism, in particular allowing for cheap transfer of potential parallelism. Considering the more challenging nature of the parallelism, GpH and Par monad achieve good speedups. The Eden version, however, suffered most severely from the irregular parallelism. This case shows the limitations of a purely skeleton-based approach, that relies on the existence of a wide range of skeletons for many different architectures. Since Eden is not limited to such a pure skeleton-based approach, but is a skeleton implementation language in its own right, further optimisation should be possible, by fine tuning an existing skeleton for this application.

Multicore Challenge input specification Finally, Table 3.14 shows the speedup results for the tuned versions of all-pairs and Barnes-Hut using the SICSA Multicore Challenge Phase II[6] input specification of 1024 bodies and 20 iterations, so we can compare with other systems. As expected, the speedups are slightly lower for the smaller input set and for an execution which requires synchronisation between the iterations. Still, the speedups of 5.23 for GpH and 5.63 for Par monad for the Barnes-Hut version are remarkable, for less than a dozen lines of code changes, and no structural changes to the original Haskell implementation. In particular, we surpass the calculated sequential overhead of a factor 3.5 compared to C on a moderate multi-core architecture and deliver superior, scalable performance with a high-level language model.

	All-Pairs				Barnes-Hut			
	GpH		Par monad		GpH		Par monad	
no. PE	RT (s)	SP	RT (s)	SP	RT (s)	SP	RT (s)	SP
Seq.	1.67	1.00	1.70	1.00	1.36	1.00	1.35	1.00
1	1.71	0.98	1.66	1.02	1.40	0.97	1.38	0.98
2	0.94	1.78	0.93	1.83	0.78	1.74	0.72	1.88
4	0.51	3.27	0.52	3.27	0.44	3.09	0.42	3.21
8	0.30	5.57	0.30	5.67	0.26	5.23	0.24	5.63

Input: 1024 bodies, 20 iterations; Runtime(RT); Speedup(SP)

Table 3.14: GpH and Par monad runtimes and speedups; Multicore Challenge input specification

3.5.3 Comparison of Models

All three models build on a sophisticated runtime system that automatically manages the synchronisation, coordination and communication necessary to achieve high parallel performance. The resulting programming model is one of semi-explicit parallel programming for GpH and Eden, where the programmer only has to identify the useful sources of parallelism, but explicit for Par monad, which allows one to encode archetypical runtime system functionality as high-level Haskell code. More commonly, however, pre-defined parallel skeletons are used to simplify the parallelisation and help portability.

The three variants differ in the way they facilitate tuning of the initial parallel algorithm, though. Being first class objects, evaluation strategies can be modified to enable different dynamic behaviour. For example, adding chunking to a data parallel algorithm can be done by composing strategies. This modularity is one of its main advantages. However, control of data locality is significantly more difficult,

[6]SICSA Multicore Challenge Phase II - http://www.macs.hw.ac.uk/sicsawiki/index.php/Challenge_PhaseII

because GpH relies on an implicit, work stealing scheduler to distribute the work. In contrast, in Eden, thread creation is mandatory on process application, and it provides finer control of co-location, by using partial applications. These features provide more opportunities for tuning the parallel program without abandoning the high level of abstraction. Finally, the Par monad is the most explicit form of controlling parallelism. Here, threads are explicit entities in the program, that have to be explicitly synchronised using established mechanisms such as `IVars`, which raises all the usual issues about parallel programming. However, by providing parallel patterns of computation, skeletons, these low-level issues can be hidden from the programmer. By implementing runtime system functionality on the Haskell level, an expert parallel Haskell programmer can also tailor the application, e.g. by implementing a custom scheduling algorithm.

Despite the high level of abstraction, the performance results show good speedups on 8 cores for GpH and Par monad (5.45 and 6.50, respectively), and 3.62 for Eden, all using the Barnes-Hut algorithm. Most notably, these results were achieved changing only a few lines of code. Introducing top-level data-parallelism changes only one line of the original code. Further optimisation code, e.g. for chunking, adds less than a dozen lines of auxiliary functions.

The main outcome from conducting this model comparison can be summarised as follows:

- All three parallel Haskell variants are able to achieve competitive multi-core speedups not only for the simple, regular all-pairs algorithm on lists but also for the more sophisticated, irregular Barnes Hut algorithm operating on custom oct-trees.
- The performance of the parallel all-pairs version surpasses the calculated performance of the sequential C version from the Computer Language Benchmarks Game (Fulgham, 2012) and achieves scalable performance up to the maximum of 8 cores.
- The ease of parallelisation allowed us to develop 6 versions (see Table 3.1), using three different variants of parallel Haskell and implementing both an all-pairs and a Barnes-Hut version.
- Well documented program transformations, in the form of local changes to the sequential program, reduce both heap and stack space consumption considerably and improve sequential performance by a factor of 12.0.
- Established techniques for tuning parallel performance, in particular chunking, were important to tune the GpH and Eden implementations of the algorithms.

- The Par monad version already achieves good parallel performance in its initial version, due to a highly optimised, work-inlining scheduler. In contrast, both the GpH and Eden versions require explicit chunking to achieve the same level of performance, but allow for more flexible tuning of performance.
- Interestingly Eden, which is designed for distributed-memory architectures, performs very well on a shared-memory setup using message passing, in particular for the all-pairs version. This strengthens the argument for a shared-nothing design of parallel runtime-system (Berthold et al., 2015).

3.6 Summary

In this chapter, we have implemented two versions of the n-body problem that use list and tree data structures, and parallelised them using GpH by implementing general purpose strategies on custom data structures. The implementation and results are compared against two other variants of parallel Haskell: Par monad, which offers an explicit way of controlling threads; and Eden, which provides process abstractions akin to lambda abstractions to define parallel computations. Compared to the other parallel Haskell dialects, GpH is minimally intrusive – the specification of the parallel execution (the coordination) is orthogonal, and separate from the specification of the computational code. GpH–Evaluation Strategies proved to be a powerful programming model, which can be exploited for data-oriented strategies. Most importantly, we began to develop a step-by-step approach to writing data-oriented strategies, which we will base on and further expand in the next chapter for tree-based data structures.

Chapter 4

Lazy Data-Oriented Evaluation Strategies

In this chapter, we extend our strategies development method from the previous chapter and apply it to tree data structures to achieve data-oriented parallelism. We present a number of flexible parallelism control mechanisms in the form of evaluation strategies for tree-like data structures implemented in Glasgow parallel Haskell. Additional flexibility is achieved by using laziness and circular programs in the coordination code. Heuristics-based parameter selection is employed to autotune the strategies for improved performance on a shared-memory machine without programmer-specified parameters. In particular, we demonstrate improved performance for unbalanced trees on a multi-core server.

4.1 Introduction

The previous chapter has covered evaluation strategies for list and the ease of specifying parallel operations on the flat data structure. Specifically, we looked at the definition of `parList` for element-wise parallelism and `parListChunk` for grouping computations to improve granularity and performance. Parallel sub-components are usually homogeneous and work well with a basic implementation. The existing strategies library also specifies generic strategies for traversable types. However, these achieve significantly worse performance on irregular input data (further discussed in Section 4.5), which is notoriously difficult to parallelise.

In the Data Parallel Haskell (DPH) extension (Jones et al., 2008), for example, flattening transformation techniques are used for nested arrays and other irregular structures to enable even partitioning and hence even distribution of work across processors. A similar transformation is used in the Manticore implementation of Parallel ML (Bergstrom et al., 2013) based on a "rope" representation for lists.

While this is efficient, it necessitates change to the compiler and base libraries, heavy use of arrays and intermediate data structures, and often employs mutable operations to achieve the best results. Our design goal is to achieve improved performance through more flexible management of the available parallelism, without having to modify the structure of the program or relying on compiler-driven source code transformations.

In our approach to parallelism we take a data-centric view and provide strategies as traversals over tree-like data structures. This offers the perspective of good re-use of such strategies for different applications, and a clean separation of computation from coordination, which was one of the main design goals for the existing evaluation strategies module.

4.2 Tree-Based Representation

The hierarchical structure of trees make them an ideal fit for parallel processing. Section 2.8.2 motives tree-based representation and their advantage over linear representation. The recursive definition of a tree data structure is analogous to a divide and conquer structure – each branch can be seen as a separate tree or sub-computation which can be evaluated independently.

Our motivation to develop advanced parallel tree evaluation strategies is due to their wide applications in many algorithms, including the Barnes-Hut algorithm seen in the previous chapter. Additionally, tree can be used as an alternative underlying representation for linear data structures. Trees offer more flexibility when it comes to their parallel processing. Below, we present an example of alternative underlying representation for the list data structure to enable intuitive parallel processing through natural decomposition of sub-computation.

Inherent sequential nature of list.

The representation of lists in Haskell (and in many other functional languages) is a sequential one. Implemented as a singly-linked list where elements are arranged in a linear order (as depicted in Figure 4.1(a)), operations on the list proceed in a sequential fashion. For instance, a fold traverses the list from left to right, and accumulates the result of applying a function to the current element and the accumulated value. Processing list in parallel always has to start from left to right.

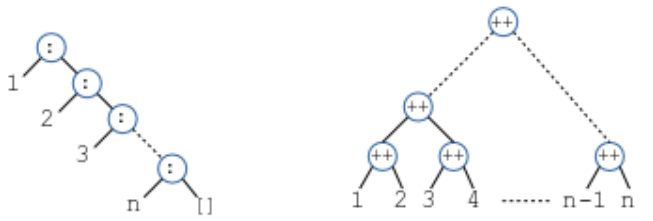
The following code example shows a parallel algorithm using `par` and `pseq` directly and requires explicit divide and combine functions on the list.

Listing 4.1: List explicit split for parallel evaluation.

```
-- explicit split
parListmap f []  = []
parListmap f [x] = [f x]
parListmap f xs  =
  let
    -- determine a pivot or chunk size
    midway = length xs `div` 2
    -- split list
    (xs1,xs2) = splitAt midway xs

    res1 = parListmap f xs1 -- in parallel
    res2 = parListmap f xs2 -- in parallel

    combine = (++)

  in res1 `par` res2 `pseq` combine res1 res2
```

Other design considerations – Prefer tree over linear representation.

Sequential data representation is bad in the context of parallel execution. A more effective policy for parallelism uses a tree. Changing the underlying representation of lists to trees, while keeping the same interface to the programmer, may lead to additional cost and overhead but this can be reduced over time, or by exploiting the new data layout for parallel evaluation.



(a) Sequential list representation

(b) Append tree representation

Figure 4.1: Alternative representation of list using append tree

Both program structures (e.g. loops) and data structures (e.g. linked lists) that generate a sequential computation tree that is detrimental to performance. Our approach is in a data structure context whereby the inherent sequential nature of list data structure, is changed to an append tree representation. Trees naturally exhibit a divide-and-conquer pattern (Cormen et al., 2009) – lending itself easily for parallelisation. Tree branches correspond to implicit independent components or partitions (as depicted in Figure 4.2) that can be evaluated in parallel. In addition, tree representation has the advantage of storing additional administration information in the inner nodes e.g. sub-tree sizes, which can be exploited during parallel evaluation.

```
-- no explicit split and combine required.
-- assuming tree is balanced, sub-trees are of equal size.
-- traversal results in implicit parallel generation.
```

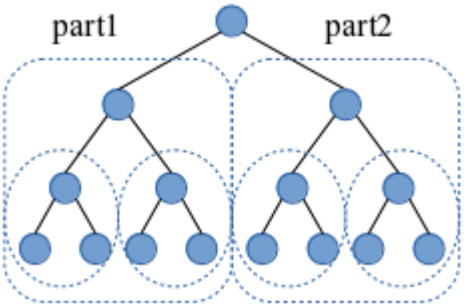


Figure 4.2: Tree implicit partitions

```
4
5  parTreemap f E          = E
6  parTreemap f (N x l r) = l' `par` r' `pseq` N (f x) l' r'
7    where
8      l' = parTreemap f l
9      r' = parTreemap f r
```

Figure 4.1 depicts the use of a (balanced) append tree as an alternative representation which allows traversal in O(log n) instead of O(n) time. Additionally, a tree naturally exhibits parallel "components" with no explicit splitting or partitioning needed. Obviously, this represents an ideal case only if that tree is balanced. To deal with irregular trees, we propose the use of self-balancing trees, e.g. red-black tree, in (Totoo, 2011), which also covers performance comparison of standard Haskell list against random-access list. The latter uses a tree-based representation.

4.3 Tree-Based Strategies Development

In the following sections, we develop a number of strategies for tree data structures: basic strategy with no parallelism control; and traditional thresholding mechanism using depth and sub-tree size as thresholds; and advanced strategies depending on lazy evaluation and fuel-based control.

Before we proceed, we revisit the development steps we put forward in Section 3.1.1 and expand on some steps for tree-based strategies below.

Revisted steps to writing tree-based strategies

1. New data type definition.

   Data-oriented strategies operate over data types. Previously we started with the built-in type `[a]`. Here we use a custom-defined tree type as example.

```
1  data Tree a = Node a (Tree a) (Tree a) | Leaf
```

A strategy defined over the data type describes how evaluation of the type and its element proceeds when its value is demanded. It defines both order (e.g. seq or par) and degree of evaluation.

2. Degree of evaluation possible on defined type. Typically, at one end, a strategy may perform no evaluation at all (`r0`), and, at the other end, may completely evaluate (`rdeepseq`) the type to normal form. Anything in between is called WHNF. We may define several intermediate evaluation degrees for different data types. For example, in the case of a tree, we may have `evalLeftBranch`, `evalRightBranch`, and `evalUntilNLevel`, which evaluates the left branch, the right branch, and up to depth N of the tree, respectively.

   For any new type, we need to implement an instance of the `NFData` type class if we want to be able to evaluate the type to normal form.

```
instance NFData (Tree a) where
  rnf Leaf         = ()
  rnf (Node x l r) = rnf a `seq` rnf l `seq` rnf r
```

3. Generalise type definition to accommodate wider application. This allows one to re-use strategies of a generic type in various scenarios. For example, a tree definition in Step 1 can be generalised to:

```
data Tree a = Node a (k (Tree a))
data Pair a = P a a
type BinTree a = Tree Pair a
```

4. Generalise type operations. Implement type instances of generic type classes, such as `Traversable`, `Foldable`, `Functor`, in order to define a default operation for each. That is, a way to traverse the type, perform a fold operation on it, and map a function over its elements.

5. Exploit strategy composition. Separate different evaluation strategies and compose appropriately. For e.g. `evalList` takes a strategy to specify the order of evaluation, and another for the degree of evaluation. Other example we will see is parCluster for separating list clustering (decomposition) from parallel processing.

6. Encode administrative information in type definition. This may be useful for parallel evaluation. For instance, for trees, size annotations help determine if it is useful and cost-effective to evaluate sub-tree in parallel.

7. Tune strategy by improving granularity and selecting correct parameters.

4.4 Tree Data Type

One of our main design goals is to develop data-centric parallelism control mechanisms that can be applied across a range of commonly used data structures and that are not tied to a particular application. Our target are quad-trees – used in two non-trivial problems later – but our formulation is general enough to cover rose trees, and the strategies should not be limited to just this data type.

Two important decisions in defining the tree data structure are the arity of the nodes and the value attribution (to nodes or leaves). In order to remain flexible, we parameterise our definition over both aspects, and arrive at the following generic definition of a k-ary tree (Cormen et al., 2009):

Definition 4.1 (k-ary tree)**.** *A k-ary tree is a rooted tree in which each node has at most k children.*

A tree of the form

```
1 data Tree k tl tn = E | L tl
2                   | N tn (k (Tree k tl tn))
3 data Quad a       = Q a a a a
4 type QTree tl tn  = Tree Quad tl tn
```

with data elements of type `tl` in leaf nodes and data elements of type `tn` in the inner nodes is called a k-ary tree.

The number of sub-trees is parameterised by introducing a type variable k to specify the sub-tree container. Using a 4-tuple, defined as `Quad`, gives the well-known quad-tree data structure, which we will use in the Barnes-Hut simulation in Section 4.12. Other common choices for the container argument are: a 2-tuple, defined as `Bin`, for a binary tree; an 8-tuple, defined as `Oct`, for oct-tree (used for 3-dimensional nbody simulation); and a list, written as `[]`, for a rose tree with potentially varying arities in different nodes. We focus on defining strategies on this k-ary tree data structure in order to enable parallelism over the data structure, without fixing the amount of parallelism or tying the evaluation to one class of architectures.

4.5 T1: Unconstrained `parTree` Strategy

The basic `parTree` strategy creates a spark for every element in the tree. Depending on whether the tree is node- or leaf-valued, the variants `parTreeL` and `parTreeN` will spark just leaf or node elements, respectively. In our discussion we focus on the most generic version, `parTree`.

The implementation of `parTree` in Listing 4.2, uses the `Traversable` class to arrange the traversal in a way that is not restricted to a tree data structure. Furthermore, the definition of `parTree` demonstrates that we can compose more complex strategies from simpler ones: we pass `(rpar ‘dot‘ strat)` as argument to the sequential `evalTree` strategy, specifying that each element should be evaluated in parallel (`rpar` – strategic equivalent of `par` primitive), using the parameter `strat` to specify the evaluation degree. This compositionality is inherited from the design of evaluation strategies as described in (Marlow et al., 2010).

Listing 4.2: Data element sparking

```
1 evalTree :: (Traversable k) =>
2             Strategy a -> Strategy (Tree k a a)
3 evalTree = traverse
4
5 -- parallel evaluation of inner node and leaf values
6 parTree  :: (Traversable k) =>
7             Strategy a -> Strategy (Tree k a a)
8 parTree strat = evalTree (rpar ‘dot‘ strat)
```

Note that `parTree` does not attempt to control, or throttle, spark creation. Thus, if the tree is large, this results in an abundance of parallelism, which can be detrimental to its performance due to the excessive overhead. Additionally, the current definition applies the same strategy to both inner and leaf nodes. This can be changed by adding a new parameter to the definition specifying different strategies for the two types of node, for example, `Strategy a -> Strategy b -> Strategy (Tree k a b)`, and tweaking the traversal function definition.

4.6 Parallelism Control Mechanisms

Uncontrolled parallelism creates overheads through generation of excessive sparks in GpH – many of which, if converted, will carry the usual thread management cost, and many will be overflown and never taken to execution. This negatively affects parallel performance. In this section, we highlight the techniques and mechanisms used to throttle the amount of parallelism in the evaluation of tree data structure. Subsequent sections look at the implementation details and issues.

Node-level sparking: In order to control granularity, we want to spark branches or sub-trees of appropriate size instead of individual tree elements. This is particularly useful for very large tree data structures where the overhead of element-wise sparking exceeds the performance gain expected from parallelisation. The concept is analogous to chunking in the context of list data structures to ensure sufficient amount of work for each thread, but identifying which branches are of adequate size

is tricky. Next we address these issues where we use a number of dynamic techniques to throttle the amount of parallelism that is generated.

Listing 4.3: Node-level or branch sparking

```
1
2  parTreeBranch :: Strategy (Tree Quad tl tn)
3                -> Strategy (Tree Quad tl tn)
4  parTreeBranch strat (N n (Q nw ne sw se)) =
5      (N n <$> (Q <$> parTreeBranch strat nw
6                  <*> parTreeBranch strat ne
7                  <*> parTreeBranch strat sw
8                  <*> parTreeBranch strat se))
9      >>= (rpar `dot` strat)
10 parTreeBranch _ (L x)  = pure $ L x
11 parTreeBranch _ E      = pure E
```

In Listing 4.3, sparks are created to evaluate branches in parallel. In this version, no restriction is placed yet, so all branches (i.e. inner nodes) are sparked. Therefore, for quad-trees, a branch will compute between 1 and 4 elements in parallel, one for each non-empty sub-tree. Note that the strategy function type definition is changed as the first argument is applied to a branch, i.e., of a `Tree` type. Note also that for this implementation we need to specify `k` is a `Quad` so pattern matching can be done, unlike the more generic implementation of `parTree` and variants. In this example, we write the first component of the first pattern match compactly using applicative style as previously seen and the explicit bind operator (`>>=`) at the outer level to compose the two actions with the specified strategy, for parallel evaluation of the second action in this case.

Thresholding: A common technique to throttle parallelism is depth-based thresholding. This involves specifying an additional parameter d used to limit sparks creation to the top d levels of a tree. This is most effective for a regular tree layout and small d as the number of sparks increases exponentially at each level.

Depth-based thresholding works well under the assumption that the major source of parallelism occurs within depth d in the tree. Spark creation is controlled, but still statically determined. The mechanism to control the amount of parallelism is fairly crude, since the number of sparks is exponential in the depth of the tree.

Alternatively, size-based thresholding checks sub-nodes for a minimum size s before sparking. Parallelism generation may go deeper inside the tree. However, the size information needs to be readily encoded in the inner nodes. Depending on the problem, this information might be already available, otherwise, a first pass to annotate the tree is required.

Fuel splitting: We use the notion of fuel – a limited, explicit resource – to control parallelism in a more flexible way. We use a fuel splitting technique to distribute resources to sub-computations to throttle parallelism. Our implementation of fuel splitting offers the flexibility of defining custom functions specifying how fuel is distributed among sub-nodes, thus influencing which path in the tree will benefit most of the parallel evaluation. This technique is related to the use of engines in Scheme 84 as a notion of timed preemption for processes (Haynes and Friedman, 1984). An engine is given a quantity of fuel and computation lasts until fuel runs out. In our context, parallelism generation, rather than evaluation, is based on fuel being available for a particular sub-tree.

Bi-directional fuel transfer: To enhance flexibility in splitting and transferring fuel, a bi-directional mechanism of transfer is advantageous. This way, fuel that is unused in one sub-tree, can be used in another sub-tree. To implement this behaviour, we need lazier representation of numbers, for example, implementing Peano number sequence through the use of lists of unit type for fuel instead of an integer type. This enables us to check sub-node bounds, for example, if it has at least n elements. This does not force the entire sub-node to normal form, i.e. full evaluation, before returning true or false.

One instance of strategy definition relies on circular program definition (as mentioned in Section 2.6) enabled only in a lazy language (Allison, 1989). Traditionally used to improve sequential performance by eliminating multiple traversals of a data structure, here it used for the first time for parallel programming. Specifically, we use it in a fuel distribution function with a giveback technique, i.e. unused fuel in any sub-node is pushed back up in the tree to be re-distributed elsewhere. We discuss this technique in detail in Section 4.10.

Abstraction: By defining splitting functions separate from the strategy definition, we can parameterise strategy to custom-defined split functions. This employs a similar concept as used for clustering strategies in the original library.

Annotations: Size (if not readily encoded) and fuel information need to be attached to the tree in an annotation run. Therefore, some strategies are defined over an annotated tree type (`AnnQTree`). The type constructor k in the `Tree` type in Definition 4.1 allows to parameterise over an annotated container, e.g. annotated pair (`AnnPair`) or quad (`AnnQuad`), as opposed to adding another field in its definition.

Listing 4.4: Annotated tree type

```
1 data Ann a = A a
```

```
-- an annotated quad type
data AnnQuad ta a = AQ (Ann ta) a a a a
-- annotated quadtree definitions
type AnnQTree ta tl tn = Tree (AnnQuad ta) tl tn
```

Size annotations are synthesised bottom-up, making sure that the tree is annotated in a single pass. Fuel annotations depend on the distribution function and are propagated top-down. For this reason, the context for the fuel-based strategies differ. Some require size information, for example, perfect split, and others require only local lookup. Depending on how the splitting works, the annotation function will assign an amount of fuel to each node, until fuel runs out. Generic annotation implementations, for example, as used for the AST for Hume space analysis (Jost et al., 2010), or the attribute-grammar (Swierstra et al., 1999) style have been investigated. At present, we settle for a simple annotation function, with a parameterised split function for fuel distribution.

Heuristics: The advanced strategies are parameterised by additional variables to specify the depth d, size s, and fuel f thresholds. These can be programmer-specified or established through a heuristics-based parameter selection determined by a number of other parameters, for instance, input size and number of PEs. The latter ensures that available information are used to tune the strategies.

Table 4.1 gives an overview of the strategies that we develop, classifying them by some basic properties of their dynamic behaviour. The information flow column indicates whether administrative information for controlling parallelism is passed down or up. For example, depth-based thresholding passes a depth counter down the tree. Fuel-based versions can also pass information up, if resources have been unused. The context column indicates how much of context information is required in a node to implement this strategy. For example, depth-based thresholding requires only information about the length of the path to the current node, whereas a lookahead strategy examines a fixed number of nodes in the sub-trees to make its decision. In the extreme, a perfect-split strategy requires complete (global) size information about the tree, and therefore incurs the highest amount of overhead. The final two columns specify the parameters that are used in the various strategies to control their behaviour and whether the parameter can be auto-specified by heuristics in our implementation. We will elaborate on these aspects in the following sections.

	Strategy	Type	Info flow	Context	Param	Heuristics
T1	parTree	element-wise sparks	-	-	-	-
T2	parTreeDepth	depth threshold	down	path length	d	yes
T3	parTreeSizeAnn	annotation	up	global	-	-
T4	parTreeLazySize	lazy size check	down (lazy)	local	s	yes
T5	parTreeFuelAnn	fuel with annotation			f	yes
T51	- pure	equal fuel distr	down	local		
T52	- lookahead	check next n nodes	down/limited	N	N	
T53	- giveback	circular fuel distr	up/down (lazy)	local		
T54	- perfectsplit	perfect fuel distr	down	global		

Table 4.1: Strategies overview and classification

4.7 T2: Depth-Thresholding (`parTreeDepth`)

The simple `parTreeDepth` strategy, shown in Listing 4.5, introduces some degree of control of spark creation by using depth as a threshold for parallelism generation. Efficient for throttling parallelism on regular trees, this technique limits sparking of sub-nodes to evaluate in parallel at the top d levels in the tree. The strategy is a recursively-defined function which stops generating sparks when depth 0 is reached. It is useful to have sparks as early as possible, which is why sparks are created from the root node to the specified level d.

Listing 4.5: Depth-based thresholding

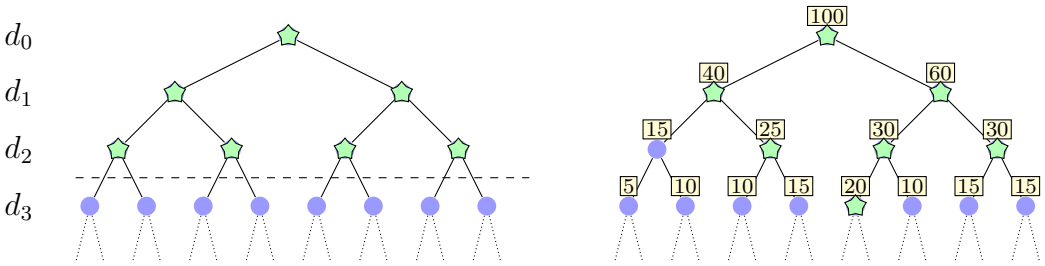
```
parTreeDepth::Int->Strategy (QTree t1)
                 ->Strategy (QTree t1)
parTreeDepth 0 _     t = return t
parTreeDepth d strat (N (Q nw ne sw se))  =
    (N <$> (Q <$> parTreeDepth (d-1) strat nw
              <*> parTreeDepth (d-1) strat ne
              <*> parTreeDepth (d-1) strat sw
              <*> parTreeDepth (d-1) strat se))
    >>= rparWith strat
parTreeDepth _ _     t = return t
```

The main advantages of this strategy are the simplicity of its implementation, low overhead, and predictable parallelism. However, it lacks flexibility in particular for unbalanced trees, where potentially useful parallelism may reside outside of the given depth threshold. Though we gain improved control over parallelism as opposed to element-wise sparking, the amount of sparks usually remains flat as an upper bound for d is specified to avoid excessive sparks. For instance, for a regular tree, d_{max} can be set at 6 to generate a maximum of 4^6 sparks. In Section 4.11, we look at ways to automatically select the depth threshold d to gain optimal performance.

4.8 T3: Synthesised Size Info (parTreeSizeAnn)

Depth-thresholding imposes a horizontal limit in the depth of the tree, not taking into account the concentration of elements in specific sub-trees, for parallelism generation. Using sub-tree size information and a size threshold s, sparks can be restricted only for sub-trees of at least s elements, that is, sub-trees deemed to have sufficient work to be sparked in parallel. Smaller sub-trees not meeting the size threshold are not sparked, thus avoiding unnecessary spark pointers that may overflow the pool.

Data structures that keep track of "region" size provide useful information for processing. If size information is not encoded e.g. in the inner-nodes of a tree, an initial traversal is needed to annotate the tree. In our implementation, size information is synthesised from the bottom up, making sure the size annotations are attached in a single traversal. Figure 4.3 is a visual depiction of the difference between depth and size-based thresholding for a tree structure. A star denotes a spark.



(a) $d < 3$ (b) $s < 20$

Figure 4.3: Depth vs (strict) size thresholding

4.9 T4: Lazy Size Check (parTreeLazySize)

Often we do not need to perform a full top-level deconstruction of a data structure in order to establish a bound. As a simple example, the standard list *length* function is implemented using atomic integer which requires a complete traversal of the spine of the list to return a value. Listing 4.6 shows an example of how we can reduce the amount of evaluation by using natural numbers, especially if we require a partial traversal of a data structure to find out a minimum length. Listing 4.7 shows that the lazier length function is faster for every large and potentially unbounded data structure (0.46 sec vs 8.02 sec to return).

Listing 4.6: Defining a lazier list length function using Natural numbers

```
1 -- standard length
2 len::[a]->Int
3 len (_:xs) = 1+len xs
```

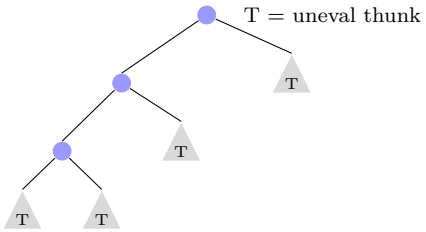


Figure 4.4: Lazily de-constructing subtrees and establishing node size ($s = 3$)

```
len []     = 0

-- lazier variant
lazylen::[a]->Natural
lazylen (_:xs) = succ (lazylen xs)
lazylen []     = zero
```

Listing 4.7: `len` vs `lazylen` in GHCi

```
> let xs = [1..10000000] -- large list
> :p xs                  -- print value without forcing its computation
xs = (_t::[Int])         -- initially xs is an unevaluated thunk

-- standard length
> len xs > 1000000
True
(8.02 secs, 1943155832 bytes)
> :p xs
xs = [(_t::Int),(_t::Int),(_t::Int), ... ,(_t::Int)]
     -- forces complete top-level cons (:) de-construction

-- lazy length
> let ys = [1..10000000]
> lazylen ys > 1000000
True
(0.46 secs, 200919400 bytes) -- NB: lazier version is faster!
> :p ys
ys = (_t::Int) : (_t::Int) : (_t::Int) : ... : (_t::[Int])
     -- only de-construct first (1000000+1) cons to establish True
```

Applying the lazy size check technique to define a lazy tree strategy. The size threshold strategy depends on a full initial traversal to attach size information to inner-nodes. This can be eliminated by implementing a *lazier* size checking function to establish size bounds without a full deconstruction of the tree. The basic idea remains the same – spark sub-trees with at least s elements. However, the size check function is implemented using an algebraic natural instead of an atomic integer type to perform lazy size computation without forcing complete evaluation. The function `isBoundedSize` in Listing 4.8 returns true when it has established that the sub-tree contains at least `s` nodes, without traversing the rest of the tree.

Listing 4.8: Lazy size check using Natural number representation

```
isBoundedSize::Natural -> QTree tl tn -> Bool
isBoundedSize s t = numLeafNodes_lazy t > s
```

```
3
4  -- | Lazy leaf node count.
5  numLeafNodes_lazy::QTree tl tn -> Natural
6  numLeafNodes_lazy (N _ (Q nw ne sw se)) =
7     numLeafNodes_lazy nw + numLeafNodes_lazy ne
8     + numLeafNodes_lazy sw + numLeafNodes_lazy se
9  numLeafNodes_lazy (L _)  = succ zero
10 numLeafNodes_lazy _      = zero
```

4.10 T5: Fuel-Based Control (`parTreeFuel`)

Fuel-based strategies are defined over an annotated tree type. In terms of implementation, this is similar to `parTreeDepth` where instead of a depth threshold, the strategy stops creating sparks once fuel runs out. This is seen in the pattern match for the fuel check in Listing 4.9.

Listing 4.9: Fuel strategy function

```
1  parTreeFuel :: Strategy (AnnQTree Fuel tl)
2              -> Strategy (AnnQTree Fuel tl)
3  parTreeFuel strat t@(N (AQ (A f) nw ne sw se))
4    | f>minfuel = (N <$> ( AQ (A f)
5                        <$> parTreeFuel strat nw
6                        <*> parTreeFuel strat ne
7                        <*> parTreeFuel strat sw
8                        <*> parTreeFuel strat se))
9                  >>= rparWith strat
10   | otherwise = return t
11 parTreeFuel _     t = return t
```

A variant of this definition (which we append with `marked`) sparks only when the condition `f>minfuel` is met. With this variant the total number of sparks generated is not cumulative from the root, but only at the specific (or marked) nodes within the tree that pass the lazy size bound test. Thus, the spark number closely corresponds to the amount of fuel distributed, assuming we allocate one unit of fuel per eligible node. This allows to have a better estimation on spark creation for a given amount of fuel. However, spark creation is delayed, which may not be desirable in cases where there are not enough computations to execute in parallel. However, our performance results show that this variant (`fuelpuremarked`) performs well compared to other fuel-based strategies in our test applications.

Annotation-based strategies, such as `parTreeFuel`, require an annotation pass before, and an unannotation pass after, the strategy application. The annotation step incurs a small cost which is reported later. Figure 4.5 and Listing 4.10 show the sequence of functions composition. Fuel annotation is parameterised by the amount of fuel, determined through heuristics which takes a number of variables into account, and a fuel splitting function.

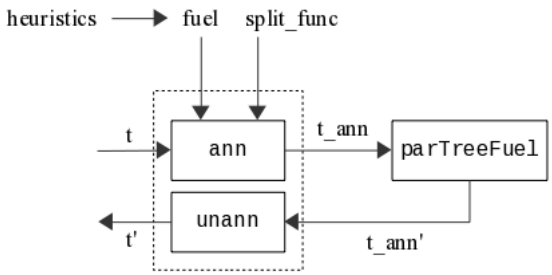


Figure 4.5: Use of annotation-based strategies.

4.10.1 Fuel Splitting Methods

The general usage of the fuel-based function is as follows, with the specification of the amount of fuel and a distribution function of type `SplitFunc`:

Listing 4.10: Annotation and strategy application function composition.

```
t' = (unann
      . withStrategy strat
      . fmap f
      . ann) t

-- pure fuel annotation example
ann = annFuel (annFuel_pure fuel)
```

A split function can be generalised with the type `SplitFunc`, then the actual implementation is parameterisable, to switch between different distribution modes.

Listing 4.11: Recursive fuel splitting and distribution among sub-trees.

```
type SplitFunc = Fuel->[Fuel]

annFuel::SplitFunc->Fuel->QTree tl
                   ->AnnQTree Fuel tl
annFuel splitfunc _    E               = E
annFuel splitfunc _    (L x)           = (L x)
annFuel splitfunc fuel (N (Q nw ne sw se)) =
    let (f1:f2:f3:f4:_) = splitfunc fuel
    in N $ AQ (A fuel) (annFuel splitfunc f1 nw)
                       (annFuel splitfunc f2 ne)
                       (annFuel splitfunc f3 sw)
                       (annFuel splitfunc f4 se)

annFuel_pure::Fuel->QTree tl->AnnQTree Fuel tl
annFuel_pure = annFuel (fuelsplit_pure _numSubnodes)
```

The following gives implementation details of four distribution methods: *pure*, *lookahead*, *giveback* and *perfectsplit*.

T51: Pure fuel distribution splits fuel evenly among the sub-nodes, ignoring the node type. Fuel is lost on hitting outer nodes (empty and leaf nodes), and on division. A version that avoids any such loss has been implemented, but does not perform significantly better and is therefore not discussed any further.

Listing 4.12: Pure fuel distribution function.

```
type Fuel=Int   -- fuel as int

fuelsplit_pure::Int->Fuel->[Fuel]
fuelsplit_pure numnodes fuel =
    replicate numnodes (fuel `div` numnodes)
```

T52: Lookahead/LookaheadN fuel distribution method, as the name suggests, looks ahead one level down the tree before distributing unneeded fuel to outer nodes. In the second variant, *lookaheadN* (defined in Listing 4.13), we can specify a parameter N on how far down the tree we can look, thus having a better idea of nodes distribution in a particular branch, leading to better fuel distribution.

Listing 4.13: Lookahead fuel distribution function.

```
fuelsplit_lookaheadN::Int->QTree tl
                         ->Fuel->[Fuel]
fuelsplit_lookaheadN n (N (Q nw ne sw se)) fuel =
   [f1,f2,f3,f4]
 where
  Q nw' ne' sw' se' = fmap (numInnerNodesUntil n)
                       (Q nw ne sw se)
  numsubnodes       = nw' + ne' + sw' + se'
  (f1:f2:f3:f4:_)   = fuelsplit_perfect fuel
                       numsubnodes [nw',ne',sw',se']
fuelsplit_lookaheadN _ _ _ = [0,0,0,0]
```

T53: Giveback fuel distribution is based on the same idea as lookahead distribution which avoids fuel loss on meeting outer nodes. This is, however, achieved differently. Instead of looking ahead down N levels of the tree, giveback employs a circular programming technique to allow passing fuel up the tree, if it is unused in a sub-tree. Thus, information flow is bi-directional in this implementation as shown in Figure 4.6. This technique depends on laziness to enable circular reference.

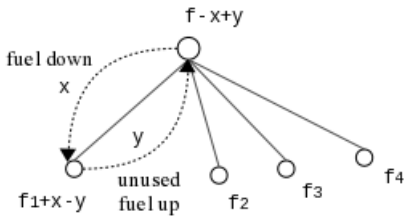


Figure 4.6: Giveback fuel mechanism.

In the implementation, the giveback mechanism uses a list as an administrative data structure to represent units of fuel instead of an atomic integer type to work with the circular nature of its definition.

Figure 4.7(c) depicts the bi-directional fuel flow in a giveback strategy. We note that unused fuel is passed to the next node on the right, and, if it is still unused, is passed up in the tree to be re-used in another, typically deeper, sub-tree. This behaviour is achieved by using a circular definition (Bird, 1984). The input to the

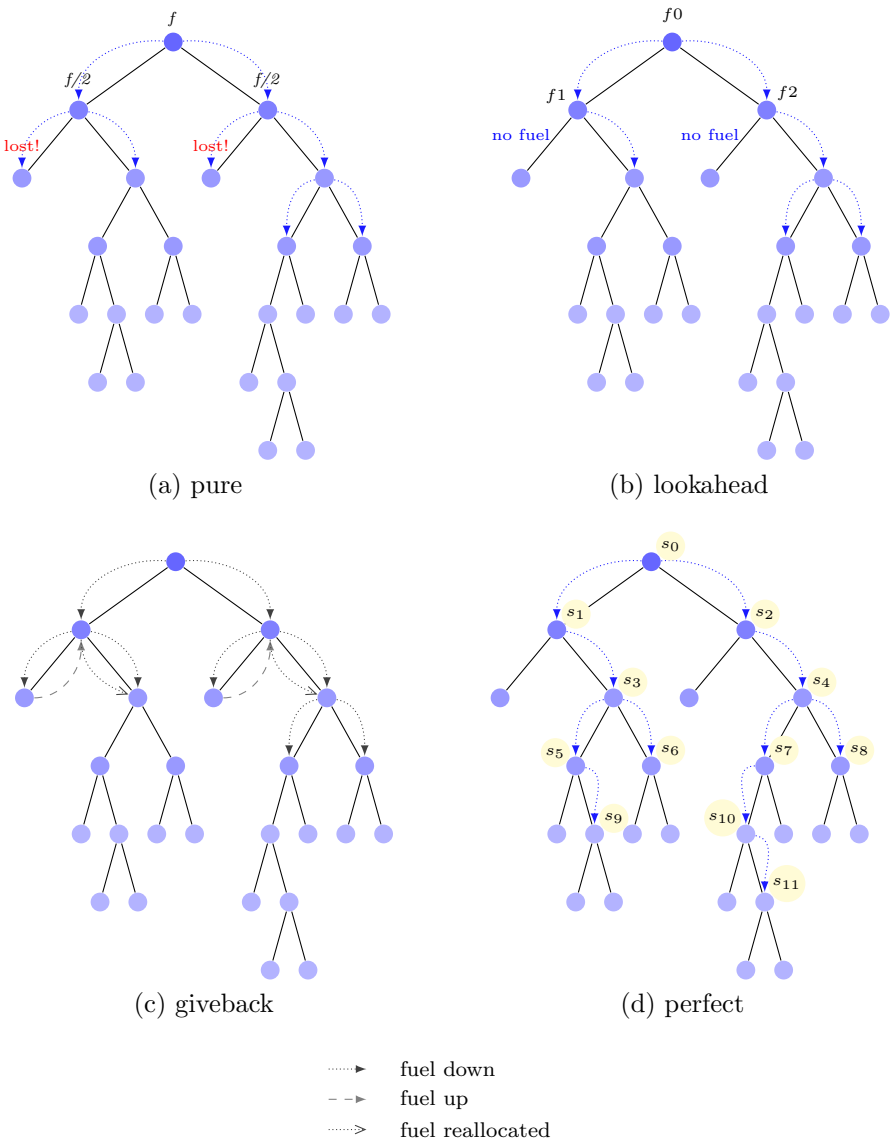


Figure 4.7: Fuel flow with different distribution function.

annotation function (`ann`) takes an initial share of the fuel and any fuel that is returned from the left. In the code of Listing 4.14, the definition `f1_out` (Line 14) depends on `f4_out`, which, in three steps, depends again on `f1_out` (Line 15). In order to guarantee that this definition is productive, fuel must not be represented as an (atomic) integer, but needs to be a list of values, which is expanded by this circular definition and requires lazy evaluation. Note that this strategy only requires a local context to distribute fuel (without wasting fuel in leaves) and thus should incur less overhead.

Listing 4.14: Fuel with giveback annotation

```
1 -- | Fuel with giveback annotation
2 annFuel_giveback::Fuel->QTree tl
3                       ->AnnQTree Fuel tl
4 annFuel_giveback f t = fst $ ann (fuelL f) t
5  where
6   ann::FuelL->QTree tl->(AnnQTree Fuel tl,FuelL)
```

```
  ann f_in E                    = (E,f_in)
  ann f_in (L x)                = (L x,f_in)
  ann f_in (N (Q nw ne sw se)) =
   (N (AQ (A (length f_in)) nw' ne' sw' se'),emptyFuelL)
   where
    (f1_in:f2_in:f3_in:f4_in:_) =
      fuelsplit_unitlist _numSubnodes f_in
    (nw',f1_out) = ann (f1_in++f4_out) nw
    (ne',f2_out) = ann (f2_in++f1_out) ne
    (sw',f3_out) = ann (f3_in++f2_out) sw
    (se',f4_out) = ann (f4_in++f3_out) se
```

Listing 4.15: Auxiliary functions for giveback fuel distribution.

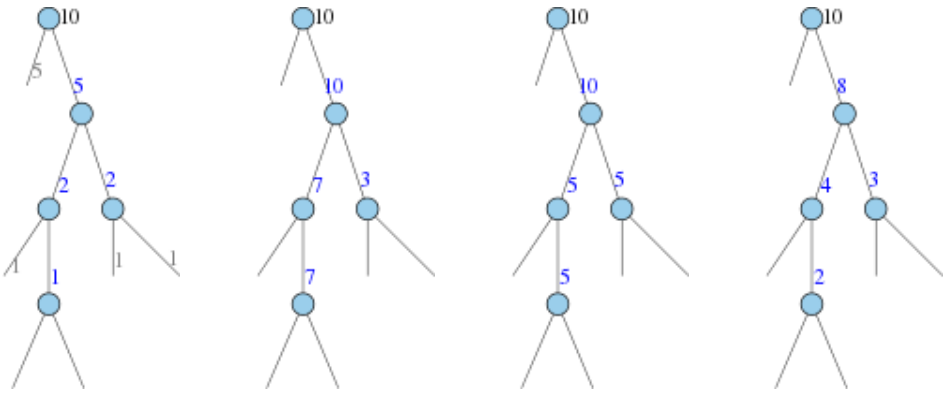
```
type FuelL=[()] -- fuel as unit list

emptyFuelL=[]   -- empty fuel list

fuelL::Fuel->FuelL
fuelL x = replicate x ()

fuelsplit_unitlist::Int->FuelL->[FuelL]
fuelsplit_unitlist numnodes fuel =
 split 0 fuel [emptyFuelL,emptyFuelL,
               emptyFuelL,emptyFuelL]
 where
  split _ []     xs = xs
  split x (_:fs) xs =
    split (x+1) fs (addfuelat (x `mod` numnodes) xs)

  addfuelat 0 [a,b,c,d] = [():a,b,c,d]
  addfuelat 1 [a,b,c,d] = [a,():b,c,d]
  addfuelat 2 [a,b,c,d] = [a,b,():c,d]
  addfuelat 3 [a,b,c,d] = [a,b,c,():d]
  addfuelat _ xs        = xs
```

T54: Perfect fuel splitting distributes fuel based on sub-node sizes. It depends on the size information being available, otherwise an annotation run, requiring a full traversal of the tree, is needed before any parallel sub-computation (spark) is generated.

Listing 4.16: Perfect fuel splitting.

```
fuelsplit_perfect::Fuel     -- amount of fuel
                  ->Size    -- parent node size
                  ->[Size]  -- children node sizes
                  ->[Fuel]  -- fuel split among children
fuelsplit_perfect fuel s ss =
  fmap (\x -> (x*fuel) `div` s) ss
```

Figure 4.7 depicts the fuel flow inside the tree shown by the arrow direction. Figure 4.8 provides an actual distribution example of fuel using the different splitting methods on a small binary tree. In the simple pure distribution, fuel is lost (shown in grey) on empty nodes. The lookahead distribution avoids this loss (see Fig. 4.8(b)),



(a) pure (b) lookahead (c) giveback (d) perfect

Figure 4.8: Fuel distribution example on a binary tree ($f = 10$)

so does the giveback version. Though perfect-splitting seems to be more precise, it needs a global context.

4.11 Heuristics

The strategies require parameters to specify the depth, size and fuel. Choosing the right strategy parameter for a given input size, tree layout, and other variables, is crucial for optimal performance. While this can be programmer-specified, it is difficult to deduce the right values for best performance given the number of variables involved. Table 4.2 lists a number of variables that are used in determining an optimum parameter for the strategies during execution.

Variable name	Description
T_{in}	Number of inner nodes in tree T
T_{out}	Number of outer nodes
T_l	Number of non-empty (leaf) outer nodes
T_e	Number of empty outer nodes i.e. $T_{out} - T_l$
H_{min}	Shortest path from root to any outer node
H_{max}	Longest path from root to any outer node
S	Number of sparks generated by a strategy
S_{max}	Maximum number of sparks
P	Number of processing elements (PEs)

Table 4.2: Heuristic parameters

The number of sparks S generated during execution roughly corresponds to the "amount" of parallelism desired. The granularity of these parallel computations will vary. S_{max} refers to an upper bound that we may set in order not to have excessive sparks created by a strategy, which would otherwise cause a higher overhead. In practice, we set this to about 8000 sparks which is the maximum number of sparks

that can be fitted in the spark pool at one time. However, S_{max} can be set higher, given that sparks are created and converted during execution, and the total number of sparks that may have been created at the end of execution could be greater than 8000, but the spark pool was never overflowed. This depends on the sequence in which parallel computations are specified and happen in the algorithm.

The programmer-specified parameters are d, s and f, for the depth-threshold, lazy size and fuel-based strategies, respectively. For good performance it is important that these parameters fulfill basic properties on the tree structure. For instance, the selected value of a parameter may be out of range for a given tree. In the following sub-sections, we lay down these properties — what we expect from each strategy — and elaborate how the heuristics preserve them.

The heuristics work based on the assumption of how much information about the tree is available. In some cases, a traversal to extract this information is justifiable if the computation involved in the nodes is substantial enough. For instance, one version of the depth heuristics (**D2**) works well if the number of nodes at each level is known.

4.11.1 Determining Depth Threshold d

The heuristics should guarantee and satisfy the following properties:

Invariant for d

1. d should be within the range

$$0 < d < H_{max} \wedge d \leq d_{max}$$

   d cannot be outside the depth bounds of a tree. d_{max} is a maximum set by the programmer (hard-coded) or estimated in the more advanced heuristic functions.

2. For any selected d, $S < T_{in}$

3. For any selected d, $S < S_{max}$

   The number of sparks generated for a given d should not exceed the maximum spark limit. S_{max} is the user defined sparks cutoff point.

Below we define some of the heuristics used to determine d.

Level	Nodes at Level	Cumu. Nodes (=Sparks)
0	1	1
1	4	5
2	16	21
3	64	85
4	256	341
5	1024	1365
6	**4096**	**5461**
7	16384	21845
8	65536	87381

Table 4.3: Number of nodes at each depth for a complete tree.

D0 $d = H_{max}/2$

This assumes that the major sources of parallelism, that is, most compute-intensive branches, reside within the upper half of the tree. This version does not take P into account and generates a fixed number of sparks for any P.

D1 $d = min\ (P-1)\ d_{max}$

where d_{max} in this version is hard-coded. The depth parameter is selected based on the maximal depth of the input tree.

The heuristic is implemented through a counter that starts with $d = 0$ on 1 PE and increases d by one from 2 PEs onward until d_{max} is met. This leads to at most $\sum_{x=0}^{P-1} 4^x$ sparks on P PEs. There is no maximum sparks control (Property 3) in the version.

D2 $d = min\ (P-1)\ d_{max}$, where the number of cumulative nodes at d is less than S_{max}

In this version, d_{max} is computed (not specified) to enforce the "where" condition. d_{max} is dependent on the input size and is determined by building a table consisting of number of nodes and cumulative number of nodes at each level. d_{max} is the level at which the cumulative number of nodes is just before S_{max}. From Table 4.3, $d_{max} = 6$ for $S_{max} = 8000$. Note that $d_{max} < H_{max}$.

D2 is based on information obtained from an initial traversal of the tree. However, where not possible, we work on an estimate. For any tree, there are at most (upper bound) 4^i nodes at level i, and the cumulative nodes at this level is $\sum_{x=0}^{i} 4^x$. For instance, the upper bound for the number of nodes at level 5 is 1024, and the number of cumulative nodes is 1365. The cumulative nodes corresponds to the upper bound of sparks that is to be created at level. We work within these bounds to determine a maximum d. Any d greater than this will generate sparks in excess. For instance, at level 6, at most 5461 sparks are created, and at level 7, 21845 sparks.

4.11.2 Determining Size Threshold s

At present we use the same heuristics to determine s for both `parTreeSizeAnn` and `parTreeLazySize`. The choice of an s can limit or create more parallelism. Small s will create more sparks, while big s will create less. When P increases, we want to be able to have more sparks, thus smaller s.

Invariant for s

1. s should be within the range $0 < s < T_{in}$

The number of sparks created for the size annotation and lazy size strategies is directly related to s. If s is big, fewer sparks are likely to be created. s is seen as the minimum size threshold for `parTreeSizeAnn`. Sparks are created until a sub-tree size is less than s, which translates to the amount of computation in that node is small given its size.

Used in a slightly different "context" with the `parTreeLazySize` strategy, s refers to the minimum number of nodes check, performed lazily by the strategy in order to decide whether to create a spark or not.

S0

$$s = \begin{cases} \frac{T_l}{(P \times X)} & \text{if } P > 1 \\ 0 & \text{otherwise} \end{cases}$$

where X is an approximate number of sparks per PE.

We use an estimate function to classify computation in nodes as S (small), M (medium), and L (large), and based on this "weight" we determine X. For instance, for a small amount of computation, it is fine to have many sparks per core (for example, 100–200), but for a large amount of computation, sparks are restricted to ca 5–10.

4.11.3 Determining Fuel f

As the number of processors (P) increases, we want to provide more fuel such that more sparks can be created. f_{min} is the minimum amount of fuel that is needed for a spark.

Properties for f: In principle, f should be within the range $0 < f < T_{in}$ (same as s) for a particular sub-tree, to effectively control the potential parallelism. If $f > T_{in}$, all nodes have at least one unit of fuel. Thus, the default fuel check of at least 1, $f > 0$, changes, for example, to $f > 5$ in order to ensure the mechanism works. Otherwise sparks are created at each inner node in the tree, defaulting to the naive `parTreeBranch` strategy. Thus, we require these properties:

1. if $f < T_{in}$, then $f_{min} < 0$
   This means that fuel will run out during distribution, and thus nodes with 0 fuel (or negative) will not be sparked.
2. if $f > T_{in}$, then $f_{min} > 0$
   This means that potentially all nodes will have some fuel, and we need to determine f_{min}, that is, the new minimum fuel threshold a node needs to have to be eligible for a spark.
3. f should be less than sparks upper bound ($f < S_{max}$)

In giveback fuel distribution, the fuel coming into a node from its parent is the sum of fuel passed down to children (plus giveback fuel). If the size of a sub-tree is smaller than the fuel it gets, there will be some giveback fuel. If the size of the tree is larger than the fuel, there will be as many sparks as prescribed by the fuel, irrespective of the shape of the tree.

We explore the following heuristic formulae for computing the fuel.

F0 The initial heuristic is designed to be simple:

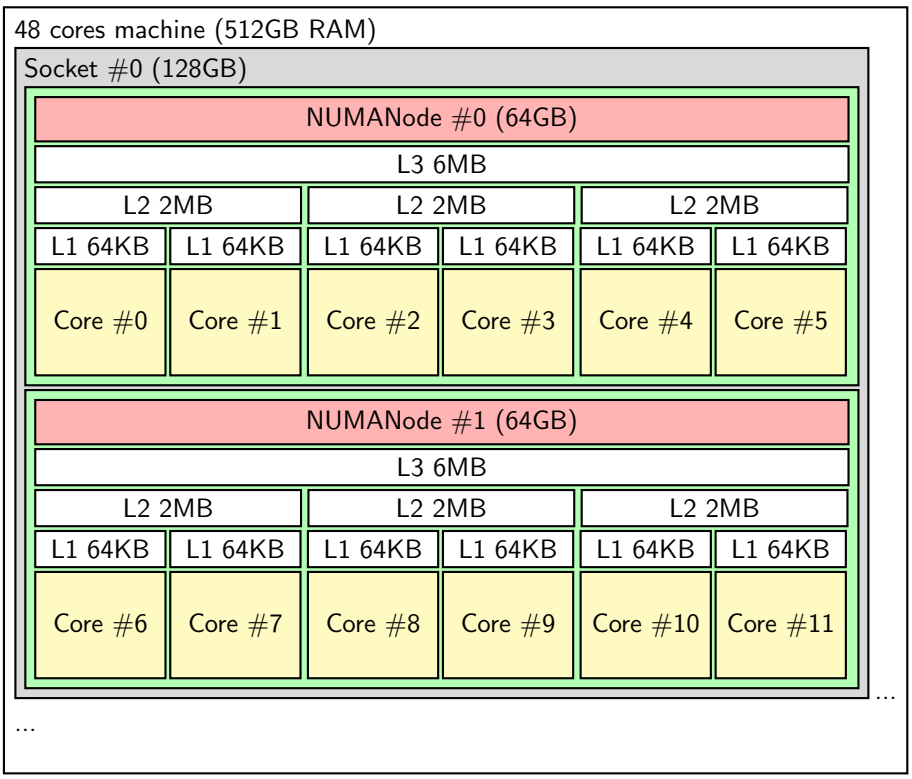
$$f = \begin{cases} \frac{T_l}{X} \times P & \text{if } P > 1 \\ 0 & \text{otherwise} \end{cases}$$

F1 The final heuristics is defined as follows: $f = a^{h(P)}$ where h is a function over the number of processors and the constant a is the arity of the tree, e.g. 4 for quad-trees. We use h in the exponent of this formula to reflect the exponential amount of potential parallelism in the tree structure.

4.12 Performance Evaluation

4.12.1 Experimental Setup

Machines: Two different machines are used for performance evaluation. For the initial test runs, we use the same desktop-class multi-core machine used in comparing



(showing 1 of 4 sockets)

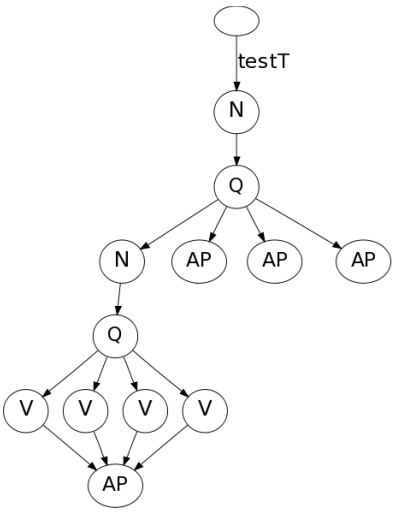
Figure 4.9: Server-class 48 cores machine topology map.

the parallel Haskell models in Chapter 3: an 8-core machine arranged in 2 sockets (2 X Intel Xeon CPU E5410 processors/4 cores @ 2.33GHz each), with 7870MB memory and 2 levels of cache hierarchy (detailed machine specification in Figure 3.4).

For performance scalability test, a server-class many-core machine consisting of 48 cores arranged in 4 sockets each with 2 NUMA nodes and each node with 6 AMD Opteron 6348 CPU cores (1400 MHz). Two cores share a 64kB L1 and a 2MB L2 cache, and the 6MB L3 cache is shared by all cores in one NUMA region (detailed NUMA structure of machine in Figure 4.9). Each region has 64GB memory, amounting to a total of 512GB RAM. The server machine runs at a lower frequency which means compute intensive tasks on a core will run slower. However, it exploits six times more processing units and thus more tasks i.e. sparks can execute in parallel to improve performance. Both machines run CentOS 6.5 64-bit Linux with kernel version 2.6.32. With a NUMA architecture, memory latencies vary depending on the region. Using the `numactl`[1] tool shows that access to a remote region is by a factor of 2.2 more expensive than access to a local region.

Compiler and libraries The Haskell compiler used is `ghc-7.6.1`. Among the main packages installed from Hackage through Cabal is `parallel-3.2.0.3` con-

[1]numactl - Control NUMA policy for processes or shared memory
http://linux.die.net/man/8/numactl



Example of using `isBoundedSize` (Listing 4.8) on a quadtree. Evaluation stops after establishing size is at least 4. AP=Unevaluated Expression

Figure 4.10: ghc-vis: visualise partially evaluated tree structure.

sisting of the primitives and original strategies. Our implementation is in our `pardata-0.1` package consisting of the new strategies. All programs are compiled with optimisation flag `-O2` on.

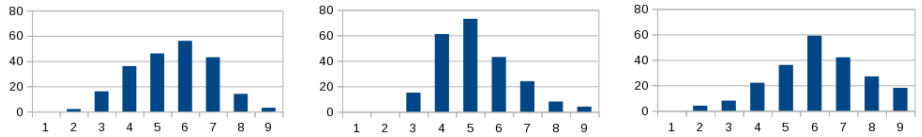
Tools We identify two useful visualisation tools to help in the implementation of the strategies and verify the actual with expected behaviour which is important in a non-strict language:

GHood (Reinke, 2001) allows one to observe intermediate states in the data structure as evaluation proceeds. Different colour schemes are used to highlight unevaluated thunks and evaluated structures.

ghc-vis (Felsing, 2012) is a tool to visualise live Haskell data structures in GHCi (see Figure 4.10 for an example). Evaluation is not forced and we can interact with the visualised data structures. This allows seeing Haskell's lazy evaluation and sharing in action.

4.12.2 Benchmark Program

The strategies are first tested on a benchmark program taking algorithmic complexity out of the picture and focusing on the strategies' behaviour. The program performs a parallel map on irregular trees with different depth distribution — normal, left-skewed and right-skewed (Fig. 4.11) — and fixed (homogeneous) or variable (heterogeneous) computations in the node.



(a) normal (b) left-skewed (c) right-skewed

Figure 4.11: Depth distribution for test program input.

The desktop machine run is mainly intended as a check for initial performance for the advanced strategies. Initial results on the 8-core machine with a small input size show good speedups for all the strategies on up to 8 cores. However, the advanced strategies do not outperform the naive element-wise `parTree`, as this strategy works well on a small scale.

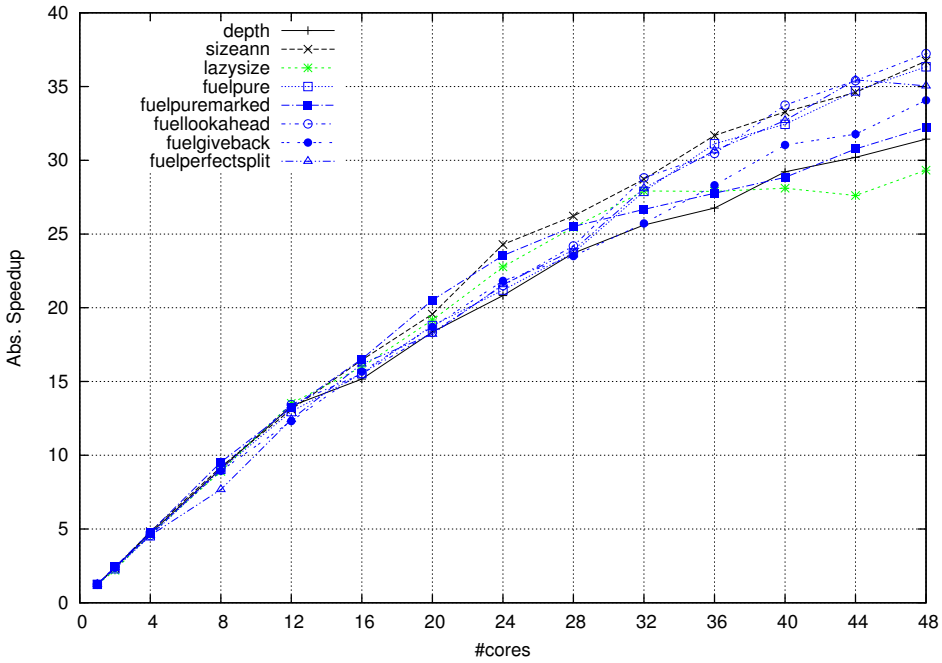
Moving on to a more realistic setup, Table 4.4 shows the running time and absolute speedup against the sequential runtime of the strategies on up to 48 cores (on the many-core server machine). On this machine, we use a larger input size of 100k tree elements and normal depth distribution across the tree. The benchmark program speedup is plotted in Figure 4.12. We make the following observations. The depth-based thresholding strategy works well with a speedup close to 32 on 48 cores. However, the main problem with this basic thresholding method is that the number of sparks generated remains flat after 8 cores. This is highlighted by the horizontal bars for the number of sparks in Figure 4.13(a). This is due to the maximum depth we encode in our heuristics as an upper bound in order to avoid excessive parallelism potential due to the exponential growth of nodes at each level. Moving to a more advanced parameter selection criteria, we see an improvement from using heuristics D2 over D1 as seen in Figure 4.13 (b). This demonstrates that for irregular trees, we can have a high depth threshold determined dynamically to go deeper down the tree in order to generate sufficient parallelism, while avoiding a hard-coded maximum d.

The lazy size strategy performs well on up to 32 cores with a speedup of 28 compared to 26 on the same core count for the depth-threshold strategy. This is explained by the fact that spark creation grows with an increasing number of cores. The speedups range from 28 to 30 on 36 to 48 cores, which is attributed to many more sparks being created on higher core numbers, which introduces some overhead and motivates the need for throttling parallelism.

The pure fuel strategy also gives good results compared to depth-based thresholding. This result is further improved by extending the fuel strategy with a lookahead mechanism — in this case, the amount of sparks generated is the same as the pure version, however, the performance gain comes from the improved efficiency in fuel distribution, marking the most eligible nodes to be sparked based on additional information from the lookahead mechanism.

	depth		sizeann		lazysize		fuelpure		fuelpuremarked		fuellookahead		fuelgiveback		fuelperfectsplit	
#pe	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP
1	196.42	1.25	194.92	1.25	196.52	1.24	196.03	1.25	195.46	1.25	195.82	1.25	196.11	1.25	195.61	1.25
2	102.32	2.39	103.56	2.36	110.49	2.21	106.00	2.31	100.56	2.43	100.41	2.44	103.32	2.37	102.19	2.39
4	51.89	4.71	50.77	4.82	52.67	4.64	53.89	4.54	51.02	4.79	53.64	4.56	52.20	4.69	53.53	4.57
8	26.93	9.08	26.57	9.21	27.46	8.91	27.06	9.04	25.67	9.53	26.74	9.15	27.36	8.94	31.86	7.68
12	18.33	13.34	18.66	13.11	18.09	13.52	18.87	12.96	18.51	13.21	18.20	13.44	19.86	12.32	19.68	12.43
16	16.12	15.17	14.86	16.46	15.30	15.99	15.74	15.54	14.81	16.52	15.75	15.53	15.61	15.67	15.10	16.20
20	13.32	18.36	12.50	19.57	12.76	19.17	13.02	18.79	11.92	20.52	13.35	18.32	13.08	18.70	13.43	18.21
24	11.74	20.83	10.07	24.29	10.74	22.77	11.54	21.20	10.39	23.54	11.37	21.51	11.21	21.82	11.34	21.57
28	10.32	23.70	9.33	26.22	9.61	25.45	10.31	23.72	9.59	25.51	10.11	24.19	10.41	23.50	10.23	23.91
32	9.55	25.61	8.52	28.71	8.76	27.92	8.77	27.89	9.17	26.67	8.49	28.81	9.51	25.72	8.70	28.11
36	9.14	26.76	7.72	31.68	8.77	27.89	7.86	31.12	8.81	27.76	8.03	30.46	8.64	28.31	7.98	30.65
40	8.37	29.22	7.35	33.28	8.70	28.11	7.54	32.44	8.48	28.84	7.25	33.74	7.88	31.04	7.48	32.70
44	8.10	30.20	7.07	34.60	8.86	27.61	7.05	34.70	7.95	30.77	6.91	35.40	7.70	31.77	6.90	35.45
48	7.78	31.44	6.66	36.73	8.34	29.33	6.73	36.34	7.59	32.23	6.57	37.23	7.18	34.07	6.98	35.04

Table 4.4: Benchmark program – Runtime and Speedup on 48 cores.



Input: 100k elements

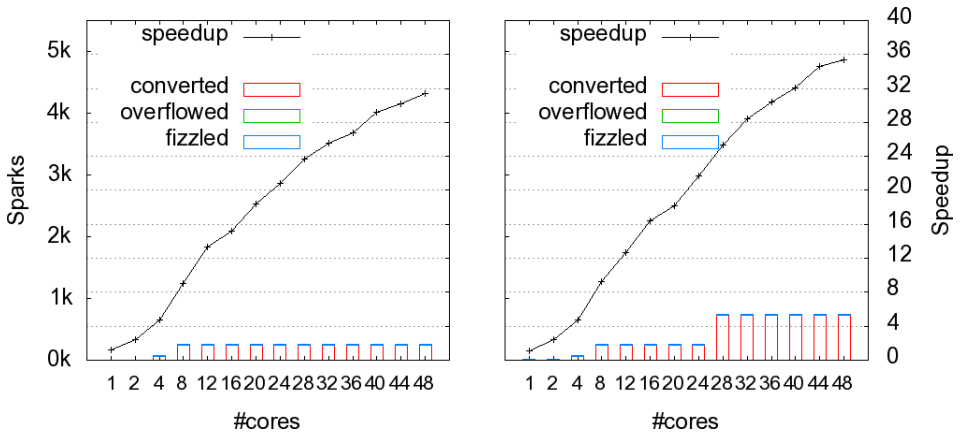
Figure 4.12: Test program speedups on 1-48 cores.

The giveback fuel strategy has performance close to the depth-based thresholding strategy, even though we note that the giveback technique generates far more sparks than the remaining strategies with the same amount of fuel. In analysing the giveback mechanism, we measure how often fuel was given back to a parent node. With $f = 50$, the fuel hit-rate (the number of times an outer node is hit, thus fuel is passed back upward) is 247. For $f = 100, 500$ and 1000, the hit-rates are 478, 2357 and 4417, respectively. These numbers demonstrate that the giveback mechanism is effective in enabling additional parallelism for irregular trees. Due to the circular distribution of fuel, we expect more fuel to be available for nodes inside the tree — that is, distribution carries on deeper inside the tree, explaining why higher numbers of sparks are generated.

4.12.3 Barnes-Hut Algorithm

The Barnes-Hut algorithm for the n-body problem is used as a concrete application to assess the performance of our new strategies. The step-by-step implementation of the algorithm is provided in the previous chapter. In short, the algorithm simulates the movement of objects in space and uses a quad-tree representation for 2D space. The algorithm involves two phases:

1. *Tree construction:* a tree containing all the bodies in the given space is constructed.

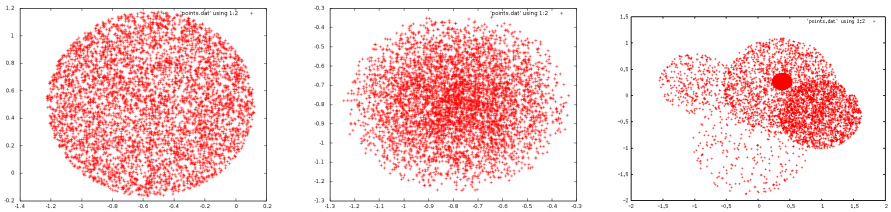


(a) Depth heuristic – D1 (b) Depth heuristic – D2

Figure 4.13: Depth heuristics performance comparison: D1 vs D2

2. *Force calculation:* iteration-wise force calculation for each body with respect to the rest, followed by positions change.

The main source of parallelism is in the force calculation step, where the force for each body is computed independently from the other bodies. We adapt the list-based algorithm used in Chapter 3 by performing the main map operation in the second phase over a quad-tree data structure. This enables us to use the newly developed tree-based strategies, while comparing the performance with an already well-tuned implementation. Most notably, no restructuring of the sequential code was necessary to enable parallelism. We also extend our experiments to include a number of different body distributions — single uniform cluster, single normally distributed cluster, and multiple clusters of bodies, as depicted in Figure 4.14. This is aimed at studying the performance of the new strategies with irregular data distributions.



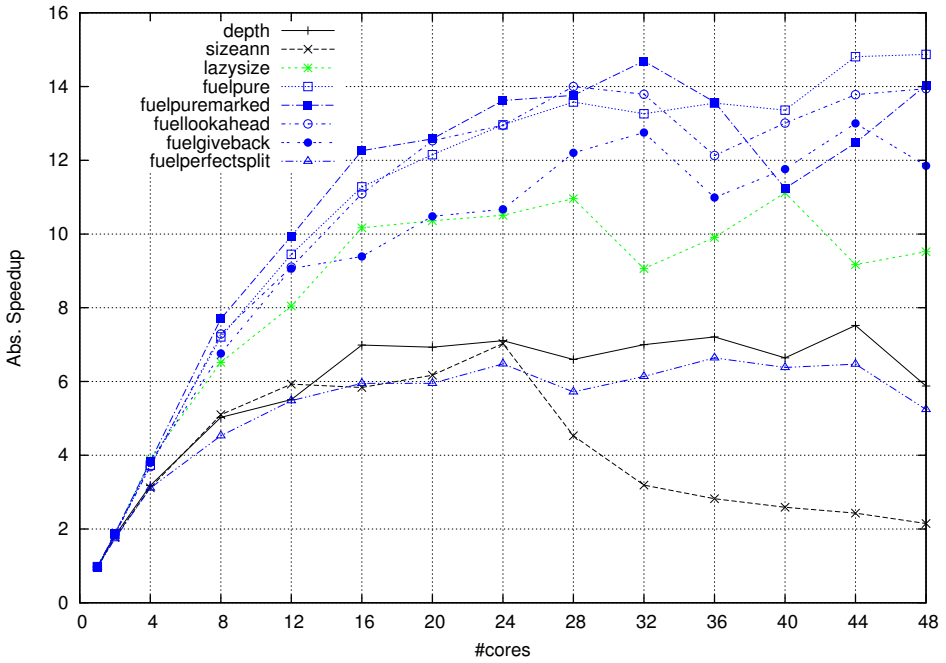
(a) uniform cluster (b) normal cluster (c) multiple clusters

Figure 4.14: Bodies distribution

Performance Results: We focus our discussion on a particular set of results obtained for one cluster of normally distributed bodies across the space (Figure 4.14 (b)) as input and compare these with results from a multiple clusters input (Figure 4.14 (c)). The latter is an example of higher irregularity in the data distribution.

	depth		sizeann		lazysize		fuelpure		fuelpuremarked		fuellookahead		fuelgiveback		fuelperfectsplit	
#pe	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP
1	205.92	0.96	203.61	0.97	200.58	0.99	201.39	0.98	203.49	0.97	203.46	0.97	202.80	0.98	207.26	0.96
2	109.09	1.82	112.03	1.77	107.08	1.85	106.93	1.85	105.24	1.88	106.05	1.87	107.06	1.85	113.88	1.74
4	62.45	3.18	63.58	3.12	51.13	3.88	53.14	3.73	51.72	3.83	53.82	3.69	52.13	3.80	63.73	3.11
8	39.45	5.03	38.86	5.10	30.44	6.52	27.52	7.21	25.77	7.70	27.21	7.29	29.35	6.76	43.81	4.53
12	36.00	5.51	33.47	5.93	24.64	8.05	20.98	9.45	19.96	9.94	21.79	9.10	21.88	9.06	36.18	5.48
16	28.37	6.99	33.98	5.84	19.51	10.17	17.59	11.28	16.18	12.26	17.89	11.09	21.12	9.39	33.35	5.95
20	28.60	6.93	32.17	6.17	19.15	10.36	16.32	12.15	15.77	12.58	15.83	12.53	18.92	10.48	33.31	5.95
24	27.88	7.11	28.23	7.03	18.88	10.51	15.30	12.96	14.56	13.62	15.31	12.95	18.59	10.67	30.63	6.48
28	30.06	6.60	43.80	4.53	18.09	10.96	14.61	13.58	14.41	13.76	14.17	14.00	16.26	12.20	34.66	5.72
32	28.35	7.00	62.21	3.19	21.90	9.06	14.96	13.26	13.50	14.69	14.38	13.79	15.55	12.75	32.29	6.14
36	27.50	7.21	70.31	2.82	20.02	9.91	14.64	13.55	14.60	13.58	16.35	12.13	18.05	10.99	29.86	6.64
40	29.85	6.64	76.66	2.59	17.84	11.12	14.85	13.36	17.64	11.24	15.24	13.01	16.87	11.76	31.10	6.38
44	26.38	7.52	81.70	2.43	21.64	9.17	13.39	14.81	15.90	12.47	14.39	13.78	15.26	13.00	30.65	6.47
48	33.74	5.88	92.46	2.15	20.83	9.52	13.34	14.87	14.13	14.04	14.23	13.94	16.74	11.85	37.83	5.24

Table 4.5: Barnes-Hut algorithm – Runtime and Speedup on 48 cores.

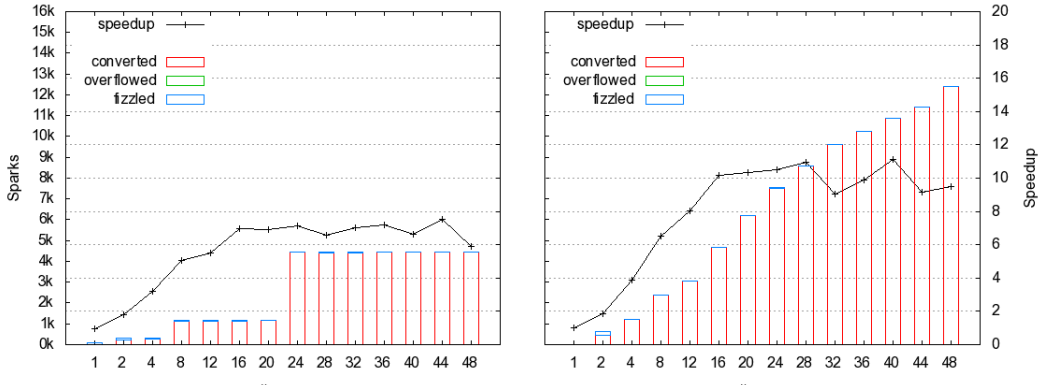


Input: 2 million bodies, 1 iteration

Figure 4.15: Barnes-Hut speedups on 1–48 cores.

Table 4.5 summarises the running time and absolute speedup showing the mean values of 3 runs and on up to 48 cores. The speedup graph in Figure 4.15 show that across the range of core numbers, the fuel-based strategies (highlighted in blue) are consistently more efficient than `parTreeDepth` or `parTreeSizeAnn`, for depth-based and size-based thresholding, respectively. In particular, a pure fuel version, using just simplistic but cheap fuel splitting, performs best on 48 cores, exhibiting a speedup of 15, and the marked variant of the fuel strategy performs best for core numbers up to 36. These results indicate that a fuel-based approach to controlling parallelism is more flexible than a thresholding approach. The latter performs very poorly from 16 cores onwards and drops in performance on the high end of the spectrum. One inherent problem with depth-based thresholding is that it provides only a very crude mechanism for controlling the amount of parallelism that is generated, because the number of sparks is exponential in the depth that is used as threshold. Furthermore, it misses out on opportunities of re-using potential parallelism late in the computation, where parallelism typically diminishes, due to having hit the depth threshold at this point in the computation. This shows up as a step function in the profile plotting sparks over cores (shown in Figure 4.16(a)). In contrast, the same profile for the lazysize strategy shows a continuous function, where sparks steadily increase over the number of cores (Figure 4.16(b)).

Among the fuel-based strategies, the pure variant performs best, but other variants remain fairly close to it. In particular, the giveback variant is within 20% and the lookahead variant is within 6% of the best result. The fuel strategies invest more



(a) depth (b) lazysize

Figure 4.16: Depth vs Lazy Size sparks creation.

work into orchestrating the parallelism, by passing fuel through the tree: measuring this overhead shows that for an input of 2 million bodies, annotating the tree takes about 11% and unannotating the tree takes about 4% of the time needed to build the tree. The overall performance is therefore a balance between this overhead and the more flexible distribution of fuel. For example, using a give-back mechanism to distribute the fuel both down-wards and up-wards, shows that for a tree with 100 thousand elements, and a fuel of 2000, there are 7682 instances of give-back, due to the irregular distribution of the input data.

We also observe that a perfect split strategy performs poorly in Figure 4.15. Again we believe that this is due to the additional overhead incurred by this strategy. Notably, the three worst performing strategies in this figure, are the ones that need a global context in order to decide how to arrange the parallelism. This can be seen in column 5 of Table 4.1.

The limited scalability beyond 28 cores can be attributed to specifics of the algorithm, the runtime system and the hardware. The algorithm itself is a standard Barnes-Hut algorithm without further optimisations to minimise data exchange and facilitate scalability. Using an increasing number of cores will naturally generate a higher number of threads in our programming model, which increases the amount of live data and thus the garbage collection overhead. Additionally, global synchronisation across all cores is necessary to perform major collections. The underlying physical shared-memory hardware is a NUMA architecture with higher memory latencies across remote NUMA regions, of 6 cores. Thus, involving several regions in the computation will result in a significant percentage of expensive memory accesses. This effect is even more pronounced in Haskell programs, since the underlying graph reduction machinery typically requires frequent memory accesses across a large, dynamic heap. For a more detailed study of the parallel memory management performance see (Aljabri et al., 2014a).

4.12.4 Sparse Matrix Multiplication

Matrix multiplication arises in many numerical and scientific applications. For large matrices, optimised multiplication and other operations that run in parallel are essential. Sparse matrices have a high ratio of nonzero to zero elements. Therefore, it is sensible to store only nonzero elements to avoid unnecessary space usage.

Sparse Matrix Representation The standard representation of sparse matrices is the column-value pairs for each nonzero value for each row. For instance, the sparse matrix:

$$A = \begin{pmatrix} row & col & 0 & 1 & 2 & & \cdots & & & 7 \\ 0 & & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & & 1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 \\ 2 & & 0 & 1 & 2 & 3 & 0 & 0 & 0 & 0 \\ & & 0 & 0 & 1 & 2 & 3 & 0 & 0 & 0 \\ \vdots & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

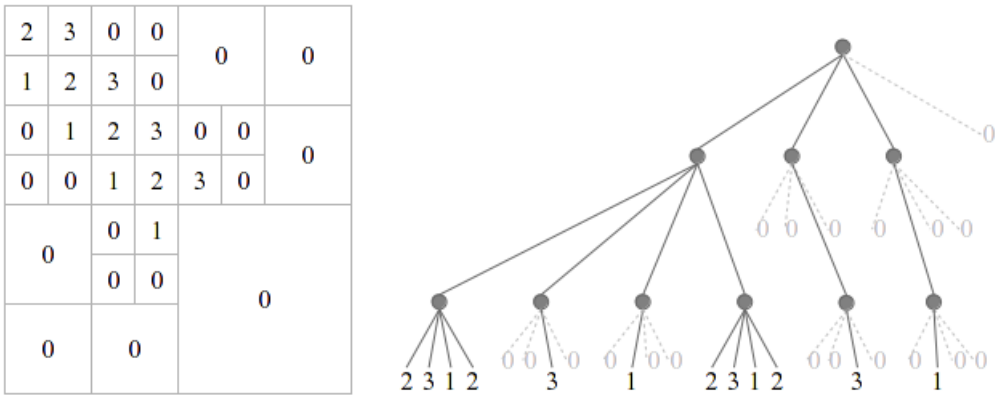
is represented in a compact form as follows:

$A =$
$[[(0,2),(1,3)],$
$[(0,1),(1,2),(2,3)],$
$[(1,1),(2,2),(3,3)],$
$[(2,1),(3,2),(4,3)],$
$[(3,1)],$
$[],$
$[],$
$[]\ ]$

The sparse matrix is compacted in a sequence of sequence where each element of the outer sequence represents a row and inner sequence is the index-value pairs for each nonzero elements. This representation is usually expanded back to its full form, that is, including the zeros before doing operations such as multiplication.

Tree-based Matrix Representation Sparse matrices can be represented using quad-trees. This allows us to use the sparse matrix multiplication as an additional application to test our tree-based strategies. Figure 4.17 shows sparse matrix A represented as a quad-tree where whole a region containing non-zero elements is represented as a single zero node.

Wainwright et al (Wainwright and Sexton, 1992) report good sequential performance



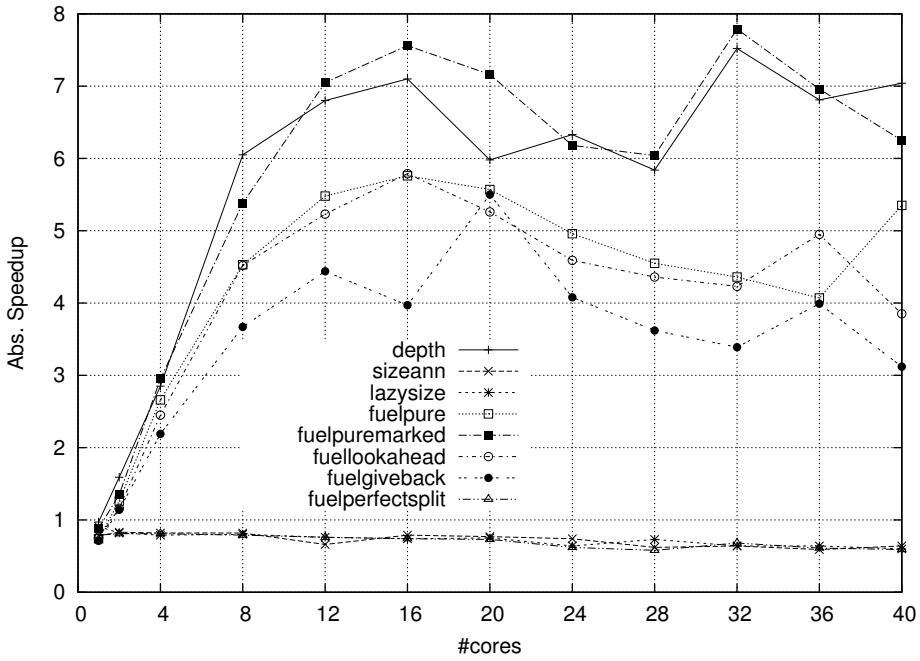
(a) (b)

Figure 4.17: Quad-tree representation of a sparse matrix.

and reduced space overhead for sparse input data, using a quad-tree representation for sparse matrix multiplication. We adapt the implementation in Haskell and aim for further performance gains through parallelisation using our new strategies. This is achieved by demanding the result matrix in parallel. Again, we do not have to change the sequential algorithm to obtain a parallel version. In Listing 4.17, we define a matrix as a tree using the type synonym (`type Matrix a = QTree a`). Two input matrices are converted to quad representations (`quadrep`) before the multiplication function is called (`qmul`), producing a result matrix in tree format. The result matrix is demanded in parallel by applying a parallel strategy e.g. `parTreeFuel` as previously defined.

Listing 4.17: Sparse matrix multiplication code extract.

```
type Matrix a = QTree a

run m n s v vv = do
  a <- ... -- read m x n matrix input a and b
  b <- ...
  let ma = quadrep a
      mb = quadrep b
      res = qmul ma mb `using` parTreeStrategy
  ... -- print result in readable format

-- build a quadtree from a list
quadrep::(Eq a,Num a) => [a] -> Matrix a
quadrep [x]      = L x
quadrep xs
  | all (0==) xs = E
  | otherwise    = N $ Q nw ne sw se
  where
    nw = quadrep (buildw firsthalf)
    ne = quadrep (builde firsthalf)
    sw = quadrep (buildw secondhalf)
    se = quadrep (builde secondhalf)

    len = length xs
    y   = floor . (/2) . sqrt . fromIntegral $ len
    y2  = y*2

    (firsthalf,secondhalf) = splitAt (len `div` 2) xs

    -- build a west region
    buildw [] = []
    buildw x  = (take y x) ++ (buildw (drop y2 x))

    -- build an east region
    builde [] = []
    builde x  = (take y (drop y x)) ++ (builde (drop y2 x))

-- main multiplication function
qmul::(Eq a,Num a) => Matrix a -> Matrix a -> Matrix a
qmul E        _        = E
qmul _        E        = E
qmul (L x)    (L y)    = L (x*y)
qmul n1@(N _) n2@(N _) = let
                           (N (Q nw1 ne1 sw1 se1)) = n1
                           (N (Q nw2 ne2 sw2 se2)) = n2
                           nw = (qmul nw1 nw2) `qadd` (qmul ne1 sw2)
                           ne = (qmul nw1 ne2) `qadd` (qmul ne1 se2)
                           sw = (qmul sw1 nw2) `qadd` (qmul se1 sw2)
                           se = (qmul sw1 ne2) `qadd` (qmul se1 se2)
                         in N (Q nw ne sw se)

qadd::(Eq a,Num a) => Matrix a -> Matrix a -> Matrix a
qadd E        x        = x
qadd x        E        = x
qadd (L x)    (L y)    = L (x+y)
qadd n1@(N _) n2@(N _) = let
                           (N (Q nw1 ne1 sw1 se1)) = n1
                           (N (Q nw2 ne2 sw2 se2)) = n2
                         in N $ Q (qadd nw1 nw2) (qadd ne1 ne2)
                                  (qadd sw1 sw2) (qadd se1 se2)
```

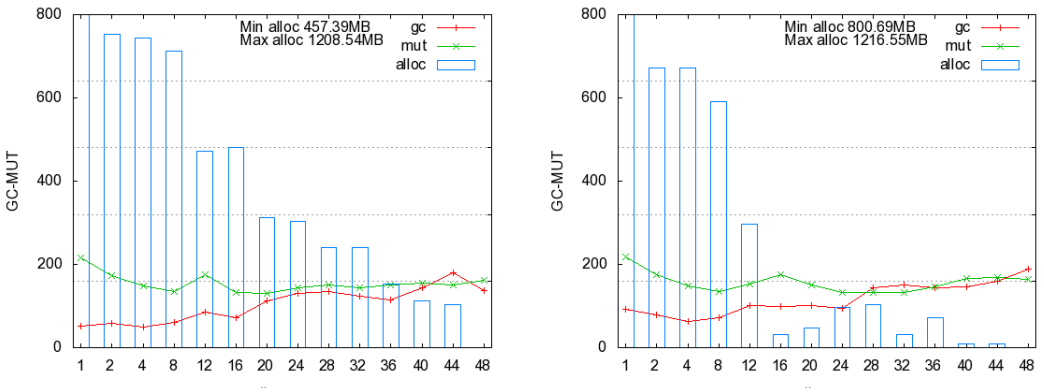


Input: 4096x4096 matrices with 5% sparsity

Figure 4.18: Sparse matrix multiplication speedups on 1-40 cores.

Our results for 4096x4096 input matrices with 5% of all elements containing non-zero values (sparsity), in Figure 4.18, show fairly good performance on core numbers up to ca. 12 or 20, i.e. typical sizes for current desktop machines. However, there is a significant drop in performance thereafter and therefore poor scalability. One specific characteristic for this application is its fairly high memory allocation throughout the execution: total allocation is 4 times and memory residency is 3 times that of the Barnes-Hut algorithm. As a result, the garbage collection (GC) overhead is high and steadily increasing for higher core numbers. We note that at the point where speedups drop, around 20 to 28 core, the GC% in the execution surpasses the MUT%, i.e. the mutation time spent doing actual graph reduction. This is shown for the depth and giveback strategies in Figure 4.19. This is an indication that all versions suffer from high GC overhead. Thus, it would be profitable to throttle the parallelism more aggressively, and this is the direction we want to explore in the future.

Comparing the performance of the different strategies reveals that again the marked variant of a fuel strategy performs best for core ranges between 12 and 20. The depth-based thresholding variant performs significantly better in this application, probably because the result of matrix multiplication is much denser than its input. Since our lazy strategies typically generate parallelism by traversing, and thus forcing evaluation of the result data structures, this means that the data is more regularly distributed compared to the Barnes-Hut program. Because we are not performing any operation on the elements of the result matrix, it is expected that `sizeann` and



(a) depth (b) fuelgiveback

Figure 4.19: GC-MUT and allocation for depth and giveback.

`fuelperfectsplit` do not give good performance as both require a first pass over the result matrix to attach administrative information.

4.13 Summary

In this chapter, we have presented novel parallelism control mechanisms, building on laziness to achieve additional flexibility. We have encoded these as evaluation strategies over tree-like data structures and demonstrated improved parallel performance over established methods for throttling parallelism on a 48-core shared-memory server using three benchmark programs. Our new strategies are more flexible in controlling the available parallelism, by re-using previously unused potential parallelism and thus obtaining better performance on structures with irregular data distribution. Our performance results show that they out-perform classic techniques that are often used, such as depth-based thresholding. Core techniques that we use in the implementation to gain this added flexibility are circular programs, requiring lazy evaluation, and annotating the tree structure with administrative information. To the best of our knowledge, this is the first use of circular programming techniques to improve parallel performance. The best-performing strategy is based on the notion of fuel that is passed down the tree and controls whether parallelism should be generated or not. While our implementation and performance results have been obtained from an quad-tree data structure, the techniques, such as bi-directional flow of fuel, are not restricted to trees nor to specific applications.

Chapter 5

Graph Evaluation Strategies

In the previous chapter, the focus has been on tree data structures, in particular, quad-trees, suitable for the Barnes-Hut algorithm and sparse matrix representation. The techniques that we developed for trees are not limited to this particular structure. In this chapter, we apply these techniques to graph data structures. We present a number of sequential and parallel strategies for graph data structures, specifically the use of recursive tree definition for graph which offers a naturally functional data representation and depends on laziness. This representation allows us to carry over the mechanisms for effective parallelism control previously presented for trees.

Graphs are increasingly important for high-performance computing in the context of big-data applications (Gilbert et al., 2007). Many problems can be modelled using graphs, for example, social networks (relationship between persons), computer networks (node connections) and web content representation (pages and links). Graphs are at the core of data intensive applications and there is an increased interest for efficient algorithms with a number of benchmark suites becoming prominent, for example, the Graph 500, HPC Graph Analysis and Lonestar (Graph500, 2015; Bader et al., 2015; Kulkarni et al., 2009).

5.1 Graph Definitions

Below we briefly explain the terminologies used in discussing graphs (Gondran et al., 1984); including definitions and basic concepts, and the types of graph we look at.

Definition 5.1. A graph $G = (V, E)$ is defined as

i) a set $V = \{v_1, v_2, ...v_n\}$ whose elements are called vertices (or nodes), and

ii) a set E whose elements $e \in E$ are pairs of vertices from V, i.e. $V \times V$, and called edges (or arcs, or connections).

The number of vertices in the graph is given by $n = |V|$. Thus, the graph is said to be of order n.

A path from a vertex v to a vertex u is a sequence $\{v_0, v_1, v_2, ...v_k\}$ of vertices where $v_0 = v$, $v_k = u$, and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, ...k - 1$.

A graph is a network model, with either directed or undirected connections. The latter connection can be viewed as bi-directional.

In an undirected graph, the edge $e = (v_i, v_j)$ represents a connection between vertex v_i and vertex v_j, i.e. $\forall(v_i, v_j) \cdot (v_i, v_j) \in E \Leftrightarrow (v_j, v_i) \in E$.

In a directed graph, the edge $e = (v_i, v_j)$ represents a connection from vertex v_i to vertex v_j, i.e. an arrow $v_i \rightarrow v_j$. v_i is the initial endpoint and v_j is the terminal endpoint of e. A direct loop is when both endpoints are the same, i.e. (v_i, v_i). A cycle (indirect loop) consists of a sequence of vertices starting and ending at the same vertex, i.e. $(v_i, ...v_i)$.

The in-degree of a node n_{in} is the number of terminal endpoints to that node, that is, the number of arrows pointing to that node. The out-degree of a node n_{out} is the number of initial endpoints from the node. A node is a shared node when $n_{in} > 1$.

5.1.1 Graph Types

A tree is a hierarchical model with implicit one-way connections always originating from parents to children nodes. A tree structure can be viewed as a specialised type of graph that restricts upward references and sharing.

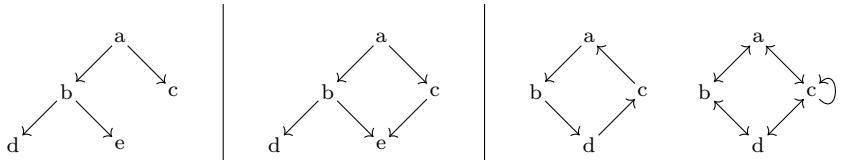
For translating our tree strategies to graph strategies, we distinguish between three types of graph:

Definition 5.2. A tree graph contains no cycles and has a distinguished root node with an in-degree of zero. A node has exactly one parent and can have any number of children.

All strategies defined in the previous chapter work on tree graphs only. By extending the properties of a tree graph, we can obtain two other widely used graph types. They differ in two important properties: node sharing turns a tree into an acyclic graph, and cycles turn a tree into a cyclic graph. The classification reflects these properties which affects design decisions and underlying representation and implementation.

Definition 5.3. An acyclic graph contains no cycles but nodes can share other nodes. Thus, a node can have more than one parent, i.e. an in-degree of more than one.

Definition 5.4. An cyclic graph contains cycles where any node in the graph may have a path back to it, including through a self-loop, or dual reference.



(a) tree (b) acyclic (c) cyclic

Figure 5.1: Types of Graph

5.2 Graph Representations

The representation of a graph has a major effect on the complexity of an algorithm. The choice of representation is influenced by a number of factors including graph size and connection intensity.

Given a graph $G = (V, E)$, it can be represented in the following ways (Gondran et al., 1984).

- *List of edges.* This representation maintains a list of edges represented as pairs of initial and terminal endpoints. It is probably the simplest representation.

```
{ (a,b),
  (a,c),
  (b,d),
  (c,d) }
```

- *Adjacency matrix.* This is a square matrix representation where rows and columns correspond to the vertices of the graph, and a connection between two vertices is represented by boolean or numeric value. This representation requires N^2 space and is preferred for dense graphs, that is, graphs with a high degree of connections between nodes.

```
  a b c d
a   1 1
b       1
c       1
d
```

- *Adjacency list.* A more space efficient representation which is suited for sparse graph. Vertices are listed once with their connections.

```
{ (a, {b,c}),
  (b, {d}),
  (c, {d}),
  (d, {}) }
```

- *Recursive representation.* An unusual but naturally functional representation of graph using recursive representation. The connections in this representation are implicit, i.e. there is no list of explicit connections or list of edges. Instead, each graph node points to children nodes by keeping references to them, analogous to a linked list or a tree data structure. We use this representation to implement graph strategies.

5.3 Related Work in Functional Graphs

Functional graphs and graph algorithms have been studied and it has been shown that while it may be difficult to achieve the same efficiency as imperative implementations, functional implementation of graphs offer a high level of abstraction as with other functional data structures (Okasaki, 1999). Many studies have focused mainly on achieving asymptotic complexity of graph algorithms which is as good as imperative implementation. Often, the choice of representation encourages an imperative style of programming in a functional language, for example, through the use of functional arrays and monad to maintain states during search or other operations on graphs.

The following briefly describes two graph libraries in Haskell.

5.3.1 Data.Graph

The `Data.Graph` module from the `containers`[1] package is based on (King and Launchbury, 1994) and uses an adjacency list representation and arrays (with the indexes representing nodes and values representing neighbours of each node) as the underlying data structure. The library is concerned only with depth-first search using a spanning forest. The approach facilitates formal reasoning of algorithms based on depth-first search. However, the implementation is close to an imperative programming style.

5.3.2 FGL

Erwig (2001) presents an inductive view of graphs, similar to lists or trees. Erwig's work covers implementations in both ML (using imperatively updatable arrays) and Haskell in the FGL library[2]. The Haskell library provides two graph implementations. The first one is an efficient implementation of graph using `Data.IntMap` from

[1]The containers package – https://hackage.haskell.org/package/containers
[2]The fgl package – https://hackage.haskell.org/package/fgl

the `containers` package which is based on big-endian patricia trees. The second one is a tree-based implementation of static and dynamic graphs. Our approach and choice of representation is closer to this functional and recursive style.

Our focus is on traversal and evaluation in parallel. In the next section, we will start from the same type definition as a binary tree for graph, and only slightly extending it to work with acyclic and cyclic graph and different traversal strategies. Our aim is to gain performance through parallelisation. For this, the underlying representation of a graph is important. We choose a representation that is a natural fit for parallelisation.

5.4 Data Type Implementation

5.4.1 Adapting Tree Data Type

We stay with a tree-based representation which is conveniently expressed using an algebraic data type in Haskell, and where the connections are implicit. This choice of representation is unusual, especially in an imperative context where the standard would be an adjacency list or matrix representation, which facilitates a number of tasks including search operations (King and Launchbury, 1994). In a tree-based representation, there is no collection of vertices, or explicit connections: a pointer points to an arbitrary node from the graph, and the vertices and implicit connections can be collected through a traversal. Using a recursive tree structure allows us to re-use strategies for trees and carry forward parallelism control mechanisms applied to trees to graphs.

In principle, the same data type definition of a binary tree can be used to represent a simple directed graph where each node can connect to at most two other nodes, and where sharing and cycles are allowed.

Definition 5.5. A binary tree-based graph (type graph G) is either:
i) an empty graph (E), which has no nodes, or
ii) a graph which consists of:
– a root node (type G)
– a left and a right subnodes of the root (type pair of graph G) isomorphic to a binary tree.

This structure is recursive since a graph node may contain other nodes, each of which may contain others. Additionally, a node can point to itself, or any other nodes, including shared nodes. The Haskell implementation in Type Def. 5.1 is close to the

mathematical notation.

Type Def. 5.1 (Binary tree-based graph representation)**.**

A graph of the form

```
data G v = N v (G v) (G v) | E
```

with data elements, i.e. vertices, of type `v` is called a binary graph using recursive tree structure definition.

In Type Def. 5.1, the outdegree for each node is limited to two, but there is no structural restriction to define other graph types in addition to tree graphs. For instance, we can capture acyclic and cyclic graphs as depicted in Figure 5.1(b) and (c) and instantiated as shown in Listing 5.1 below using a tree-based representation for graphs in Haskell.

Listing 5.1: Acyclic and cyclic graph instances using tree-based type

```
-- acyclic graph example (Fig 5.1(b))
acyclic_g = a
 where
  a = N 1 b c
  b = N 2 E d -- b and c share the same node d
  c = N 3 d E
  d = N 4 E E

-- cyclic graph example (Fig 5.1(c))
cyclic_g = a
 where
  a = N 1 b c
  b = N 2 a d
  c = N 3 a d
  d = N 4 b c
```

Type Def. 5.1 presents a simple case of using the same tree data type to implement a *limited* graph type. Building on this, we can define a more generic and advanced graph type which can be used for other forms of graphs. In particular, many applications require nodes to have arbitrary number of connections. This can be implemented using a rose-tree definition (see Section 4.4), substituting the pair of left and right nodes, with a list of nodes.

5.4.2 Extended Graph Data Type

Type Def 5.2 represents a more powerful representation, i.e. it can represent more forms of graph and better suited for many algorithms.

Type Def. 5.2 (Extended rose-tree-based graph type)**.**

A graph of the form

```
type NId = Int
data Graph v = N NId v [Graph v]

-- list of connected components
type ConnGraph v = [Graph v]

-- to enforce graph type
data Gtype = Tr | Acyc | Cyc
data G v = G Gtype (Graph v)
```

– with data elements (vertices) of type `v` has an arbitrary number of connections,
– allows one to uniquely identify a node through `NId`, for instance, to keep track of visited nodes during a traversal,
– may have a number of connected components, and
– may enforce a specific graph type, through an extension to augment the type with additional information to distinguish between graph types.

In Type Def. 5.2, we extend the basic definition with a node Id and there is no limit to the outdegree a graph node may have. The node Id is important to uniquely identify a node in the graph. This is necessary during traversal to keep track of already visited nodes. More information at node level can be encoded, for e.g. the graph type, which could help to switch between different traversals specialised for specific subgraphs, or independent components. Predicate functions can distinguish between different graph types by determining nodes sharing or cycles in instances of this type, and accordingly select an appropriate traversal for a specialised type. We use laziness on data-type constructors to encode cycles in this data representation. This would not work in a language with strict data type constructors e.g. ML.

5.4.3 Administration Data Structures

We need to implement a way to check if nodes have been visited before; otherwise we may re-process them, or even run into an infinite loop. In an imperative language we can mark a node as visited after it has been encountered the first time. However, in a (state-less) functional language, this is done in a different way – we maintain a data structure of visited nodes, which is passed through each recursive call into the graph. This requires an additional parameter for passing the state around, and if not done in an efficient way, it may affect parallelism by introducing unnecessary sequencing.

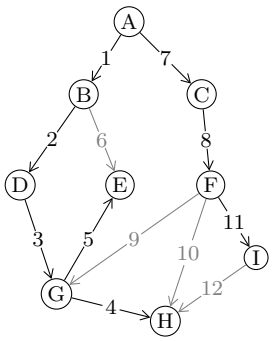


Figure 5.2: Depth-first traversal order

5.5 Graph Traversal Strategies

In this section, we describe two graph traversal algorithms, depth-first and breadth-first, and the implementation of graph evaluation strategies based on each traversal order. To simplify the explanation, we use binary graph examples in the code listing (i.e. Type Def. 5.1).

5.5.1 G1: Depth-First

Definition 5.6. A depth-first search or traversal starts at an arbitrary node of a graph and proceeds as far as possible along the current path before backtracking.

"In this pattern of search, each time an edge to a new vertex is discovered, the search is continued from the new vertex and is not renewed at the old vertex until all edges from the new vertex are exhausted." (Tarjan, 1972)

Figure 5.2 depicts a depth-first traversal where the nodes are visited in the order `ABDGHECFI`. Starting from node A, the traversal proceeds as far as possible (until `H`), then it backtracks and visits `E`. The number on the edges indicate the order in which each path is explored. An edge that leads to an already visited node is coloured grey. The following is an imperative algorithm for depth-first traversal. All nodes

are assumed to be marked unvisited initially.

```
Input  : Graph
Output: List of reachable nodes from start node
initialise empty result list;
proc dftrav(n)
    if not marked n then
        process and add node to result list;
        mark node as visited;
        for each neighbour n of node do
            dftrav(n);
        end
    end
```

Algorithm 1: Depth-first traversal

Implementation A simple depth-first graph traversal is obtained by extending tree traversal with a container used to retain references of already visited nodes. This ensures that when visited nodes are encountered again, e.g. through *shared* pointers, they are not processed again. This implementation is an alternative to using monadic state which requires more code changes to the naive traversal to mark visited nodes. In Listing 5.2, we use a map data structure (dictionary) from `Data.Map` module to store visited node IDs as keys and their corresponding updated nodes as values. If a node has already been visited (Line 9-10) (determined by a key look up query on the map), a reference to the updated node is returned. Otherwise, each path of that node is traversed recursively, with the output (updated visited nodes map) of the preceding path used as input to the next one (Line 13-14). This is another example of use of circular reference in our strategy implementation.

Dealing with non-termination Note that the map stores the traversed nodes IDs as well as a reference to the updated nodes, so if a path leads back to a visited node, we just return the reference. This ensures that the reconstructed graph after traversal is a homomorphism, i.e. no structural changes occurs on traversal and all connections, even if they lead to an visited node, are retained. This step also solves an important issue of non-termination when the updated node reference for each visited node is not kept and referred to when needed.

Line 16-20 executes in the `Eval` monad and is where the strategy `f` is applied to the node value. Note that this is equivalent to using the applicative notation `N idx <$> f x <*> new_l <*> new_r` (see Section 4.6), in this example, to extract the connected nodes from the `Eval` context. We use the `do` notation so we can rearrange the imposed sequence on `f x`, `new_l` and `new_r` to avoid a potential loop when dealing with cyclic graphs. Note that the data structure `new_n` defined in the main branch is used in its definition.

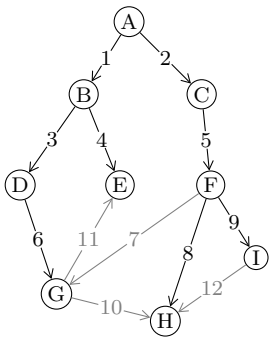


Figure 5.3: Breadth-first traversal order

Listing 5.2: Depth-first traversal over binary tree graph.

```
evalGraph_df::NFData a =>Strategy a -> Strategy (Graph a)
evalGraph_df f g = g'
  where
    (g',_) = dftrav g vs
    vs     = Map.empty -- visited node ids

    dftrav E             vs = (return E,vs)
    dftrav (N idx x l r) vs =
      case Map.lookup idx vs of
        Just n  -> (n,vs)
        Nothing ->
          let
            (new_l,vs' ) = dftrav l (Map.insert idx new_n vs)
            (new_r,vs'') = dftrav r vs'

            new_n = do
                      x'  <- f x
                      ll' <- new_l
                      rr' <- new_r
                      return (N idx x' ll' rr')
          in (new_n,vs'')
```

5.5.2 G2: Breadth-First

Definition 5.7. A breadth-first traversal visits the nodes at level i before the nodes of level $i + 1$.

The traversal is depicted in Figure 5.3, where the edges are numbered in the order they are followed. The traversal visits the node in the order `ABCDEFGHI`. The algorithm is given below.

	Okasaki's `bfnum`	Our `bftrav`
function signature	`T a -> T Int`	`(a->b) -> T a -> T b`
containers used	queue implemented using list	batched queue or stack[a] list or set for visited nodes
works on	tree	tree, acyclic, and cyclic graphs

Table 5.1: Breadth-first numbering (tree) and traversal (graph)

[a]to switch between breadth- and depth-first

```
Input  : Graph
Output: List of reachable nodes from start node
initialise empty result list;
initialise empty queue q;
add starting node to q;
bftrav(q);
proc bftrav(q)
    while q is not empty do
        n ← dequeue(q);
        if n not visited then
            append n to result list;
            mark n as visited;
            for each neighbour n' of n do
                enqueue(q,n');
            end
        end
    end
```

Algorithm 2: Breadth-first traversal

Implementation A strategy must satisfy the identity property by returning the same data structure initially passed to it. This leads to a challenge in implementing a breadth-first strategy: the reconstruction of the graph afterwards. Using the old strategy formulation from the earlier strategies library version (`parallel-1.x`) where `type Strategy = a -> ()`, there was no need to rebuild the graph. However, with the intent on preserving the compositionality benefit of the new formulation (from `parallel-3.x` and onwards) where a strategy type is defined as `a -> Eval a`, we implement a graph reconstruction solution.

Graph reconstruction after traversal Okasaki (2000) describes an elegant solution for breadth-first numbering of a tree. It is based on a queue-based breadth-first tree traversal algorithm and on generalising the problem of breadth-first tree traversal to breadth-first forest traversal. Our implementation of breadth-first graph traversal strategy is based on Okasaki's implementation of breadth-first numbering which preserves the structure after traversal. The definition is extended and is summarised in Table 5.1.

Breadth-first traversal order container The breadth-first traversal requires the use of an intermediate queue data structure to arrange the traversal level-by-level ensuring a breadth-first order. We generalise the implementation such that the container is parameterisable. Therefore, passing a stack instead of queue results in a depth-first traversal (version G1.2 – `DF_Stack` from Table 5.2) which is equivalent to the container-less depth-first traversal (version G1.1 – `DF_None`). This also helps to switch between the two orders in a hybrid version, which is important for efficient parallelism, described later. The typeclass interface for the container is given below. Three implementing types are queue, batched queue and stack. We implement a batched queue based on (Okasaki, 1996) which is an improvement compared to using a single list to implement queue. Batched queue uses a pair of lists, one for enqueue operation (*enq*) and one for dequeue (*deq*). If the *deq* is empty, it is replaced by reversing *enq* and resetting *enq*.

Listing 5.3: Traversal order container interface

```
class Container c where
  -- add item to container
  (+>)::a -> c a -> c a
  -- remove item from container
  (-<)::c a->(a,c a)
  -- check if container is empty
  isempty::c a->Bool
  -- return an empty container
  emptyc::c a
```

Defining a lazier version of the Eval monad to ensure termination for cyclic graph. Strategic traversals are arranged using the monadic bind operator (`>>=`), either explicitly or through syntactic sugar, e.g. through the `do` or applicative style programming. The following three arrangements are equivalent.

```
-- Property: 1 == 2 == 3
-- 1. explicit bind operator
n = f x >>= \x' ->
    l   >>= \l' ->
    r   >>= \r' ->
    return (N nid x' l' r')
-- 2. do notation
n = do
      x' <- f x
      l' <- l
      r' <- r
      return (N nid x' l' r')
-- 3. applicative style
n = N nid <$> f x <*> l <*> r
```

The bind operator in the Eval monad imposes a sequential evaluation order in which the expressions get evaluated. The ability to define circular data structures depends on laziness. Graph instances with cycles require laziness but the added strictness introduced by the Eval monad instance definition causes non-termination problem.

We demonstrate the problem with a simple example.

1. Define a simple data type which can allow circular reference.

```
type NId=Int
data N a = N NId a (N a) | E
```

The above definition is equivalent to that of a list type, except that we encode an ID to uniquely identify each data constructor to detect any shared references and cycles.

2. A non-circular instance of the new type.

```
non_circ_ds::N Char
non_circ_ds = N 1 'a' (N 2 'b' (N 3  'c' (N 4 'd' E)))
```

3. A circular instance of the type (depicted in Figure 5.4, where there is a circular reference from node `d` to `a`).

```
circ_ds::N Char
circ_ds = N 1 'a' (N 2 'b' (N 3  'c' (N 4 'd' circ_ds)))
```

In Listing 5.4, a traversal strategy on the simple data type works fine on `non_circ_ds` but will not terminate on a circular instance(i.e. `circ_ds`), since the default bind definition is strict on the pattern match performed on `Done x`. In Listing 5.5, we implement a lazier version of the Eval monad bind operator to work around this issue.

Listing 5.4: Strict (default) bind non-termination issue.

```
-- default (strict) bind (from Control.Parallel.Strategies)
data Eval a = Done a
Done x >>= k = lazy (k x) -- pattern 'Done x' makes >>= strict.

-- using strict bind through do notation
-- works on non_circ_ds, but NOT on circ_eg
strat_noncirc::NFData a => Strategy a -> Strategy (N a)
strat_noncirc f E          = return E
strat_noncirc f (N k x ns) = do
  x'  <- rparWith f x
  ns' <- strat_noncirc f ns
  return $ N k x' ns'
```

Listing 5.5: Lazy bind example on a simple circular data structure.

```
-- lazy bind
data Eval a = Done { runDone :: a }
m >>== k = k (runDone m)

-- strategy that will work on circular ds due to
-- lazier bind
strat_circ::NFData a => Strategy a -> Strategy (N a)
strat_circ f ns = evalcirc Map.empty ns
  where
    evalcirc vs E          = return E
    evalcirc vs (N k x ns) =
```

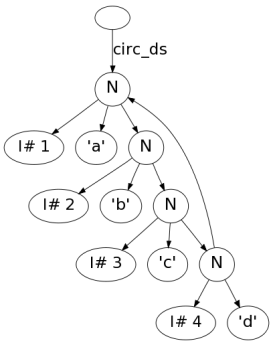


Figure 5.4: ghc-vis: circular data structure

```
12        case Map.lookup k vs of
13          Just n  -> n
14          Nothing ->
15            let
16              vs'    = Map.insert k new_n vs
17              new_n = rparWith f x     >>= \ x'   ->   -- strict
18                      evalcirc vs' ns >>== \ ns' ->   -- lazy
19                      return $ N k x' ns'
20            in new_n
```

The main issue from Listing 5.5 is to determine when a circle exists, in this case, using a container to keep references (`NId`), and identifying where too much strictness does not work for circular definitions and appropriately change the stricter bind to a lazier one (Line 18).

It is crucial to deal with circular definitions appropriately in lazy languages. For instance, implementing `show`, `fold` or other functions for the data structure depicted in Figure 5.4 necessitates additional steps to detect loop and accordingly terminate the program (the derived versions of these functions do not work automatically and will enter in a loop). The same problem is dealt with in our definitions of graph strategies that cater for cycles.

Graph breadth-first strategy We have, until now, considered the issues of graph reconstruction after traversal, the breadth-first traversal order container, and the use of lazier bind to ensure termination of a simple data type with cycles. Listing 5.6 provides the main breadth-first strategic traversal function. The main point to highlight is its more elaborate implementation compared to the previous container-less depth-first traversal. In the breadth-first implementation, traversal starts by adding the root node to a new queue container. This step is repeated recursively, level-by-level for each connected node (Line 4 and 18). This effectively arranges the order we want to achieve. Building on the queue-based breadth-first tree traversal idea, `evalGraph'` takes the queue of graph nodes and the visited node map container as arguments, and dequeues one node at a time. The node is checked

if already visited (Line 12) in which case, a reference to the updated node is returned (`nb`). Otherwise, the current node ID is inserted in the visited node container and each connected node is processed recursively. Note the use of lazy bind at Line 23 and 24. This enforces the application of `f x`, i.e. a strategy application to the node value, strictly in order to enable any parallel behaviour specified in `f`, as parallelism requires a certain degree of strictness to kickstart. The remaining expressions in the sequence are delayed using the lazy bind – to allow termination for cyclic graphs. Note that this implementation is an identity on the input graph, provided `f`, i.e. the strategy, is an identity as well.

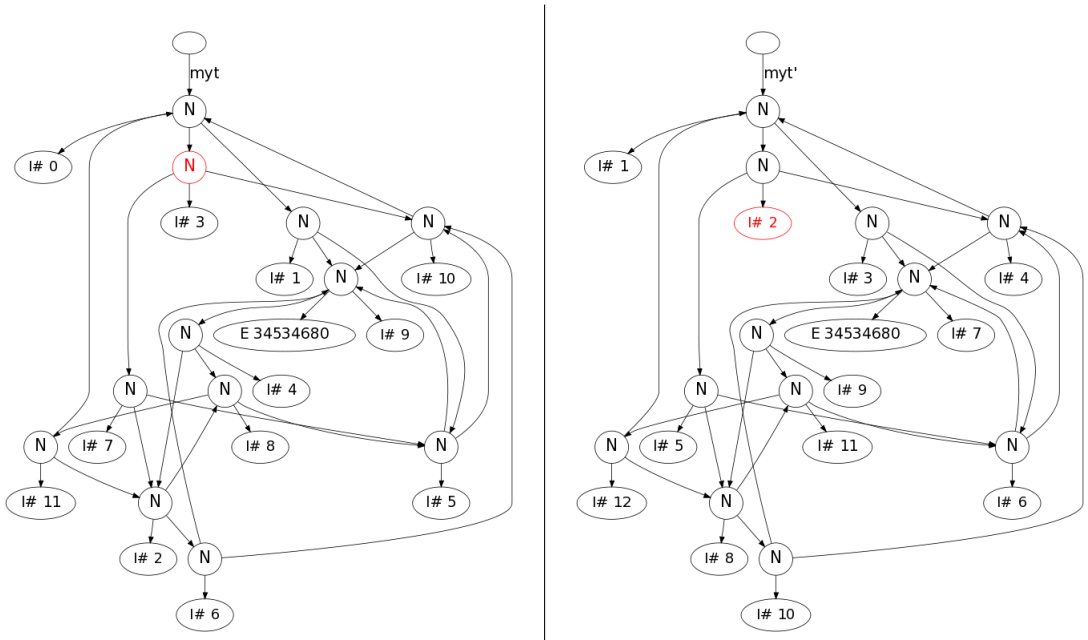
In comparison to the queue-based breadth-first traversal and numbering of tree, `evalGraph_bf` works on the three graph types defined earlier, that is, tree, acyclic and cyclic graphs.

Listing 5.6: Breath-first traversal

```
1  evalGraph_bf::NFData a => Strategy a -> Strategy (G a)
2  evalGraph_bf f g = g'
3    where
4      newq   = g +> empbq
5      (g',BQu _ _) = (-<) (evalGraph' newq Map.empty)
6
7      evalGraph' q vs          -- q queue structured container
8        | isempty q = emptyc   -- vs accumulated list of nodes
9        | otherwise =
10       case (-<) q of
11         (E          ,ts) -> (return E) +> (evalGraph' ts vs)
12         (N idx x l r,ts) -> case Map.lookup idx vs of
13            Just nb -> let
14                         ts'= evalGraph' ts vs
15                       in nb +> ts'
16            Nothing -> let
17                         vs' = Map.insert idx new_n vs
18                         q'  = evalGraph' (r +> (l +> ts)) vs'
19                         (new_r,q'') = (-<) q'
20                         (new_l,ts') = (-<) q''
21                         -- f x apply strategy on node
22                         new_n = f x   >>= \x'  ->  -- strict
23                                 new_l >>== \ll' ->  -- lazy
24                                 new_r >>== \rr' ->  -- lazy
25                                 return (N idx x' ll' rr')
26                     in new_n +> ts'
```

Preserving traversal homomorphic and strategy identity property Understanding the behaviour of lazy programs is non-trivial. The correctness of functions is verified by a number of tests by checking pre- and post-conditions on traversal. In particular, traces, command-line debugging and graphical visualisation tools help in confirming expected behaviour. Circularity is best depicted in Figure 5.5. The result of performing an operation `f` on each node of the graph using a breadth-first strategy (`travGraph_bf f`) shows that: 1) the structure is preserved; 2) the strategy

application terminates with graphs containing cycles.



(a) before (b) after

Figure 5.5: Graph structure preserved on traversal.

5.6 Limiting Parallelism

The depth-first and breadth-first strategies generate unconstrained parallelism, leading to overhead for large graphs – the same issue we faced for tree strategies. Limiting parallelism entails sparking sub-graphs. Next we describe a number of graph strategies that efficiently throttle parallelism.

5.6.1 G3: Implementing a Hybrid Traversal Order

Depth-first traversal incurs a lower memory overhead as opposed to breadth-first since the latter uses an additional intermediate data structure to organise the traversal order. As such, breadth-first requires more space but can be faster if a solution is not far from the starting node. On the other hand, depth-first may be slower to find a path to an element in a search operation especially if the graph goes very deep in one path that does not contain a solution.

A combination of these two traversal strategies may yield the benefits of both. This technique is used in artificial intelligence, for example, in a technique called iterative deepening search which combines positive elements of breadth- and depth-first traversal (Korf, 1985).

We implement a hybrid version of graph traversal that proceeds breadth-first until a depth or level l in the graph, from which point, it resumes or continues in a depth-first order. This combination of breadth-first and depth-first in that order is particularly useful in exposing parallelism across the graph, before switching to depth-first, which is generally faster given its simple implementation.

In Listing 5.7, using a tree graph example, nodes 1, 2 and 3 are visited in breadth-first order, then, at $l = 2$, traversal proceeds depth-first with the output at Line 11.

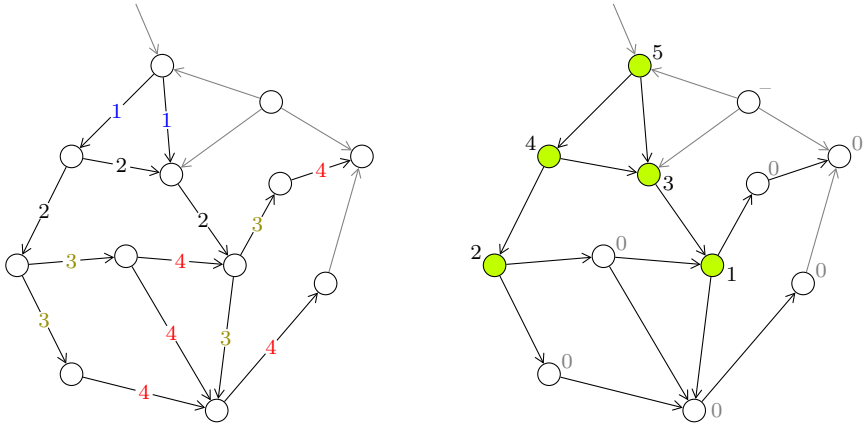
Listing 5.7: Example of breadth-depth hybrid traversal output.

```
1            1                    -- 1
2          /   \
3        /       \
4      2           3              -- 2 <- switch here
5     /  \        /   \
6    4    5      6     7          -- 3
7   / \  / \    / \   / \
8   8 9 10 11  12 13 14 15        -- 4
9
10 bdf: bf -- 1,2,3
11      df -- 4,8,9, 5,10,11, 6,12,13, 7,14,15
```

The hybrid traversal in Listing 5.8 requires an additional parameter to activate the switch from breadth- to depth-first. This is implemented by a conditional check at Line 12. To maintain the breadth-depth order, the graphs in the queue are held until the depth-first traversal returns for each node where the switch happens, before appending the result with the breadth-first part of the traversal.

Listing 5.8: Hybrid traversal (BDF)

```
1  travGraph_bdf::Int->(a->b)->G a->G b
2  travGraph_bdf depth f g = g'
3   where
4    newq  = (g,0) +> empbq
5    ((g',_),BQu _ _) = (-<) (bftrav Map.empty newq)
6
7    bftrav vs q
8      | isempty q = emptyc
9      | otherwise =
10        let ((n,d),qs) = (-<) q
11        in
12          -- condition to switch from breadth- to depth-first --
13          if d==depth then
14            -- continue with df --
15            let (n',vs') = dftrav n vs
16            in (n',d) +> (bftrav vs' qs)
17          else
18            -- proceed with bf --
19            case n of
20              E           -> (E,d) +> (bftrav vs qs)
21              N idx x l r ->
22                case Map.lookup idx vs of
23                  Just n2 -> (n2,d) +> (bftrav vs qs)
24                  Nothing ->
```



(a) depth (b) fuel

Figure 5.6: Depth threshold vs. fuel passing in graphs.

```
                    let
                      vs'    = Map.insert idx new_n vs
                      d1     = d+1
                      q'     = bftrav vs' ((r,d1) +> ((l,d1) +> qs))
                      ((r',_),q'')  = (-<) q'
                      ((l',_),q''') = (-<) q''
                      new_n = N idx (f x) l' r'
                    in (new_n,d) +> q'''

dftrav E              vs = (E,vs)
dftrav (N idx x l r) vs =
  case (Map.lookup idx vs) of
    Just n2 -> (n2,vs)
    Nothing ->
      let
        (l',vs' ) = dftrav l (Map.insert idx new_n vs)
        (r',vs'') = dftrav r vs'
        new_n     = N idx (f x) l' r'
      in (new_n,vs'')
```

5.6.2 G4: Depth Threshold

This technique of limiting parallelism is similar to that described in the previous chapter only here it is applied to a graph. The base traversal order of this strategy is breadth-first so it proceeds level-by-level inside the graph. As we proceed at each level, a counter is incremented. Sparks are created as long as the counter is less than a specified threshold. Figure 5.6(a) depicts the how depth is determined for graph (labels on edges represent the current depth). The implementation is more complicated than depth-thresholding for tree structure. The queue data structure used to arrange the traversal order keeps a tuple of current node and its depth in the graph: $[(n_1, d1), (n_2, d2), (n_3, d2)...(n_i, dX)]$, where i is number of nodes and dX the maximum depth or height of the graph.

5.6.3 G5: Fuel Passing

The fuel concept is efficient in controlling parallelism as shown in the previous chapter. The implementation approach is slightly different for graphs. We do not perform an annotation run on the graph structure to attach fuel to the graph nodes, consequently removing the overhead. We use a fuel passing mechanism instead of fuel splitting. Fuel is passed from node to node. Starting with an initial amount of fuel, each unvisited node takes one unit of fuel and passes the remaining to the next nodes during traversal as illustrated in Figure 5.6(b). Only nodes that have retained fuel are sparked. Given that the graph traversals are more elaborate, this choice of a simple port of the fuel concept to graphs is first considered.

The fuel passing implementation (visualised in Fig. 5.6(b)) ensures the following properties:

1. Starting node n_0 is given an amount of fuel $0 < f < n$ where n is the number of vertices.
2. The current node keeps one unit of fuel and passes the remaining $f - 1$ to the next neighbour. If $n_1, n_2, ...$ are neighbours of n_0, then fuel passed to n_i is $f_i = f - i$.
3. If the traversal proceeds in the order $n_0, n_1, n_2, ...$, the fuel available at each node in that order is one less than the previous node $f, f - 1, f - 2, ...0$, until fuel is 0; the condition stops any further parallelism generation.
4. Sub-graphs with $f > 0$ are sparked in parallel.

5.7 Traversal Strategies Summary

Graph traversals require additional administrative data structures to: a) keep track of visited nodes (the visited nodes container); b) and, optionally, to enforce a particular order (the traversal order container). We have described two common traversal strategies, based on depth-first and breadth-first order, for graph data structures represented using a recursive tree definition. Depth-first can be implemented without an intermediate container to arrange order and using the dynamic call stack implicitly. By default traversal proceeds in a depth-first order. We have described three more advanced graph strategies that attempt to reduce overhead by throttling parallelism. Table 5.2 summarises the different traversal strategies for graphs.

	Trav. func.	Trav. order	Inter. container	Sparking	Param
G1.1	DF_None	Depth-first	Container-less	Unconstrained	
G1.2	DF_Stack	Depth-first	Stack	Unconstrained	
G2	BF_BQu	Breadth-first	Batched-queue	Unconstrained	
G3	BDF	Hybrid breadth-depth	Batched-queue[a]	Throttled	l
G4	DepthThres	Breadth	Batched-queue	Throttled	d
G5	FuelPassing	Breadth	Batched-queue	Throttled	f

Table 5.2: Summary of graph traversal strategies

[a]until level l, container-less thereafter

5.8 Performance Results

In this section, we report on the performance results of the different traversal strategies used to implement graph search algorithms. The machine setup for this experiment is the same desktop-class 8 core machine used in Chapter 3 (Section 3.4). We discuss the results and explain their limitations.

5.8.1 Graph Search Algorithm

Search algorithms on a graph proceed by traversing the data structure in a specific order. This can be implemented by composing a traversal function taking a predicate as an argument and an outer fold to return true or false if an element is contained in the graph.

```
travG::(a->b)->G a->G b
cond::a->Boolean -- predicate condition
reduce::G a->Boolean
reduce = foldr (||) False

searchG = reduce . travG cond
```

5.8.2 Input Set

We use the following two input sets for performance test. These are generated through a graph generator function, which takes as parameters: the number of nodes, a boolean to specify if cycles are permitted, a boolean to specify if self-loop is permitted, node values (which may reflect computation at node level), and a ratio of number of connections/edges for a node (i.e. node outdegree ratio).

Graph input 1: Acyclic, 20k nodes, 22231 edges, 2232 shared nodes

Graph input 2: Cyclic, 20k nodes, 31997 edges

We test on a single connected components, though graphs with multiple connected components, are easier to arrange such that different component with different characteristics can be traversed using a particular traversal function.

5.8.3 Traversal Performance

Table 5.3 summarises the runtime and speedup of the different traversal strategies both for acyclic and cyclic graphs. A discussion of performance for both follows.

Acyclic Graph

The first measurements on acyclic graphs show promising results in Figure 5.7. Table 5.4 complements the results with additional runtime statistics including spark numbers and heap allocation. Given the large number of nodes for the input graph, the first three strategies (`DF-None`, `DF-Stack` and `BF-BQu`), which do not limit parallelism in any way, create sparks for every node, and thus do not perform well. The depth-thresholding (`Depth-Thres`) strategy shows good performance, due to its straight-forward throttling mechanism. The hybrid version (`BDF`), as expected, is efficient in creating useful parallelism during the breadth traversal. It also incurs less overhead associated with the breadth-first traversal as it switches to depth-first after a specified level. This can be seen from Table 5.4 which shows it incurs the lowest heap allocation after the container-less depth-first (`DF-None`). Most notably, the fuel passing method works best among all the strategies, which could be attributed to the simple fuel distribution model applied in this implementation, without performing an annotation run as previously done for tree strategies. Overall, the control mechanisms work very efficiently on acyclic graphs which shows that our techniques from the previous chapter is applicable to other data structures. In particular, the best speedup is near 5 on an desktop-class 8-core machine. This indicates that we can expect further speedup on the server-class machine used in Chapter 4. The more advanced strategies have potential for further tuning of the parameters, for e.g., through a heuristic-based method to select appropriate values for l, d and f.

Cyclic Graph

As expected, the cyclic graph traversals take longer to run as depicted in Figure 5.8. This is due to the number of cycles that may be present in the graph. Interestingly, we note better performance for `DF-None`, `DF-Stack` and `BF-BQu` compared to acyclic performance, for e.g., the best speedup is 3 compared to 2 for similar strategy on acyclic graph. However, the main issue to report at this point is the performance

Acyclic

	DF-None		DF-Stack		BF-Bqu		Hybrid-BDF		Depth-Thres		Fuel-Pass	
#pe	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP
1	115.28	0.97	115.78	0.96	115.38	0.97	115.20	0.97	115.12	0.97	115.55	0.97
2	109.76	1.02	111.62	1.00	75.58	1.48	68.23	1.64	68.19	1.64	70.51	1.58
3	84.82	1.32	88.46	1.26	72.00	1.55	52.67	2.12	52.61	2.12	49.86	2.24
4	75.05	1.49	80.56	1.39	70.58	1.58	36.75	3.04	42.25	2.64	39.43	2.83
5	69.34	1.61	76.41	1.46	69.64	1.60	34.80	3.21	30.32	3.68	33.13	3.37
6	64.74	1.73	73.81	1.51	69.10	1.62	32.03	3.49	27.63	4.04	28.35	3.94
7	60.99	1.83	71.96	1.55	68.76	1.62	30.49	3.66	25.54	4.37	25.43	4.39
8	57.82	1.93	70.82	1.58	68.09	1.64	25.91	4.31	25.60	4.36	23.14	4.83

Cyclic

	DF-None		DF-Stack		BF-Bqu		Hybrid-BDF		Depth-Thres		Fuel-Pass	
#pe	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP	RT	SP
1	141.81	1.20	144.58	1.07	130.84	1.19	133.36	1.08	140.93	1.00	136.93	1.05
2	99.40	1.71	100.34	1.54	78.36	1.99	150.30	0.96	148.47	0.95	149.43	0.96
3	65.18	2.62	82.33	1.88	60.21	2.59	152.91	0.94	151.50	0.93	149.80	0.96
4	63.80	2.67	73.88	2.09	51.97	3.00	158.44	0.91	149.19	0.95	151.47	0.95
5	58.23	2.93	69.48	2.22	49.56	3.14	150.10	0.96	152.53	0.93	153.12	0.94
6	59.92	2.84	67.63	2.28	54.61	2.85	152.76	0.94	149.32	0.95	150.39	0.96
7	58.91	2.89	67.30	2.30	53.64	2.90	150.61	0.96	151.13	0.94	146.98	0.98
8	58.85	2.90	67.75	2.28	51.59	3.02	150.91	0.96	149.95	0.94	149.82	0.96

Table 5.3: Graph search – Runtime and Speedup.

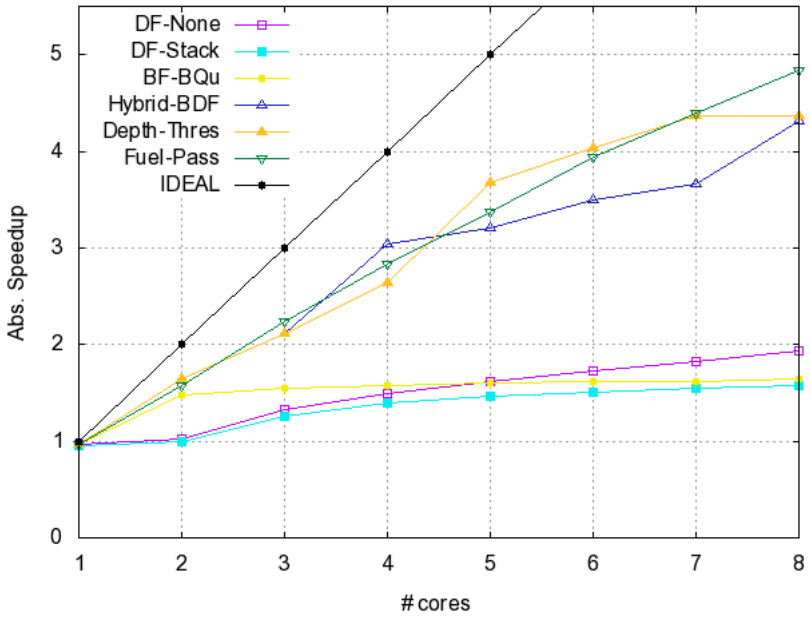


Figure 5.7: Acyclic graph traversals speedup on 8 cores, 20k nodes.

	created	converted	overflowed	param	heap alloc
DF_None	20000	10882	9118		103MB
DF_Stack	20000	8611	11389		108MB
BF_Bqu	20000	8618	11354		112MB
BDF (hybrid)	35	33	0	l=5	104MB
ThresDepth	35	33	0	d=5	117MB
FuelPass	2445	2357	0	f=1000	123MB

Table 5.4: Sparks and heap allocation statistics.

of the three advanced strategies. Given that we use the lazy bind in the implementation, and due to cycles being present in the graph, sparks are created, but parallel execution hardly kicks in. The interplay of laziness and parallel evaluation is a tricky one. While we managed to ensure termination for cyclic instances, this comes at the cost of delaying and, possibly, enforcing a sequential ordering.

In the implementation, we make two uses of laziness: one to use a tree-based representation that enables cyclic graph instances, i.e. circular data structure, which will not work in a strict language; and one to use laziness in the fuel strategy (laziness for performance, as in the previous chapter). The latter should still carry over to graphs, and the results on acyclic graphs show this. The problem is the complexity from the interplay of both uses of laziness and this has not been fully resolved and explains the poor results for a fuel strategy for cyclic graphs. A good direction for improving on these results would be to add explicit strictness annotation in the cyclic strategy definitions, which when used in combination with the lazier bind, would provide just the amount of eagerness needed to start parallel evaluation. Alternatively, we can define the cyclic graph strategies using the earlier formulation of

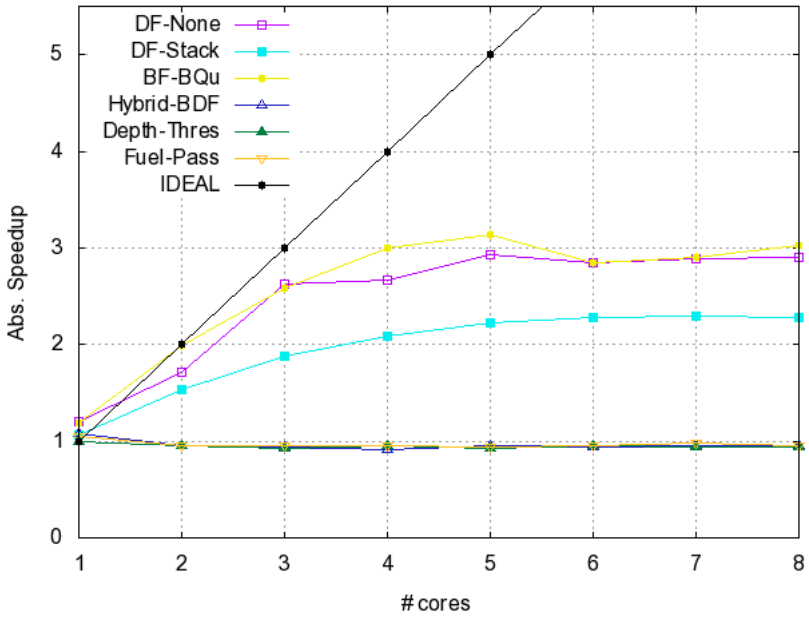


Figure 5.8: Cyclic graph traversals speedup on 8 cores, 20k nodes.

evaluation strategies, which do not use the Eval monad, thus no implicit sequencing of actions is encoded, making it easier to implement.

5.9 Summary

In this chapter, we have implemented a number of traversal strategies for graph data structures which uses a tree-based representation to facilitate adaptation of previously defined techniques for trees. As noted by (King and Launchbury, 1994), it is harder to work with recursive tree representation for graph data structures, and this was illustrated by a number of issues we had to deal with in the implementation, in particular, a lazier Eval monad instance use case. Most importantly, we were able to show that the previous parallelism control mechanisms are not restricted to trees only. The techniques work well with acyclic graphs as shown in the results. The use of a hybrid traversal order and fuel passing proves to be more efficient in controlling parallelism. Finally, we need to find a fix for the advanced traversals when used on cyclic graphs, by using the new formulation of evaluation strategies, thus preserving its compositionality benefit.

Chapter 6

Conclusion

6.1 Summary

To avoid many of the difficulties of traditional parallel programming, this thesis investigates the benefits of high-level parallel programming models. Through their use of high level programming models, most of the complexity of parallel programming is shifted to the system, giving way to a more abstract model to the programmer. A high level model removes a lot of issues and challenges, such as synchronisation, communication, task distribution, and load balancing, associated with conventional low level models. Modern mainstream languages, like C# and Go (Campbell et al., 2010; Google, 2015), are introducing higher level constructs, for example, one that only requires a programmer to identify parallelism, and the management of the parallel execution is dealt with by the system. It is widely acknowledged (Hammond and Michaelson, 2000) that functional languages are particularly well-suited for parallel programming as they are free from side-effects, offer powerful abstractions and a semi-explicit model with an adequate balance of control required by application programmers. As basis for our main studies on high-level parallelism, this thesis surveys three modern functional languages and their parallelism support in Chapter 2 before focusing on one (GpH) for more detailed study. A detailed empirical study of their performance and programmability is given in (Totoo et al., 2012).

Good parallel performance can be achieved through the use of inherently parallel data structures in algorithms. This approach to introduce data-oriented parallelism requires little change to existing sequential algorithms, and hence can be applied to a wide variety of applications. The main emphasis of this thesis is the development of advanced evaluation strategies over commonly used data structures, which provide a clear separation of coordination from computation aspects of a parallel program. We study lists in Chapter 3, trees in Chapter 4 and graphs in Chapter 5. The main strategies are summarised in Table 6.1. These strategies allow top-level par-

allelisation of a number of applications including the Barnes-Hut and graph search algorithms in a minimally intrusive way, making parallelism more accessible.

Data structure	Strategy	
	General purpose	*Custom*
list	parList (Sec. 3.4)	chunking/clustering (Sec. 3.4.1)
tree	parTree (Sec. 4.5)	depthThres (Sec. 4.7) sizeann, lazysize (Sec. 4.8, 4.9) fuel-based control (Sec. 4.10)
graph	depthFirst (Sec. 5.5.1) breadthFirst (Sec. 5.5.2)	hybrid, depthThres (Sec. 5.6.1, 5.6.2) fuelpassing (Sec. 5.6.3)

Table 6.1: Evaluation strategies implementation overview

Comparison: By systematically implementing a naive and a more realistic application in three parallel Haskell variants in Chapter 3, we compare the parallelisation methodology in GpH using evaluation strategies against two other models in Haskell: the Par monad, a deterministic model built on top of Concurrent Haskell; and Eden, which is targeted for distributed memory and offers scaling on clusters. Our comparison highlights the difference in implementing various parallel list map operations in all models, and, in particular, evaluates the use of evaluation strategies as a model that offers the highest level of abstraction, enabling the specification of parallel behavioural code separate from the algorithmic code. While the Par monad and Eden offer more explicit control to certain aspects of parallel execution, GpH delegates all aspects of parallel execution to a sophisticated runtime system. It comes at the cost of limited control over parallel execution, but it is very well suited for data-structure-driven parallel evaluation. GpH retains the important property of compositionality, and this is further exemplified through granularity control of parallel list evaluation – we consider three ways of introduing chunking: explicit, strategic and an even more generic implicit clustering. In this comparison, we observe the ease of introducing and modifying clustering strategies over data structures.

Tree-based parallelism: The first part of Chapter 3 motivates the use of an alternative representation for lists, which are inherently sequential data structures. We stress the advantage of using tree-based representations. In Chapter 4 we demonstrate that more flexible parallel control mechanisms can be implemented for tree-based data structures through a number of basic and advanced tree-based evaluation strategies.

Non-strictness and parallelism: One central issue tackled in this thesis is whether non-strictness can be beneficial for the parallel manipulation of data structures. We use laziness to improve parallel performance. Laziness and, for the first time, circular programming are used in the coordination code to achieve additional flexibility. A number of flexible parallelism control mechanisms are developed in the form of evaluation strategies. The thesis highlights the use of circular programming techniques

in a number of occasions in our implementation: Section 4.10.1 (circular distribution of fuel using a giveback mechanism), Section 5.5.2 (circular data structure – cyclic graph), both in instances of data types that allow circular references, and in expressions depending on each other in function definitions. Finally, we demonstrate improved performance on a benchmark program and two non-trivial applications, in particular, with irregular trees.

Graph strategies: Building on the work from Chapter 4, we extend tree strategies for graphs in Chapter 5. We implement a number of traversal strategies for graphs represented as recursive tree structures. Laziness is further exploited as we demonstrate that strategies requiring a local context, through being less eager in evaluating sub-expressions (e.g. lazy size computation in Section 4.9), have same or better performance (and less overhead) than those requiring a global context, i.e. more information (see Table 4.1). Cyclic graph instances require non-strictness as a language property. Several challenges in implementing efficient parallel algorithms in a purely lazy functional language are addressed. For instance, understanding the behaviour of lazy programs is difficult. Through a systematic approach, the use of profiling and tools that help to analyse programs and live data structure evaluation, we are able to identify issues such as memory leaks, and non-termination. Specifying parallel behaviour is more subtle but can be achieved by using a relaxed bind operator for the Eval monad. Defining evaluation strategies on graph data structures requires reconstruction of the graph, which we have implemented by extending a technique of tree reconstruction after breadth-first traversal by Okasaki (2000), and making sure the structure is preserved for all graph instances.

6.2 Contributions

This section summarises the main contributions of the thesis.

1. Performance evaluation of three parallel Haskell implementation (Totoo and Loidl, 2014a).

   We have assessed the advantages and disadvantages of high-level parallel programming models for multi-core programming by implementing two versions of the n-body problem. We have compared three different parallel programming models based on parallel Haskell, differing in the ways how potential parallelism is identified and managed. We have assessed the performance of each implementation, discuss in detail the sequential and parallel tuning steps leading to the final versions, and draw general conclusions on the suitability of high-level parallel programming models for multi-core programming.

2. Lazy data-oriented evaluation strategies (Totoo and Loidl, 2014b).

   We have developed a number of flexible parallelism control mechanisms in the form of evaluation strategies for tree-like data structures implemented in GpH. We have demonstrated that the use of laziness and circular programs in the coordination code can achieve additional flexibility and better performance. We have employed heuristics-based parameter selection to auto-tune these strategies for improved performance on a shared-memory machine without the need for programmer-specified parameters.

3. Graph evaluation strategies.

   By systematically extending our tree-based techniques to graph structures, and allowing for shared references and cycles, we have extended our advanced tree strategies to graphs. We have implemented a number of traversal strategies that work on acyclic and cyclic graphs. We make use of laziness again in the core implementation, specifically to allow cyclic graph instances by resolving to a less eager bind operator for the Eval monad, and to control traversals defined on them.

6.3 Limitations and Future Work

The source code for the strategies and applications used in this thesis is available online[1], but these need to be consolidated in a single package. To facilitate up-take of our techniques we will bring our code onto Hackage as a library of parallel data structures in the form of a `pardata` package consisting of the advanced strategies defined for list, tree and graph data structures.

Evaluation strategies have proven to be an effective model to specify high level parallelism, which works particularly well on data-oriented parallelism, for data-structure-driven parallelism, in the way we used them in this thesis. The results of the evaluation strategies that we developed demonstrate a clear improvement in flexibility and performance. But these can be further improved through a number of measures.

To further improve the results presented in Chapter 4, the heuristics for parameter selection can be enhanced. In particular, similar heuristics used to automatically determine the right parameter to tree strategies can be extended and developed for graph structures.

The prospect of using an attribute-grammar style (Swierstra et al., 1999) of spec-

[1] http://www.macs.hw.ac.uk/~pt114/phd-thesis/

ifying the propagation of synthesised and inherited parameters through the data structure as a generalisation of the fuel-based control of parallelism can be explored. This style could provide a framework for fine-tuning parallelism, while still defining behaviour and desired properties in a high level.

Additionally, the compositionality of the mechanisms can be further improved. It would be desirable to provide high-level constructors, to freely combine a general fuel mechanism, for driving the parallelism, with a lookahead mechanism, for controlling the precision of contextual information, and a giveback mechanism, for adding flexibility in managing the parallelism.

To reduce the annotation pass overhead currently incurred in the tree strategies, an annotation-free fuel strategy may be investigated. This could be achieved, for example, by using techniques to combine the separate annotation and parallelism generation traversals into one.

Scaling to many-cores and clusters: We compared GpH with Eden in Chapter 3. Eden has the advantage of allowing us to scale in a distributed memory setting. One crucial advantage of GpH is it minimises code changes to the sequential algorithm. This fact can be exploited for a large number of applications. More importantly, as the model is not tied to the underlying system, using GUMSMP (Aljabri et al., 2014b), a multilevel parallel Haskell implementation for clusters of multi-cores, allows extending the use of strategies on clusters and thus they can be tested on larger data input. Unlike Eden, GUMSMP would allow us to re-use the same strategies and not different skeletons.

Another direction of further work, though not covered in this thesis, is to improve our runtime support for hierarchical, heterogeneous parallel architectures, e.g. clusters of multi-cores, and to integrate the different Haskell variants into one unified language that makes use of these variants on different levels in the hierarchy. Eden, based on a distributed-memory model, is a natural match with clusters, whereas GpH and Par monad are natural matches for physical shared-memory architectures. GpH also supports virtual memory, which can be efficiently exploited on closely connected clusters.

Improved Graph Strategies: The main purpose of Chapter 5 is to demonstrate that the techniques applied for tree strategies can be adapted to work with graphs. By implementing the depth threshold and fuel-passing techniques for limiting parallelism, and further relying on lazy evaluation, especially for cyclic graphs, we have been successful in achieving this primary aim. There are, however, opportunities for improvement. The graph strategies firstly need to be optimised. Further performance tuning and the application of heuristic-based methods to graph strategies

are envisaged.

The graph results presented in Chapter 5 are based on graph searches implemented using different traversal strategies. We have developed a naive implementation of the maximum clique algorithm which is also the basis for the SICSA Multicore Challenge Phase III[2]. We try to map the algorithm in such a way that the traversal strategies can be used out-of-the-box. However, this program re-structuring is not very efficient. And it also indicates that it is not always possible to use evaluation strategies directly for problems that do not exhibit top-level parallelism. We plan to refine the current algorithm, use the DIMACS[3] dataset to verify result and report on performance. This could be used as a basis for performance comparison against other graph benchmark, for e.g., the Graph500[4] benchmark suite.

[2]SICSA Multicore Challenge Phase III - http://www.macs.hw.ac.uk/sicsawiki/index.php/Challenge_PhaseIII

[3]DIMACS - http://dimacs.rutgers.edu/

[4]The Graph 500 List - http://www.graph500.org/

Bibliography

Aarseth, S. J. (2003). *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press.

Adapteva (2015). The Parallella Computer. http://www.adapteva.com/parallella/.

Aljabri, M., Loidl, H.-W., and Trinder, P. (2014a). Distributed vs. Shared Heap, Parallel Haskell Implementations on Shared Memory Machines. In *Draft Proc. of Symp. on Trends in Functional Programming*, TFP'14, Univ. of Utrecht, The Netherlands.

Aljabri, M., Loidl, H.-W., and Trinder, P. W. (2014b). The Design and Implementation of GUMSMP: A Multilevel Parallel Haskell Implementation. In *Proc. of the 25th Symp. on Implementation and Application of Functional Languages*, IFL '13, pages 37:37–37:48, New York, NY, USA. ACM.

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G. L., and Tobin-Hochstadt, S. (2008). The Fortress Language Specification Version 1.0. Technical report, Sun Microsystems, Inc.

Allison, L. (1989). Circular programs and self-referential structures. *Software: Practice and Experience*, 19(2):99–109.

ANL (2012). The Message Passing Interface Standard. http://www.mcs.anl.gov/research/projects/mpi/.

Armstrong, J., Virding, R., Wikström, C., and Williams, M. (1995). *Concurrent Programming in Erlang*. Prentice Hall, second edition. ISBN 978-0135083017.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., and Vanneschi, M. (1995). P^3L: a Structured High-level Programming Language and its Structured Support. *Concurrency and Computation: Practice and Experience*, 7(3):225–255.

Backus, J. (1978). Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641.

Bader, D. A., Gilbert, J. R., Kepner, J., and Madduri, K. (2015). HPC Graph Analysis. http://www.graphanalysis.org/.

Barnes, J. and Hut, P. (1986). A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449.

Barthe, G. and Coquand, T. (2002). An introduction to dependent type theory. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 1–41, London, UK, UK. Springer-Verlag.

Belikov, E., Deligiannis, P., Totoo, P., Aljabri, M., and Loidl, H.-W. (2013). A Survey of High-Level Parallel Programming Models. Technical report, Heriot-Watt University.

Benoit, A., Cole, M., Gilmore, S., and Hillston, J. (2005). Flexible Skeletal Programming with eSkel. In *EuroPar'05*, LNCS 3648, pages 761–770. Springer.

Bergstrom, L., Fluet, M., Rainey, M., Reppy, J., Rosen, S., and Shaw, A. (2013). Data-Only Flattening for Nested Data Parallelism. In *Proc. of the ACM SIGPLAN Symp. on Princ. Pract. of Par. Program.*, PPoPP'13, pages 81–92.

Berthold, J., Loidl, H.-W., and Hammond, K. (2015). PAEAN: Portable Runtime Support for Physically-Shared-Nothing Architectures in Parallel Haskell Dialects. *Journal of Functional Programming*.

Berthold, J., Marlow, S., Hammond, K., and Zain, A. A. (2009). Comparing and Optimising Parallel Haskell Implementations for Multicore Machines. *Memory*, pages 386–393.

Bird, R. (1984). Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250.

Blelloch, G. E. (1995). NESL: A Nested Data-parallel Language (version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University.

Boehm, H.-J., Atkinson, R., and Plass, M. (1995). Ropes: an alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330.

Campbell, C., Johnson, R., Miller, A., and Toub, S. (2010). *Parallel Programming with Microsoft .NET – Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press.

Cann, D. (1992). Retire fortran?: A debate rekindled. *Commun. ACM*, 35(8):81–89.

Chakravarty, M. M. T., Keller, G., Jones, S. P., and Marlow, S. (2005). Associated types with class. In *In POPL 05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press.

Chakravarty, M. M. T., Keller, G., Leshchinskiy, R., and Pfannenstiel, W. (2001). Nepalnested data parallelism in Haskell. *EuroPar 2001 Parallel Processing*, pages 524–534.

Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., Keller, G., and Marlow, S. (2007). Data Parallel Haskell: A Status Report. In *Proc. of the Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA. ACM.

Chamberlain, B., Callahan, D., and Zima, H. (2007). Parallel Programmability and the Chapel Language. *Int. Journal on High Performance Computing Applications*, 21(3):291–312.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An Object-Oriented Approach to Non-uniform Cluster Computing. In *Proc. of OOPSLA'05*, pages 519–538. ACM Press.

Chrysos, G. (2012). Intel Xeon Phi Coprocessor - the Architecture. Online.

Ciechanowicz, P., Poldner, M., and Kuchen, H. (2010). The Münster Skeleton Library: Müsli. Technical Report ERCIS Working Paper No. 7, University Münster.

Cilk (1998). *Cilk 5.4.6 Reference Manual*. MIT, Supercomputing Technologies Group MIT Laboratory for Computer Science.

Cole, M. (1991). *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA.

Cooper, P., Dolinsky, U., Donaldson, A. F., Richards, A., Riley, C., and Russell, G. (2010). Offload–automating code migration to heterogeneous multicore systems. In *High Performance Embedded Architectures and Compilers*, pages 337–352. Springer.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition.

Curry, H. and Feys, R. (1967). Combinatory Logic. Volume I. *Journal of Symbolic Logic*, 32:267–268.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113.

Diaz, J., Munoz-Caro, C., and Nino, A. (2012). A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386.

Dice, D., Hendler, D., and Mirsky, I. (2013). Lightweight contention management for efficient compare-and-swap operations. *CoRR*, abs/1305.5800.

Enmyren, J. and Kessler, C. (2010). SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proc. of the 4th Int. Workshop on High-Level Parallel Programming and Applications*, pages 5–14. ACM.

Epstein, J., Black, A. P., and Jones, S. P. (2011). Towards Haskell in the Cloud. In *Proc. of the 4th ACM Symp. on Haskell*, Haskell '11, pages 118–129, New York, NY, USA. ACM.

Erwig, M. (2001). Inductive Graphs and Functional Graph Algorithms. *Journal of Functional Programming*, 11(5):467–492.

Felsing, D. (2012). Visualization of lazy evaluation and sharing. Bachelor's thesis, Karlsruhe Institute of Technology, Germany.

Fluet, M., Rainey, M., Reppy, J., and Shaw, A. (2010). Implicitly-threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5–6):537–576.

Fluet, M., Rainey, M., Reppy, J., Shaw, A., and Xiao, Y. (2007). Manticore: A Heterogeneous Parallel Language. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*, pages 37–44, Nice, France. ACM Press.

Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

Foltzer, A., Kulkarni, A., Swords, R., Sasidharan, S., Jiang, E., and Newton, R. (2012). A meta-scheduler for the Par-Monad: composable scheduling for the heterogeneous cloud. In *Proc. of the 17th ACM SIGPLAN Int. Conf. on Functional programming*, ICFP '12, pages 235–246. ACM.

Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Friedman, D. and Wise, D. (1976). CONS Should not Evaluate its Arguments. In Michaelson, S. and Milner, R., editors, *Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, Edinburgh.

Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The Implementation of the Cilk-5 Multithreaded Language. In *PLDI98: Conf. on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

Fulgham, B. (2012). The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/.

Furber, S. (2000). *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Commun. ACM*, 35(2):96–107.

Gilbert, J. R., Reinhardt, S., and Shah, V. B. (2007). High-Performance Graph Algorithms from Parallel Sparse Matrices. In *Proc. of the 8th Int. Conf. on Applied Parallel Computing: State of the Art in Scientific Computing*, PARA'06, pages 260–269, Berlin, Heidelberg. Springer-Verlag.

Goetz, B. (2013). Lambda Expressions for the Java Programming Language. *Java Community Process*.

Gondran, M., Minoux, M., and Vajda, S. (1984). *Graphs and Algorithms*. John Wiley & Sons, Inc., New York, NY, USA.

González-Vélez, H. and Leyton, M. (2010). A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Software: Practice and Experience*, 40(12):1135–1160.

Google (2015). The Go Programming Language . https://golang.org/. Accessed: 2016-02-08.

Google Inc. (2011). Renderscript. http://developer.android. com/guide/topics/renderscript/compute.html.

Graph500 (2015). The Graph 500 List. http://www.graph500.org/.

Gregory, K. and Miller, A. (2012). *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. Microsoft Press.

Grelck, C. and Scholz, S.-B. (2006). SAC: A Functional Array Language for Efficient Multi-threaded Execution. *Int. Journal on Parallel Programming*, 34(4):383–427.

Haller, P. and Odersky, M. (2009). Scala Actors: Unifying Thread-based and Event-based Programming. *Theoretical Computer Science*, 410:202–220.

Hammond, K. (1990). *Parallel SML: A Functional Language and Its Implementation in DACTL (Research Monographs in Parallel & Distributed Computing)*. Financial Times Prentice Hall.

Hammond, K. and Michaelson, G., editors (2000). *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, UK.

Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2008). Composable memory transactions. *Commun. ACM*, 51(8):91.

Haynes, C. T. and Friedman, D. P. (1984). Engines Build Process Abstractions. In *Proc. of the ACM Symp. on LISP and Functional Programming*, LFP'84, pages 18–24, New York, NY, USA. ACM.

Hejlsberg, A., Wiltamuth, S., and Golde, P. (2003). *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Henderson, P. and Morris, Jr., J. H. (1976). A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, POPL '76, pages 95–103, New York, NY, USA. ACM.

Hoare, C. (1972). Notes on Data Structuring. In Dahl, O.-J., Dijkstra, E., and Hoare, C., editors, *Structured Programming*, pages 83–174. Academic Press.

Hudak, P., Peterson, J., and Fasel, J. H. (1999). A gentle introduction to Haskell 98. *Online tutorial*.

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J. (1992). Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164.

Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107.

IntelTBB (2012). *Intel(R) Threading Building Blocks Reference Manual*. Intel.

Jarvi, J. and Freeman, J. (2010). C++ lambda expressions and closures. *Science of Computer Programming*, 75(9):762–772.

Jones, S. P. (2007). Beautiful concurrency. *Beautiful Code: Leading Programmers Explain How They Think*, pages 385–406.

Jones, S. P., Gordon, A., and Finne, S. (1996). Concurrent Haskell. In *Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA. ACM.

Jones, S. P., Leshchinskiy, R., Keller, G., and Chakravarty, M. M. T. (2008). Harnessing the multicores: Nested data parallelism in Haskell. *Theoretical Computer Science*, pages 1–32.

Jones, S. P., Washburn, G., and Weirich, S. (2004). Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389.

Jost, S., Hammond, K., Loidl, H.-W., and Hofmann, M. (2010). Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proc. of the 37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '10, pages 223–236, New York, NY, USA. ACM.

Kaplan, H. (2001). Persistent data structures. In *In Handbook on Data Structures and Applications*. CRC Press.

Kelly, P. (1989). *Functional Programming for Loosely-coupled Multiprocessors*. Pitman/MIT Press.

King, D. and Launchbury, J. (1994). Lazy Depth-First Search and Linear Graph Algorithms in Haskell. Technical report, Glasgow Workshop on Functional Programming.

Kirsch, C., Lippautz, M., and Payer, H. (2013). Fast and Scalable, Lock-Free k-FIFO Queues. In Malyshkin, V., editor, *Parallel Computing Technologies*, volume 7979 of *Lecture Notes in Computer Science*, pages 208–223. Springer Berlin Heidelberg.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search . *Artificial Intelligence*, 27(1):97 – 109.

Kranz, D., Halstead Jr., R., and Mohr, E. (1989). Mul-T: A High-Performance Parallel Lisp. In *PLDI'91 — Programming Languages Design and Implementation*, volume 24(7) of *SIGPLAN Notices*, pages 81–90, Portland, Oregon, June 21–23.

Kulkarni, M., Burtscher, M., Casçaval, C., and Pingali, K. (2009). Lonestar: A Suite of Parallel Irregular Programs. In *IEEE Int. Symp. on Performance Analysis of Systems and Software ISPASS '09*.

Lea, D. (1999). *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Lea, D. (2000). A Java fork/join Framework. In *Java'00 — ACM Conf. on Java Grande*, pages 36–43. ACM Press.

Leijen, D., Schulte, W., and Burckhardt, S. (2009). The Design of a Task Parallel Library. In *Int. Conf. on Object Oriented Programming Systems Languages and Applications OOPSLA'09*, pages 227–242. ACM Press.

Leimanis, E. and Minorsky, N. (1958). *Dynamics and nonlinear mechanics: Some recent advances in the dynamics of rigid bodies and celestial mechanics*. Surveys in applied mathematics. Wiley.

Leyton, M. and Piquer, J. M. (2010). Skandium: Multi-core Programming with Algorithmic Skeletons. In *IEEE Euro-micro PDP 2010*.

Lindholm, T. and Yellin, F. (1999). Java (tm) virtual machine specification, the.

Loidl, H.-W., Rubio, F., Scaife, N., Hammond, K., Horiguchi, S., Klusik, U., Loogen, R., Michaelson, G., Pea, R., Priebe, S., Rebn, ., and Trinder, P. (2003). Comparing Parallel Functional Languages: Programming and Performance. *Higher-Order and Symbolic Computation*, 16:203–251.

Loidl, H.-W., Trinder, P., and Butz, C. (2001). Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4).

Loidl, H.-W. and Trinder, P. W. (1997). Engineering Large Parallel Functional Programs. In *Implementation of Functional Languages, 1997*, LNCS, St. Andrews, Scotland. Springer-Verlag.

Loogen, R., Ortega-Mallén, Y., and Peña Marí, R. (2005). Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475.

Lowenthal, D. K., Freeh, V. W., and Andrews, G. R. (1996). Using Fine-grain Threads and Run-time Decision Making in Parallel Computing. *Journal of Parallel and Distributed Computing*, 37(1). Special issue on multithreading for multiprocessors.

Maier, P. and Trinder, P. (2012). Implementing a High-Level Distributed-Memory Parallel Haskell in Haskell. In *Proc. of the 23rd Int. Conf. on Implementation and Application of Functional Languages*, IFL'11, pages 35–50, Berlin, Heidelberg. Springer-Verlag.

Mainland, G., Jones, S. P., Marlow, S., and Leshchinskiy, R. (2013). Haskell beats C using generalised stream fusion. In *ICFP'13*. Submitted.

Marlow, S. and Jones, S. P. (2012). The glasgow haskell compiler.

Marlow, S., Jones, S. P., and Singh, S. (2009). Runtime Support for Multicore Haskell. In *Proc. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP '09, pages 65–78, New York, NY, USA. ACM.

Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K., and Trinder, P. (2010). Seq No More: Better Strategies for Parallel Haskell. In *Proc. of the 3rd ACM Haskell Symp. on Haskell*, Haskell '10, pages 91–102, New York, NY, USA. ACM.

Marlow, S., Newton, R., and Jones, S. P. (2011). A monad for deterministic parallelism. In *Proc. of the 4th ACM Symp. on Haskell*, Haskell '11, pages 71–82, New York, NY, USA. ACM.

Matthews, D. C. (1986). An overview of the Poly programming language. Technical Report UCAM-CL-TR-99, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500.

Matthews, D. C. J. and Wenzel, M. (2010). Efficient Parallel Programming in Poly/ML and Isabelle/ML. In *DAMP10: Declarative Aspects of Multicore Programming*, Madrid, Spain.

Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition.

McBride, C. and Paterson, R. (2008). Applicative Programming with Effects. *Journal of Functional Programming*, 18(1):1–13.

McCarthy, J. (1962). *LISP 1.5 Programmer's Manual*. The MIT Press.

Mohr, E., Kranz, D., and Halstead Jr., R. (1991). Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280.

Moir, M. and Shavit, N. (2007). Concurrent Data Structures. In *Handbook of Data Structures and Applications*, pages 47–14 – 47–30. Chapman and Hall/CRC Press.

Newburn, C. J., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S. D., et al. (2011). Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *9th Int. Symp. on Code Generation and Optimization*, pages 224–235. IEEE/ACM.

Newton, R., Chen, C.-P., and Marlow, S. (2011). Intel Concurrent Collections for Haskell. Technical Report MIT-CSAIL-TR-2011-015, MIT.

Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Nikhil, R. and Arvind (2001). *Implicit parallel programming in pH*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Numrich, R. and Reid, J. (2005). Co-arrays in the Next Fortran Standard. *ACM SIGPLAN Fortran Forum*, 24(2):4–17.

Numrich, R. W. and Reid, J. (1998). Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17(2):1–31.

NVIDIA (2008). CUDA Programming Guide 2.0.

Odersky et al., M. (2006). An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2006-001, EPFL Lausanne, Switzerland. Second edition.

Okasaki, C. (1996). Functional Data Structures. In *Advanced Functional Programming*, pages 131–158.

Okasaki, C. (1999). *Purely Functional Data Structures*. Cambridge University Press. September.

Okasaki, C. (2000). Breadth-first Numbering: Lessons from a Small Exercise in Algorithm Design. In *Proc. of the 5th ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP '00, pages 131–136, New York, NY, USA. ACM.

OpenMP (2012). The OpenMP API specification for parallel programming. http://openmp.org/wp/.

Orale (2015). Java Platform, Standard Edition (Java SE) 8 Documentation.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A. E., and Purcell, T. (2007). A survey of general-purpose computation on graphics hardware.

Pfalzner, S. and Gibbon, P. (1996). *Many-Body Tree Methods in Physics*. Cambridge University Press. ISBN: 9780511529368.

PGAS (2015). Partitioned Global Address Space. http://www.pgas.org/.

Prokopec, A., Bagwell, P., Rompf, T., and Odersky, M. (2011). A Generic Parallel Collection Framework. In *Int. Conf. on Parallel Processing EuroPar'11*, pages 136–147. Springer Verlag.

Reinders, J. (2007). *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly.

Reinke, C. (2001). GHood – Graphical Visualisation and Animation of Haskell Object Observations. In Hinze, R., editor, *ACM SIGPLAN Haskell Workshop*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 182–196. Elsevier Science.

Reppy, J., Russo, C., and Xiao, Y. (2009). Parallel Concurrent ML. In *Int. Conf. on Functional Programming (ICFP 2009)*.

Richardson, H. (1996). High performance fortran: history, overview and current developments. Technical report, 1.4 TMC-261, Thinking Machines Corporation.

Risset, T. (2011). *Encyclopedia of Parallel Computing*, chapter SoC (System on Chip), pages 1837–1842. Springer US, Boston, MA.

Sadrozinski, H. and Wu, J. (2010). *Applications of Field-Programmable Gate Arrays in Scientific Research*. CRC Press.

Schmidt, D. A. (1986). *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA.

Scholz, S.-B. (2003). Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059.

Shan, H., Singh, J. P., Oliker, L., and Biswas, R. (2003). Message passing and shared address space parallelism on an smp cluster. *Parallel Comput.*, 29(2):167–186.

Shavit, N. (2011). Data structures in the multicore age. *Commun. ACM*, 54(3):76.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2012). *Operating System Concepts*, chapter 5: Process Synchronization. Wiley Publishing, 9th edition.

Silva, L. and Buyya, R. (1999). Parallel Programming Models and Paradigms. In Buyya, R., editor, *High Performance Cluster Computing: Programming and Applications*, chapter 1. Prentice Hall.

Skedzielewski, S. K. (1991). Parallel functional languages and compilers. chapter Sisal, pages 105–157. ACM, New York, NY, USA.

Skillicorn, D. B. (1995). Towards a higher level of abstraction in parallel programming. *In: Proc. of Programming Models for Massively Parallel Computers*, pages 78–85.

Sondergaard, H. and Sestoft, P. (1989). Referential transparency, definiteness and unfoldability. *Acta Inform*, pages 505–517.

Spetka, S., Hadzimujic, H., Peek, S., and Flynn, C. (2008). High productivity languages for parallel programming compared to mpi. In *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, pages 413–417.

Steele, G. (2009). Organizing Functional Code for Parallel Execution. Talks and Posters.

Steuwer, M., Kegel, P., and Gorlatch, S. (2011). SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IEEE Int. Symp. on Parallel and Distributed Processing*, pages 1176–1182.

Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73.

Sulzmann, M., Lam, E. S. L., and Marlow, S. (2009). Comparing the performance of concurrent linked-list implementations in Haskell. *ACM Sigplan Notices*, 44(5):11.

Sunderam, V. S. (1990). Pvm: A framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339.

Swierstra, S., Azero Alcocer, P., and Saraiva, J. (1999). Designing and Implementing Combinator Languages. In *Ad. Funct. Program.*, LNCS 1608, pages 150–206. Springer.

Syme, D., Granicz, A., and Cisternino, A. (2007). *Expert F#*. Apress Academic. ISBN 1590598504.

Tanenbaum, A. S. and Austin, T. (2012). *Structured Computer Organization*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition.

Tarjan, R. (1972). Depth first search and linear graph algorithms. *SIAM Journal on Computing*.

The GHC Team (2015). The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.10.2.

TIOBE Software (2013). TIOBE Programming Community Index. www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Totoo, P. (2011). *Inherently Parallel Data Structures*. MSc Thesis, Heriot-Watt University.

Totoo, P., Deligiannis, P., and Loidl, H.-W. (2012). Haskell vs. F# vs. Scala: A High-level Language Features and Parallelism Support Comparison. In *Proc. of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC '12, pages 49–60, New York, NY, USA. ACM.

Totoo, P. and Loidl, H.-W. (2014a). Parallel Haskell Implementations of the N-body Problem. *Concurrency and Computation: Practice and Experience*, 26(4):987–1019.

Totoo, P. and Loidl, H.-W. (2014b). Lazy Data-oriented Evaluation Strategies. In *Proc. of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC '14, pages 63–74, New York, NY, USA. ACM.

Toub, S. (2010). Patterns of Parallel Programming – Understanding and Applying Parallel Patterns with the .NET Framework 4. Online.

Trinder, P., Hammond, K., Loidl, H.-W., and Jones, S. P. (1998a). Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60.

Trinder, P., Loidl, H.-W., and Pointon, R. F. (2002). Parallel and Distributed Haskells. *Journal of Functional Programming*, 12:469–510.

Trinder, P. W., Barry Jr., E., Davis, M. K., Hammond, K., Junaidu, S. B., Klusik, U., Loidl, H.-W., and Peyton-Jones, S. L. (1998b). GpH: An architecture-independent functional language. Unpublished draft.

Turing, A. M. (1937). Computability and -definability. *The Journal of Symbolic Logic*, 2:153–163.

Turner, D. (1986). An overview of miranda. *SIGPLAN Not.*, 21(12):158–166.

Turner, D. A. (1979). A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49.

Turner, D. A. (2013). *Trends in Functional Programming: 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, chapter Some History of Functional Programming Languages, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg.

UPC Consortium (2005). Unified Parallel C Language Spec. v1.2 LBNL-59208. Technical report, Lawrence Berkeley National Lab.

Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. *New Computational Paradigms for Computer Music*, pages 9–47.

Wadler, P. (1990). Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231 – 248.

Wadler, P. (1995). Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK. Springer-Verlag.

Wadsworth, C. (1971). *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford.

Wainwright, R. L. and Sexton, M. E. (1992). A study of sparse matrix representations for solving linear systems in a functional language. *Journal of Functional Programming*, 2(01):61–72.

Weiland, M. (2007). Chapel, Fortress and X10: novel languages for HPC. Technical report, EPCC, University of Edinburgh.

Wirth, N. (1978). *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Wolfe, M. (2013). The OpenACC Application Programming Interface. http://www.openacc.org/.