PROFILING A PARALLEL DOMAIN SPECIFIC LANGUAGE USING OFF-THE-SHELF TOOLS

Majed Mohammed Abdullah Al-Saeed

Submitted in Fulfilment of the Requirements for the Degree of Doctor of Philosophy

University of Glasgow

College of Science and Engineering

School of Computing Science

July 2015

Abstract

Profiling tools are essential for understanding and tuning the performance of both parallel programs and parallel language implementations. Assessing the performance of a program in a language with high-level parallel coordination is often complicated by the layers of abstraction present in the language and its implementation. This thesis investigates whether it is possible to profile parallel Domain Specific Languages (DSLs) using existing host language profiling tools. The key challenge is that the host language tools report the performance of the DSL runtime system (RTS) executing the application rather than the performance of the DSL application. The key questions are whether a correct, effective and efficient profiler can be constructed using host language profiling tools; is it possible to effectively profile the DSL implementation, and what capabilities are required of the host language profiling tools?

The main contribution of this thesis is the development of an execution profiler for the parallel DSL, Haskell Distributed Parallel Haskell (HdpH) using the host language profiling tools. We show that it is possible to construct a profiler (HdpHProf) to support performance analysis of both the DSL applications and the DSL implementation. The implementation uses several new GHC features, including the GHC-Events Library and ThreadScope, develops two new performance analysis tools for DSL HdpH internals, i.e. *Spark Pool Contention Analysis*, and *Registry Contention Analysis*.

We present a critical comparative evaluation of the host language profiling tools that we used (GHC-PPS and ThreadScope) with another recent functional profilers, EdenTV, alongside four important imperative profilers. This is the first report on the performance of functional profilers in comparison with well established industrial standard imperative profiling technologies. We systematically compare the profilers for usability and data presentation. We found that the GHC-PPS performs well in terms of overheads and usability so using it to profile the DSL is feasible and would not have significant impact on the DSL performance.

We validate HdpHProf for functional correctness and measure its performance using six benchmarks. HdpHProf works correctly and can scale to profile HdpH programs running on up to 192 cores of a 32 nodes Beowulf cluster. We characterise the performance of HdpHProf in terms of profiling data size and profiling execution runtime overhead. It shows that HdpHProf does not alter the behaviour of the GHC-PPS and retains low tracing overheads close to the studied functional profilers; 18% on average. Also, it shows a low ratio of HdpH trace events in GHC-PPS eventlog, less than 3% on average.

We show that HdpHProf is effective and efficient to use for performance analysis and tuning of the DSL applications. We use HdpHProf to identify performance issues and to tune the thread granularity of six HdpH benchmarks with different parallel paradigms, e.g. divide and conquer, flat data parallel, and nested data parallel. This include identifying problems such as, too small/large thread granularity, problem size too small for the parallel architecture, and synchronisation bottlenecks.

We show that HdpHProf is effective and efficient for tuning the parallel DSL implementation. We use the Spark Pool Contention Analysis tool to examine how the spark pool implementation performs when accessed concurrently. We found that appropriate thread granularity can significantly reduce both conflict ratios, and conflict durations, by more than 90%. We use the Registry Contention Analysis tool to evaluate three alternatives of the registry implementations. We found that the tools can give a better understanding of how different implementations of the HdpH RTS perform.

Dedication

In The Name Of Allah (God), The Most Gracious, The Most Merciful.

I dedicate this thesis to the memory of my mother. I miss her so much, but I am glad that she saw the process of this thesis through to its completion, offering the support to make it possible, as well as plenty of prayers, may Allah rest her soul in peace amen.

Acknowledgements

My praises to Allah (God) for giving me the wellness, the strength of determination, the patience, and support to complete this work successfully.

I would like to express my special appreciation and thanks to my supervisor Professor Phil Trinder, for all his encouragement, guidance and support to my research, and for allowing me to grow as a research scientist. I have benefited greatly from his advice, inspiration, and vast experience.

Also, I would like to express my very great appreciation to my supervisors Dr. Lilia Georgieva and Dr. Patrick Maier. Lilia's professional guidance and valuable support have been a great help in my research. Patrick has been a tremendous mentor for me. I have been lucky enough to work with him over the course of my PhD, he was invaluable in providing guidance and support from a different perspective. His willingness to give time so generously has been very much appreciated.

Many thanks to Professor Greg Michaelson and Dr. J. Paul Siebert for their useful and constructive feedback and recommendations regarding this thesis. Also, I would like to thank my friends at the Dependable Systems Group (DSG) at Heriot-Watt University Dr. Khari Armih, Dr. Mustafa Aswad, and Dr. Rob Stewart for their advice and valuable technical support.

I would also like to extend my thanks to the computing officers at Heriot-Watt University for their support and allowing hardware access for the performance evaluation of HdpHProf and the critical analysis study.

I would like to thank King Faisal University and the Ministry of Higher Education in Saudi Arabia for offering me this scholarship. Also I would like to thank them for providing the finance support throughout the years of study for me and my family.

Last, but by no means least, a special thanks to my family. Words cannot express how grateful I am to my father, my wife, my daughters, and my son for all of the sacrifices that you have made on my behalf. Your prayers for me were what has sustained me thus far.

Declaration

I declare that, except where explicit reference is made to the contribution of others, that this dissertation is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution.

Majed Al-Saeed

Table of Contents

List of Tables

List of Figures

List of Acronyms

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

COTS Commercial Off-The-Shelf.

DSL Domain Specific Language.

GHC Glasgow Haskell Compiler.

GHC-ELF GHC EventLog Format.

GHC-PPS GHC Parallel Profiling System.

GHC-SMP GHC on a Shared-Memory Multiprocessor.

GpH Glasgow Parallel Haskell.

GUI Graphical User Interface.

HdpH Haskell Distributed Parallel Haskell.

HEC Haskell Execution Context.

HPC High-Performance Computing.

MIMD Multiple Instruction Multiple Data.

MISD Multiple Instruction Single Data.

MPI Message Passing Interface.

NUMA Non-Uniform Memory Access.

OTF Open Trace Format.

PE Processing Element.

PVM Parallel Virtual Machine.

RTS Runtime System.

SDDF Self-Describing Data Format.

SICSA Scottish Informatics and Computer Science Alliance.

SIMD Single Instruction Multiple Data.

SISD Single Instruction Single Data.

SMP Shared-Memory Multiprocessor.

TSO Thread State Object.

UMA Uniform Memory Access.

Chapter 1

Introduction

The manycore revolution has both made parallelism mainstream, and sparked interest in functional languages. The underlying memory model has a big influence on parallel languages: in shared-memory languages computations can share state, but in a distributed-memory language computations must communicate any common state. The increasing demand for parallel machines to solve larger problems raises the need for advanced performance analysis tools to help programmers effectively and efficiently optimise parallel applications [42].

Performance analysis and tuning for parallel environments is important due to the complexity of parallel technology [51, 30], and presents more challenges than on a sequential machine [145]. Profiling is a key element of effective parallel programming and performance optimisation: it is essential that the programmer understands the parallel behaviour in order to improve it [146, 98, 6, 20].

In languages that provide high-level parallelism, like most parallel functional languages, profiling is especially important. The high level abstraction means that the conceptual gap between the program and its execution on the hardware is greater [122]. Therefore, language implementers and programmers must have a profiler to help understand parallel behaviour and identify performance bottlenecks in the implementation [54, 35, 66].

This thesis investigates a new approach to constructing profiling infrastructure for parallel Domain Specific Languages (DSLs), namely using host language profiling tools to construct an effective and efficient profiling tool for a parallel DSL. Specifically, **H**askell **D**istributed **P**arallel **H**askell (**HdpH**) is a distributed-memory parallel DSL [84, 82] that is built solely on the standard Glasgow Haskell Compiler (GHC) [46] runtime. We present the design, implementation, validation and evaluation of Hd-

pHProf, an execution profiler, i.e. time profiler, for the DSL HdpH constructed from Haskell profiling tools.

1.1 Thesis Statement

This thesis investigates whether it is possible to profile parallel DSLs using existing host language profiling tools. The key challenge is that the host language tools report the performance of the DSL Runtime System (RTS), executing the application rather than the performance of the DSL application. The key questions are as follows: can a correct, effective and efficient profiler be constructed using host language profiling tools for applications written in the parallel DSL? Is it possible to construct a correct, effective and efficient profiler to tune the parallel DSL implementation? What capabilities of the host language profiling tools facilitate parallel DSL profiling? Can shared-memory profiling tools be extended to work effectively for a distributed-memory parallel DSL?

1.2 Contributions

This thesis investigates how to use commodity pre-existing profiling tools for a host language to profile the performance of a distributed-memory parallel DSL. The main contribution is to develop, validate and evaluate a new profiler, HdpHProf, for a distributed-memory parallel functional DSL, i.e. HdpH which runs on a Beowulf cluster of multicore. The thesis makes the following research contributions:

1. We report a critical analysis of parallel functional profilers [4, 5], comparing two functional profilers, GHC Parallel Profiling System (GHC-PPS) with its trace viewer ThreadScope [67, 134] and EdenTV [11]; alongside four important imperative profilers, i.e. Vampir [139], Score-P [124], mpiP [141], and ompP [37]. The comparison is based on the SICSA Concordance benchmark [23], covers both shared and distributed-memory parallel languages, and is performed on common parallel architectures. We compare the runtime overheads and amount of profiling data generated by the profilers, analysed by whether the parallelism is shared/distributed memory and whether the profiler is imperative/functional, and tracing/summative. We systematically compare the profilers for usability and data presentation (Chapter 3).

2. We investigate the feasibility of constructing an effective profiler for a distributed-memory parallel DSL using the host language profiling tools. We do so by designing and implementing HdpHProf [3], a post mortem, multi-stage, and extensible profiler for a distributed-memory Haskell parallel DSL (HdpH). HdpHProf requires no changes to the host language (GHC) unlike EdenTV [11], nor to the HdpH programs that are profiled. Importantly, the implementation uses several new GHC features including the GHC-Events Library and ThreadScope, to build profiling infrastructure for a parallel DSL. We introduce two novel analysis tools for monitoring HdpH internals, i.e. *Spark Pool Contention Analysis* and *Registry Contention Analysis* (Chapter 4).

3. We validate HdpHProf for functional correctness and characterise its performance. We ascertain that HdpHProf works correctly and accurately records the behaviour of parallel programs. Also, we validate HdpHProf for scalability by using 5 HdpH benchmarks to show that it can scale to profile applications running on a large number of cores (192 cores on 32 cluster nodes). We characterise the performance of HdpHProf in terms of data size and execution runtime overhead, depending on computation size and the number of Processing Elements (PEs) as in [4]. We show that HdpHProf retains the low time and space overheads of the GHC-PPS. Moreover, profiling the parallel DSL occupies less than 2.2% of tracing overheads of GHC-PPS, unless thread granularity is excessively small (Chapter 5).

4. We show that an effective and efficient DSL profiler (HdpHProf) can be constructed for applications using host language profiling tools. We use HdpHProf to identify performance issues and to tune the thread granularity of six HdpH benchmarks. We demonstrate how to identify performance problems in execution behaviour of HdpH applications with different parallel paradigms, e.g. divide and conquer, flat data parallel, and nested data parallel. This includes problems such as too small/large thread granularity, problem size too small for the parallel architecture, synchronisation bottlenecks, and combinations of these factors. We demonstrate how HdpHProf can be used to tune thread granularity in HdpH applications (Chapter 6).

5. We show that an effective and efficient profiler can be constructed to tune the parallel DSL implementation (i.e. the HdpH RTS) using the host language profiling

tools. We do so by using HdpHProf analysis tools to investigate the behaviour of HdpH RTS. The *Spark Pool Contention Analysis* tool reveals, for example, how the spark pool implementation behaves during high concurrent access demand and how contention changes with task granularity. We present how appropriate task granularity can significantly reduce contention on the spark pool by more than 97%. We use the *Registry Contention Analysis* tool to evaluate three alternative registry implementations. This shows how HdpHProf can identify different execution behaviours for the different implementations which can help debug or improve the parallel DSL implementation (Chapter 7).

1.3 Authorship and Publications

Parts of this thesis are closely based on the work reported in the following papers:

- **HdpHprof— A Profiler for Haskell Distributed Parallel Haskell** [3]. In *The Draft Proceedings of the Symposium on Trends in Functional Programming (TFP12)*, St Andrews, Scotland, 2012. With Patrick Maier, Phil Trinder and Lilia Georgieva. The paper presents the initial design and implementation of HdpHProf and shows preliminary profiling results of HdpH's programs run on a Beowulf cluster comprising 32 8-cores nodes.

- **A Critical Analysis of Parallel Functional Profilers** [4]. In *The Draft Proceedings of The 25th symposium on Implementation and Application of Functional Languages(IFL13)*, Radboud University Nijmegen, The Netherlands, 2013. With Patrick Maier, Phil Trinder and Lilia Georgieva. This paper presents an evaluation of two parallel Haskell profilers, GHC-PPS and EdenTV, in comparison with four important profilers for imperative languages. The comparison covers both shared and distributed memory parallel languages, and is performed on common parallel architectures. The comparison uses a published benchmark, namely the Concordance application set as the first Multicore Challenge [23]. We compare the amount of profiling data generated by the profilers analysed by whether the parallelism is shared/distributed memory and whether the profiler is imperative/-functional, and tracing/summative. We investigate the runtime overheads of the profilers, again analysed by whether the parallelism is shared/distributed memory and whether the profiler is imperative/functional, and tracing/summative. Also, we systematically compare the profilers for usability and data presentation.

Chapter 2

Background

Parallelism has become the norm in commodity computers. Unfortunately developing applications for parallel computers is more difficult than developing software for sequential computers, hence it consumes more time and money [110]. A parallel programmer must specify additional coordination aspects, such as how to divide load among different processors, and how these processors communicate and synchronise. Moreover, parallelism makes debugging and optimisation of programs more complicated and error prone [74, 34, 147].

To get good performance, tools for performance analysis and tuning of parallel programs are crucial to give programmers an insight about the execution behaviour and assistance in identifying performance problems [97, 51]. Performance tools help programmers identify performance bottlenecks and under-utilisation of computing resources. Every mainstream parallel programming language, such as C with MPI and/or OpenMP, has one or more performance analysis tools, e.g. Seecube [25], HyperView [86], Pablo toolkit [115], and ParaGraph [60].

This chapter surveys the state of the art of parallel computer architectures, parallel programming models, and performance analysis of parallel programs. We present parallel computer systems depending on the physical memory organization: shared-memory and distributed-memory (Section 2.1). We discuss parallel programming models (Section 2.2). We discuss ways of profiling performance of parallel programs and explain what makes profiling distributed-memory programs different from shared-memory programs (Section 2.3). We present state-of-the-art performance analysis tools for imperative languages such as C/C++ with MPI and/or OpenMP, and high-level functional parallel Haskells (Section 2.4).

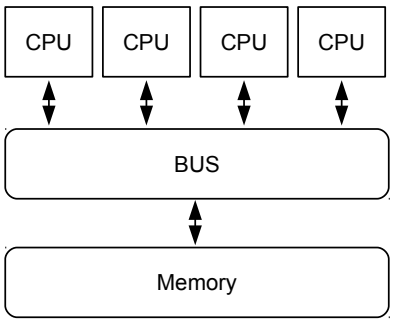


Figure 2.1: An SMP Architecture

2.1 Parallel Architectures

Parallel architectures can be classified according to the machine's instruction and data streams as in Flynn's taxonomy [31, 32]. He identified four categories of architecture: SISD (e.g. uni-core), SIMD (e.g. vector processor), MISD (e.g. systolic array), and MIMD (e.g. multicore). Today most general-purpose parallel computers are based on MIMD [112]. MIMD is too broad to be useful on its own so it is split into two classes of parallel systems according to memory organisation, shared-memory, and distributed-memory [92].

2.1.1 Shared-Memory Architectures

All the processors in a shared memory system share a single physical or logical address space and communicate with each other by reading and writing variables from the shared-memory. Shared-memory systems could be divided further into two categories based on how the memory is accessed by the processors, i.e. Uniform Memory Access (UMA), and Non-Uniform Memory Access (NUMA).

UMA. Uniform Memory Access (UMA) is a class of shared-memory systems where the cost of accessing memory is the same for all collaborating processors.

Shared-Memory Multiprocessor (SMP) is one form of the UMA shared memory systems in which all processors can access all the memory locations at equal speed via a shared bus [110]. Figure 2.1 depicts a SMP architecture. SMP is considered attractive and easy to program because of the convenience of sharing data among processors [144]. On the other hand, SMP is not scalable beyond a relatively small number of processors because increasing the number of processors causes contention on the bus [92].

A Multicore system [45] is a special kind of SMP where two or more "execution

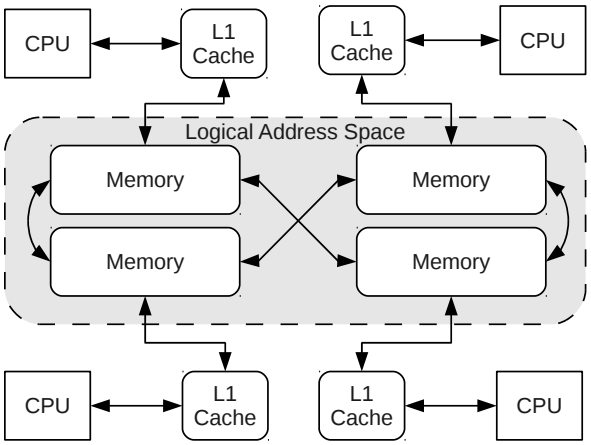


Figure 2.2: A NUMA Architecture

cores" are implemented within a single processor. The cores reside in a single die and are considered as individual processors which have their own set of execution and architectural resources [1]. Multicore systems, may, or may not, share a large on-chip L2 cache between cores. Multicore systems are more efficient than the SMP since the cores share the L2 cache, reducing the memory bandwidth bottleneck and communication problem. As with SMP, multicore systems' scalability is limited [29].

NUMA. Non-Uniform Memory Access (NUMA) is another class of shared-memory systems in which all processors share a memory which is nonuniformly addressable for them, where some processors may physically reside more closely in some memory blocks than other processors. The access time for data in the memory can vary considerably depending upon whether the data is located in the local memory of the processor, or the local memory of another processor [110]. Figure 2.2 depicts a shared memory NUMA architecture. Unlike the SMP the NUMA is more scalable because of the lower memory bandwidth bottleneck involved. On the contrary, the time it takes to access location on the memory could vary considerably from one processor to another depending on how far the memory is from the processor [92].

2.1.2 Distributed-Memory Architecture

Each processor in a distributed-memory architecture has its own memory and typically communicates with other processors by sending and receiving messages. Figure 2.3 depicts a distributed-memory architecture [92, 110]. The speed of distributed systems depends not only on the speed of CPUs but also on the speed and topology of the

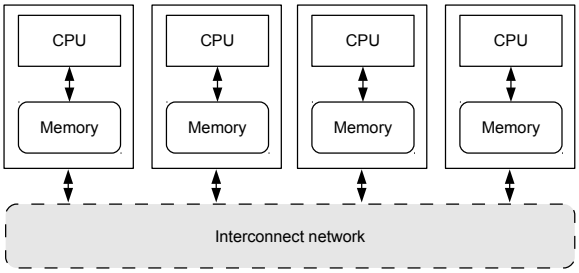


Figure 2.3: A Distributed Memory Architecture

network. Most importantly, in distributed systems it is more probable that the programmer will be responsible for explicitly specifying all the communication between processors, and specifying the distribution of data [92]. We will discuss the ubiquitous cluster and hybrid architectures used later in the thesis, but will not discuss Grid [33, 18], cloud technologies [26, 27] or the more exotic manycore [22, 81] .

Cluster Computing. A cluster [92, 121] is a set of computers connected with each other by some networking technology, e.g. Ethernet. Beowulf clusters [127, 119] are constructed by networking Commercial Off-The-Shelf (COTS) nodes, typically running free to use operating system and software packages, e.g. Linux and MPI/PVM libraries. Beowulf clusters are powerful and inexpensive compared with High-Performance Computing (HPC) architectures [118].

Hybrid Architectures. Hybrid systems are built by combining shared-memory and distributed-memory technology. When a cluster is built of shared-memory nodes, e.g. multicores –which is probably the most common platform for PCs and workstations these days– the cluster is considered a hybrid parallel system. Cores in each separate node can communicate through passing pointers to shared-memory regions. On the other hand, message-passing is used for communication between nodes within the cluster. We will profile parallel programs running on a hybrid Beowulf cluster of multicores in Chapter 6.

2.2 Parallel Programming Models

To program a parallel computing architecture programmers need a programming model that allows the design and implementation of parallel applications. Moreover, programming languages, tools, and environments are essential elements for designing and

implementing parallel applications [130]. To make it possible for programmers to build and run parallel programs they need a set of tools, features, and an Application Programming Interface (API) [92].

According to Rauber and Rünger [113] a parallel programming model "specifies the programmers view on the parallel computer by defining how the programmer can code an algorithm. This view is influenced by the architectural design and the language, compiler, or the runtime libraries." Unlike in sequential programming there are many possible parallel programming models, depending on the architecture of the parallel machine [130, 92]. This makes it hard for programmers to write portable programs. The most commonly used models for parallel programming are the shared-memory, distributed-memory, or hybrid models.

There are many programming languages and environments for parallel programming [92]. Here we only present the two parallel programming environments most widely used by the parallel programming community, i.e. OpenMP [101] for shared-memory, and Message Passing Interface (MPI) [95] for distributed-memory. In addition, we will discuss high-level functional parallel languages.

2.2.1 Shared-Memory Parallelism

OpenMP [101] is a model for shared-memory architectures that currently support Fortran, C, and C++ on Linux and Windows platforms. OpenMP is mostly used by adding a compiler directive around a loop to add parallelism to sequential code where the compiler takes care of the majority of the detailed thread creation and management [92]. OpenMP is the most widely used communication standard in shared-memory programming in parallel computing [15].

2.2.2 Distributed-Memory Parallelism

The Message Passing Interface (MPI) [95] is a standard message passing library used for distributed-memory architectures where processes do not share data. There are multiple implementations of the MPI specification in the form of libraries that provide functions, subroutines, and methods for languages, e.g. Fortran, C, and C++. Moreover, it is supported in a variety of HPC and commodity clusters [110]. Programming distributed-memory machines with MPI is difficult because the programmer must specify the data distribution and inter-process communication between processes

using messages [92]. MPI is probably the most widely used communications standard for programming distributed-memory parallel applications [15].

2.2.3 Parallel Functional Languages

This section discusses parallel implementations of the general purpose lazy functional programming language Haskell [58]. We will focus on Haskell and not discuss other functional languages e.g. Erlang [7] and ML [143] because we will profile Haskell DSLs. Haskell is different from most current languages that are closely related to the underlying hardware, where programming is based upon the concept of changing stored values. Instead, Haskell promotes a programming style with higher level of abstraction, based upon the idea of applying functions to arguments [65]. Haskell is a functional computation language that is the base for a range of parallel and distributed languages [135]. We will outline various Haskell extensions for parallel and distributed programming.

Shared-Memory Haskells

GpH. Glasgow Parallel Haskell (GpH) [136] is an extension of Haskell that provides parallelism and keeps the programmer away from the details of the parallel execution. GpH provides semi-explicit parallelism with little programmer control of parallel behaviour where the compiler and the runtime system do most of the work. The programmer only writes the parallel algorithm and *explicitly* controls some aspects of the parallel algorithm. There are two available implementations for GpH: GHC-SMP [90] for shared-memory, and GUM [138] for distributed-memory. Evaluation strategies, i.e. coordination abstractions, have been introduced [137, 88] to specify parallelism (evaluation order and degree) at a higher level (e.g. parallel data structures).

GHC-SMP. GHC on a Shared-Memory Multiprocessor (GHC-SMP) [57, 90] is a present full-scale implementation of shared-memory parallel Haskell based on the GHC. GHC-SMP provides lightweight parallel evaluation and deterministic parallelism: the parallel program has the same semantics as the sequential program. The GHC-SMP runtime can support explicit thread-based parallelism [106] and semi-explicit deterministic parallelism [137].

Parallel Monad. `Par` Monad [89] is a programming model for deterministic parallel computation in Haskell that provides monadic control of parallelism and retains determinism and purity. Furthermore, `Par` Monad's work-stealing scheduler allows it to lift system-level functionality to the Concurrent Haskell level. Despite this, performance results show that `Par` Monad keeps performance overhead at a level comparable with other parallel programming models in Haskell.

Distributed-Memory Haskells

Eden. Eden [12, 80] is a Haskell extension for distributed-memory parallelism. Eden provides a high-level of abstraction for parallel programming by controlling the parallel evaluation of processes. Processes in Eden can by defined explicitly; meanwhile communication between the processes remains implicit. Eden provides a rich library of predefined skeletons that cover many common patterns of parallel algorithms, e.g. parallel divide-and-conquer, and parallel map. Eden programmers can choose or adapt one of these skeletons for the problem at hand instead of writing programs from scratch. Eden provides dynamic load balancing by its replicated workers' skeleton which frequently gives better performance than purely static schemes such as tasks farms.

Cloud Haskell. Cloud Haskell [28] is a new domain-specific (cloud computing) language which is shallowly embedded in Haskell for distributed-memory parallelism. Cloud Haskell has been influenced by the successful message-passing model used in Erlang [7] along with the purity, types, and monads of Haskell. A message-passing communication model is provided in Cloud Haskell. Unlike Eden, Cloud Haskell contributes a new method for serialising functions' closures to be transferred over the network with no need to extend the compiler or the runtime system.

2.2.4 HdpH

Haskell Distributed Parallel Haskell (HdpH) [84, 83] that is profiled in the reminder of the thesis is a new Haskell embedded DSL for distributed-memory parallelism. It provides semi-explicit parallelism i.e. the programmer specifies only a few key parallelism aspects, e.g. the creation of tasks. HdpH scales to run on HPC, e.g. the 90K core HECToR supercomputer [63]. HdpH is designed to not rely on a bespoke low-level runtime system. Like Cloud Haskell it requires no more than GHC, primarily to minimise the language maintenance effort. HdpH is implemented in a modular and

layered approach and, importantly, coded in vanilla Concurrent Haskell [106]. As this thesis presents the design, implementation and evaluation of a profiler (HdpHProf) for HdpH, in Chapters 4, 5, 6 and 7 this section gives an overview of HdpH.

Listing 2.1: HdpH Primitives [84].

```
1  data Par a -- Par monad
2  eval :: a -> Par a
3
4  fork :: Par () -> Par ()
5
6  data IVar a -- buffers
7  new :: Par (IVar a)
8  put :: IVar a -> a -> Par ()
9  get :: IVar a - Par a
10
11 data NodeId  -- explicit locations
12 allNodes :: Par [NodeId]
13
14 data Closure a    -- explicit, serialisable closures
15 spark  :: Closure(Par ()) -> Par ()
16 pushTo :: Closure(Par ()) -> NodeId - Par ()
17
18 data GIVar a  -- global handles to IVars
19 glob :: IVar  (Closure a) -> Par (GIVar (Closure a))
20 rput :: GIVar (Closure a) -> Closure a -> Par ()
21 at   :: GIVar (Clusure a) -> NodeId
```

Listing 2.1 illustrates the basic primitives that the programmer can use to express parallelism in HdpH. The lines from 1 to 9 show the shared-memory primitives, and in lines 11 to 21 are the distributed-memory primitives. The `Par` type constructor is used to encapsulate a parallel computation. The `fork` primitive creates a new thread and returns nothing, and is used for generating shared-memory parallelism. `IVar`s are mutable variables (writable exactly once) used by threads to communicate computational results. There are three operations that allow the programmer from accessing `IVar`s: creating new one (`new`), blocking read (`get`), and write (`put`). The `put` does not normalise its argument hence the `eval` primitive is used, instead, to evaluate an expression to weak-head normal form.

HdpH support distributed-memory parallelism with abstract data types for explicit locations, explicit closures, and global IVars. To generate distributed-memory parallelism HdpH exposes to the programmer the basic primitives `spark` and `pushTo`. The `spark` generates computation (called a *spark*), a future computation that may be executed on different node. A created spark resides in a spark pool and waits to be distributed or scheduled by an on-demand work-stealing scheduler. The `pushTo` primitive is similar to `spark` but eagerly sends a computation to a target node for instant execution.

HdpH retrieves the results of remote distributed computations by using *global IVars*, which are global references to IVars that support remote write, and local only

read. Global IVars are exposed to the programmer by three operations: create new one (`glob`) by globalising a local IVar, remote write (`rput`), and location information (`at`) of the underlying IVar. The operations restrict the base type of their underlying IVars to closures to ensure the serialisability of the values that the `rput` writes. All transported values between nodes both computation and results are closures which means that a result might be again a computation.

The following example and descriptive text are closely based on [84]. Listing 2.2 shows an HdpH Fibonacci program (`dpfib`) that employs the HdpH primitives, and can be executed on shared or distributed memory architectures using the HdpH implementation. The program must globalise the IVar `v`, yielding global IVar `gv`, and wrap the first recursive call in an explicit closure generated by the Template Haskell splice `$(mkClosure [|...|])`, before sparking. Also, it must convert the result of the sparked computation to an explicit closure with `toClosure` before writing to `gv`, and that closure must be eliminated again with `unClosure` before adding the results of both recursive calls.

Listing 2.2: HdpH Fibonacci Program [84].

```
1  dpfib :: Int -> Int -> Par Int
2  dpfib t n
3    | n <= t    = return $ fib n
4    | otherwise = do
5        v  <- new
6        gv <- glob v
7        spark $ (mkClosure [| dpfib t (n-1) >>=
8                              eval >>=
9                              reput gv . toClosure |])
10       y      <- dpfib t (n-2)
11       clo_x <- get v
12       return (unClosure clo_x + y)
```

2.3 Parallel Performance Analysis

Performance monitoring and analysis tools play a very important role in the process of developing parallel software that efficiently utilises the parallel machine [6, 71]. According to Liang and Viswanathan [76] the term profiling in its broad sense is, "the ability to monitor and trace events that occur during run time, the ability to track the cost of these events, as well as the ability to attribute the cost of the events to specific part of the program." Moreover, the goal of performance analysis of parallel programs is to provide programmers with an insight about the behaviour and performance issues during execution by efficiently recording and intuitively presenting performance data [116]. This thesis uses the term *profiler* to mean a tool that is used to monitor

and analyse the behaviour and performance of parallel applications, for the purpose of helping programmers to tune and improve their parallel code.

2.3.1 Performance Profiling Process

Performance profiling is commonly broken into different stages where the output of one stage is the input for the next [74, 30]. The process consists of: Gathering critical data that can give an insight about the behaviour of the parallel system, persisting the performance data for processing, e.g. in memory, in a database or a trace file, the analysis of performance data to order events and calculate facts, and finally, the presentation of system behaviour to the user which can be in the form of graphical, or text-based (summarised) visualisation. Figure 2.4 illustrates the performance profiling process workflow.

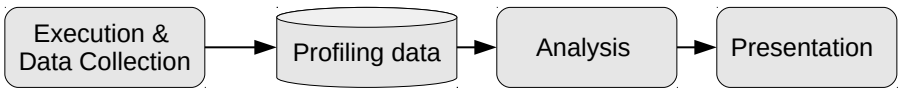


Figure 2.4: Performance Profiling Process Workflow.

Data Collection

The profiling process starts by collecting performance data from the executing program. The data collection process can take different forms, e.g. summative profiling, sampling, or tracing [24]. In the following we describe three common data collection approaches.

Summative Profiling. In this approach the profiler collects aggregated information about particular events during execution [71]. The profiler counts routine invocations or execution times of various events during the program execution and derives statistics from this data, e.g. gprof [52], CPPRO [54], and Haskell profiler [122, 66]. A profile can be useful for improving the behaviour of a program by showing routines that are responsible for most counts and execution times and comparing different alternative implementations [52]. An advantage of summative profiling is the low-overhead compared to other data collected methods such as tracing. However, profiling can only be useful for high-level analysis because it does not preserve the structure and temporal ordering of events [35, 99].

Sampling. Sampling takes snapshots at time intervals during the program execution, and the runtime must be sufficiently long relative to the sample period so that meaningful information can be derived [116]. Figure 2.5 demonstrates how performance data is collected during the sampling process[1]. Periodic interrupts during program execution occur to take measurement. Statistical inference is used to derive program behaviour from the sampled data [16, 43].

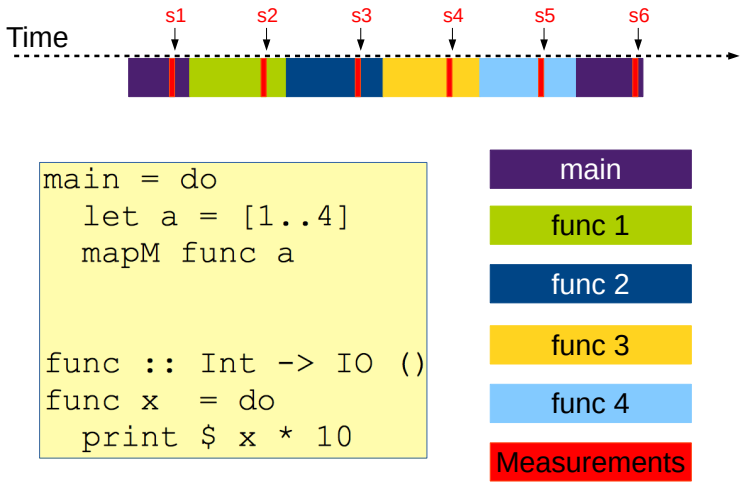


Figure 2.5: Sampling of Performance Data.

Tracing. Tracing is the recording of individual program execution events against time. With tracing detailed analysis of the interaction of processes or threads is possible with time-stamped events [35, 71]. Many performance analysis tools use tracing for collecting more detailed and comprehensive information about the behaviour of the executing program [68]. However, tracing is more intrusive than profiling and sampling. The main disadvantage of tracing is that it is considered the most expensive approach for performance data collection in terms of perturbation to program execution, space required for tracing data, and the post-processing of the trace events [6, 98, 116, 71, 99]. Importantly, tracing allows performance analysis that summative profiles cannot: for instance, identifying variation in dynamic behaviour of a function over several iterations. Besides, from tracing data, profiles can be computed, but not vice versa [71].

Instrumentation is used to emit trace events at certain places in the parallel code. Tracing is implemented by either instrumenting the runtime system, or the parallel application, to emit trace events during the execution. A trace event consists

[1] With permission from Holger Brunst and Brian Wylie the figure is re-produced with slight modifications from original [16, 43] for the purpose of illustration in this thesis.

of a time stamp which indicates when the event occurred and a string that describes where/why the event occurred [115]. Figure 2.6 illustrates how individual trace events are recorded in a program for different functions or procedures[1].

The emitted trace events are then saved into a kind of storage system. As can be seen from Figure 2.7, monitors record trace events at the time of the executions then all collected tracing data is stored into a trace file[1]. Trace events are most commonly stored in a trace file with a predefined data structure to help analysis tools to read and process the data more efficiently. We use tracing to collect data for HdpH performance analysis in Chapter 4.

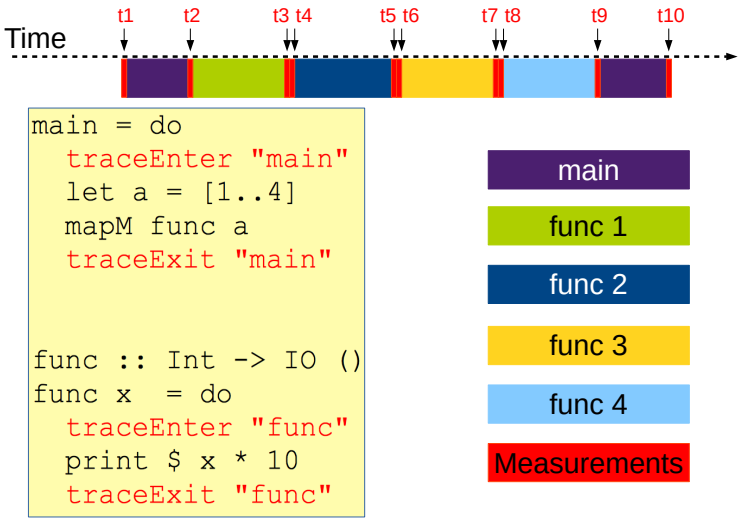


Figure 2.6: Tracing Performance Data.

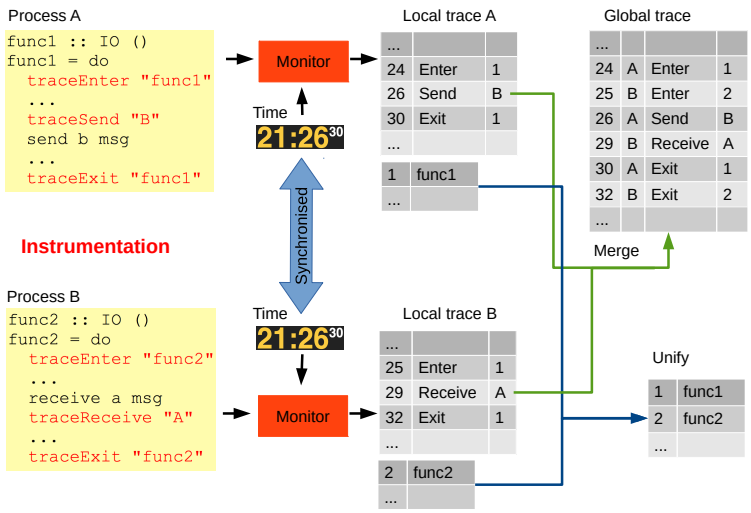


Figure 2.7: Collecting Tracing Data.

Data Analysis

Once the performance has been collected it is analysed [74]. The analysis process starts by reading the raw trace events. After that, the trace events can be processed for categorisation, grouping, calculation of statistics, etc. The results can be shown to the user instantly, saved into a file, or fed to a presentation tool for visualisation.

Presentation

Presenting the performance data is about presenting the profile data in a form that usefully reflects the execution behaviour of the parallel program. For the presentation to be meaningful, it must relate the information in a context [62]. The data presentation can be graphical or text-based. In the graphical case data presentation graphs, such as Gantt charts or Kiviat diagrams [59], are used to map trace events to a physical or logical computation resource, e.g. a processor or a thread. Text-based presentation uses summaries [35], tables, and statistics to provide information about the execution behaviour of the program.

It is very important that the performance data is presented to the user in a way that allows them to identify performance problems [85]. Presenting performance data using graphical visualisation tools is a powerful tool for understanding, tuning, and optimising the behaviour of parallel systems and program execution [74, 53, 39]. Figure 2.8 demonstrates how a Gantt chart can be used to give an insight into the parallel behaviour of an application in terms of processor utilisation by visually presenting trace events to the user[1]. Trace events are used to construct the dynamic behaviour of the parallel program, and time stamps indicate when things happened (x-axis), whereas description is used to identify where things happened (y-axis).

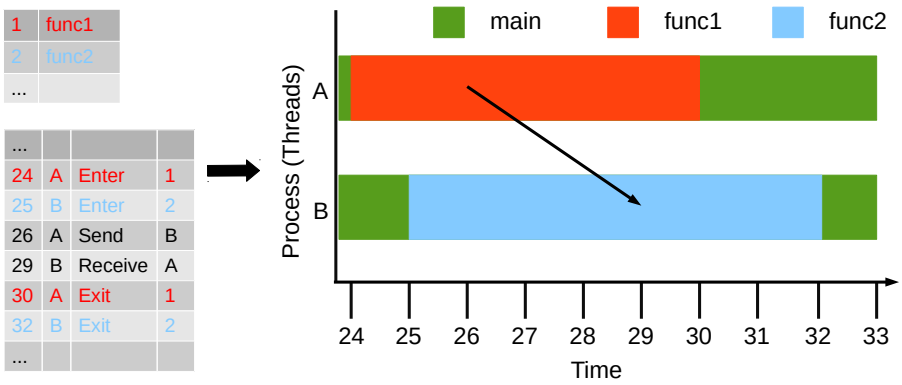


Figure 2.8: Visualising Tracing Data.

Shared and Distributed Memory Profiling

Parallel performance profiling depends on the programming model of the profiled program. Distributed-memory profiling poses several challenges absent in the shared-memory model; for instance, how to profile highly scalable systems, monitor communication, manage multiple trace files, synchronise trace events, and resolve different clock rates. Therefore, profiling distributed-memory parallelism is more challenging than shared-memory parallelism. In Chapter 4 we will present HdpHProf a profiler for the distributed-memory HdpH DSL on a hybrid architecture. It uses tracing to collect HdpH performance data and provides trace analysis tools.

2.4 Parallel Profilers

2.4.1 Imperative Profiling

For decades imperative parallel languages have been supported by a variety of performance analysis tools. Early profilers, like ParaGraph [60], Pablo [117], and XPVM [73], provided parallel programmers with useful information about the parallel execution. Parallel profiling faces new challenges in new architectures; for instance, larger-scale systems and higher-level parallel languages. These challenges encourage the research community to develop advanced performance tools such as mpiP [141], ompP [37], Score-P [124], Vampir [139], Scalasca [123, 42], and TAU [125]. We will use some of the imperative profilers in our study for parallel functional profilers in Chapter 3.

ParaGraph. ParaGraph [60] is a trace based performance analysis tool for message-passing programs. It was probably the commonly accepted performance visualisation technology in the mid '90s [62]. ParaGraph uses the portable tracing library PICL [44] that runs on variety of message-passing parallel computing systems. However, ParaGraph does not support performance analysis of shared-memory parallel programs. It post-processes produced trace information of actual executions to present performance behaviour. ParaGraph provides multiple graphs that depict dynamic behaviour of message-passing parallel applications, and shows overall graphical performance summaries [61]. It provides users with a variety of displays that can give detailed analysis about parallel performance from different prospectives, e.g. utilisation, processor count, and concurrency profile. Figure 2.9 shows some ParaGraph performance graphs, for example, the Kiviat diagram display (bottom-right in Figure 2.9) shows a geometric

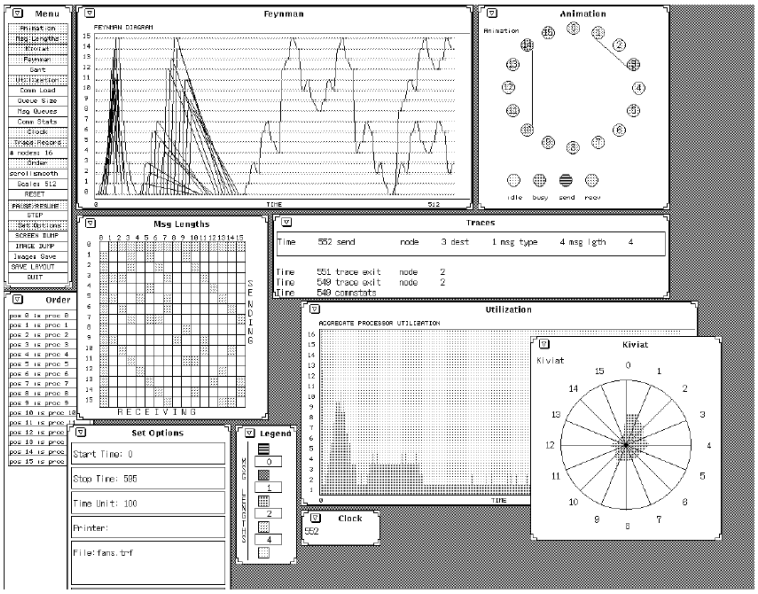


Figure 2.9: Some ParaGraph displays [51]

representation of individual processor's usage and how the overall load is balanced among all processors.

Pablo Toolkit. Pablo [114, 117] is a performance analysis environment that supports a variety of scalable parallel computing architectures. It focuses on scalability, portability, and extensibility. The Pablo environment toolkit consists of two main components: a portable instrumentation library, and a portable software analysis tool [115]. Pablo uses its own Self-Describing Data Format (SDDF) [8] for recording tracing data into a trace file [116]. The analysis of trace data is post-mortem. Pablo analysis tools can present performance behaviour to the user, using graphics to help understand the parallel program behaviour and identify performance bottlenecks [117].

mpiP. mpiP [142, 141] is a profiler for MPI applications. mpiP monitors the performance of MPI by collecting statistical information about MPI functions from the MPI profiling layer. Users can configure mpiP to collect aggregate metrics for statistical analysis [99]. mpiP does not capture all MPI calls; it avoids communication during profiling, and it can limit the profiling scope to reduce the profiling overhead. mpiP has no Graphical User Interface (GUI) and does not provide performance graphs but outputs text profiles to show statistical information about the execution of the parallel program. Figure 2.10 shows an extract of mpiP profile of a parallel program.

```
R00003 openMP_version.c (158-184) PARALLEL
 TID      execT      execC      bodyT   exitBarT   startupT   shutdwnT      taskT
   0       0.13          1       0.10       0.03       0.00       0.00       0.00
   1       0.13          1       0.11       0.03       0.00       0.00       0.00
   2       0.13          1       0.11       0.02       0.00       0.00       0.00
   3       0.13          1       0.13       0.00       0.00       0.00       0.00
 SUM       0.53          4       0.45       0.08       0.00       0.00       0.00
```

Figure 2.11: ompP Profile for Parallel Section.

```
@ mpiP
@ Command : ./mpi_concordanceP 4 /u1/pg/mmaa10/TextFiles/400KB.txt
@ Version                  : 3.3.0
@ MPIP Build date          : Oct 17 2012, 14:27:14
@ Start time               : 2013 01 25 11:26:28
@ Stop time                : 2013 01 25 11:26:32
@ Timer Used               : PMPI_Wtime
@ MPIP env var             : [null]
@ Collector Rank           : 0
@ Collector PID            : 2102
@ Final Output Dir         : .
@ Report generation        : Single collector task
@ MPI Task Assignment      : 0 bwlf01
@ MPI Task Assignment      : 1 bwlf05
@ MPI Task Assignment      : 2 bwlf03

---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0       4.91       3.91    79.52
   1       4.91        4.7    95.72
   2       4.91        4.7    95.68
   *       14.7       13.3    90.30
---------------------------------------------------------------------------
```

Figure 2.10: mpiP profile Header and MPI Time Sections.

ompP. ompP [36, 37] is a performance analysis tool for shared-memory programming with OpenMP. Tracing is used to gather performance data in the memory to produce a post-mortem profile. ompP measures the performance of an OpenMP application by calculating statistical information about the parallel execution. Also, it can support performance monitoring of hybrid applications of OpenMP + MPI [38]. ompP helps identify performance problems, e.g. most time-consuming regions and load imbalance. ompP is similar in spirit to mpiP [141]; it has no GUI and it does not present performance graphs, instead it presents the performance information in a text profile. Figure 2.11 shows an extract of ompP profile for a parallel program.

Score-P and Vampir. Score-P [124] and Vampir [139] are elements of a bigger set of performance analysis tools for optimising the performance of parallel applications. Score-P is used for performance data collection and Vampir is a visualisation and analysis tool.

Score-P is a performance measurement infrastructure for parallel programming. Score-P is highly scalable and can support HPC facilities [17]. It equips its users with tools for profiling, event tracing, and online analysis of parallel application. In addition,

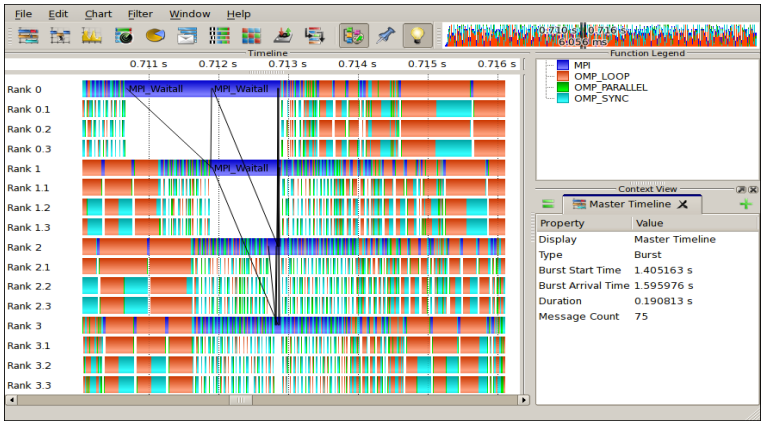


Figure 2.12: Screenshot of Vampir Visualising a Score-P Trace File.

Score-P can work with a number of performance analysis tools [72]; for example, Vampir [139], Scalasca [123, 42], and TAU [125]. Score-P made this possible by adopting standardised output formats such as the Open Trace Format (OTF) [70], the CUBE4 profiling formats [41] and using instrumentation tools like Opari2 [94].

Vampir is a performance analysis tool which was introduced in the mid '90s by Nagel et al. [96] and improved upon to become one of the most advanced and sophisticated performance analysis tools available nowadays [71, 15, 14]. Vampir is a GUI which provides the capability to read, analyse and present graphically the performance monitoring data for different parallel imperative languages, e.g. C or Fortran with MPI, or OpenMP, or CUDA. Vampir provides its users with multiple views to help the understanding of the execution behaviour of parallel programs. Also, it is capable of working on large scale computing infrastructures, e.g. HPC. Figure 2.12 shows a screenshot of Vampir Visualising a Score-P trace file. It offers different performance graphs the can be selected for performance inspection (from the tool bar top-left in Figure 2.12), e.g. process and thread activities and messages between them. Also, it provides overall activity display (top-right in Figure 2.12) depicting the average utilisation of the parallel machine.

2.4.2 Functional Profilers

Parallel functional languages also have performance analysis tools. hpcpp [120] was one of the earliest attempts to profile a parallel Haskell. GranSim [55, 79] was the first performance analysis tool for the parallel implementations of the Glasgow Haskell Compiler (GHC), GpH-GUM [138], and GpH [108]. GranSim provides a variety of performance graphs such as overall activity, threads activity, and granularity profile.

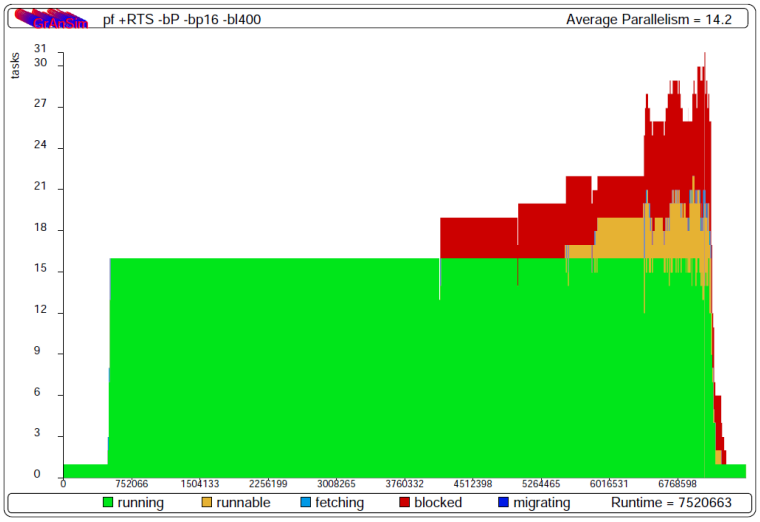


Figure 2.13: An example GranSim overall activity [78]

Other variants of GranSim also have been developed [56, 69]. Influenced by GranSim, more recent parallel Haskell profilers have been developed utilising GUI technology, e.g. EdenTV [11] and ThreadScope [134]. We will present a critical analysis of these two functional profilers in Chapter 3.

GranSim. GranSim [55, 78] is a simulator built around the GHC threaded runtime system. It simulates parallel execution of Haskell programs and provides profiling tools for tuning the performance and granularity of parallel programs. Each thread is given a statistics buffer attached to its Thread State Object (TSO) in which the trace events of the execution are recorded. When the thread terminates, all the contents of the buffer will be dumped to a trace file. Moreover, other important events, such as communication between processes, can be written to the trace file during the execution time. The profiling tools focus on visualising the granularity profile but they also provide general activity profiles such as overall activity, per-processor activity, and per-thread activity. Figure 2.13 shows an example of GranSim overall activity profile for a parallel program where the overall runtime is measured in machine cycles and the average parallelism is determined by the area covered by the continuing green or medium-grey threads.

Eden Tracing and EdenTV. The Eden programming language [12, 80] is supported with performance analysis tools. The profiling system of Eden provides tracing and visualisation.

Eden tracing is the performance monitoring tool for the parallel Haskell Eden [12,

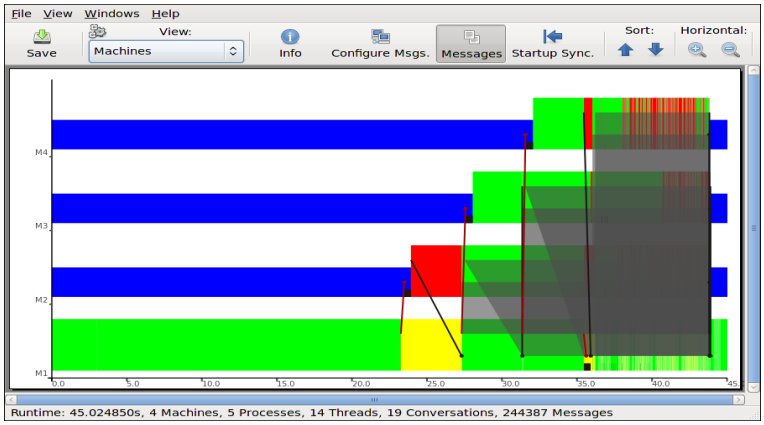


Figure 2.14: Screenshot of EdenTV Visualising an Eden Trace File.

80]. The runtime of GHC-Eden [48] is instrumented to produce trace events. This is can be activated from the runtime options. Previously, Eden tracing was implemented by using the Pablo Toolkit [117] and adopting its SDDF [8] for the trace files. The most recent version of Eden adopts the new tracing of GHC Parallel Profiling System (GHC-PPS) and uses its GHC EventLog Format (GHC-ELF) [67].

The Eden Trace Viewer (EdenTV) is a post-mortem visualisation tool which provides the capability to read, analyse, and present graphically the performance monitoring data (trace file) of the parallel functional language Eden from the level of the parallel runtime system [11]. EdenTV presents Eden's processes mapped to the machines they were executed on, Eden threads activities, garbage collection phases, process generation tree, and the stream of communication between Eden processes on different machines. Figure 2.14 shows an example of EdenTV profile (Machines view) for Eden program where each machine's status is depicted in a coloured bar on the y-axis and the execution time is represent on the x-axis, e.g. green is active, blue is idle. The black lines between the machines shows the stream of messages between the machines.

GHC-PPS. The GHC Parallel Profiling System (GHC-PPS) [67] is the current performance analysis tools for the GHC on multicore [57]. The GHC-PPS consists of tracing facility, analysis tools (GHC-Events Library [49]), and a GUI for browsing the trace events called ThreadScope [134]. Figure 2.15 shows the workflow model of the GHC-PPS.

The GHC-PPS tracing is built into the GHC runtime. To monitor the performance of a program, tracing flags are added to both the compilation and the execution

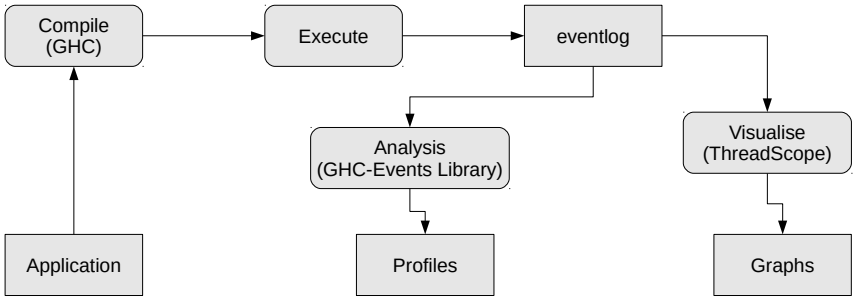


Figure 2.15: Work Flow model of the GHC-PPS.

options (known as event logging in the Haskell community). This will produce a trace file with the GHC EventLog Format (GHC-ELF) called (eventlog).

The GHC-Events library is for reading and processing performance data in the eventlogs [49]. It includes a variety of analysis tools and functionalities that allow the user to investigate the eventlog contents; for example, to sort and print out the trace events to the user in human readable format. Importantly, the library is extensible, so users of the GHC-PPS can develop custom analysis tools to satisfy their needs.

ThreadScope [134] is the post-mortem trace analyser for the GHC-SMP [57]. It is the standard GUI tool to read, analyse, and display performance data generated by the GHC-PPS. Figure 2.16 shows ThreadScope profile of shared-memory parallel Haskell program. The main display of the profile shows (on the y-axis) the overall activity of the program and a list of Haskell Execution Contexts (HECs) beneath it (normally each HEC represents a thread that is mapped to a physical core), and the execution time is represented on the x-axis.

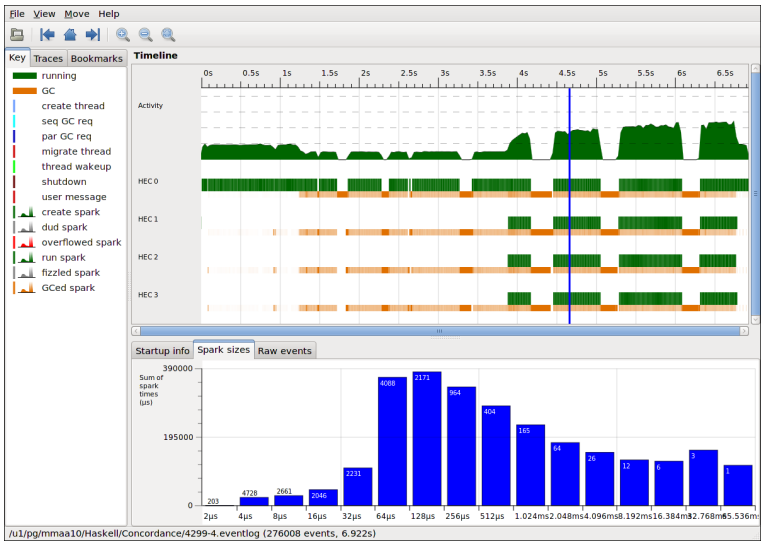


Figure 2.16: Screenshot of ThreadScope Visualising a GHC Trace File.

2.5 Summary

This chapter covered a background about parallel computing, parallel programming languages and profiling parallel performance. We presented parallel systems architectures based on the underlying physical memory organisation: shared-memory architectures, distributed-memory architectures and hybrid architectures(Section 2.1). Later in this thesis we will measure performance on a hybrid architecture, a Beowulf cluster of multicores. We presented parallel programming models that are widely used in the parallel programming domain; shared-memory programming with OpenMP and distributed-memory programming with MPI. Also, we discussed parallel programming with high-level parallel functional languages like HdpH that we will use for the rest of the thesis (Section 2.2). We discussed the analysis of parallel performance, illustrating the performance profiling process that we will use in Chapter 4 to build the profiler HdpHProf. We also discussed performance analysis tools of both imperative parallel languages and functional parallel languages that we will study in Chapter 3 for the comparative analysis of parallel functional profiler (Section 2.4).

Chapter 3

A Survey of Parallel Functional Profilers

This chapter presents a survey of parallel functional profilers alongside important imperative profilers. We evaluate two parallel Haskell profilers, GHC-PPS and EdenTV, in comparison with four important profilers for imperative languages. The functional profilers are relevant as our new HdpHProf profiler exploits GHC-PPS capabilities to profile a distributed-memory DSL, and hence EdenTV is a natural comparator.

The comparison covers profilers of both shared/distributed- memory parallel languages, and is performed on common parallel architectures. The comparison uses a published benchmark, namely the Concordance application which was set as the first SICSA Multicore Challenge [23].

The GHC-PPS performs tracing profiling of shared-memory parallel Haskell, and EdenTV performs tracing profiling of the Eden distributed-memory parallel Haskell. The imperative profilers are the tracing and graphical Score-P/Vampir for MPI, Score-P/Vampir for OpenMP, the two summative profilers mpiP for MPI, and ompP for OpenMP (Section 3.1). We compare the amount of profiling data generated by the profilers classified by whether the parallelism is shared/distributed-memory, whether the profiler is imperative/functional, and tracing/summative. The study reveals some interesting results, e.g. both functional tracing profilers generate one or two orders of magnitude less data than the imperative tracing profilers (Section 3.2).

We investigate the runtime overheads of the profilers, again classified by whether the parallelism is shared/distributed memory, whether the profiler is imperative/functional, and tracing/summative. The results of this study shows, for example, both tracing functional profilers induce overheads of an order of magnitude less than the

imperative tracing profilers. A more complete account of our studies is available as a technical report [5] (Section 3.3). We systematically compare the profilers for usability and data presentation, and found that the results reflect the design philosophy of the tools. Summative tools report a small set of key data with minimal intrusion into program execution. The functional tracing profilers provide more information, together with some graphical visualisation, with little more intrusion. Vampir offers the greatest range of information at the cost of significant intrusion (Section 3.4).

We discuss a number of related studies that evaluate other parallel profilers. We compare the experimental methodology used to evaluate the profilers with the methodology we use to evaluate the functional profilers (Section 3.5). After that, we outline the findings and summarise the work of this chapter (Section 3.6).

3.1 Experimental Methodology

3.1.1 Experimental Set-up

The profilers were measured on a Beowulf cluster comprising 32-nodes, each node comprising two Intel quad-core processors (Xeon E5504) running at 2.00GHz, sharing 4MB of L3 cache and 12GB of RAM. The machines were connected via Gigabit Ethernet and ran CentOS Linux distribution [19] version 6.3 x86_64. Table 3.1 specifies the compilers and profiling tools used for the experiments.

Compiler/Profiling Tool	Version
GNU Compiler Collection (GCC) Red Hat [40]	4.4.6-4
The Glorious Glasgow Haskell Compilation System (GHC) [46]	7.2.1
The Parallel Haskell Compilation System (GHC-Eden) [48]	7.4.2
ompP to profile OpenMP [36]	0.7.1
mpiP to profile MPI [141]	3.3
Vampir to profile MPI & OpenMP [139]	8.0.0 Demo
Opari2 to profile OpenMP [100]	1.0.6
Score-P to profile MPI [124]	1.1
ThreadScope to profile GHC-SMP [134]	0.2.1
EdenTV to profile GHC-Eden [10]	4

Table 3.1: Compilers and Profiling Tools.

3.1.2 Concordance Benchmark Versions

The profilers were compared using implementations of the same algorithm Concordance benchmark that was published as Phase I of the SICSA MultiCore Challenge [23]. The Concordance benchmark takes as input a text file and an integer (N). It processes the text file to find all sequences of words in the text, up to the length of N, together with the number of occurrences of this sequence and a list of start indices. As the profilers work on different languages we obtained four parallel implementations of a Concordance benchmark application, i.e. Eden, GHC-SMP, MPI and OpenMP.

3.1.3 Experiments

We used the Concordance benchmark implementations to measure the profilers' data size and runtime overhead as compared to non-profiling execution. We measured the performance of the profilers in dependence of 2 parameters; i.e. application computation size, and the number of PEs. Firstly, we studied how the increase in the computation size changed the performance of the tools. Secondly, we evaluated how the increase in the number of PEs affected these profilers. The total number of experimental executions was 2400. Each experiment was repeated 5 times and the reported figures are medians.

To increase the computation size, we used different sizes of input files because computation size grows with input size for the concordance. The SICSA MultiCore Challenge provides two input files: the smallest file is 35 KB and the largest is 4300 KB. To carry out the experiments we needed more input files with a gradual increase in size. Therefore, we used the 4300 KB file to produce files with different sizes starting with 100 KB and doubling up to 3200 KB. Our analysis is based on the data sets 100 KB to 3200 KB. However, for completeness we also included the 35 KB and the 4300 KB files in the experiment as they are the standard set of input in the SICSA MultiCore Challenge.

Similarly, we doubled the number of PEs from 1 PE to 8 PEs as this is the maximum number of cores on our system. However, the MPI Concordance implementation requires a minimum of 2 PEs in to work; one as master and the other as worker. As consequence, this study reports the results based on the number of workers used in the computation. The master PE only distributes the work and waits for termination, and hence does not generate profiling data. We also included measurements of

6 PEs as using all the available cores on a machine is known to sometimes perturb performance [88].

3.2 Profiling Data Size

This section investigates the amount of profiling data generated by the profilers. The results are presented in the following order: imperative/functional distributed-memory profilers; imperative/functional shared-memory profilers; then summative distribute/share-memory profilers.

3.2.1 Profiling Data Size in Relation to Computation Size

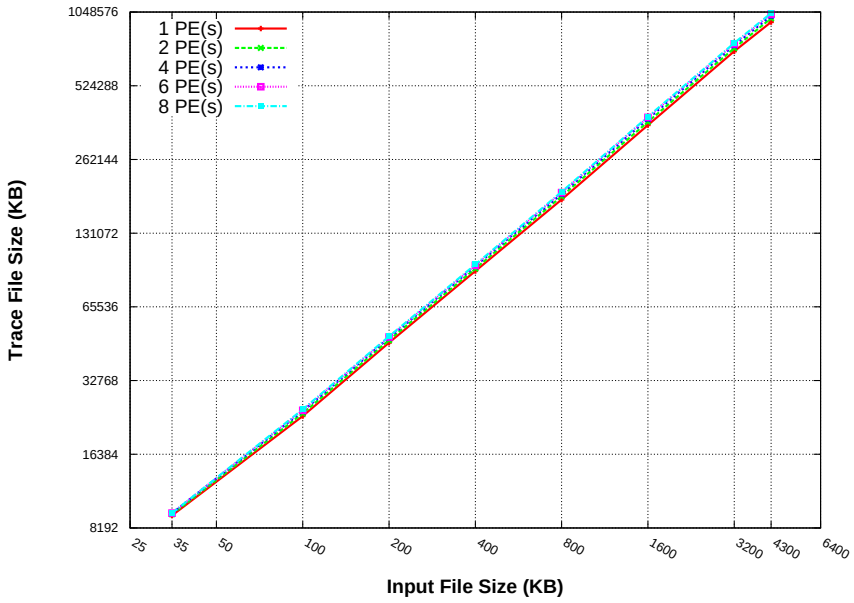


Figure 3.1: Score-P (MPI) Profiling Data Size in Relation to Input Size.

Score-P (MPI). Figure 3.1 shows how the tracing data size of Score-P (MPI) changes as the input size increases. The tracing data size of profiling MPI with Score-P increases substantially as the input size increases. Increasing the input size by a factor of 2 will result in a significant increase to the size of the trace file by about 99% on average.

Eden Tracing. Figure 3.2 shows how the tracing data size of Eden tracing changes as the input size increases. All the curves show an increasing data size. The tracing

data size increases dramatically as the input size gets bigger. Increasing the input size by a factor of 2 causes a significant change in the size of tracing data, on average the tracing data will increase by 160%. The ratios appear to be uniform on 1 and 2 PEs between 82% and 109% but beyond that the ratios are noisy.

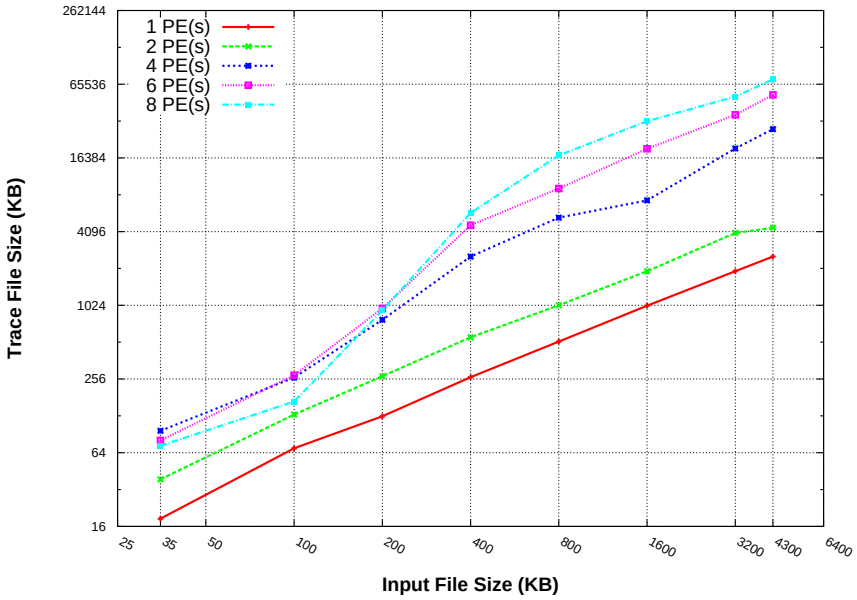


Figure 3.2: Eden Tracing Profiling Data Size in Relation to Input Size.

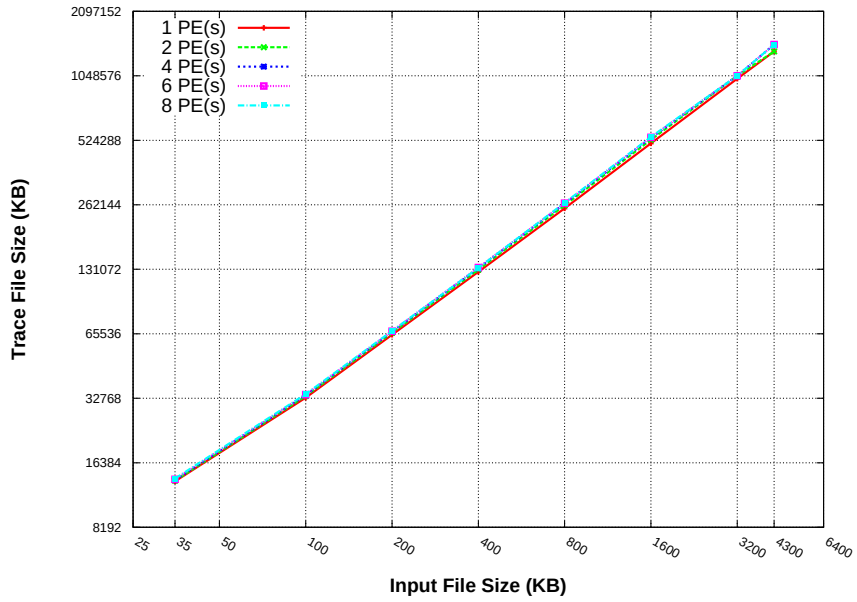


Figure 3.3: Score-P (OpenMP) Profiling Data Size in Relation to Input Size.

Score-P (OpenMP). Figure 3.3 presents how the tracing data size of Score-P (OpenMP) changes as the input size increases. The increase in the profiling data size of profiling OpenMP with Score-P is similar to profiling MPI with Score-P. The tracing data size of profiling OpenMP with Score-P increases dramatically as the input size increases. Our results show that increasing the input size by a factor of 2 will result in a significant increase to the trace data size by about 99% on average.

.

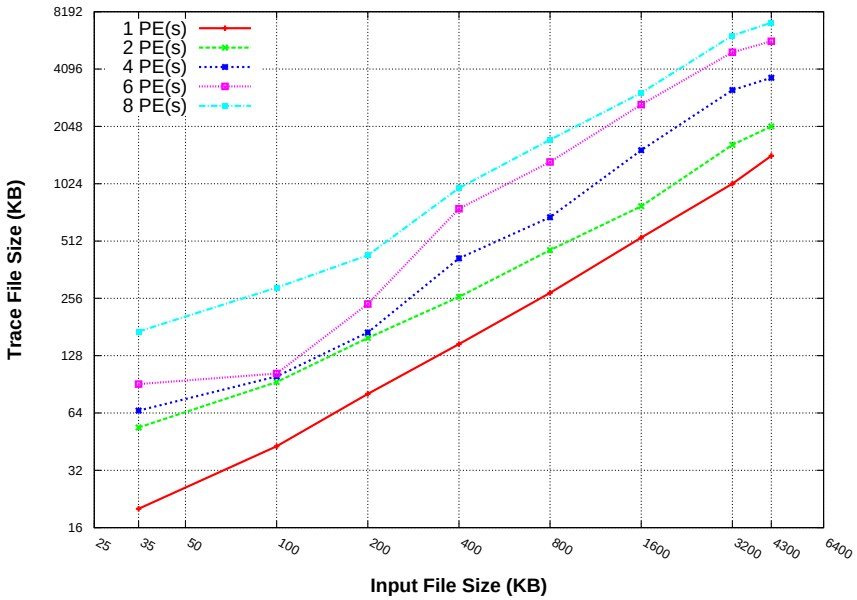


Figure 3.4: GHC-PPS Profiling Data Size in Relation to Input Size.

GHC-PPS. Figure 3.4 shows how the tracing data size of GHC-PPS changes as the input size increases. Again, all the curves show an increase in data size. The tracing data size increases dramatically as the input size gets bigger. Our results show that increasing the input size by a factor of 2 will result in a significant increase in the trace data size; on average trace data size will increase by 90%.

mpiP. Figure 3.5 shows that the profiling data size of mpiP does not change as the input size increases.

ompP. Figure 3.6 illustrates that the profiling data size of ompP does not change as the input size increases. This is to be expected because ompP is a summative profiler like mpiP.

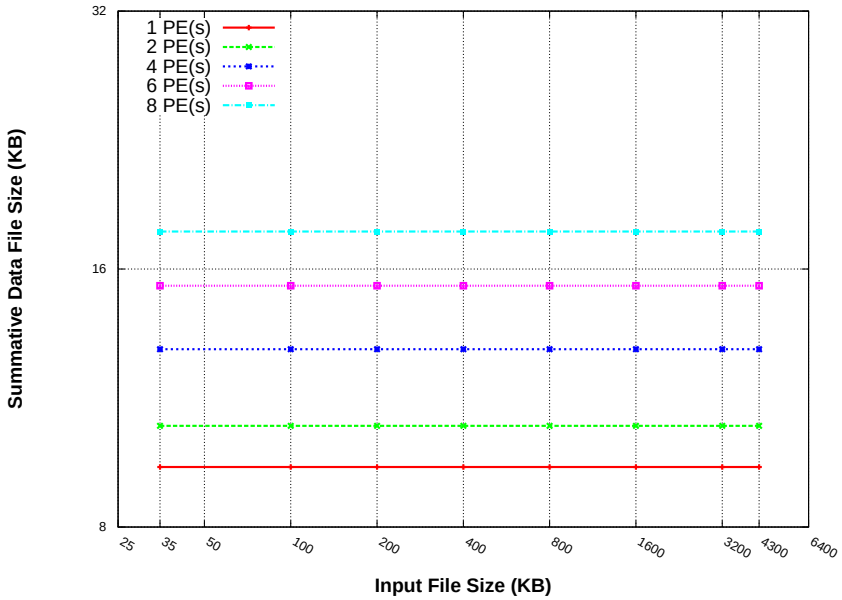


Figure 3.5: mpiP Profiling Data Size in Relation to Input Size.

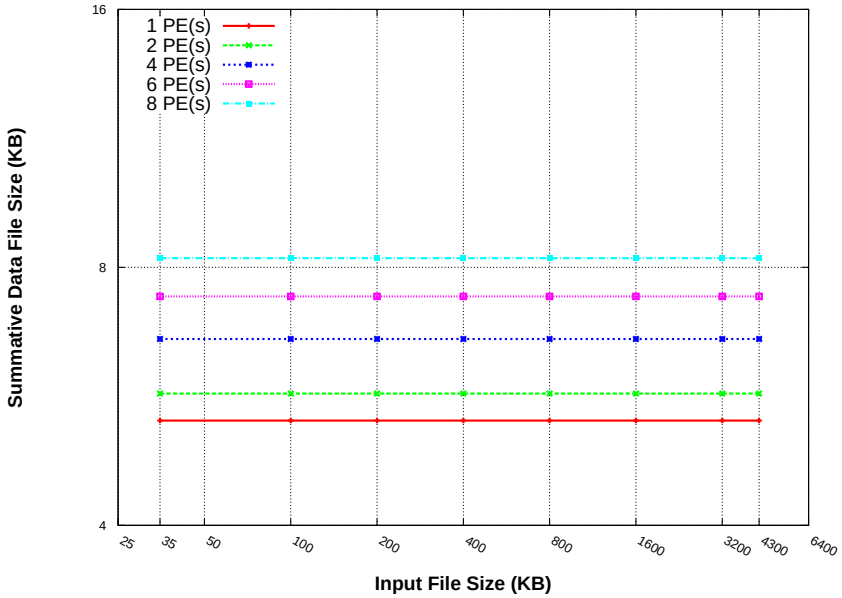


Figure 3.6: ompP Profiling Data Size in Relation to Input Size.

3.2.2 Profiling Data Size in Relation to Number of Processing Elements (PEs)

Score-P (MPI). Figure 3.7 shows how the tracing data size of MPI Score-P changes as the number of PEs increases. The figure shows that increasing the number of PEs will result in a slight increase to the tracing data size when profiling MPI with Score-P.

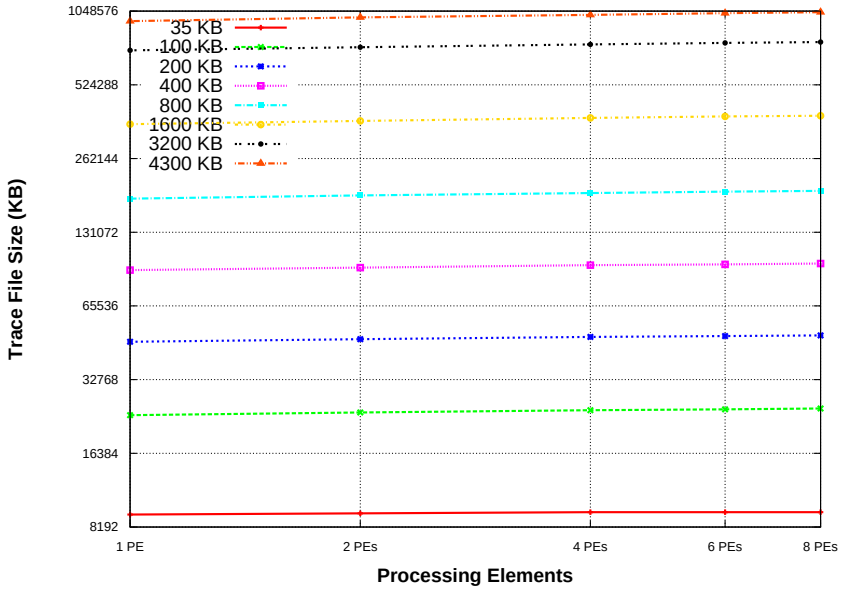


Figure 3.7: Score-P (MPI) Profiling Data Size in Relation to Number of PEs.

Increasing the number of PEs by a factor of 2 will result in a slight increase to the size of the tracing data ranging between 1.5% and 3.1%.

Eden Tracing. Figure 3.8 depicts how the increase in the number of PEs changes the trace data size of Eden tracing. All curves show an increase in data size. We noticed that the 35 KB, 100 KB, and 200 KB curves tailed off at 4 PEs, and may even decrease below 4 PEs. We think that the input size was too small to effectively use more than 4 PEs, and that the data points at 35 KB, 100 KB, and 200 KB below 4 PEs should be disregarded. Increasing the number of PEs by a factor of 2 increases the data size significantly by between 97% and 420%.

Score-P (OpenMP). Figure 3.9 depicts how the increase in the number of PEs changes the tracing data size of Score-P (OpenMP). The increase in the profiling data size of profiling OpenMP with Score-P is similar to profiling MPI with Score-P. We found that increasing the number of PEs only results in a slight increase in the profiling data size. Our results show that increasing the number of PEs by a factor of 2 will result in a slight increase to the size of the tracing data size between 0.1% and 4.0%.

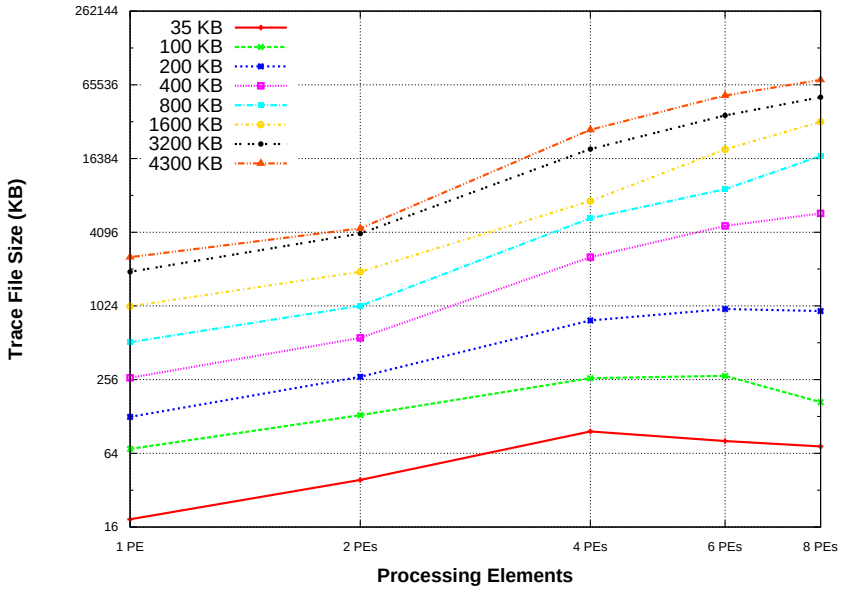


Figure 3.8: Eden Tracing Profiling Data Size in Relation to Number of PEs.

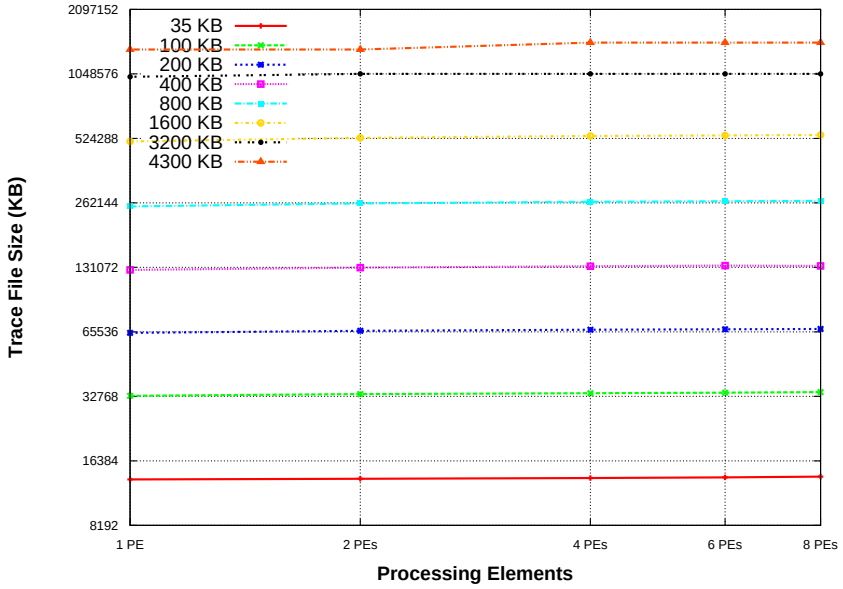


Figure 3.9: Score-P (OpenMP) Profiling Data Size in Relation to Number of PEs.

GHC-PPS. Figure 3.10 shows how the increase in the number of PEs affects the tracing data size of GHC-PPS. As the Figure demonstrates, increasing the number of PEs results in an increase in the tracing data size. We noticed that the smallest data inputs, i.e. 35 KB, 100 KB, and 200 KB, remained fairly steady after 2 PEs, however, they showed an unexpected increase at 8 PEs. This effect was likely caused by the input being too small, and the data points at 35 KB, 100 KB, and 200 KB beyond 2

PEs should be disregarded. Increasing the number of PEs by a factor of 2 will result in increase to the tracing data ranging between 46% and 155%.

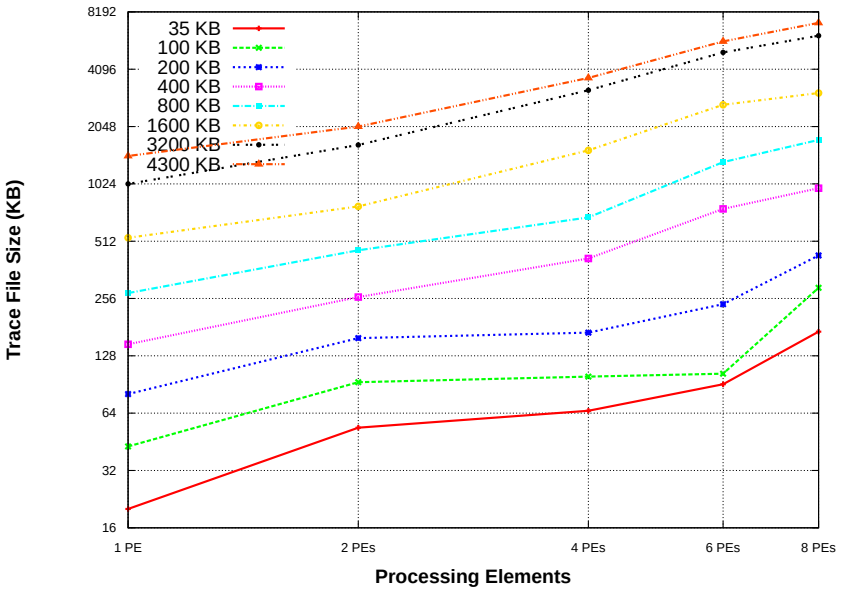


Figure 3.10: GHC-PPS Profiling Data Size in Relation to Number of PEs.

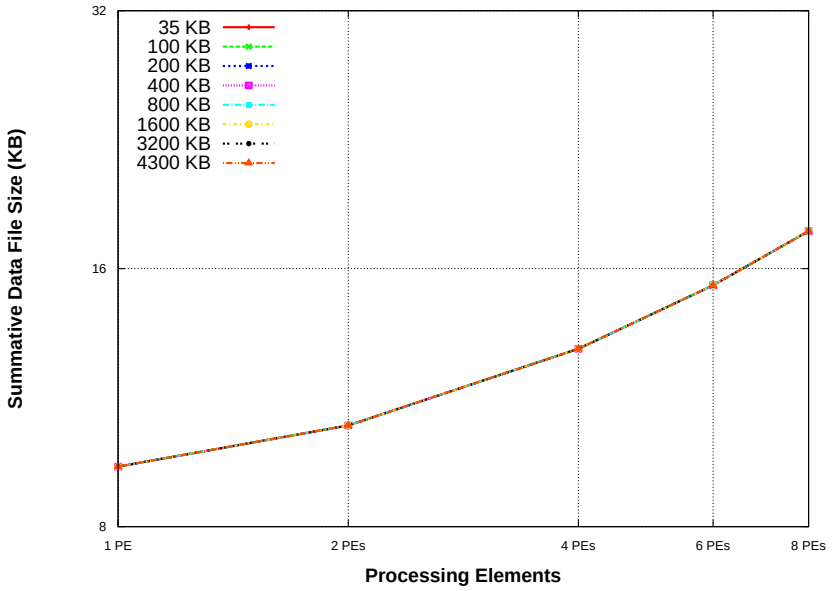


Figure 3.11: mpiP Profiling Data Size in Relation to Number of PEs.

mpiP. Figure 3.11 illustrates how the increase in the number of PEs changes the size of the profiling data of mpiP. Increasing the number of PEs results in an increase in

the summative data size. Increasing the number of PEs by a factor of 2 changes the profiling data by between 11.7% and 37.2%.

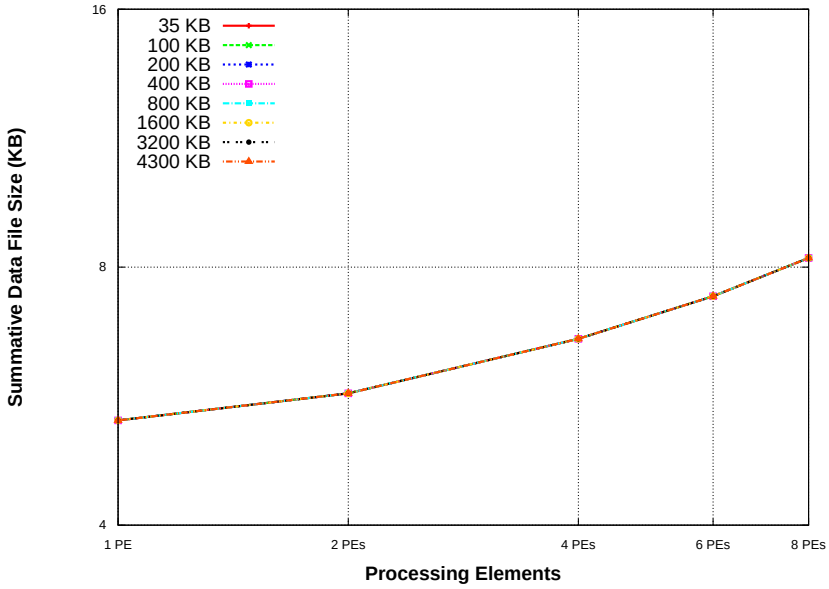


Figure 3.12: ompP Profiling Data Size in Relation to Number of PEs.

ompP. Figure 3.12 illustrates how the increase in the number of PEs changes the profiling data size of ompP. As the Figure shows, increasing the number of PEs increases the profiling data size. We found that increasing the number of PEs by a factor of 2 changes the profiling data size by between 7.5% and 24.2%.

3.2.3 Profiling Data Size Discussion

This section compares the profiling data size of the parallel profilers reported in sections 3.2.1 and 3.2.2. The goal of this comparison is to see how the functional profilers compare to the imperative profilers in terms of profiling data size. We compare the profilers with increasing input size on 4 PEs. Comparisons with other fixed numbers of PEs, and comparisons with fixed input sizes show similar results, and can be found in [5].

Figure 3.13 compares 6 profiling tools in terms of how an increase in the input size changes the profiling data size on a fixed number of PEs. Table 3.2 demonstrates how the profiling data size of these profiling tools are compared to each other in terms of minimum, mean, and maximum values of the profiling data size for each line from Figure 3.13.

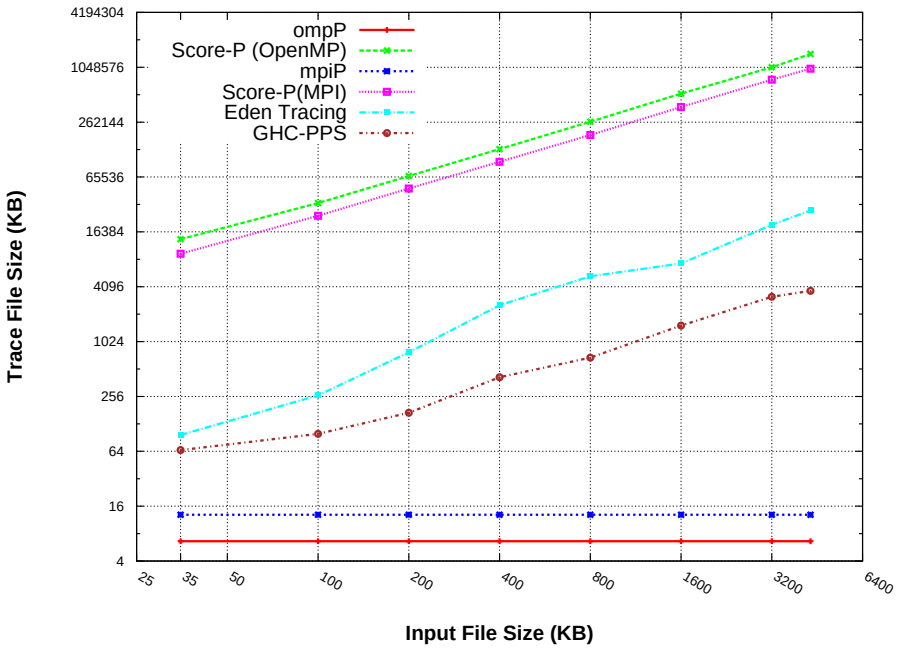


Figure 3.13: Synopsis of Profiling Data Sizes in Relation to Input Size (on 4 PEs).

We based the following comparison on the mean values from Table 3.2(a) and Table 3.2(b). We observed that the profiles generated by Score-P were about two orders of magnitude bigger than those generated by the functional profilers, which in turn were two to three orders of magnitude bigger than the data generated by the summative profilers.

Distributed Memory: Imperative (Score-P) vs Functional (Eden). In Table 3.2(a) the mean profiling data size of Score-P (MPI) was 316 MB; whereas, the mean profiling data size of the Eden tracing was 8014.4 KB. The functional Eden tracing generated significantly smaller profiles than the imperative Score-P (MPI).

Shared Memory: Imperative (Score-P) vs Functional (GHC-PPS). There was an even larger difference between the functional, and the imperative shared-memory profilers. The mean profiling data size of GHC-PPS was 1228.7 KB; whereas, the mean profiling data size of Score-P (OpenMP) was 445 MB, see Table 3.2(b).

The reason why Score-P profiles are so much bigger than profiles generated by the functional profilers is twofold.

- Firstly, Score-P collects many more events than both GHC-PPS and Eden. To give an example, for an input of 800KB (and on 4PEs), Score-P (OpenMP)

(a) Distributed-Memory Profilers

Profiling Data Size (KB)	Imperative		Functional
	Summative	Tracing	Tracing
	mpiP	Score-P (MPI)	Eden Tracing
Min	12.9	**9420.8**	**96.7**
Mean	12.9	**316428.8**	**8014.4**
Max	12.9	**1012121.6**	**28160.0**

(b) Shared-Memory Profilers

Profiling Data Size (KB)	Imperative		Functional
	Summative	Tracing	Tracing
	ompP	Score-P (OpenMP)	GHC-PPS
Min	6.6	**13619.2**	**65.9**
Mean	6.6	**445952.0**	**1228.7**
Max	6.6	**1468006.4**	**3686.4**

Table 3.2: Minimum, Mean, and Maximum Profiling Data Sizes (on 4 PEs).

generates 490 times as many events as GHC-PPS, and Score-P (MPI) generates 8 times as many events as Eden.

- Secondly, Score-P records events in the OTF format [70], which defines a human-readable line-by-line ASCII encoding of events. Even after compression, this format is less compact than the (not directly human-readable) binary formats adopted by GHC-PPS and Eden.

Trace Based vs Summative. Figure 3.13 and Table 3.2 illustrate that the summative profiling tools require significantly smaller storage space for the profiling data than trace based profiling tools. This is to be expected since the summative profiling tools only summarise the parallel execution behaviour in a text format. Moreover, the space reported does not grow with input size, or number of PEs.

3.3 Runtime Overheads of Profiling

This section investigates the runtime overheads of profilers as compared with non-profiling executions. The results are presented in the following order: imperative/-functional distributed-memory profilers; imperative/functional shared-memory profilers; then summative distribute/share-memory profilers.

3.3.1 Runtime Overhead in Relation to Computation Size

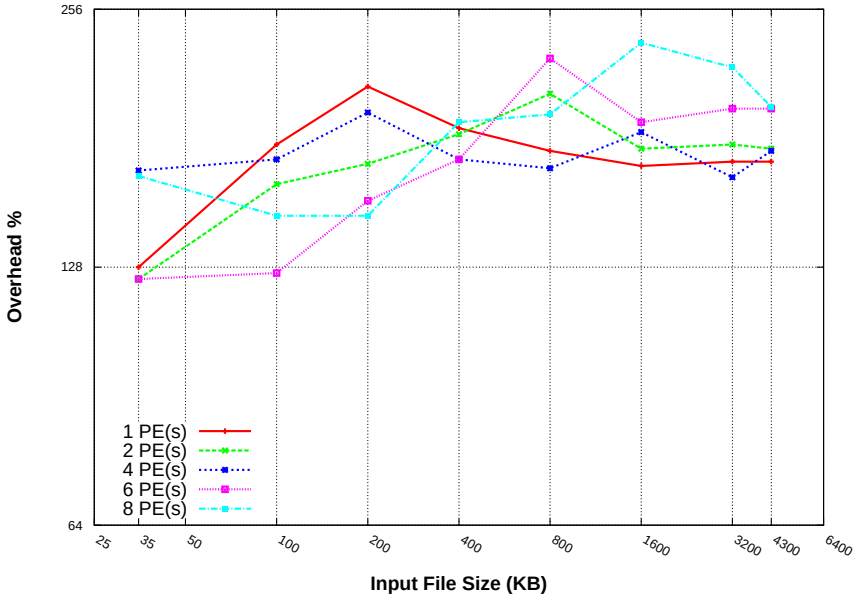


Figure 3.14: Score-P (MPI) Runtime Overhead in Relation to Input Size.

Score-P (MPI). Figure 3.14 shows how the relative runtime overhead of Score-P (MPI) changes as the input size increases. The data is noisy with overhead curves rising slightly as the input size increases from 100 KB to 3200 KB. Overall, the relative runtime overhead of Score-P (MPI) remains between 126% and 234%.

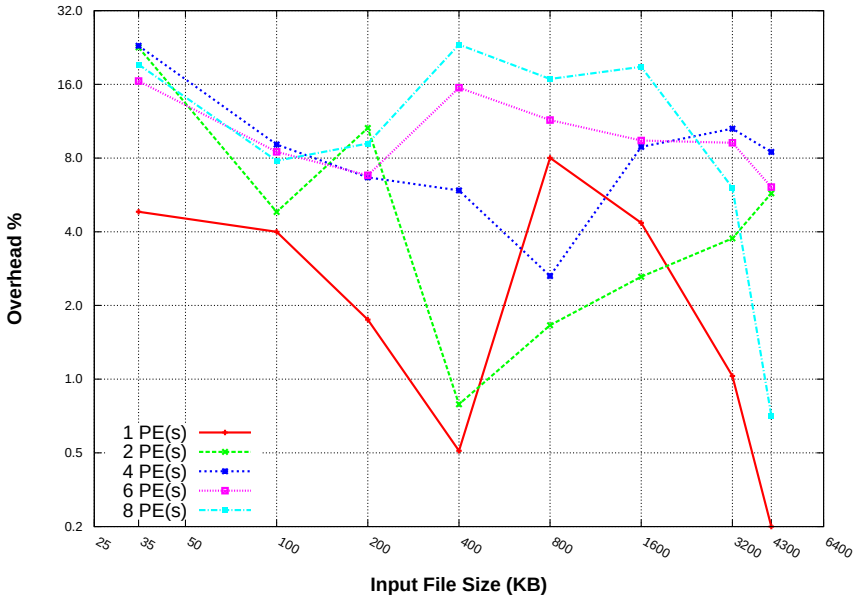


Figure 3.15: Eden Tracing Runtime Overhead in Relation to Input Size.

Eden Tracing. Figure 3.15 shows the relative runtime overhead of Eden tracing. The data is very noisy, with curves fluctuating extremely as the input size increases. It is difficult to determine how increasing the input data size can affect the overhead change as the overhead decreased in some cases, and increased in other cases. However, the majority of curves show that the overhead decreased as the input size increased. Overall the relative runtime overhead of Eden tracing is less than 23% .

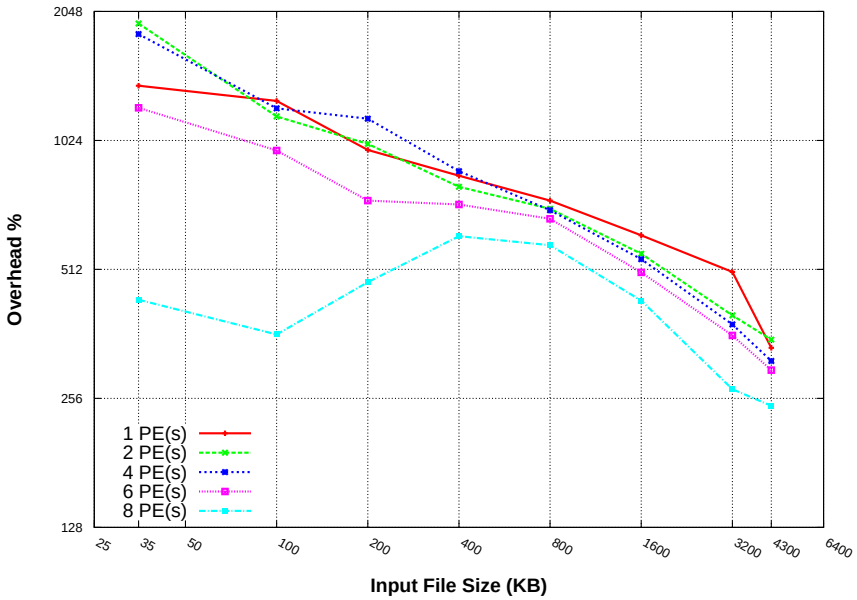


Figure 3.16: Score-P (OpenMP) Runtime Overhead in Relation to Input Size.

Score-P (OpenMP). Figure 3.16 shows the change of relative runtime overhead of Score-P (OpenMP). There is a drop of all curves as the input size increases. We think that the 8 PEs curve is an outlier because of the fact that 8 was the maximum number of cores on our system. The relative runtime overhead of profiling with Score-P (OpenMP) decreases as the input size increases from 100 KB to 3200 KB. Overall the relative runtime overhead remains between 269%, and 1266%.

GHC-PPS. Figure 3.17 shows how the relative runtime overhead of GHC-PPS changes as the input size increases. The data is noisy, and all curves fluctuate widely. There is a decreasing pattern as the input sizes increases. The relative runtime overhead of profiling with GHC-PPS declines as the input size increases from 100 KB to 3200 KB. Overall the runtime relative overhead is less than 20%.

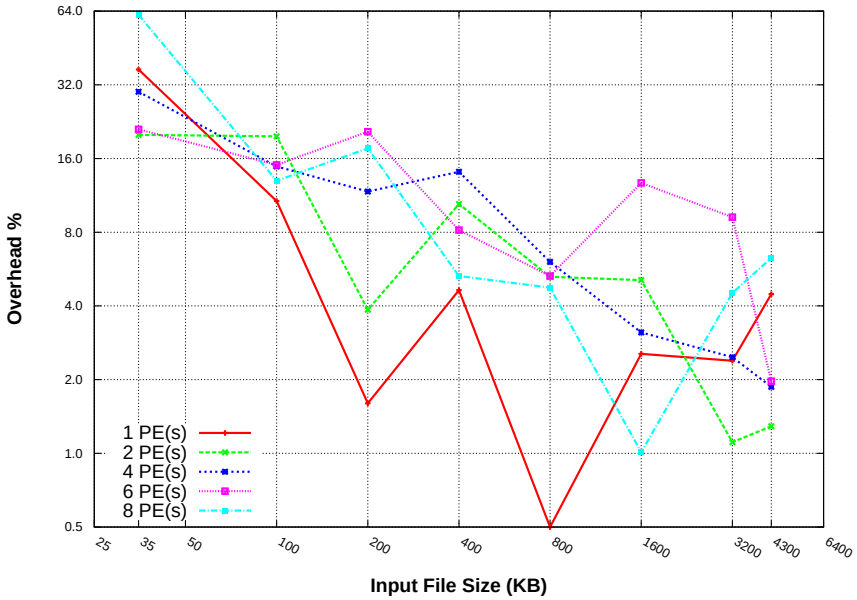


Figure 3.17: GHC-PPS Runtime Overhead in Relation to Input Size.

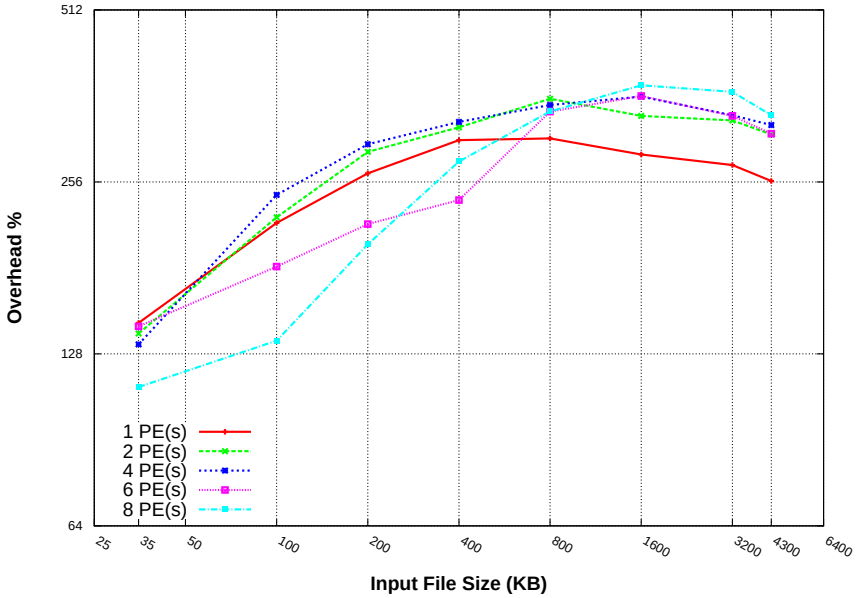


Figure 3.18: mpiP Runtime Overhead in Relation to Input Size.

mpiP. Figure 3.18 shows how profiling runtime relative overhead of mpiP changes as the input size increases. From the figure we can see that there is a growth pattern between all the PEs curves as the input size increases. The figure shows that runtime relative overhead grows as the input size increases. Overall the relative runtime overhead remains between 135% and 378%.

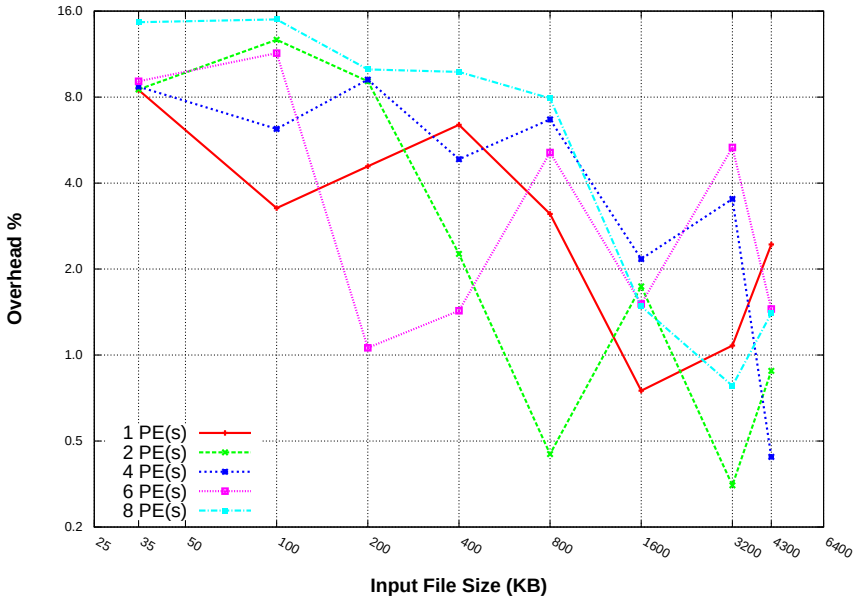


Figure 3.19: ompP Runtime Overhead in Relation to Input Size.

ompP. Figure 3.19 shows how the profiling relative overhead of ompP changes as the input size increases. As the figure illustrates, the data is noisy where all curves fluctuate. However, all curves show that, to some extent, there is a decreasing pattern as the input sizes increases. Generally speaking, we can say that the overhead decreases as the input size increases from 100 KB to 3200 KB. Overall the runtime relative overhead is less than 15%.

3.3.2 Profiling Overhead in Relation to Number of PEs

Score-P (MPI). Figure 3.20 shows how the relative runtime overhead of Score-P (MPI) changes as the number of PEs increases. The data is noisy beyond 4 PEs; and it appears that changing the number of PEs does not change the relative overhead but increases variability.

Eden Tracing. Figure 3.21 shows how the relative runtime overhead of Eden tracing [11] changes as the number of PEs increases. The data is noisy; Eden tracing appears to contribute significant variability to the overheads. However, there is a trend towards increasing relative overheads as the number of PEs increases.

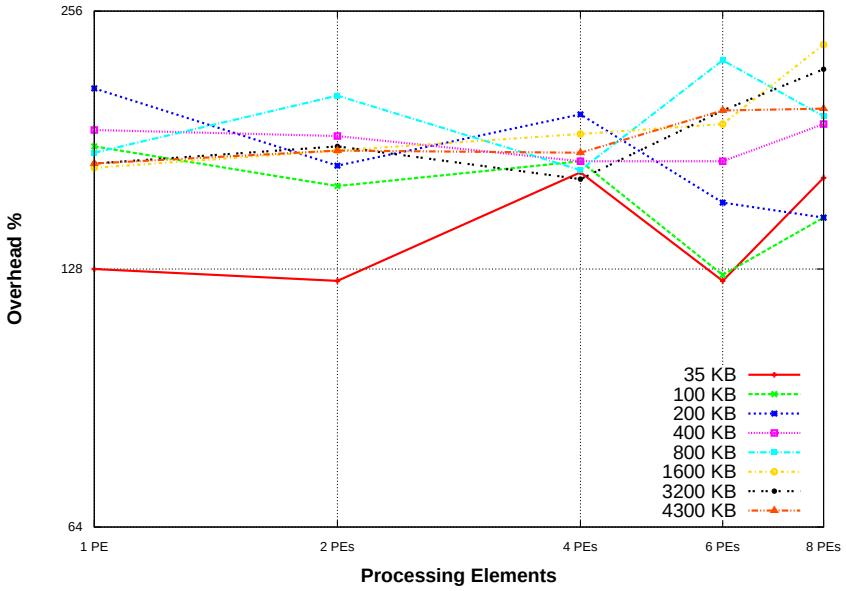


Figure 3.20: Score-P (MPI) Runtime Overhead in Relation to Number of PEs.

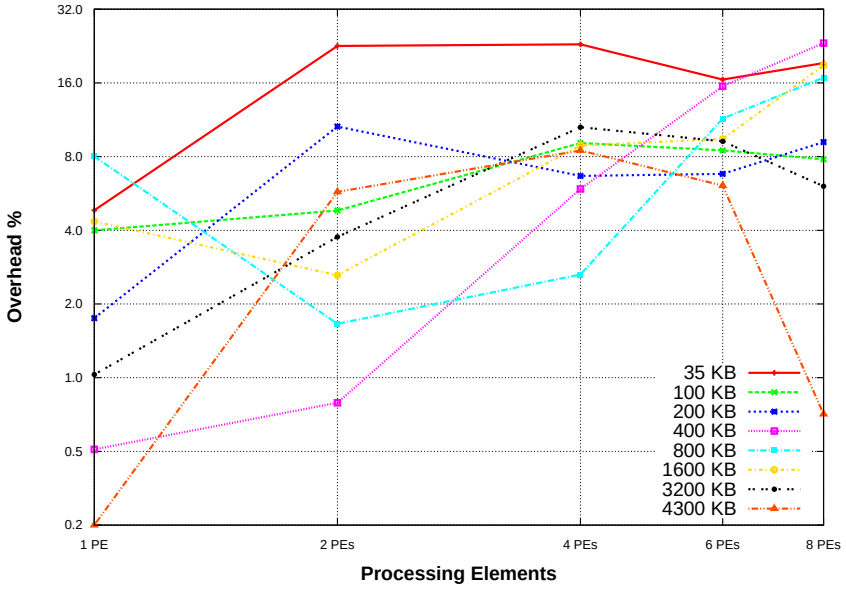


Figure 3.21: Eden Tracing Runtime Overhead in Relation to Number of PEs.

Score-P (OpenMP). Figure 3.22 shows how the relative runtime overhead of Score-P (OpenMP) changes as the number of PEs increases. As the figure demonstrates, the data is fairly steady up to 6 PEs, where increasing the number of PEs does not increase the relative overhead. The sudden drop in relative overheads on 8 PEs for the smaller input sizes 35 KB, 100 KB, and 200 KB are outliers, possibly caused by too little work.

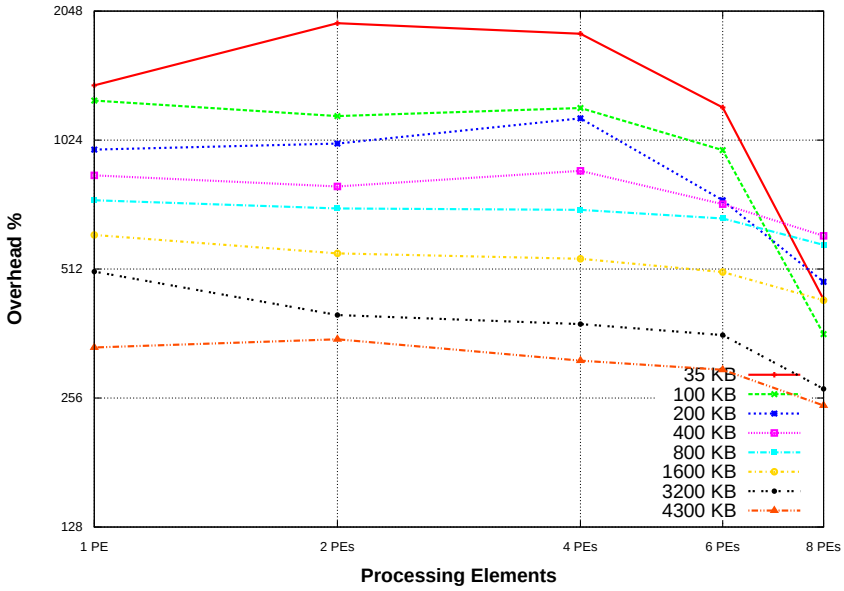


Figure 3.22: Score-P (OpenMP) Runtime Overhead in Relation to Number of PEs.

GHC-PPS. Figure 3.23 shows the relative runtime overhead of GHC-PPS [67]. The data is noisy as for the Eden tracing, and GHC-PPS appears to contribute significant variability to the overheads. However, unlike the Eden tracing, there is no clear trend showing an increase in relative overheads with increasing number of PEs.

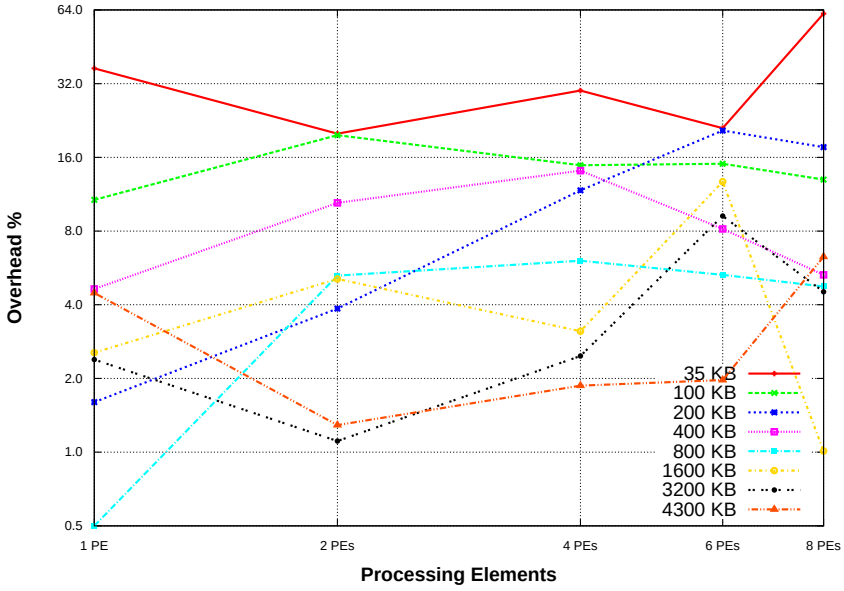


Figure 3.23: GHC-PPS Runtime Overhead in Relation to Number of PEs.

mpiP. Figure 3.24 shows how the profiling runtime relative overhead of mpiP changes as the number of PEs increases. The data is fairly stable up to 4 PEs. As a result,

we can say that the relative runtime overhead does not change as the number of PEs increases. However, it contributes to variability in the overheads.

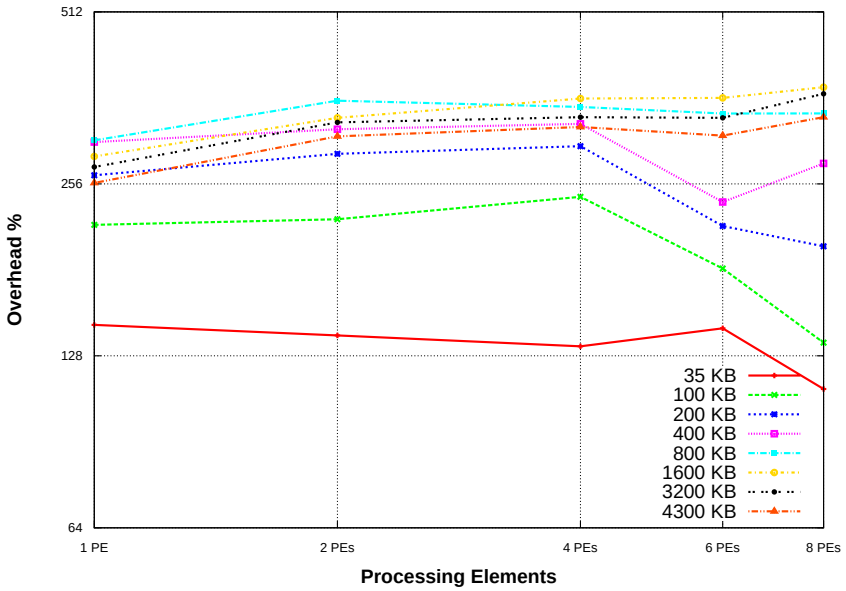


Figure 3.24: mpiP Runtime Overhead in Relation to Number of PEs.

ompP. Figure 3.25 shows how the profiling relative overhead of ompP changes as the number of PEs increases. The data is noisy, with high variability in overheads; there is no clear trend towards an increase in overheads with an increasing number of PEs.

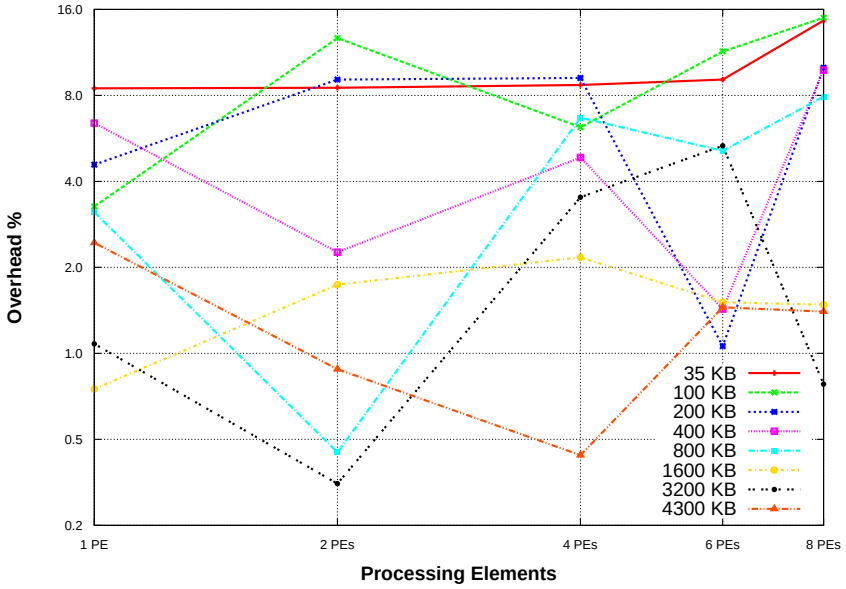


Figure 3.25: ompP Runtime Overhead in Relation to Number of PEs.

3.3.3 Runtime Overhead Discussion

This section discusses the results presented from Sections 3.3.1 and 3.3.2, aiming to compare the functional and imperative profilers in terms of relative runtime overhead. We compare the profilers with increasing Input Size on 4 PEs. Comparisons with other fixed numbers of PEs, and comparisons with fixed input sizes show similar results and can be found in [5].

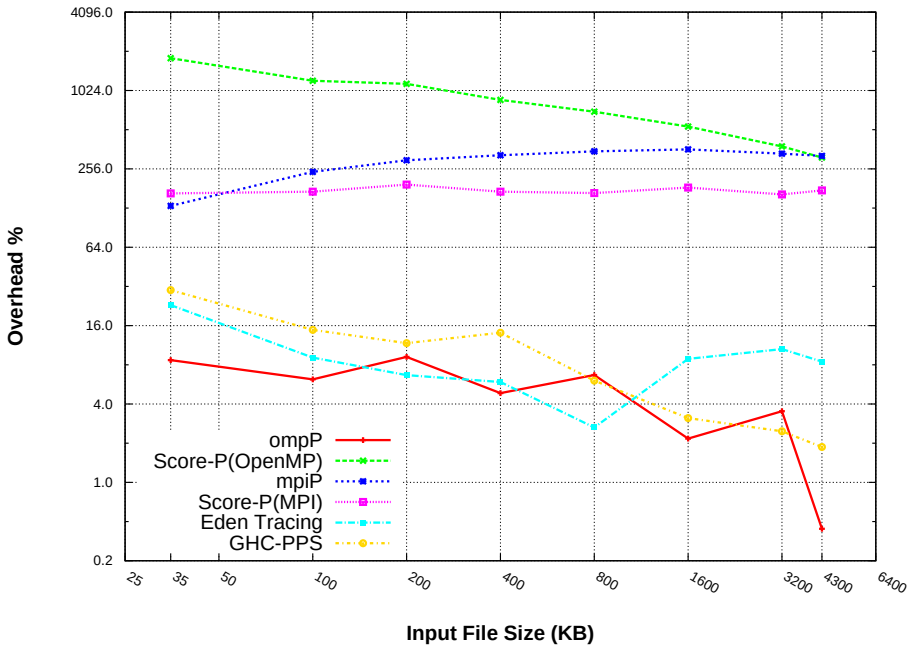


Figure 3.26: Synopsis of Relative Runtime Overheads in Relation to Input Size (on 4 PEs.)

Figure 3.26 compares the relative runtime overhead of the 6 profiling tools. To demonstrate how the overheads of these profiling tools compared to each other we also summarised the minimum, mean, and maximum values of the relative overhead for each curve from Figure 3.26 into Table 3.3(a) and Table 3.3(b). In this comparison we used the mean value from the table for each curve.

The figures show that profilers are clearly divided into two groups; those with high overheads (over 100%), and those with low overheads (below 25%). The group with high overheads comprises, in increasing order of overheads, Score-P (MPI), mpiP, and Score-P (OpenMP). The last has a mean relative overhead of 873%, which is surprising for a shared-memory profiler. The group with low overheads comprises both functional profilers and ompP. Because relative overheads for these profilers are already low, they are also more susceptible to perturbations, resulting in a higher variability

(a) Distributed-Memory Profilers

Overhead %Value	Imperative		Functional
	Summative	Tracing	Tracing
	mpiP	Score-P (MPI)	Eden Tracing
Min	133.00%	**163.00%**	**2.64%**
Mean	295.88%	**173.88%**	**9.40%**
Max	361.00%	**194.00%**	**23.00%**

(b) Shared-Memory Profilers

Overhead %Value	Imperative		Functional
	Summative	Tracing	Tracing
	ompP	Score-P (OpenMP)	GHC-PPS
Min	0.44%	**313.00%**	**1.87%**
Mean	5.22%	**873.50%**	**10.53%**
Max	9.21%	**1813.00%**	**30.00%**

Table 3.3: Minimum, Mean, and Maximum Relative Runtime Overheads (on 4 PEs).

of overheads.

Distributed Memory: Imperative (Score-P, mpiP) vs Functional (Eden). Table 3.3(a) shows the mean overheads of Score-P (MPI), mpiP, and Eden tracing as 173.88%, 295.88%, and 9.40%, respectively. Thus, the relative overhead of both Score-P and mpiP is more than an order of magnitude higher than the overhead of Eden. The higher overheads of Score-P (MPI) may be partially explained by the fact that Score-P (MPI) collects more data in a less compact format than Eden, as discussed in Section 3.2.3. However, this cannot explain the overheads of mpiP, which collects far less data than Eden.

Shared Memory: Imperative (Score-P, ompP) vs Functional (GHC-PPS). The mean overheads of Score-P (OpenMP), ompP, and GHC-PPS are 873.50%, 5.2%, and 10.5%, respectively (see Table 3.3(b)). Surprisingly, the overhead of Score-P (OpenMP) is almost two orders of magnitude higher than GHC-PPS and ompP.

Trace Based vs Summative. Here, the picture is not clear cut. The summative ompP has the lowest overhead (5.2%), yet this is closely followed by the trace based Eden (9.4%) and GHC-PPS (10.5%). Despite being a summative profiler, the overheads of mpiP (295%) are greater than the overheads for the trace based Score-P (173% on MPI).

3.4 Data Presentation and Visualisation

This section compares the performance data presentation tools of the functional profilers and imperative profilers. We critically compare the main features and facilities of these tools. In particular, we investigate how the visualisation tools of parallel Haskells i.e. EdenTV and ThreadScope, compare to Vampir, the well established technology which is used by mainstream manufacturers for visualising the performance data of imperative parallel languages. Table 3.4 summarises how the visualisation tools of functional profilers compare to imperative profilers.

3.4.1 Programming Model

The Programming Mode section of Table 3.4 highlights the differences between the tools. Vampir can visualise the performance data of multiple programming models, e.g. MPI [96], OpenMP, MPI + OpenMP [15], or MPI + Accelerator [13]. mpiP can only support MPI [141]. ompP mainly supports OpenMP [37], but can also profile hybrid applications, e.g. MPI + OpenMP [38]. EdenTV only supports the distributed-memory parallel Eden [11]. Likewise, ThreadScope only supports the shared-memory parallel Haskell (GHC-SMP) [67].

In terms of the variety of programming models and parallel languages that these tools support, we can say that Vampir is more flexible. Moreover, EdenTV and ThreadScope are both visualisation tools for two parallel variants of the general purpose programming language Haskell, which are Eden and GHC-SMP respectively. Nonetheless, EdenTV cannot present the parallel behaviour of GHC-SMP, and ThreadScope cannot present the parallel behaviour of Eden.

3.4.2 Presentation of Performance Data

Vampir, EdenTV, and ThreadScope provide the user with a browser to visualise graphically the performance data as well as textual data. However, mpiP and ompP do not provide such a facility to the user; instead the performance data is summarised into a text file. Presenting the performance data graphically is important because graphs can help the user to quickly identify any performance problems. Text based visualisation is useful for presenting statistical information, e.g. ratios, counters, and repetitive patterns about the parallel behaviour.

	Features	Imperative Profilers			Functional Profilers	
		Vampir	mpiP	ompP	EdenTV	ThreadScope
Programming Model	Distributed Memory	+	+	-	+	-
	Shared Memory	+	-	+	-	+
	Hybrid	+	-	+	-	-
Data Presentation	GUI	+	-	-	+	+
	Graphs	+	-	-	+	+
	Text Profile	+	+	+	+	+
Software Properties	License (Open Source)	-	+	+	+	+
	Interoperability	+	N/A	N/A	-	-
	Heterogeneity	+	-	+	-	-
	Scalability	+	+	N/A	-	N/A
Usability	Zooming	+	-	-	+	+
	Filtering	+	-	-	-	-
	Find	+	-	-	-	-
Visualisation Displays/Views	Machines	+	+	N/A	+	-
	Processes	+	-	N/A	+	N/A
	Threads	+	-	+	+	+
	Synchronisation	+	-	+	-	-
	Messages	+	+	-	+	-
	Communications	+	-	-	+	-
	Overall Activity	+	-	-	-	+
	Two Profiles Comparison	+	-	-	-	-

Table 3.4: Synopsis of Visualisation Tools.

3.4.3 Software Properties

Performance Visualisation tools are software systems which are used by programmers to tune and improve the behaviour of their parallel programs [60, 51]. A software system has properties that can make it the ideal choice for its users. Here we will compare these visualisation tools based on their software properties as shown in Table 3.4 (Software Properties).

License. The only proprietary visualisation tool is Vampir; whereas, mpiP, ompP, EdenTV, and ThreadScope are all open source software.

Interoperability. Here our concern is the ability of the visualisation tool to process trace files from different trace generation tools, and Vampir is the only interoperable tool. Vampir can process trace files of the format OTF2 [104], which is the standard trace format adopted by performance monitoring tools of Vampir, i.e. Score-P [72] and VampirTrace [140]. In contrast, EdenTV and ThreadScope are not interoperable visualisation tools, e.g. EdenTV can only process trace files generated by GHC-Eden.

Heterogeneity. In this context heterogeneity is the ability of the tools to visualise the performance of heterogeneous parallel applications, e.g. MPI + OpenMP. Vampir and ompP are the only tools that present performance information of heterogeneous applications. ompP can only profile OpenMP applications or hybrids of MPI + OpenMP [38]. In contrast, Vampir is more heterogeneous since it can support hybrid applications of different paradigms, e.g. MPI + Accelerator + Threads, MPI + CUDA, PGAS + CUDA, and MPI + PGAS [13].

Scalability. Scalability means a tool can visualise long executions on large numbers of processors [53, 68]. Large-scale problems mean performance data gathered becomes increasingly more challenging to process [111]. In particular, scalability is an important issue for visualising the performance of distributed-memory applications since the number of processors on distributed-memory grows exponentially. However, scalability is not such a critical issue for shared-memory applications because the number of processors in a single shared-memory machine is limited and typically small. Therefore, we compared the imperative visualisation tools for distributed memory with the functional visualisation tools for distributed memory, i.e. Vampir and mpiP vs EdenTV.

Vampir is designed to target scalability [14]. Vampir can scale to a large number of processors, and can process large numbers of execution events, e.g. up to 220,000 cores and up to 10^{12} recorded events [13]. In addition, mpiP can scale up to 65536 processes [141]. However, we could not find any publication claiming EdenTV to be scalable, or revealing the maximum number of processors it can handle. On the other hand, we have used EdenTV to profile a Concordance application on a Beowulf cluster, and found that increasing the number of processors significantly increased the size of the trace file. When the trace file becomes too big to fit into main memory, EdenTV

is not able to open it. Therefore, EdenTV's scalability is limited to common cluster architectures.

3.4.4 Usability

We focus on three main facilities which we think important to help identify performance problems. These are zooming in and out of the performance graphs, filtering the performance data to show a particular group of events in the performance graphs, and finding a specific event in a performance graph. All profiling tools with GUI provide zooming facilities, i.e. Vampir, EdenTV, and ThreadScope. However, the graph zooming of both EdenTV and ThreadScope is quite basic. In contrast, Vampir has more advanced zooming facilities; for example, the user can use the mouse to select a specific part of the performance graph to be magnified on the screen. Moreover, Vampir is the only profiling tool that provides filtering and finding facilities. In terms of the variety of performance graphs each profiling tool provides a selection of views, see Table 3.4 (Displays/Views). However, we emphasise that Vampir provides more views than the other profiling tools.

3.4.5 Discussion

Functional profilers provide good facilities for identifying parallel performance problems. However, comparing them with imperative profilers shows that functional profilers can benefit from the more mature imperative profilers. Our study shows that imperative profilers support different programming models, provide more facilities, and have adopted standardised formats. In addition, scalability is an important feature of imperative distributed-memory profilers. However, we found that scalability has not been considered in EdenTV. Furthermore, even though EdenTV and ThreadScope are both visualisation tools for two variants of the functional language Haskell, they are based on different trace file formats. This means that neither can display trace files produced for the other, making them incompatible systems. The tools in the established imperative world such as Vampir [139], Scalasca [123], Periscope [105], and, TAU [132], already address challenges such as interoperability, heterogeneity, and scalability, as these are important demands for profiling parallel performance [72]. Therefore, the functional tools would have to do the same.

Since Eden processes communicate via MPI message-passing one could in princi-

ple use a standard MPI profiler, like Score-P (MPI), to profile Eden programs. While such an approach can provide some summary information, e.g. about overall resource utilisation, it is insufficient to profile high-level parallel functional languages. Berthold et al. [11] for instance, investigated whether the imperative profiler XPVM [73] could be used to profile Eden programs (using PVM-based message passing [109]) and found that XPVM lacks the ability to relate the gathered performance data to the Eden language constructs, e.g. processes and threads, that matter to the programmer.

3.5 Related Work

Chung et al. [21] investigated how to reduce the cost of tracing by selectively recording only certain classes of events using a set of standard HPC profiling tools. They evaluated their approach with an experimental study of the cost of five profiling tools: IBM HPCT, Paraver, KOJAK, TAU, and mpiP. In a similar approach to our work, they used two metrics to characterise the profiling tools: the runtime overheads and the size of the collected profiling data. There are a number of differences between their study and our work. Firstly, their study was restricted to one programming model, imperative programming with MPI, whereas we covered a range of different programming models (shared vs distributed memory) and paradigms (imperative vs functional). However, their study used 4 benchmark applications, whereas we were limited to one because it was difficult to find multiple and similar benchmarks for all the programming models we considered. Finally, their study investigated the cost of profiling on a larger scale than ours did. We selected small numbers of processors, so that we could compare profiling tools for both distributed and shared memory, inheriting the low processor limit of shared-memory architectures.

Malony et al. [87] investigated overhead compensation in a prototype extension of the TAU [125] profiling tool. They performed experiments to evaluate the performance of their tool, measuring the runtime overhead but not the data size of profiles. However, they did not vary the computation size or the number of PEs in their experiments. They also did not compare their results with the overheads of other profilers.

Jones Jr. et al. [67] introduced the GHC parallel profiling system and the ThreadScope visualizer to the Haskell community. To demonstrate the overheads of parallel profiling, the paper presents the runtime overheads and trace file sizes of two microbenchmarks (parallel Fibonacci and parallel quicksort). However, the authors do

not investigate the impact of computation size and number of cores on the cost of profiling, nor do they compare their overheads with those of other profiling tools.

3.6 Summary

We have evaluated two functional profilers, GHC-PPS and EdenTV, alongside four important imperative profilers. The comparison is based on the SICSA Concordance benchmark [23], which covers both shared and distributed-memory parallel languages, and is performed on common parallel architectures.

Key findings are as follows: The summative profilers generated the least profiling data. More interestingly both functional tracing profilers generated one or two orders of magnitude less data than the imperative tracing profilers. While generating so much data risks distorting the parallel execution, the benefit is that tools like Score-P/Vampir can potentially assist the programmer by providing more detailed information about program execution (Section 3.2). More work is needed to establish the cost/benefit trade-off between profiling data size and the programmer's understanding of program behaviour.

Both tracing functional profilers induce very low runtime overheads by an order of magnitude less than the imperative tracing profilers. Both functional profilers runtime overheads, whether distributed or shared-memory, are no more than twice as much than the best summative profiler in our study: for example, 9.4% of runtime overhead for EdenTV and 10.5% for GHC-PPS compared with 5.2% for ompP (Section 3.3).

Comparing the profilers for usability and data presentation, we see that the functional profilers are relatively immature when compared with tools like Vampir for popular imperative technologies. The results also reflect the profiler design philosophies: summative tools provide key information with minimal intrusion. The functional profilers provide more information and some graphical visualisation; Vampir offers the greatest range of information, and the most sophisticated and usable visualisation tools (Section 3.4).

Functional profilers could be improved in a number of ways. Currently the data collection and visualisation options are relatively modest, and both could be improved to approach the standard of leading tools like Vampir. Functional profiling architectures could better exploit techniques proven by tools like Vampir. For example, instead of different visualisation tools to visualise two variants of parallel Haskell, one

tool could be designed to visualise multiple variants. Similarly, instead of producing different trace formats for each Haskell variant, a standard format is needed which can capture monitoring data from a more generic abstract unit of computation resource. While GHC-PPS represents a move in this direction, it is closely entwined with GHC and has a relatively simple model of computation resources.

Interesting challenges lie ahead: functional profilers must soon address the issues of scalability and heterogeneity. The scalability challenge is to collect useful information as the number of cores grows exponentially and the bandwidth available to each core shrinks. The challenge of heterogeneity is to profile a program executing on a range of computing resources, e.g. multicores and GPUs.

Moreover, since Haskell has become the host language of several DSLs implementations, another important challenge for functional profilers is how to support profiling parallel DSL. The DSL profiling challenge is the ability of the profiler to monitor, analyse the performance, and present the behaviour from the high-level abstraction of the DSL. It is unclear how much profiling technologies can be shared by the various parallel Haskell DSLs like the Par Monad [89], Cloud Haskell [28], and HdpH [84].

Chapter 4

HdpHProf– Design and Implementation

This chapter presents the design and implementation of HdpHProf, a profiler for the HdpH DSL. In keeping with the HdpH philosophy of relying on nothing but the host platform, HdpHProf builds on GHC's existing profiling infrastructure, in particular on the event logging mechanism of the GHC Parallel Profiling System (GHC-PPS). HdpHProf is post-mortem, multi-stage, and extensible. Importantly, the implementation exploits several new GHC features, including the GHC-Events Library and ThreadScope, to build profiling tools for HdpH. HdpHProf faces some challenges unique to the high-level distributed-memory DSL setting: how to instrument and trace the behaviour of the parallel DSL, how to tweak event logging to generate a single profile of a distributed program execution, spanning multiple machines with independent clocks, and how to analyse and visualise such trace files. The design introduces two novel analysis tools for monitoring the DSL internals, i.e. *Spark Pool Contention Analysis* and *Registry Contention Analysis*. Furthermore, we present how HdpHProf uses ThreadScope [134], the standard GHC shared-memory performance analysis tool, to visualise the performance of the distributed-memory executions of HdpH.

4.1 HdpHProf Requirements

The requirements for HdpHProf to profile HdpH are to use the available performance analysis infrastructure from the host language, i.e. GHC [57], to profile HdpH. The GHC compiler comes with a full profiling suite called the GHC Parallel Profiling System (GHC-PPS) [67] and a trace visualiser, ThreadScope [134]. We can categorise

HdpHProf requirements into three different types as follows.

Architecture Requirements:

- HdpHProf should not require any change to the GHC platform.
- HdpHProf should use the GHC-PPS tracing to emit HdpH trace events into the eventlogs produced by the GHC-PPS on each node.
- HdpHProf should use and extend the GHC-Events library to read HdpH trace events from the eventlog, normalise the HdpH RTS start time in each eventlog and synchronise the time in the eventlogs accordingly, and merge the multiple eventlogs from a distributed run.

Functional Requirements:

- HdpHProf should provide analysis tools for HdpH performance, e.g. spark pool contention analysis and registry contention analysis.
- HdpHProf should use ThreadScope to browse the eventlogs and see how HdpH utilises the cores of a Beowulf cluster of multicores.

Performance Requirements:

- HdpHProf should scale to profile HdpH applications on clusters of multicores with large number of cores, e.g. 192 cores of a 32-node Beowulf cluster.
- HdpHProf should induce low tracing overheads to the GHC-PPS and the profiled applications.

4.2 HdpHProf Implementation Design

This section presents the design of HdpHProf. HdpHProf is a performance analysis tool for the high-level distributed parallel Haskell (HdpH). We explore the feasibility and issues of profiling a parallel DSL using the host language profiling tools. HdpHProf aims to help HdpH programmers to tune and improve the performance of their parallel programs and help HdpH implementers to debug the HdpH RTS. HdpHProf is a post-mortem, multi-stage and extensible time profiler which takes a new approach

to constructing profiling tools for the parallel DSL HdpH. The approach that HdpH-Prof takes is to use off-the-shelf commodity profiling tools of a general purpose parallel language and adapt the tools to profile the parallel DSL; in our case HdpH. HdpHProf uses the GHC-PPS tracing and the GHC-Events Library as a base to profile HdpH.

HdpH is implemented in vanilla Concurrent Haskell [84]. This means that HdpH can be profiled using the GHC-PPS. However, doing so will capture the performance behaviour of only one of the HdpH virtual parallel machines. Therefore, HdpHProf is designed to profile the performance of HdpH on all nodes and collects performance data that captures the behaviour of the parallel DSL. For example, HdpH trace events that reflect what is happing at the HdpH level are emitted into the trace file.

Figure 4.1 illustrates the work flow design[1] of HdpHProf, where an application is compiled with GHC with GHC-PPS enabled. The executor takes the compiled application, along with a list of a cluster nodes, and executes the application on these nodes to produce eventlogs; one per node. Eventlogs can be analysed before or after being merged as required to produce summative performance profiles. Similarly, an eventlog of single nodes can be visualised separately or after being merged with all eventlogs to see overall performance on the parallel architecture.

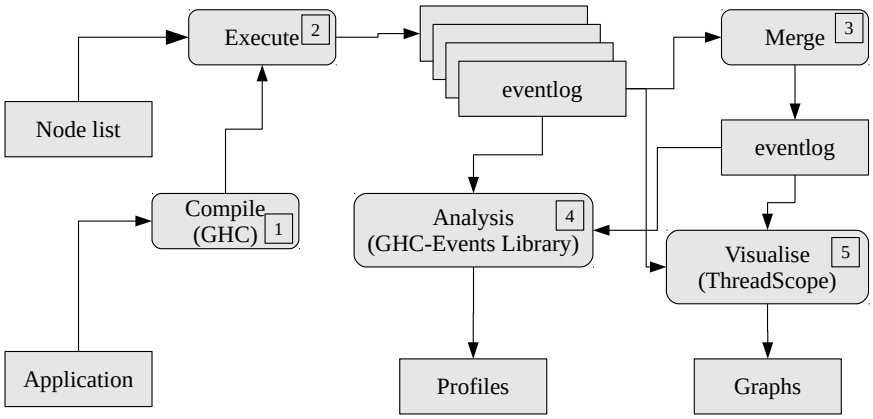


Figure 4.1: HdpHProf Work Flow Model.

4.2.1 HdpH Trace Events

Trace based data collection requires code instrumentation with trace events that are triggered during execution to emit the trace events into a log, as discussed in Sec-

[1] A rectangle with sharp corners represents an input/output for/from a process. However, a rectangle with rounded corners represents a functional phase that takes an input and produces an output [126].

tion 2.3.1. The emitted trace events represent events in the program being profiled, e.g. the start of executing a function. Having these trace events in the form of an eventlog can help to understand the execution behaviour of the program and identify performance problems. The GHC-PPS provides valuable trace functions for Haskell users who want to profile the behaviour of their programs, and especially for implementers of Haskell parallel DSLs. Therefore, to emit HdpH trace events we use this feature to instrument the HdpH runtime system.

The GHC-PPS trace function allows trace events to be emitted from the DSL level into the GHC-PPS eventlogs without the need to change the GHC. It provides general data fields with each trace event. For example, a trace event has when and where the trace event was emitted along with an event description that is specified by the user. Emitted trace events then can be read from the eventlog after execution for performance analysis. Table 4.1 gives a list of all trace events that HdpHProf traces to profile HdpH execution. Some of the trace events are one to one where we emit only one trace event. Conversely, some other trace events are paired where we have to record the beginning and the end of an event.

Category	Type		Trace Event
HdpH RTS	Single		Start-up
	Single		Shut-down
Global References	Single		Put an IVar
	Single		Get an IVar
	Paired	Enter	Globalise GRef
		Exit	GRef Globalised
	Paired	Enter	Dereference GRef
		Exit	GRef Dereferenced
	Paired	Enter	Free GRef
		Exit	GRef Freed
Sparks	Single		Put Spark
	Single		Spark Created
	Paired	Enter	Get Spark
		Exit (or)	Converted Spark
			Nothing to Spark
Messages	Single		SCHEDULE Message
	Single		FISH Message
	Single		Forward FISH
	Single		NOWORK Message
	Single		PUSH Message

Table 4.1: HdpH Trace Events.

4.2.2 Multiple Trace Files

An important point of the HdpHProf design is to produce a unique and independent trace file for each HpdH node. The trace files can be later merged into a single trace file that gives a complete picture of the execution behaviour, or they can be analysed individually for node based diagnosis, e.g. contention analysis.

The trace file produced by the GHC-PPS is designed to profile a shared-memory parallel programming model. In contrast, HdpH extends this programming model for distributed-memory parallelism on a cluster of multicores. This means the GHC-PPS on its own is insufficient to profile the execution behaviour of HdpH.

Figure 4.2 demonstrates how HdpHProf generates multiple trace files of an HdpH application. HdpHProf uses its Executor component for this task. The Executor runs the HdpH application in the eventlogging mode and makes each node produce a unique trace file which can be identified by, e.g. application name and node number. It is important that each HdpH node has a unique eventlog so each node can be analysed individually for quick and light-weight diagnosis of execution behaviour. Moreover, this design enables the process of synchronising and merging the eventlogs to take place after the execution to reduce the overhead of profiling, thus avoiding distorting profiling runs with synchronisation overheads.

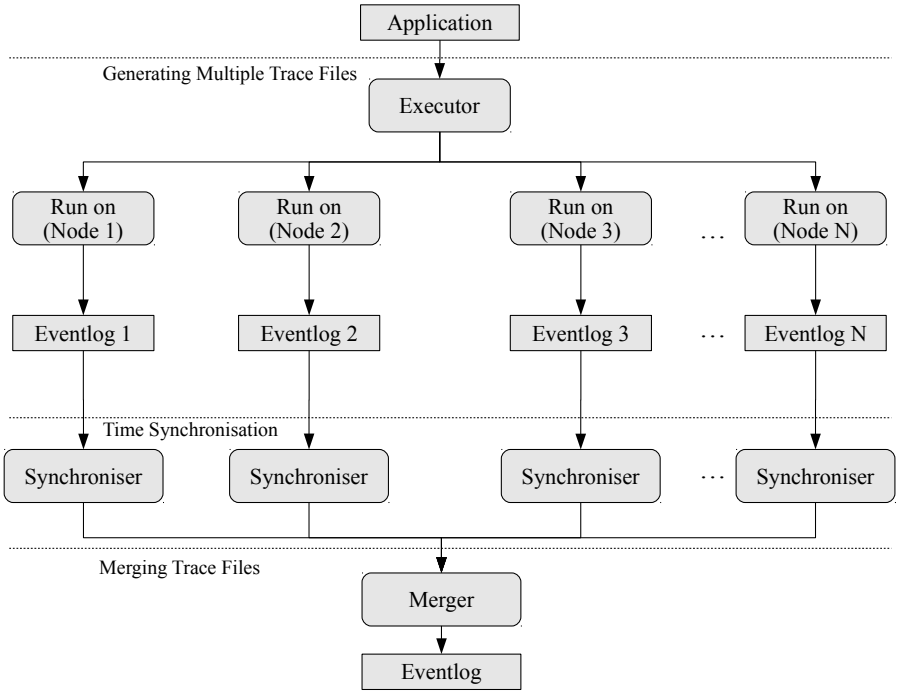


Figure 4.2: Generating, Synchronising and Merging Multiple Trace Files.

4.2.3 Time Synchronisation

HdpHProf solves the time synchronisation for the multiple trace files of HdpH. Capturing consistent time for trace events is crucial for accurate performance measurements. Tracing profilers use time stamps in trace events as a means of keeping track of time to observe points of interest in the course of program execution [71]. Time synchronisation is relatively easy in profiling shared-memory parallel programs as processes exist within the same machine and share a clock. However, in a distributed-memory execution processes reside in separated machines with their own clocks, and time needs to be synchronised in the tracing data, to reflect an accurate picture for the performance of the profiled application.

HdpHProf eventlogs contain trace events which are emitted with time stamps produced from different machines. In terms of profiling this introduces two issues. The first issue is in the case of using computing architecture that consists of multiple machines where it is likely that the clocks have some time difference. In the HdpHProf design we assume that while clock rates may not be identical they do not differ significantly during a program execution. The second issue is in how the GHC-PPS gives its trace events their time stamps. It starts by giving an arbitrary time greater than 0s for the first trace event in the log and greater time stamps for the following trace events.

HdpH has a barrier synchronisation for its RTS to start-up. However, starting the trace events with an arbitrary number in each trace file makes start-up times asynchronous in HdpHProf eventlogs. To solve the issue of time difference in the eventlogs, HdpHProf uses its Synchroniser component (Figure 4.2) to fix the time difference from the multiple eventlogs by equalising the HdpH RTS's start-up times and reflects this change for all trace events accordingly.

Figure 4.3 illustrates in more detail how the process of the time synchronisation is accomplished for the multiple eventlogs of HdpH. As the figure shows eventlogs have different HdpH RTS start-up times . To synchronise the trace events for all the eventlogs an arbitrary but uniform start-up time is needed to synchronise the profiling data based upon it. The arbitrary start-up time is architecture dependent. For instance, 1.5s is suitable for a Beowulf cluster of 32 nodes as no machine is expected to exceed this number. However, other computing architecture may need a longer time, e.g. 10s for HECToR [63]. The start-up must be greater than the real HdpH RTS start-up time otherwise time stamps get corrupted because of their unsigned integer type.

For example, if the HdpH RTS start-up time in an eventlog is 0.8s the synchronisation value will be $(1.5 - 0.8 = 0.7\text{s})$ and all the times in this eventlog will be updated by +0.7s. After synchronising all the eventlogs the profiling data is synchronised and all machines have the same start-up time.

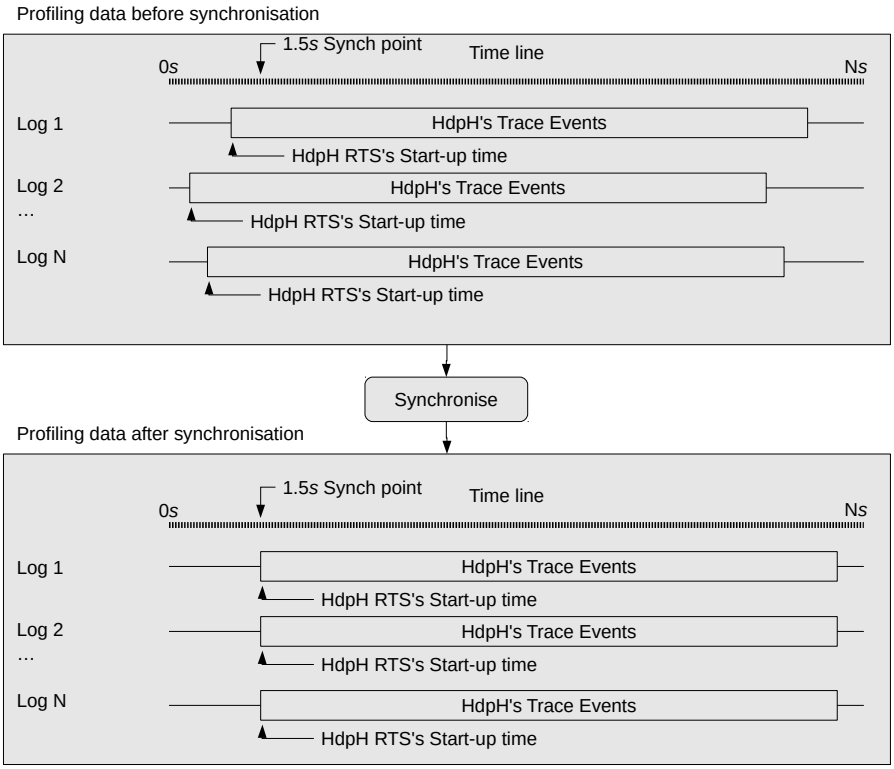


Figure 4.3: Design of HdpHProf Time Synchronisation Process.

4.2.4 Merging Trace Files

HdpHProf merges the multiple eventlogs from an HdpH application. The eventlogs are in the GHC-ELF [50]. Therefore, they can be read, analysed, and visualised individually by the GHC-PPS tools. However, to get a comprehensive picture of performance all eventlogs must be merged into one eventlog.

One alternative is to modify the GHC-Events Library and ThreadScope to read, analyse, and present performance data from multiple eventlogs. We think this is not a good design decision since it requires changing the code of both GHC-Events Library and ThreadScope. The second option is to provide a merging tool to locate and merge the multiple eventlogs of HdpH into a single eventlog.

Figure 4.2 illustrates the design of HdpHProf for merging multiple eventlogs. The Merger locates all eventlogs and merges them into a single eventlog that contains all

the trace events from the input files. In addition, the Merger keeps the output eventlog in the same format as the input eventlogs, i.e. GHC-ELF. The merger also keeps the performance data sorted by node number so it will be possible to identify performance problems for each node.

This approach is different from other functional distributed-memory profilers. For example, EdenTV [11] wraps all the trace files of distributed-memory execution into a single trace file of a different format than the GHC-ELF. Therefore, it is not possible to visualise Eden's trace file with the standard GHC trace browser ThreadScope. Instead, Eden has its EdenTV to visualise aspects that ThreadScope cannot, such as Eden processes. On the contrary, our approach in the design of HdpHProf is to keep the performance data in the GHC-ELF, so we can use the GHC-PPS to profile HdpH and extend it to introduce new profiling tools for HdpH.

4.2.5 Trace Visualisation

To present the performance data of HdpH, HdpHProf uses two approaches; summative profiles, which we discuss in Section 4.2.6, and performance graphs. HdpHProf uses the standard GHC trace viewer ThreadScope to visualise the HdpH performance date graphically. Performance data visualisation was discussed earlier in Section 2.3.1. Since HdpHProf is built based on the GHC-PPS this means that the raw performance data of HdpH has the GHC-ELF format. This means HdpHProf eventlogs can be visualised by ThreadScope directly for presentation. Figure 4.1 (Phase 5) shows how an HdpHProf eventlog is visualised with ThreadScope.

Visualising the performance of HdpH with ThreadScope gives information about how HdpH performs. HdpH is implemented using concurrent Haskell [107] where HdpH schedulers and message handlers are Haskell IO threads. Therefore, by visualising the performance data of HdpH in ThreadScope we see how well HdpH is utilising the parallel architecture. This helps see how HdpH applications behave also it can identify performance problems and tune thread granularity which we will discuss in Chapter 6.

4.2.6 Trace Analysis and Presentation

HdpHProf provides analysis tools to analyse and present the HdpH DSL implementation performance. Performance data analysis was discussed earlier in Sections 2.3.1. HdpHProf extends the GHC-Events Library with analysis tools specially designed for

the DSL HdpH. The library is equipped with tools and functionalities that can be used to read and process HdpH's eventlogs to produce performance profiles. We call the extended version the GHC-Events-HdpH Library and it is available to download online [2]. As proof of concept we designed two analysis tools for HdpH internals: *Spark Pool Contention Analysis*, and *Registry Contention Analysis*. These are tools specified for detecting contention on data structures shared between HdpH schedulers and we want to know when it occurs because contention reduces performance. Figure 4.1 (Phase 4) demonstrates how the analysis tools read and present performance of the DSL implementation. We will discuss the tools in more detail in the following sections.

Spark Pool Analysis

The spark pool is an important shared data structure in the HdpH DSL implementation that schedulers use to get sparks. This means that more than one scheduler can access the spark pool concurrently during execution. This can create contention where schedulers have to wait for sparks and this can affect the performance of HdpH. The *Spark Pool Contention Analysis* tool is designed to detect contention on accessing the spark pool by the multiple schedulers. It reads the spark pool trace events and analyses them for contention.

The tool is designed to detect two type of conflicts: 1) unproductive conflict, when two or more schedulers conflict and none of them leave the spark pool with a converted spark, and 2) productive conflict, where two or more schedulers conflict and one scheduler leaves the spark pool with a converted spark. It is important to distinguish between these two types of contentions as unproductive conflicts happen when the spark pool is empty and no scheduler leaves with a spark; whereas, productive conflicts happen when the spark pool contains some sparks and eventually one scheduler leaves with a spark while other schedulers must wait.

Figure 4.4 illustrates how contention on spark pool is identified by conflicts between sequences of spark pool trace events over time. An unproductive conflict exists when the first scheduler enters the spark pool until it exits the spark pool; meanwhile, a second scheduler enters after the first scheduler and leaves with no spark until the first scheduler exits the spark pool. With this data more statistical information can be derived; e.g. how often conflicts happen, what are their durations, and what is the maximum conflict duration. This information can be used to understand the behaviour

of the HdpH RTS and to help improve it.

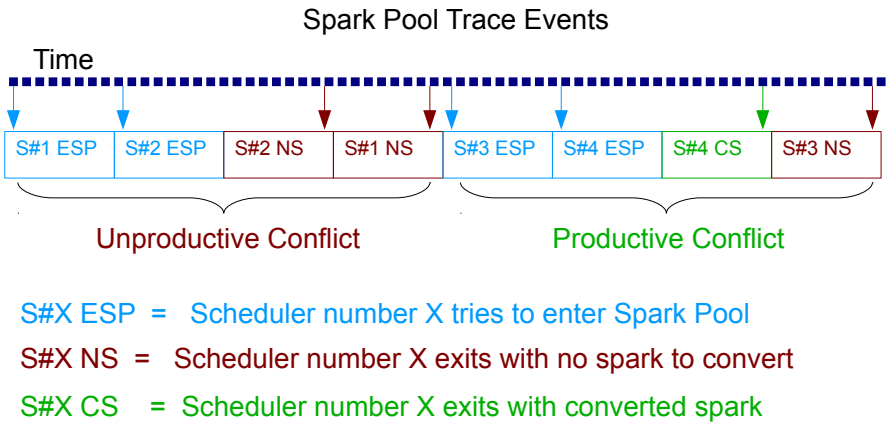


Figure 4.4: Spark Pool Conflicts.

Registry Analysis

The registry is another important data structure that manages global data in the HdpH DSL implementation. The registry can face high demand as many schedulers may try to access the registry concurrently, but only one scheduler can update the registry at a time and the others have to wait. The *Registry Contention Analysis* tool is designed to detect contention between schedulers accessing the registry.

A conflict on the registry occurs when a scheduler is holding the registry while one or more other schedulers are trying to access the registry and they have to wait until the first scheduler leaves the registry. Figure 4.5 shows how contention on the registry is detected by conflicts between the sequence of registry trace events over time. A conflict happens when a first scheduler enters the registry and a second scheduler tries to access the registry after the first scheduler and it has to wait until the first scheduler releases the registry. With this data statistical information can be derived, e.g. how often conflicts occur, and what are the durations of these conflicts. This information can help to understand the execution behaviour of HdpH RTS and improve it in the event of performance problems.

HpdH Performance Presentation

To present the HdpH RTS implementation performance data, HdpHProf uses summative profiles for presentation. We extend the GHC-Events Library with HdpHProf profiles for the DSL profiling. HdpHProf analysis tools, the Spark Pool Contention

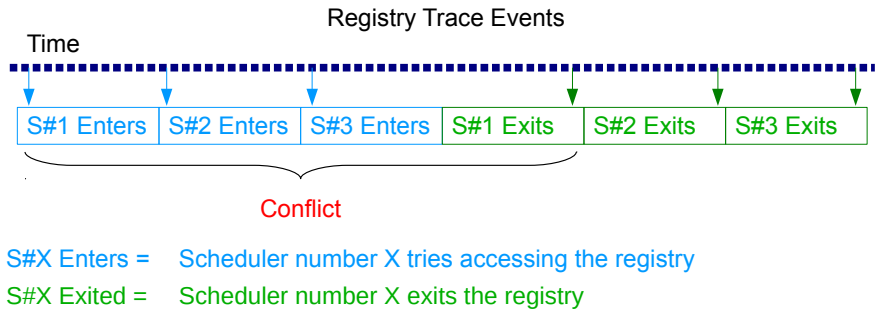


Figure 4.5: Registry Conflicts.

Analysis and the Registry Contention Analysis present their result in the form of a summative profile. Figure 4.1 (Phase 4) demonstrates how a performance profile is produced from an eventlog with HdpHProf analysis tools. The profilers are discussed and presented in more detail in the implementation (Section 4.3.5).

4.3 HdpHProf Implementation

This section presents the implementation of HdpHProf. It introduces how HdpHProf collects performance data of the execution behaviour of HdpH, how the GHC-Events Library [49] is used and extended to introduce new performance analysis tools for the DSL HdpH, and how multiple eventlogs of distributed-memory Haskell parallel DSL are tweaked to be visualised in ThreadScope [134]. The extended version of the library, GHC-Events-HdpH is available to download online [2].

4.3.1 Data Collection

This section demonstrates how HdpHProf collects HdpH performance data. The HdpH RTS is instrumented with HdpH trace events that are emitted into the GHC-PPS eventlog during execution. The following sections illustrate the emission of HdpH trace events, the code instrumentation of HdpH RTS, and the creation of HdpH multiple eventlogs.

Emitting HdpH Trace Events

HdpHProf exploits the trace function of GHC-PPS to record HdpH RTS level events in the eventlogs. We used the `traceEventIO` from the `Debug.Trace` module to emit trace events in monadic code. The function takes a string (event content) and emits

the event content along with a time stamp and the HEC that indicates the number of the virtual PE in which the event occurred.

HdpH RTS Code Instrumentation

HdpH RTS is instrumented with HdpH trace events in a number of modules. To emit the trace events enumerated in Table 4.1 we have to find in which module a particular event happens then instrument its code. For contention analysis it is important to know when an event has started and when it finished. Therefore, instrumentation will emit two trace events, before and after the actual event. In addition, an event must have a unique description so it can be identified during the data analysis process. Here we demonstrate how events instrumentation is implemented for the HdpH RTS.

Listing 4.1 illustrates how functions from the HdpH IVar module are instrumented to emit registry trace events. Three paired events occur: globalising a Global Reference (GRef), dereferencing a GRef, and freeing a GRef. For example, to emit trace events for globalising a GRef in the `globIVar` function Line 3 emits the beginning of the event and Line 5 emits the end of the event. Similarly, the other events are implemented, dereferencing GRef in Lines 10 and 12, and Freeing GRef in Lines 13 and 15.

Listing 4.1: Extract of `IVar` Module.

```
1  globIVar :: Int -> IVar m a -> IO (GIVar m a)
2  globIVar schedID v = do
3    traceEventIO $ "GlobaliseGRef {scheduleId = "++ show schedID  ++"}"
4    gv <- globalise v
5    traceEventIO $ "GRefGlobalised {scheduleId = "++ show schedID  ++"}"
6    return gv
7
8  putGIVar :: Int -> GIVar m a -> a -> IO [Thread m]
9  putGIVar schedID gv x = do
10   traceEventIO $ "DereferenceGRef {scheduleId = "++ show schedID  ++"}"
11   ts <- withGRef gv (\ v -> putIVar v x) (return [])
12   traceEventIO $ "GRefDereferenced {scheduleId = "++ show schedID  ++"}"
13   traceEventIO $ "FreeGRef {scheduleId = "++ show schedID  ++"}"
14   free gv
15   traceEventIO $ "GRefFreed {scheduleId = "++ show schedID  ++"}"
16   return ts
```

Some of HdpH trace events implementation are in none of the monadic functions and we need to lift the `traceEventIO` into the monad that HdpH uses; for instance, the `getSpark` function from the `Sparkpool` module. To get a spark a scheduler enters the spark pool then either exits with a converted spark, or with nothing to spark. Listing 4.2 illustrates the implementation on HdpH trace events. Lines 1 and 2 show how the `traceEventIO` is lifted to the `SparkM` monad. Line 7 emits the beginning trace events of entering the spark pool to get a spark. In contrast, in Lines 10 and 13 emit the exit trace events based on the result.

Listing 4.2: Extract of `Sparkpool` Module.

```
1 trace :: String -> SparkM m ()
2 trace message = liftIO $ traceEventIO message
3
4 getSpark :: Int -> SparkM m (Maybe (Spark m))
5 getSpark schedID = do
6   pool <- getPool
7   trace $ "EnterSparkPool {scheduleId = "++ show schedID ++"}"
8   maybe_spark <- liftIO $ popFrontIO pool
9   case maybe_spark of
10    Just _  -> do trace $ "ConvertSpark {scheduleId = "++ show schedID ++"}"
11                  sendFISH
12                  return maybe_spark
13    Nothing -> do trace $ "NothingToSpark {scheduleId = "++ show schedID ++"}"
14                  sendFISH
15                  return maybe_spark
```

Multiple Trace Files

This section shows the implementation of `hdphexec`, a profiling executer for HdpH applications. One of the challenges we faced in profiling the performance of the DSL HdpH was to produce a unique trace file (eventlog) for each computing node. This is because HdpH executes on multiple nodes that run multiple instances of GHC. Using the GHC-PPS on its own is not sufficient because it is designed for shared-memory profiling on a single multicore, producing a single eventlog from one machine. Therefore, we implemented the `hdphexec`, which takes a list of Beowulf cluster nodes and an HdpH application then executes the application to produce multiple eventlogs, one per node. Eventlogs are identified by the application name and the node number. For example, `SumEuler#3.eventlog` means this is the eventlog of the SumEuler application which was executed on the third node. Listing 4.3 shows the implementation code for the `hdphexec` (as a shell script).

Listing 4.3: Implementation of HdpHProf Executer (`hdphexec`).

```
1 # variables from command line, number of nodes, hosts file and the program name
2 nodes=$3
3 hosts=$4
4 execname=$5
5 # create executable for each machine.
6 N=1
7 while test "$N" -le "$nodes"
8 do
9   read hostname
10  cp $execname $execname#$N
11  N=$[N+1]
12 done<$hosts
13 #shift the command-line varibles to strat from the HdpH run commands.
14 N=1
15 while test "$N" -le "5" ; do
16         shift
17         N=$[N+1]
18 done
19 # generat command to run the program on multiple nodes
20 runcommad="mpiexec"
21 N=1
22 while test "$N" -le "$nodes"
23 do
```

```
24   read hostname
25   if test "$N" -lt "$nodes"; then
26         runcommad="$runcommad  -host $hostname  $execname#$N $@ : "
27   else
28         runcommad="$runcommad  -host $hostname  $execname#$N $@ "
29   fi
30   N=$[N+1]
31 done<$hosts
32 # execute the command
33 $runcommad
```

4.3.2 Data Analysis

This section presents the implementation aspects of HdpHProf related to HdpH performance data analysis tools we discussed previously in Section 4.2.6. This includes introducing HdpH trace event types, tools for reading and filtering trace events, and tools for performance analysis, Spark Pool Contention Analysis and Registry Contention Analysis.

HdpH Trace Event Types

Defining a data structure for trace events is crucial for performance data collection. A trace file can consist of hundred of thousands of trace events. Storing trace events in an appropriate data structure makes it easier to manage the data and derive information. Moreover, structured data makes it possible to process or apply functions on trace events based on their type, e.g. operations such as equality or ordering can be applied to trace events.

We implemented HdpH trace events data format as a module in the GHC-Events-HdpH Library, called `HdpHEventType` (Listing 4.4). The implementation follows the approach used to implement the GHC event types from the GHC-Events Library. We introduced new Haskell data type, `HdpHData`. The data type consists of a list of type `HdpHEvent`. Each HdpH event has two fields, a time stamp and an event information. The time stamp field represents when the event happens. However, the event information field can be one of the shown HdpH trace events. Keeping HdpH trace events in such a Haskell data structure makes them readable and processable to derive performance information as discussed in the following sections.

Listing 4.4: `HdpHEventTypes` Module.

```
1 module HdpH.HdpHEventTypes where
2 import GHC.RTS.Events
3 import Data.List
4 import Data.Word (Word16)
5
```

```
 6 type SchedulerId = Word16
 7 type Message     = String
 8 type Node        = String
 9
10 data HdpHData = HdpHData {
11    hevents :: [HdpHEvent]
12    }
13
14 data HdpHEvent =
15    HdpHEvent {
16      e_time :: {-# UNPACK #-}! Timestamp,
17      e_spec :: HdpHEventInfo
18    } deriving (Show, Eq, Ord)
19
20 data HdpHEventInfo
21    -- sparks
22    = SparkCreated      { scheduleId :: {-# UNPACK #-}! SchedulerId
23                        }
24    | ConvertSpark      { scheduleId :: {-# UNPACK #-}! SchedulerId
25                        }
26    | NothingToSpark    { scheduleId :: {-# UNPACK #-}! SchedulerId
27                        }
28    | EnterSparkPool    { scheduleId :: {-# UNPACK #-}! SchedulerId
29                        }
30    | PutSpark          { scheduleId :: {-# UNPACK #-}! SchedulerId
31
32    -- Globale References
33                        }
34    | GlobaliseGRef     { scheduleId :: {-# UNPACK #-}! SchedulerId
35                        }
36    | GRefGlobalised    { scheduleId :: {-# UNPACK #-}! SchedulerId
37                        }
38    | FreeGRef          { scheduleId :: {-# UNPACK #-}! SchedulerId
39                        }
40    | GRefFreed         { scheduleId :: {-# UNPACK #-}! SchedulerId
41                        }
42    | FreeGRefNow       { scheduleId :: {-# UNPACK #-}! SchedulerId
43                        }
44    | GRefFreedNow      { scheduleId :: {-# UNPACK #-}! SchedulerId
45                        }
46    | DereferenceGRef   { scheduleId :: {-# UNPACK #-}! SchedulerId
47                        }
48    | DeadGRef          { scheduleId :: {-# UNPACK #-}! SchedulerId
49                        }
50    | GRefDereferenced  { scheduleId :: {-# UNPACK #-}! SchedulerId
51                        }
52    -- Messages
53   | FishMsg            { node        :: Node
54                        , message     :: Message
55                        , target      :: Node
56                        }
57    | ScheduleMsg       { node        :: Node
58                        , message     :: Message
59                        , fisher      :: Node
60                        }
61    | ForwardFish       { node        :: Node
62                        , message     :: Message
63                        , target      :: Node
64                        }
65    | NoWorkMsg         { node        :: Node
66                        , message     :: Message
67                        , fisher      :: Node
68                        }
69    -- RTS start-up/shutdown
70    | HdpHStartup       {}
71    | HdpHShutdown      {}
72    -- mis
73    | NotHdpHEvent      {}
74
75    deriving (Show, Eq, Ord, Read)
```

Extracting HdpH Trace Events

This section demonstrates how HdpH trace events are extracted from the GHC eventlog. HdpH trace events are emitted into an eventlog as GHC trace events that we discussed earlier (Section 4.3.1). To analyse the performance of HdpH we needed to extract its trace events from the eventlog. Therefore, we implemented a function that takes trace events of an eventlog then returns HdpH trace events as result.

Listing 4.5 shows partial code of the `HdpH.Analysis` module from the GHC-Events-HdpH Library [2] that illustrates how HdpH trace events are extracted. The `getHdpHEvents` function takes a list of GHC trace events[2] then returns a list of HdpH trace events as results. The function recursively checks and extracts events from the input list then produces a new list of HdpH trace events. The `liftEvent` function is utilised by the `getHdpHEvents` to parse the content of trace events. It examines the content of the trace event; in case of a HdpH trace event it returns it as a result or skips it otherwise. This is how HdpH trace events are extracted from an eventlog trace events.

Listing 4.5: Parsing HdpH Events.

```
1  getHdpHEvents :: [Event]-> [HdpHEvent]
2  getHdpHEvents []=[]
3  getHdpHEvents (x:xs) =
4      let evt = liftEvent x
5      in case  e_spec evt of
6         NotHdpHEvent ->  getHdpHEvents xs
7         _            ->  evt :  getHdpHEvents xs
8
9  liftEvent :: Event -> HdpHEvent
10 liftEvent e = let
11   eventtime = time e
12   eventspec =  isHdpHEventInfo (reads (msg $ spec $ e) :: [(HdpHEventInfo, String)])
13   in HdpHEvent {e_time = eventtime , e_spec = eventspec}
14
15 isHdpHEventInfo ::  [(HdpHEventInfo, String)] -> HdpHEventInfo
16 isHdpHEventInfo []=  NotHdpHEvent
17 isHdpHEventInfo [(x, y)] = x
```

Spark Pool and Registry Analysis

To analyse contention on the spark pool it is necessary to acquire the spark pool trace events and pair events. Listing 4.6 shows the `pairSparkEvents` function from the `HdpH.Analysis` module of the GHC-Events-HdpH Library. The function takes a list of HdpH events and returns a list that contains lists of paired HdpH trace events. Similarly, the `pairRegEvents` function in Listing 4.7 is used to acquire and pair the

[2] The GHC trace events at this stage are filtered to be only those emitted with the `traceEventIO` function.

registry trace events. In a following section we will show performance profile with information derived from this data (Section 4.3.5). Full implementation is also available in [2].

Listing 4.6: Extracting Spark Pool Events from HdpH Events.

```
1 -- Takes a list of HdpHEvent and return a list of pairs each pair
2 -- has the events of entering the spark pool and leaving the sparkpool
3 -- for the same scheduleId
4 pairSparkEvents :: [HdpHEvent]-> [[HdpHEvent]]
5 pairSparkEvents []=[]
6 pairSparkEvents (x:xs)
7    | isEnterSparkPool x = searchForPairOf x xs : pairSparkEvents xs
8    | otherwise = pairHdpHEvents xs
```

Listing 4.7: Extracting Registry Events from HdpH Events.

```
1 -- This function takes a list of HdpHEvent and returns a list of pairs in
2 -- a list of registry events  i.e globalise, dereference and free.
3 pairRegEvents ::  [HdpHEvent] -> [[HdpHEvent]]
4 pairRegEvents  [] = []
5 pairRegEvents  (x:xs)
6   | isGlobaliseGRef x = searchForPairOf x xs : pairRegEvents  xs
7   | isFreeGRef x = searchForPairOf x xs : pairRegEvents  xs
8   | isDereferenceGRef x =  searchForPairOf x xs : pairRegEvents  xs
```

4.3.3 Trace File Time Synchronisation

Listing 4.8 presents the `HdpH.Synch` module from the GHC-Events-HdpH Library that synchronises distributed eventlogs. The `synchEventlogs` function takes a start time and an HdpH eventlog then synchronises its trace events time stamps to the required start time. For example, 1.5s as a start time will be sufficient for a Beowulf cluster up 32 nodes as we found that average HdpH RTS start-up time is about 1s. The function works by first reading the HdpH RTS start-up time from the eventlog. Then it calculates the synchronisation time value by subtracting start time (`stime`) from the HdpH RTS start-up time. After that it modifies times for all trace events in the eventlogs by adding the synchronisation value to each event time stamp. The function is used recursively over the multiple eventlogs before merging them as will be discussed in the next section.

Listing 4.8: `HdpH.Synch` Module.

```
1  module HdpH.Synch (synchEventLogs) where
2
3  import GHC.RTS.Events (Data)
4  import Data.List
5  import Data.Word (Word64)
6
7  synchEventLogs :: Word64 -> EventLog -> EventLog
8  synchEventLogs stime (EventLog h d) = EventLog h (synchData stime d)
9
10 synchData :: Word64 -> Data -> Data
11 synchData stime d@(Data (x:xs))  =
```

```
12   let sv = synchValue stime d
13       evts = syn sv (x:xs)
14         where syn s (y:ys) = Event (time y + sv )(synchBlock s (spec y)) : syn s ys
15               syn s [] = []
16       in Data evts
```

4.3.4 Merging Trace Files

HdpHProf merges eventlogs using its component `merger`, a script that implements the design described in Section 4.2.4. The `merger` locates eventlogs and recursively feeds them to the `synch` function.

Merge Executer

Listing 4.9 presents the implementation of the `merger` which is an executer for merging HdpH eventlogs. The merge executer locates the eventlogs of the profiled application. It applies the `synch` function to each eventlog for time synchronisation, then merges the eventlogs.

Listing 4.9: Implementation HdpHProf Merge Executer.

```
1 #!/bin/bash
2 programname=$1
3
4 if test "$programname" != ""; then
5 x=0
6 for i in $programname#*.eventlog
7 do
8   ghc-events-hdph synch 1500000000 temp.eventlog $i
9   mv  temp.eventlog $i
10  x=$[x+1]
11 done
12 mergecommand="ghc-events-hdph merge $programname.eventlog"
13
14 N=1
15 while test "$N" -le "$x"
16 do
17   mergecommand="$mergecommand $programname#$N.eventlog "
18   N=$[N+1]
19 done
20 $mergecommand
21
22 else
23 echo "Usage : hpmerge <program_name> "
24 fi
```

Merge Function

Listing 4.10 illustrates the implementation of the `merge` function in the `GHCEvents` module of the GHC-Events-HdpH Library. This function is a modified version of the existing `merge` function from the GHC-Events Library [49]. The original function can only merge two eventlogs in a single execution. However, our modified versions are

capable of merging non-empty list of eventlogs with fold1. We changed the function parameters to take multiple input files. We used `mapM` to read input files as a list to return a list of monadic actions for reading the input files. After that, we used the `foldr1`[3] function to apply `mergeEventLogs` function to the result from the `mapM`. This upgrade to the merge function makes it capable of merging multiple eventlogs at single execution instead of only two eventlogs. This function is used by the `merger` which is presented in the previous section to merge HdpH eventlogs.

Listing 4.10: Re-implementation of Merge Function from the GHC-Events Library.

```
1 command ("merge":out:files) = do
2     fs <- mapM readLogOrDie files
3     let m = foldr1 mergeEventLogs fs
4     writeEventLogToFile out m
```

4.3.5 Data Presentation

This section shows how HdpHProf presents HdpH performance data. HdpHProf provides two methods of presenting performance data. First, HdpHProf analysis tools which present results as summative profiles, i.e. Spark Pool Contention Analysis Profile and Registry Contention Analysis Profile. Second, HdpHProf utilises ThreadScope [134] the standard GHC-PPS eventlog time line browser to visualise eventlogs of HdpH.

Summative Profiles

HdpHProf analysis tools present their results in the form of summative profile, i.e. Spark Pool Contention Analysis Profile, and Registry Contention Analysis Profile.

Spark Pool Contention Analysis Profile. One of the analysis tools which we designed and implemented is the *Spark Pool Contention Analysis* tool. The tool analyses performance data from an HdpH eventlog then presents a performance profile. The profile presents statistical information that measures certain behaviour aspects of HdpH. Figure 4.6 shows a spark pool contention analysis profile. It is divided into four sections. The first section gives counts of how many times the spark pool has been accessed, the total exiting the spark pool with sparks converted, and the total of no spark to convert. The second section shows statistics regarding conflicts between schedulers entering the spark pool. The third section presents information

[3] The list of eventlogs must be folded from the right; a fold from the left produces a corrupted eventlog.

about conflict durations. Finally the fourth section shows the total number of conflicts grouped by the number of schedulers involved.

The tool is implemented as an extension in the GHC-Events-HdpH Library. It can be used from the command line. For example, the Spark Pool Contention Analysis tool is called to present the profile as follows:

```
$ ghc-events-hdph showsparkcont fib#1.eventlog
```

This will execute the (`showsparkcont`) tool which will read its argument (`fib#1.eventlog`) as input and produce the performance profile. We will use the Spark Pool Contention Analysis tool to profile the HdpH DSL implementation in Chapter 7.

```
--------------------------------------------------------------------------------
                                HdpHprof
                    Spark Pool Contention analysis

Total entry to sparkpool: 17753
Total sparks converted: 17710
Total No spark to convert: 43
--------------------------------------------------------------------------------
SID        Enter      All.Cof.   All.C.%    Pro.Conf.  Pro.C.%
1          2573       155        6.02       155        6.02
2          2595       174        6.71       174        6.71
3          2516       187        7.43       187        7.43
4          2492       162        6.50       162        6.50
5          2435       164        6.74       163        6.69
6          2595       147        5.66       147        5.66
7          2547       182        7.15       182        7.15
------------------------------------------------------------
Total      17753      1171       6.60       1170       6.59
--------------------------------------------------------------------------------
All times displayed are in milliseconds
SID        All.C.Dur  Mean       Pro.C.Dur  Mean
1          1.5143     0.0098     1.5143     0.0098
2          2.2808     0.0131     2.2808     0.0131
3          16.5414    0.0885     16.5414    0.0885
4          1.9529     0.0121     1.9529     0.0121
5          3.1525     0.0192     3.1436     0.0193
6          1.6001     0.0109     1.6001     0.0109
7          3.7056     0.0204     3.7056     0.0204
------------------------------------------------------------
Total      30.7475    0.0263     30.7387    0.0263

Max duration  in a productive conflict: 13.30991
--------------------------------------------------------------------------------
Conflicts grouped by total number of schedulers involved
No.Schedulers   Conf. Occurance%     Conf. Duration%      Mean
7               0.26                 3.34                 0.3424
6               0.09                 0.04                 0.0118
5               0.43                 0.33                 0.0205
4               1.71                 2.16                 0.0331
3               11.28                50.62                0.1179
2               86.24                43.51                0.0133
--------------------------------------------------------------------------------
```

Figure 4.6: Spark Pool Contention Analysis Profile.

Registry Contention Analysis Profile. Another analysis tool that HdpHProf provides is the *Registry Contention Analysis* tool. It reads HdpH performance data from the eventlog and produces a profile. The profile contains statistical information about

the HdpH RTS performance in terms of operations on the registry. Figure 4.7 illustrates a registry contention analysis profile. The registry profile is divided into five sections. The first section gives the total number of times the registry has been accessed by different operations. The second, third and fourth sections are similar to those on the Spark Pool Contention Analysis tool. The fifth section presents conflicts grouped by operations type. For example, *Glob* means conflicts that only have globalise GIVar operations involved; whereas, *Mixture* means conflicts that have more than one operation type involved. The registry contention profile is important to understand how operations on the registry behave during execution. We will use the Registry Contention Analysis tool to evaluate different implementations of the registry for the HdpH DSL in Chapter 7.

Graphical Visualisation

Figure 4.8 shows ThreadScope visualising HdpH performance. ThreadScope was introduced to visualise threads activities of GHC-SMP [57] on a single multicore machine [67]. We extended the use of ThreadScope to present the performance of HdpH and show how load is distributed between nodes of Beowulf cluster of multicores. As the figure illustrates, the first green bar is an overall activity for all cores –from multiple nodes of multicores– used in computation. Overall green colour goes up when the cores are being utilised whereas it is white when they are under-utilised. The following green bars show multiple HECs. The per-core HECs indicate if there is an active thread utilising that core or not. For instance, green means there is an active thread utilising the core, white means the core is not utilised, and orange indicates garbage collection. HECs are sorted in an ascending order, e.g. HECs from 0 to 1 report cores of node-1 and HECs from 2 to 3 report cores of node-2 etc.

```
--------------------------------------------------------------------------------
                                HdpHprof
                        Registry Contention analysis

Total entries to registry: 53130
Total globalise entries: 17710
Total free entries: 17710
Total dereference entries: 17710
--------------------------------------------------------------------------------
SID         Enter       Conflict    Conflict%
0           0           0           NaN
1           7924        2021        25.50
2           7780        1892        24.32
3           7387        1765        23.89
4           7344        1732        23.58
5           7333        1909        26.03
6           7600        1884        24.79
7           7762        1962        25.28
----------------------------------------
Total       53130       13165       24.78
--------------------------------------------------------------------------------
Displayed times are in milliseconds.
SID         Conf. Duration          Mean
0           0.0000                  0.0000
1           23.5212                 0.0116
2           39.3737                 0.0208
3           42.7713                 0.0242
4           35.7293                 0.0206
5           25.4518                 0.0133
6           19.2003                 0.0102
7           23.3062                 0.0119
----------------------------------------
Total       209.3537                0.0159
Max conflict duration : 20.75027
--------------------------------------------------------------------------------
Number of times a conlflict occured with this number of schedulers
No.Schedulers   Conf. Occurance%    Conf. Duration%     Mean
8               0.00                0.00                0.0000
7               3.63                11.45               0.0501
6               8.69                11.60               0.0212
5               13.93               13.03               0.0149
4               19.44               24.86               0.0203
3               24.91               21.43               0.0137
2               29.40               17.62               0.0095
--------------------------------------------------------------------------------
Conflict grouped by events type
Events type     Conf. Occurance%    Conf. Duration%     Mean
Glob            19.71               13.12               0.0106
Free            12.42               5.82                0.0075
Deref           13.80               15.60               0.0180
Mixture         54.07               65.46               0.0193
--------------------------------------------------------------------------------
```

Figure 4.7: Registry Contention Analysis Profile.

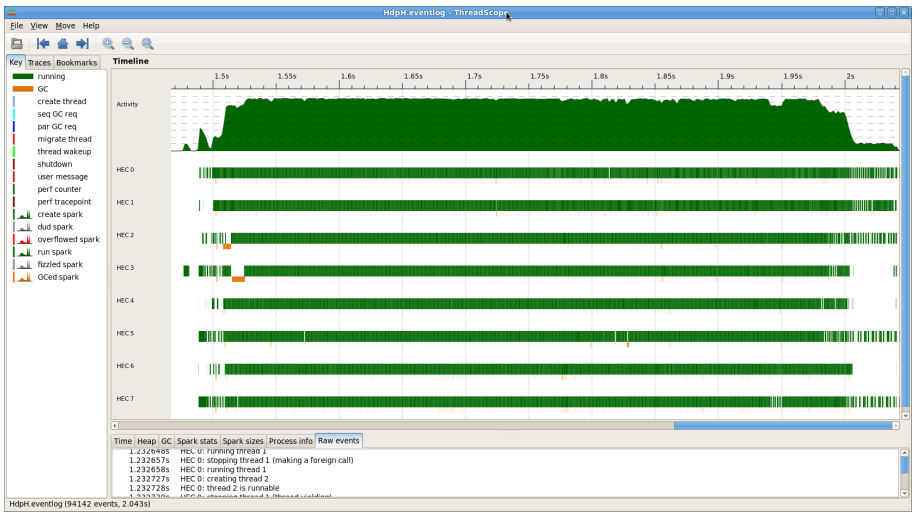


Figure 4.8: TheadScope visualises HdpH Fibonacci 40 threshold 30 on 4 nodes 2 cores each (total cores 8).

4.4 Summary

This chapter shows that it is possible to construct a profiler for a distributed-memory parallel DSL using the host language profiling tools. We presented the design and implementation of HdpHProf, a profiler for the parallel Haskell DSL, HdpH. We presented the requirements of HdpHProf to profile HdpH (Section 4.1). To meet these requirements, we introduced the design of HdpHProf to collect the performance data of HdpH using trace events and multiple trace files. Moreover, we presented how to synchronise times in HdpH trace files and how they can be merged into a trace file. Also, we showed how the performance data of HdpH can be analysed and presented for performance tuning (Section 4.2). In addition, we presented the implementation of HdpHProf. We showed the implementation of the HdpH RTS instrumentation for performance data collection. We presented how to extract HdpH trace events and use them for performance analysis. Also, we showed the implementation of time synchronisation and merging of HdpH trace files. We discussed how to extend the host language profiling tools with HdpHProf analyses for the HdpH DSL implementation. Finally, we discussed the implementation of HdpHProf performance data presentation (Section 4.3). We use HdpHProf to profile HdpH applications in Chapter 6 and use the Spark Pool Contention Analysis tool and the Registry Contention Analysis tool in Chapter 7 to analyse the implementation of the HdpH DSL internal data structures.

Chapter 5

Validating HdpHProf

This chapter validates HdpHProf for functional correctness and profiling performance. Functional correctness means that HdpHProf accurately records the behaviour of parallel programs. In addition, it is important to measure its performance and characterise it in terms of its design goals and requirements, i.e. scalability, profiling data size, and profiling execution runtime overhead. Consequently, this chapter is divided into four main sections. It starts by validating HdpHProf functional correctness (Section 5.2). Second, we validate the scalability of HdpHProf up to 192 cores of 32 Beowulf cluster nodes (Section 5.3). Third, we characterise the overheads of HdpHProf and compare it to other profilers in terms of profiling data size (Section 5.4) and profiling execution runtime overhead (Section 5.5). After that, we measure the ratio of HdpH trace events in the GHC-PPS eventlog (Section 5.6). Therefore, each section has its own experimental methodology that satisfies the purpose of the study.

5.1 Experimental Tools and Benchmarks

This section specifies the compilers and profiling tools used for experiments, and describes the benchmarks that we use for validation and evaluation. Table 5.1 specifies the compilers and profiling tools used for experiments. Table 5.2 describes the HdpH benchmarks. The benchmarks and profiling tools will also be used for experiments in Chapter 6 and Chapter 7.

Summatory Liouville. The Liouville function, denoted by $\lambda(n)$ is a completely multiplicative function, which is related to the number of prime factors of integer n, with $\lambda(n)$ being -1 when n is a prime number [131]. Summatory Liouville, denoted

Compiler/Profiling Tool	Citation	Version
The Glorious Glasgow Haskell Compilation System (GHC)	[46]	7.6.3
The HdpH RTS	[82]	0.1.0
Ghc-Events-HdpH Library	[2]	0.4.0.0
ThreadScope	[134]	0.2.1

Table 5.1: Compilers and Profiling Tools.

Benchmarks	Algorithm	Parameters	Granularity Control	Source Code Citation
Summatory Liouville	Data parallel	2	Chunk Size	[129, 128]
Mandelbrot	Divide and conquer	4	1 threshold	[129, 128]
SumEuler	Data parallel	3	Chunk Size	[82]
Fibonacci	Divide and conquer	3	2 thresholds	[82]
Queens	Nested data parallel	2	Chunk Size	[82]
NBody	Data parallel	3	Chunk Size	[82]

Table 5.2: HdpH Benchmarks.

by $L(n)$ is the sum of the values of the Liouville function $\lambda(n)$ up to n and defined as: $L(n) := \sum_{k=1}^{n} \lambda(k)$ [129]. The benchmark has a flat data parallel algorithm and gives the user control over thread granularity by a chunk size argument.

Mandelbrot. The Mandelbrot set is the set of values of c on the complex plane which remain bounded to the set when a mathematical operation is iterated on it. It is formed of all values defined by the complex numbers c for which the recursive formula $z_{n+1} = z_n^2 + c$ never approaches infinity when $z_0 = 0$ and n approach infinity [129]. The benchmark has a divide and conquer parallel algorithm and takes four arguments, X, Y, Depth, and threshold. The threshold is used to control thread granularity to determine when to evaluate sequentially.

SumEuler. The benchmarks sums Euler's totient function ϕ over long lists of integers and it has a flat data parallel algorithm. It takes three arguments, the first two are beginning and the end of the integers list, and the third argument is chunk size to control thread granularity.

Fibonacci. Fibonacci sequence, denoted by F_n is defined as: $F_n = F_{n-1} + F_{n-2}$, where $F_1 = 1, F_0 = 0$. The benchmark has a divide and conquer parallel algorithm, and it takes three arguments. The first argument n is to find the n^{th} term in the Fibonacci

sequence. The remaining two arguments are cut-off options, i.e. two thresholds to control task granularity.

Queens. Queens, a nested data parallel benchmark. The benchmark tries to solve a chessboard problem by placing N queens on an $N \times N$ board such that no queen can attack any other queen. The benchmark takes two arguments the number of queens and a chunk size for thread granularity.

NBody. The benchmark runs a simulation of X bodies for Y time steps and uses chunk size Z for parallelising each time step. In other words, the benchmark is phases of data parallel problems where each time step is data parallel. However, time steps sequentially follow each other, which requires sequential synchronisation.

5.2 Validation of Functional Correctness

HdpHProf functional correctness is validated using a combination of scenario-based tests and real performance data. We profile HdpH benchmarks running on a Beowulf cluster which is described earlier in Section 3.1.1. After that, the outcomes are examined for errors and defects, visually and by hand.

This experiment is divided into four areas. First of all, a coverage test that confirms that HdpHProf correctly emits HdpH trace events during execution. The test involves reading the trace events form eventlogs of multiple benchmarks and checks whether HdpH trace events exist or not.

Second is a visual test for the trace file time synchronisation function. The function is tested for adjusting the times in the eventlog profile as required so all HdpH nodes start at the same time. The function is applied to eventlogs of real executions produced by different HdpH benchmarks. The result, the profile before and after time synchronisation, is then compared visually in ThreadScope.

Third is testing the merge function for reading multiple HdpH eventlogs and correctly merging them into a single eventlog. The outcomes are compared visually in ThreadScope with the original eventlogs. We check that all performance data appear correctly and in the expected order.

Finally is a check on the functional correctness of the HdpHProf contention analysis tools. To validate the functional correctness of HdpHProf analysis tools performance data with known contention is needed. Due to the random work stealing in

HdpH, benchmarks cannot be used to produce predictable contentions. Therefore, we used artificial performance data in the form of scenarios that simulate contention in the HdpH RTS internals. These scenarios are carefully designed with a precision of nanoseconds that produces contention between HdpH schedulers at specific location and specific numbers at the execution. By feeding this data into the tools we know exactly what to expect to see on the performance profile. If the profile shows exactly the same results as contention planted in the performance data, it proves that the tools function correctly, and the performance profile results are valid.

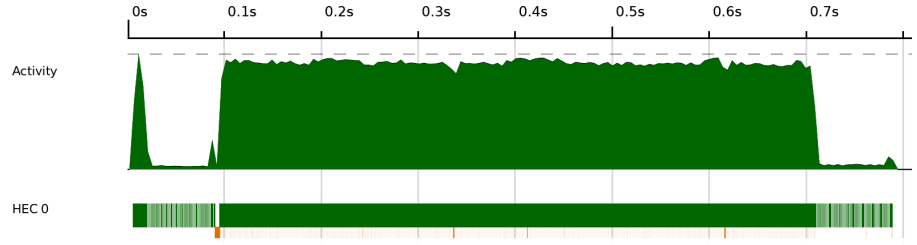
5.2.1 Code Instrumentation

This section illustrates how the HdpH RTS code instrumentation from Sections 4.2.1 and 4.3.1, for emitting HdpH trace events is validated for functional correctness, i.e. a coverage test for HdpH instrumentation. To check that the instrumentation implementation is working and trace events are emitted during the applications' executions, we check a number of profiles to see whether all the different HdpH trace events appear in the profiles. First, we had to read the eventlog and convert it to human readable format. After that, we filtered HdpH trace events from the GHC trace events. Finally, we tested that the instrumented trace events existed in the trace file, e.g. create spark event. In every case we found that all expected HdpH RTS code instrumentations emitted trace events. Indeed, unless a certain sequence of trace events exist in the eventlog, for instance, enter spark pool then convert spark or no spark to convert, then HdpHProf analysis tools will crash.

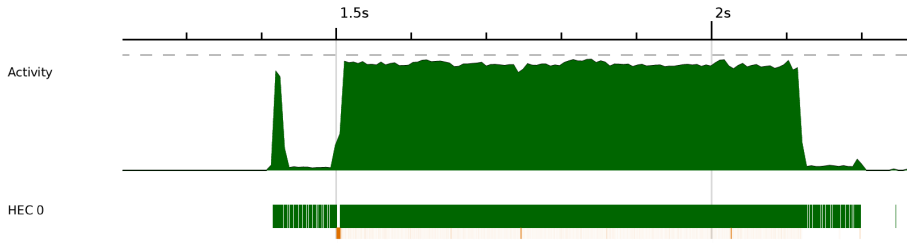
5.2.2 Time Synchronisation in Trace Files

This section presents testing the functional correctness of the trace files time synchronisation function of HdpHProf from Sections 4.2.3 & 4.3.3. In short, the function shifts the activity bar and HECs bars in the ThreadScope eventlog forward in time to be at the required synchronisation point. Figure 5.1(a) shows a profile of HdpH application before synchronising the time; whereas, Figure 5.1(b) shows the same profile after it has been synchronised with the HdpHProf time synchronisation function to start at 1.5s. In addition, Figure 5.2 shows three eventlogs of an HdpH application before synchronising the time, the profiles are synchronised and merged in Figure 5.3. As it can be seen from the figures all bars have been shifted to make the HdpH RTS start-up time

at 1.5s. However, the behaviour of the application remained intact in the ThreadScope profile. Consequently, this confirmed that the function worked correctly and it did not alter the behaviour presented in the synchronised eventlog.



(a) Before Time Synchronisation



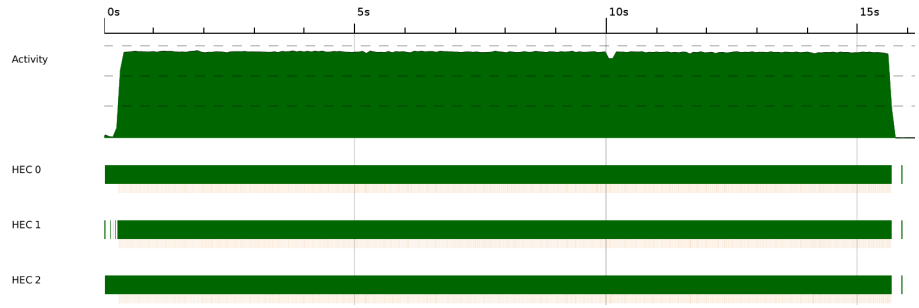
(b) After Time Synchronisation

Figure 5.1: Test of HdpHProf Time Synchronisation Function.

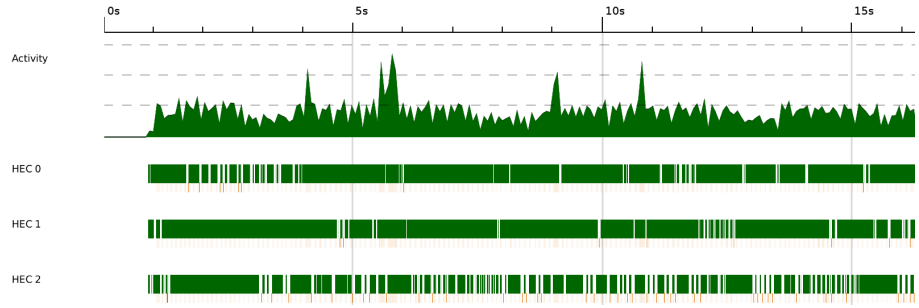
5.2.3 Merging Trace Files

The merge function is used to merge multiple HdpH eventlogs into a single eventlog before visualising the overall performance of an HdpH application (Sections 4.2.4 & 4.3.4). To validate the function we applied it to HdpH eventlogs of real executions then examined the resultant eventlog. For example, the function was used to merge eventlogs of executions on 2 Beowulf cluster nodes with 2 cores up to 32 nodes with 192 cores utilised. Figure 5.2 shows three eventlogs of an HdpH application executed on 3 Beowulf nodes, each utilised 3 cores. In this form these eventlogs do not represent the overall performance of utilising the parallel machine; instead each shows only what is happening on a single node separately.

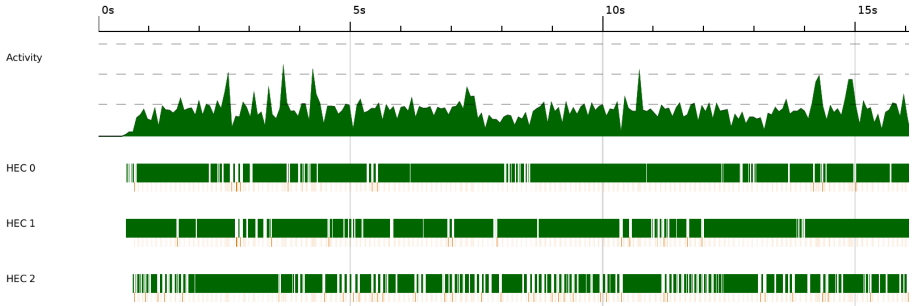
In contrast, merging the eventlogs together into a single eventlog gives an overall picture about how the parallel application utilised the parallel machine. In order to do that the merge function should behave as follows: First, all eventlogs must be merged into a single eventlog. Second, the overall active should reflect the performance of all nodes. Third, the HECs must be ranked incrementally starting from 0 assigned to the first core of the first node to $T_{TotalCores} - 1$ for the last core of the last node. Figure 5.3 illustrates the result of the HdpHProf merge function by merging the three



(a) Node 1



(b) Node 2



(c) Node 3

Figure 5.2: Eventlogs Before Synchronisation and Merging.

eventlogs shown previously. Inspecting the profile visually showed that the function worked correctly and the resultant eventlog presented the overall performance of the three machines to be as expected without any problems.

5.2.4 Contention Analysis Tools

This section presents a validation of functional correctness of HdpHProf contention analysis tools, i.e. the Spark Pool Contention Analysis tool and the Registry Contention Analysis tool (Section 4.3.5). We created synthetic performance data that contains scenarios of contentions situation on the HdpH internals. The scenarios were carefully made with a precision of nanoseconds to produce contention on spark pool at specific locations, with specific numbers of HdpH schedulers during the execution. To be sure that we had not selected a biased sample we examined some random real execution trace file sample scenarios. These scenarios are used as input for the analysis tools; by

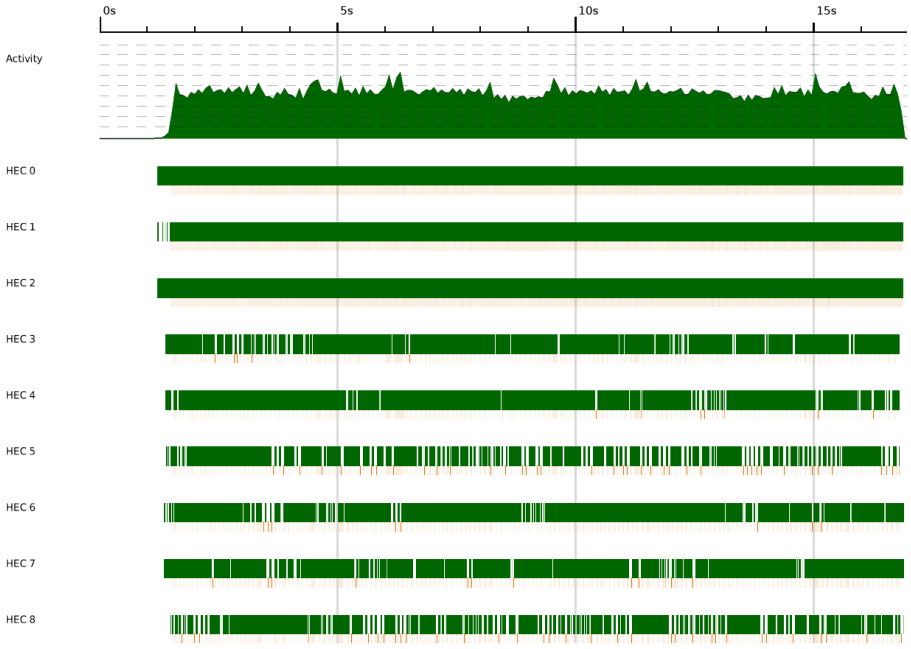


Figure 5.3: Eventlog After Merging.

examining the output profiles we know exactly what to expect to see. If the profiles confirm the performance data this shows that the tools function correctly and the performance profile results are valid.

Spark Pool Contention Analysis tool

The Spark Pool Contention Analysis tool detects contention on the spark pool which is one of the main internal data structures for the HdpH RTS. The tool's functional correctness was examined with multiple scenarios to simulate different contention situations. Listing 5.1 shows an extract of an eventlog from a spark pool contention scenario. Figure 5.4 shows the Spark Pool Contention Analysis tool profile resulting from reading the trace events from the listing.

The eventlog contains 38 spark pool trace events: 19 enter spark pool, 10 convert spark, and 9 nothing to spark. Comparing this with the produced profile of reading the eventlog confirmed that the Spark Pool Contention Analysis tool showed the correct values. Examining the second section of the profile showed that it presented the correct number of times each scheduler was involved in a conflict (all/productive) with the correct percentages of these contentions and total numbers. For example, from the eventlog scheduler number 2 entered the spark pool 4 times and had 2 productive conflicts (for definition see Section 4.2.6). The profile shows the exact numbers with the correct percentage values of occurrence for these number, i.e. $2/4 * 100\% = 50\%$. Similarly, the total row on line 18 of the listing at the end of this profile section

reflects the correct total numbers of conflicts and percentages as in the eventlog, i.e. 19 entries, 6 all conflicts with 31.58% of occurrence and 5 productive conflicts with 26.32% of occurrence.

Listing 5.1: Contention on Spark Pool.

```
1  scenario5 = [
2  -- productiv conflict for schedId 1
3  -- conflict duration 30000ns, group of 2 schedulers.
4   HdpHEvent {e_time = 100030000, e_spec = NothingToSpark {scheduleId = 1}},
5   HdpHEvent {e_time = 100010000, e_spec = EnterSparkPool {scheduleId = 2}},
6   HdpHEvent {e_time = 100000000, e_spec = EnterSparkPool {scheduleId = 1}},
7   HdpHEvent {e_time = 100020000, e_spec = ConvertSpark {scheduleId = 2}},
8  -- new non-productive conflict for schedId 3
9  -- conflict duration 40000ns, group of 2 schedulers.
10  HdpHEvent {e_time = 100070000, e_spec = NothingToSpark {scheduleId = 4}},
11  HdpHEvent {e_time = 100040000, e_spec = EnterSparkPool {scheduleId = 3}},
12  HdpHEvent {e_time = 100050000, e_spec = EnterSparkPool {scheduleId = 4}},
13  HdpHEvent {e_time = 100080000, e_spec = NothingToSpark {scheduleId = 3}},
14 -- new productive conflict for schedId 5
15 -- conflict duration 810000ns, group of 3 schedulers.
16  HdpHEvent {e_time = 100090000, e_spec = EnterSparkPool {scheduleId = 5}},
17  HdpHEvent {e_time = 100900000, e_spec = ConvertSpark {scheduleId = 5}},
18  HdpHEvent {e_time = 100100000, e_spec = EnterSparkPool {scheduleId = 6}},
19  HdpHEvent {e_time = 100110000, e_spec = ConvertSpark {scheduleId = 6}},
20  HdpHEvent {e_time = 100120000, e_spec = EnterSparkPool {scheduleId = 4}},
21  HdpHEvent {e_time = 100130000, e_spec = ConvertSpark {scheduleId = 4}},
22 -- new productive conflict for schedId 2
23 -- conflict duration 80000ns, group of 4 schedulers.
24  HdpHEvent {e_time = 100910000, e_spec = EnterSparkPool {scheduleId = 2}},
25  HdpHEvent {e_time = 100990000, e_spec = ConvertSpark {scheduleId = 2}},
26  HdpHEvent {e_time = 100920000, e_spec = EnterSparkPool {scheduleId = 1}},
27  HdpHEvent {e_time = 100930000, e_spec = ConvertSpark {scheduleId = 1}},
28  HdpHEvent {e_time = 100940000, e_spec = EnterSparkPool {scheduleId = 3}},
29  HdpHEvent {e_time = 100950000, e_spec = NothingToSpark {scheduleId = 3}},
30  HdpHEvent {e_time = 100960000, e_spec = EnterSparkPool {scheduleId = 4}},
31  HdpHEvent {e_time = 100970000, e_spec = NothingToSpark {scheduleId = 4}},
32 -- new productive conflict for schedId 2
33 -- conflict duration 8000ns, group of 2 schedulers.
34  HdpHEvent {e_time = 100991000, e_spec = EnterSparkPool {scheduleId = 2}},
35  HdpHEvent {e_time = 100999000, e_spec = NothingToSpark {scheduleId = 2}},
36  HdpHEvent {e_time = 100992000, e_spec = EnterSparkPool {scheduleId = 3}},
37  HdpHEvent {e_time = 100993000, e_spec = ConvertSpark {scheduleId = 3}},
38 -- new productive conflict for schedId 6
39 -- conflict duration 1000000ns, group of 6 schedulers, maximum conflict.
40  HdpHEvent {e_time = 101000000, e_spec = EnterSparkPool {scheduleId = 6}},
41  HdpHEvent {e_time = 102000000, e_spec = NothingToSpark {scheduleId = 6}},
42  HdpHEvent {e_time = 101100000, e_spec = EnterSparkPool {scheduleId = 7}},
43  HdpHEvent {e_time = 101110000, e_spec = ConvertSpark {scheduleId = 7}},
44  HdpHEvent {e_time = 101120000, e_spec = EnterSparkPool {scheduleId = 8}},
45  HdpHEvent {e_time = 101130000, e_spec = ConvertSpark {scheduleId = 8}},
46  HdpHEvent {e_time = 101140000, e_spec = EnterSparkPool {scheduleId = 1}},
47  HdpHEvent {e_time = 101150000, e_spec = ConvertSpark {scheduleId = 1}},
48  HdpHEvent {e_time = 101160000, e_spec = EnterSparkPool {scheduleId = 2}},
49  HdpHEvent {e_time = 101170000, e_spec = NothingToSpark {scheduleId = 2}},
50  HdpHEvent {e_time = 101180000, e_spec = EnterSparkPool {scheduleId = 3}},
51  HdpHEvent {e_time = 101190000, e_spec = NothingToSpark {scheduleId = 3}}
52  ]
```

The third section of the profile in lines 20 to 33 presents an analysis of conflict durations in milliseconds per schedulers and as total for all schedulers. Also, it shows the maximum productive conflict duration. Examining the results of reading the eventlog the profile presents the correct values. For instance, from the eventlog schedulers number 3 had a conflict that lasts for 40000ns which is shown in the profile as 0.04ms. Likewise, in the eventlog the maximum productive conflict duration of 1000000ns happened to scheduler number 6 it is presented correctly in the profile as 1.0ms.

The last section of the profile in lines 35 to 43 demonstrates an analysis of productive conflicts grouped by the number of schedulers involved in a conflict. Again

```
                                    HdpHprof
                          Spark Pool Contention analysis

Total entry to sparkpool: 19
Total sparks converted: 10
Total No spark to convert: 9
```

SID	Enter	All.Cof.	All.C.%	Pro.Conf.	Pro.C.%
1	3	1	33.33	1	33.33
2	4	2	50.00	2	50.00
3	4	1	25.00	0	0.00
4	3	0	0.00	0	0.00
5	1	1	100.00	1	100.00
6	2	1	50.00	1	50.00
7	1	0	0.00	0	0.00
8	1	0	0.00	0	0.00
Total	19	6	31.58	5	26.32

All times displayed are in milliseconds

SID	All.C.Dur	Mean	Pro.C.Dur	Mean
1	0.0300	0.0300	0.0300	0.0300
2	0.0880	0.0440	0.0880	0.0440
3	0.0400	0.0400	0.0000	0.0000
4	0.0000	0.0000	0.0000	0.0000
5	0.8100	0.8100	0.8100	0.8100
6	1.0000	1.0000	1.0000	1.0000
7	0.0000	0.0000	0.0000	0.0000
8	0.0000	0.0000	0.0000	0.0000
Total	1.9680	0.3280	1.9280	0.3856

```
Max duration in a productive conflict: 1.00000
```

Conflicts grouped by total number of schedulers involved

No.Schedulers	Conf. Occurance%	Conf. Duration%	Mean
8	0.00	0.00	0.0000
7	0.00	0.00	0.0000
6	20.00	51.87	1.0000
5	0.00	0.00	0.0000
4	20.00	4.15	0.0800
3	20.00	42.01	0.8100
2	40.00	1.97	0.0190

Figure 5.4: Spark Pool Contention Analysis Profile.

manual analysis of the eventlog confirms the profile is correct. For example, groups of 2 schedulers conflicts occurred 3 times, 2 of which are productive. The profile shows the group of 2 schedulers has an occurrence of 40% which is correct as the eventlog has 5 total productive conflicts i.e. $2/5 = 40\%$. The two productive conflicts durations are 30000ns and 8000ns; the ratio of these conflict duration to the total productive conflict duration 1928000ns is 1.97% which is reflected correctly in the profile. Similarly, the mean value of conflicts duration 30000ns and 8000ns is 19000ns which is presented correctly as 0.019ms in the profile.

Table 5.3 summarises the tests that the Spark Pool Contention Analysis profile passed during this validation. This confirms that the Spark Pool Contention Analysis tool produced correct and validated performance results for HdpH RTS' behaviour.

Registry Contention Analysis tool

The Registry Contention Analysis tool detects contention on the HdpH registry. The tool is tested with scenarios in a similar manner to the previously tested Spark Pool Contention Analysis tool in Section 5.2.4. Listing 5.2 shows an extract of an eventlog

Spark Pool Contention Analysis Profile Functionality Test		Result
Section 1	Detects total entries to spark pool	pass
	Detects total number of sparks converted	pass
	Detects total number of no sparks to convert	pass
Section 2	Detects entries per scheduler	pass
	Detects number of all conflicts per scheduler	pass
	Detects occurrence ratio for all conflicts per scheduler	pass
	Detects number of productive conflicts per scheduler	pass
	Detects occurrence ratio for productive conflicts per scheduler	pass
	Detects total number all conflicts	pass
	Detects total occurrence ratio for all conflict	pass
	Detects total number productive conflicts	pass
	Detects total occurrence ratio for productive conflict	pass
Section 3	Detects all conflicts duration per scheduler	pass
	Detects mean value of all conflicts durations per scheduler	pass
	Detects productive conflicts duration per schedulers	pass
	Detects mean value of productive conflicts durations per schedulers	pass
	Detects total all conflicts duration	pass
	Detects mean value of total all conflicts duration	pass
	Detects total productive conflicts duration	pass
	Detects mean value of total productive conflicts duration	pass
	Detects maximum duration of productive conflict	pass
Section 4	Detects productive conflicts occurrence ratio per group of schedulers	pass
	Detects productive conflicts duration ratio per group of schedulers	pass
	Detects mean value of productive conflicts duration per group of schedulers	pass

Table 5.3: Spark Pool Contention Analysis Profile Functionality Test.

with a scenario that simulates contention on the registry. Figure 5.4 shows an output profile of the Registry Contention Analysis tool reading the trace events from the listing.

The eventlog contains 34 registry trace events: 18 globalising GRef , 6 freeing GRef, and 10 dereferencing GRef. Comparing the facts from the trace events with the profile produced by reading the eventlog confirms that the Registry Contention Analysis tool shows the correct values. Validating the second section in lines 9 to 20 of the profile confirms that it presents the correct number of times each scheduler is involved in a conflict with correct percentages of these conflicts and total numbers. For example, from the eventlog scheduler number 4 enters the registry 3 times and has 1 conflict. From the figure we can see the profile shows the exact numbers with the correct percentage values of occurrence for these number, i.e. $1/3 = 33.33\%$. Likewise, the total row on line 20 at the end of this profile section presents the correct total numbers of conflicts and percentages as in the eventlog, i.e. 17 entries, 5 conflicts; giving a 29.41% occurrence.

Listing 5.2: Contention on Registry.

```
1 scenario5 = [
2 -- no conflict
3  HdpHEvent {e_time = 000001000, e_spec = GlobaliseGRef {scheduleId = 1}},
4  HdpHEvent {e_time = 000002000, e_spec = GRefGlobalised {scheduleId = 1}},
5  HdpHEvent {e_time = 000003000, e_spec = GlobaliseGRef {scheduleId = 3}},
6  HdpHEvent {e_time = 000004000, e_spec = GRefGlobalised {scheduleId = 3}},
7 -- conflict of type mixture for schedId 1
8 -- conflict duration 9000ns, group of 2 schedulers.
9  HdpHEvent {e_time = 000010000, e_spec = GlobaliseGRef {scheduleId = 1}},
10 HdpHEvent {e_time = 000012000, e_spec = DereferenceGRef {scheduleId = 0}},
11 HdpHEvent {e_time = 000019000, e_spec = GRefGlobalised {scheduleId = 1}},
12 HdpHEvent {e_time = 000013000, e_spec = GRefDereferenced {scheduleId = 0}},
13 -- new coflict of type globalise for schedId 2
14 -- conflict duration 10000ns, group of 2 schedulers, maximum duration.
15 HdpHEvent {e_time = 000020000, e_spec = GlobaliseGRef {scheduleId = 2}},
16 HdpHEvent {e_time = 000030000, e_spec = GRefGlobalised {scheduleId = 2}},
17 HdpHEvent {e_time = 000021000, e_spec = GlobaliseGRef {scheduleId = 3}},
18 HdpHEvent {e_time = 000022000, e_spec = GRefGlobalised {scheduleId = 3}},
19 -- new coflict of type globalise for schedId 4
20 -- conflict duration 4000ns, group of 2 schedulers.
21 HdpHEvent {e_time = 000031000, e_spec = GlobaliseGRef {scheduleId = 4}},
22 HdpHEvent {e_time = 000035000, e_spec = GRefGlobalised {scheduleId = 4}},
23 HdpHEvent {e_time = 000032000, e_spec = GlobaliseGRef {scheduleId = 5}},
24 HdpHEvent {e_time = 000034000, e_spec = GRefGlobalised {scheduleId = 5}},
25 -- new conflict of type free for schedId 6
26 -- conflict duration 5000ns, group of 3 schedulers.
27 HdpHEvent {e_time = 000036000, e_spec = FreeGRef {scheduleId = 6}},
28 HdpHEvent {e_time = 000041000, e_spec = GRefFreed {scheduleId = 6}},
29 HdpHEvent {e_time = 000037000, e_spec = FreeGRef {scheduleId = 4}},
30 HdpHEvent {e_time = 000038000, e_spec = GRefFreed {scheduleId = 4}},
31 HdpHEvent {e_time = 000039000, e_spec = FreeGRef {scheduleId = 3}},
32 HdpHEvent {e_time = 000040000, e_spec = GRefFreed {scheduleId = 3}},
33 -- no conflict
34 HdpHEvent {e_time = 000042000, e_spec = GlobaliseGRef {scheduleId = 2}},
35 HdpHEvent {e_time = 000043000, e_spec = GRefGlobalised {scheduleId = 2}},
36 HdpHEvent {e_time = 000043100, e_spec = GlobaliseGRef {scheduleId = 4}},
37 HdpHEvent {e_time = 000043200, e_spec = GRefGlobalised {scheduleId = 4}},
38 -- new conflict of type deref for schedId 1
39 -- conflict duration 6000ns, group of 4 schedulers.
40 HdpHEvent {e_time = 000044000, e_spec = DereferenceGRef {scheduleId = 1}},
41 HdpHEvent {e_time = 000050000, e_spec = GRefDereferenced {scheduleId = 1}},
42 HdpHEvent {e_time = 000045000, e_spec = DereferenceGRef {scheduleId = 2}},
43 HdpHEvent {e_time = 000046000, e_spec = GRefDereferenced {scheduleId = 2}},
44 HdpHEvent {e_time = 000047000, e_spec = DereferenceGRef {scheduleId = 3}},
45 HdpHEvent {e_time = 000048000, e_spec = GRefDereferenced {scheduleId = 3}},
46 HdpHEvent {e_time = 000049000, e_spec = DereferenceGRef {scheduleId = 5}},
47 HdpHEvent {e_time = 000049900, e_spec = GRefDereferenced {scheduleId = 5}}
48 ]
```

The next section of the profile, lines 22 to 34, shows an analysis of conflicts durations in milliseconds per schedulers and as a total for all schedulers. In addition, it shows the maximum conflict duration. The profile results of reading the eventlog confirms it presents the correct values. For instance, from the eventlog scheduler number 1 had 2 conflicts that lasted for 9000ns and 6000ns respectively which is shown correctly in the profile as total of 0.015ms conflict duration. Similarly, the maximum conflict duration from the eventlog, 10000ns, occurred to scheduler number 2 and is reflected correctly in the profile as it shows it is 0.01ms.

The fourth section of the profile, lines 36 to 43, presents an analysis of conflicts grouped by the number of schedulers involved in a conflict. The profile results of reading the eventlog confirm that the profile shows the correct values. For example, the eventlog contains 5 conflicts, 3 of them are group of 2 schedulers conflict. The profile shows the group of 2 schedulers has occurrence of 60% which is correct as $3/5 = 60\%$. The three conflicts durations are 9000ns, 10000ns, and 4000ns; the ratio of these conflict durations to the total conflict duration of 34000ns is 67.65%, which is

```
1                                   HdpHprof
2                          Registry Contention analysis
3
4  Total entries to registry: 17
5  Total globalise entries: 9
6  Total free entries: 3
7  Total dereference entries: 5
8  ------------------------------------------------------------------------------
9  Registry Analysis
10 ------------------------------------------------------------------------------
11 SID          Enter        Conflict     Conflict%
12 0            1            0            0.00
13 1            3            2            66.67
14 2            3            1            33.33
15 3            4            0            0.00
16 4            3            1            33.33
17 5            2            0            0.00
18 6            1            1            100.00
19 ------------------------------------------------------------
20 Total        17           5            29.41
21 ------------------------------------------------------------------------------
22 Displayed times are in milliseconds.
23 SID          Conf. Duration            Mean
24 0            0.0000                    0.0000
25 1            0.0150                    0.0075
26 2            0.0100                    0.0100
27 3            0.0000                    0.0000
28 4            0.0040                    0.0040
29 5            0.0000                    0.0000
30 6            0.0050                    0.0050
31 ------------------------------------------------------------
32 Total        0.0340                    0.0068
33 ------------------------------------------------------------------------------
34 Max conflict duration : 0.01000
35 ------------------------------------------------------------------------------
36 Number of times a conlflict occured with this number of schedulers
37 No. Schedulers   Conf. Occurance%     Conf. Duration%      Mean
38 7                0.00                 0.00                 0.0000
39 6                0.00                 0.00                 0.0000
40 5                0.00                 0.00                 0.0000
41 4                20.00                17.65                0.0060
42 3                20.00                14.71                0.0050
43 2                60.00                67.65                0.0077
44 ------------------------------------------------------------------------------
45 Conflict grouped by events type
46 Events type      Conf. Occurance%     Conf. Duration%      Mean
47 Glob             40.00                41.18                0.0070
48 Free             20.00                14.71                0.0050
49 Deref            20.00                17.65                0.0060
50 Mixture          20.00                26.47                0.0090
```

Figure 5.5: Registry Contention Analysis Profile.

reflected correctly in the profile. Similarly, the mean value of conflict duration 9000ns, 10000ns, and 4000ns, is 0.0077ms which is presented correctly in the profile.

The last section of the profile, lines 45 to 50, shows an analysis of conflicts grouped by operations (event) type (Section 4.3.5). The results shown on the profile correspond with the actual performance data from the eventlog. For instance, the group conflicts of type Globalise (Glob) occurred twice in the eventlog. The profile presented in this group with a 40% of occurrence which is correct as the total number of conflicts is 5. In addition, it shows that this group has 41.18% of conflict duration, $(10000ns + 4000ns)/34000ns = 41.18\%$, a correct result.

Table 5.4 summarises the tests that the Registry Contention Analysis profile passed during this validation of functional correctness. This confirms that the Registry Contention Analysis tool produces correct and validated performance results of HdpH RTS behaviour.

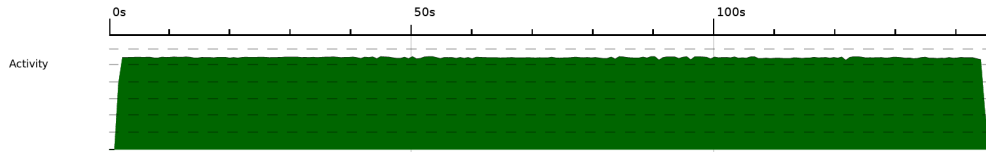
Registry Contention Analysis Profile Functionality Test		Result
Section 1	Detects total entries to registry	pass
	Detects total number of globalise operations	pass
	Detects total number of free operations	pass
	Detects total number of dereference operations	pass
Section 2	Detects entries per scheduler	pass
	Detects number of conflicts per scheduler	pass
	Detects occurrence ratio of conflicts per scheduler	pass
	Detects total number conflicts	pass
	Detects total occurrence ratio of conflict	pass
Section 3	Detects conflicts duration per scheduler	pass
	Detects mean value of conflicts durations per scheduler	pass
	Detects total conflicts duration	pass
	Detects mean value of total conflicts duration	pass
	Detects maximum conflict duration	pass
Section 4	Detects conflicts occurrence ratio per group of schedulers	pass
	Detects conflicts duration ratio per group of schedulers	pass
	Detects mean value of conflicts duration per group of schedulers	pass
Section 5	Detects conflicts occurrence ratio per group of events type	pass
	Detects conflicts duration ratio per group of events type	pass
	Detects mean value of conflicts duration group of events type	pass

Table 5.4: Registry Contention Analysis Profile Functionality Test.

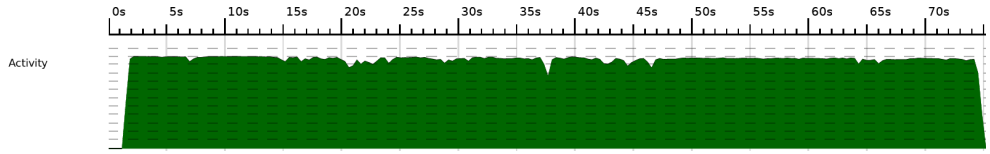
5.3 Scalability

This section evaluates the performance of HdpHProf in terms of scalability. HdpHProf was benchmarked to see how it could profile HdpH applications running on relatively large numbers of nodes and cores, i.e. 32 nodes with a total of 129 cores on a Beowulf cluster. The hardware set-up and benchmarks were described previously in Section 3.1.1 and Section 5.1 respectively. The goal of this experiment is to show that HdpHProf is able to produce and visualise the performance profiles of large scale runs on a common parallel architecture. For this demonstration we used two benchmarks with different parallel paradigms, i.e. Fibonacci, which is a divide and conquer, and Summatory Liouville, which is flat data parallel. The applications were profiled on different number of nodes, i.e. 1, 2, 4, 8, 16, and 32, to show that HdpHProf is able to scale to profile HdpH applications that run for long time (up to 148s), or run on a large number of cores. This section is not about tuning behaviour or showing a good performance of the profiled applications as these issues will be discussed later in Chapter 6.

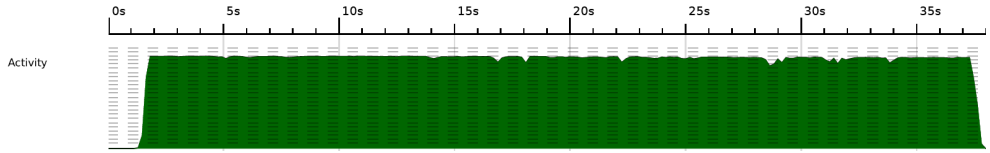
Fibonacci. Figure 5.6 illustrates performance profiles (Overall Activity) of Fibonacci 50, co-location threshold 24, sequential threshold 16. The application was profiled on 1, 2, 4, 8, 16, and 32 nodes, with 6 cores each (192 cores in total). The runtimes of the profiled application varied between 148s and 9.4s depending on the number of nodes used.



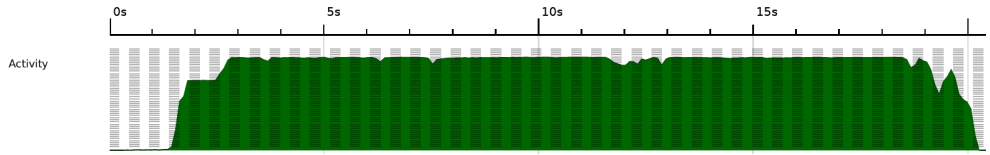
(a) 6 Cores



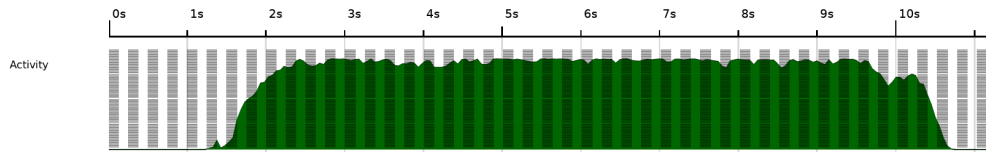
(b) 12 Cores



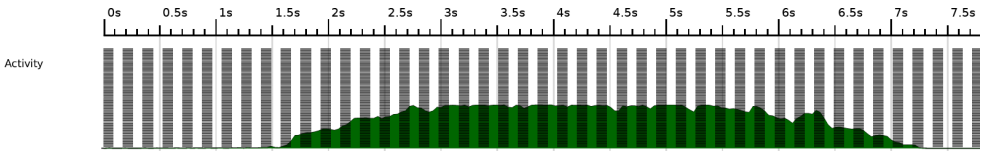
(c) 24 Cores



(d) 48 Cores



(e) 96 Cores

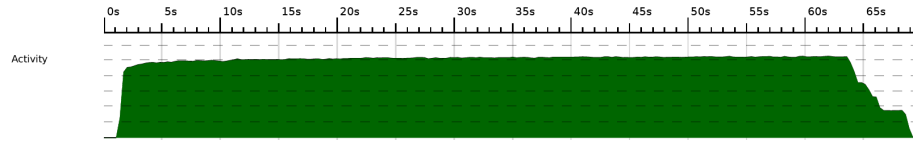


(f) 192 Cores

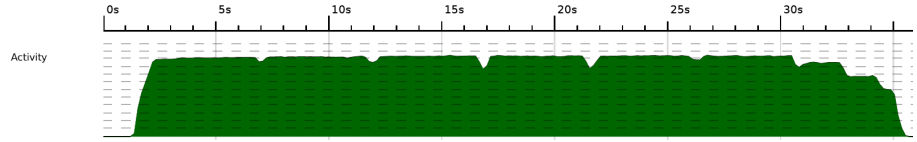
Figure 5.6: Fibonacci 50 Thresholds Co-loc. 34 Seq. 16.

Summatory Liouville. Figure 5.7 demonstrates performance profiles (Overall Activity) of Summatory Liouville 40,000,000 with chunk size 400,000. This was to measure

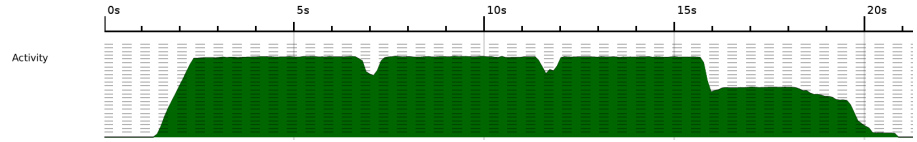
HdpHProf scalability; the application profiled on 1, 2, 4, 8, 16, and 32 nodes 6 cores each (192 cores in total). HdpHProf was able to profile all executions, runtime varied between 68s and 11s depending on the number of nodes used. These experiments show that HdpHProf is capable of profiling HdpH applications at the current scale of widespread clusters.



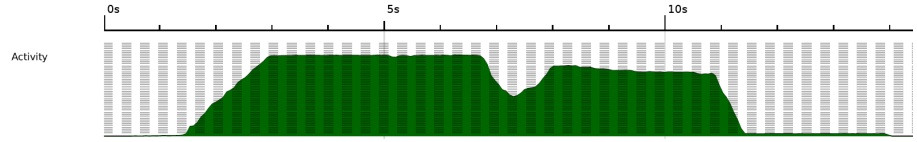
(a) 6 Cores



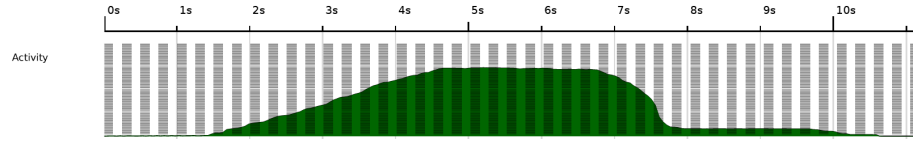
(b) 12 Cores



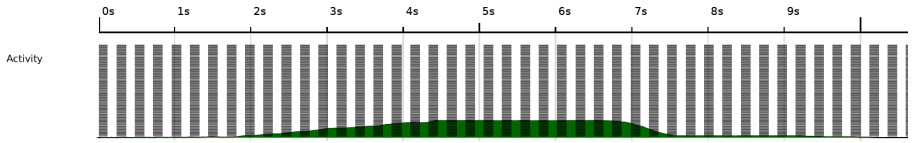
(c) 24 Cores



(d) 48 Cores



(e) 96 Cores



(f) 192 Cores

Figure 5.7: Summatory Liouville 40,000,000 Chunk Size 400,000.

5.4 Profiling Data Size

This section investigates and characterises the amount of profiling data size generated by HdpHProf. This is similar to our study of parallel function profilers in Chapter 3 and [4]. First, we will show how profiling data size changes in respect to the increase in the profiled application computation size (sparks). Then, we will present how data size changes as the number of Processing Elements (PEs) increases. We will profile the execution of four HdpH benchmarks, i.e. Queens, Mandelbrot, Fibonacci, and SumEuler, and measure the amount of storage space the profiler consumes in KB. We will report the median of 5 executions. The hardware set-up and benchmarks were described previously in Section 3.1.1 and Section 5.1 respectively.

To change the computation size we ran the benchmarks to produce different number of sparks. The number of sparks was increased by about the factor of 2 and ranged from 200 to 1000 sparks. Similarly, we increased the number of PEs (nodes) by a factor of 2 starting by 1 PE until reaching 16 PEs. We chose to stop at 16 PEs for these reasons. We wanted accurate results by running the experiments on cluster nodes with very low load (less than 0.5), and it is difficult to find more than the 16 nodes that are not heavily loaded on our cluster. These experiments ran for along time and were completed over many days so we did not want to disturb other researchers by using the whole cluster.

5.4.1 Profiling Data Size vs Computation Size

Queens. Figure 5.8 shows that the tracing data size of profiling queens increases as the computation size (sparks) increases. Doubling the computation size results in a slight increase to the size of the trace file, by about 25% on average.

Mandelbrot. Figure 5.9 shows that the tracing data size increases dramatically as the computation size gets larger. Doubling the computation size increases the tracing data by 258% on average.

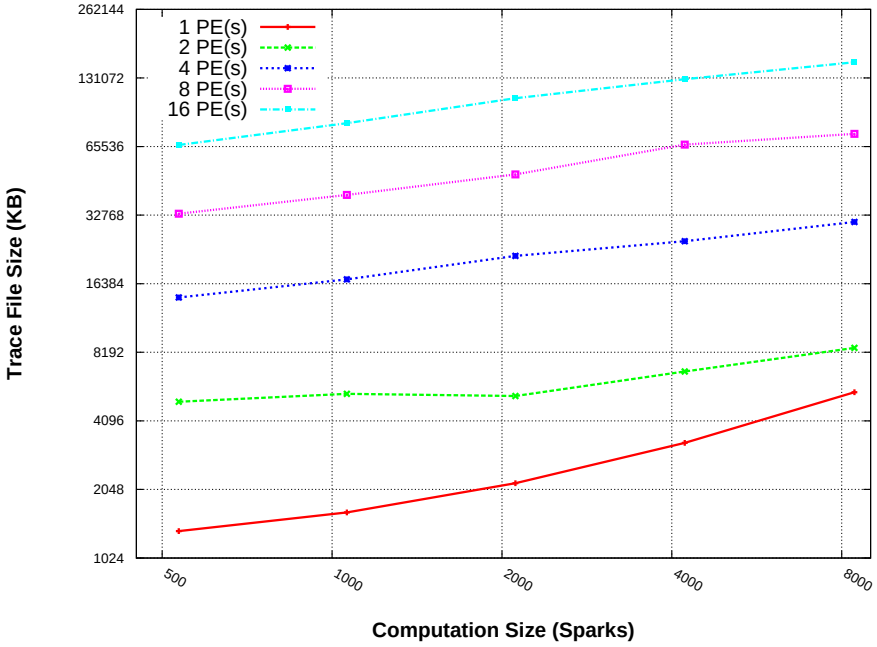


Figure 5.8: Queens Profiling Data Size vs Computation Size.

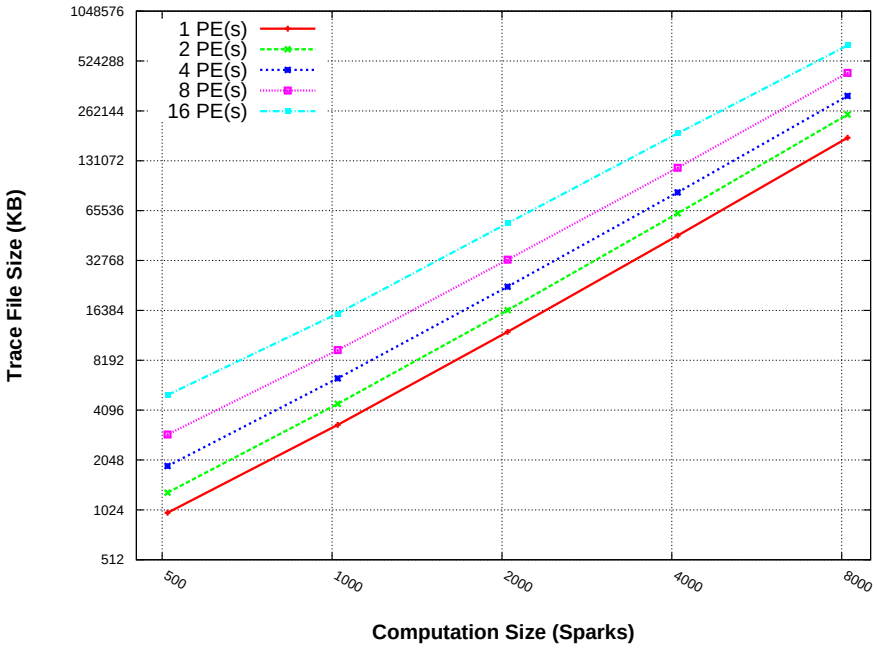


Figure 5.9: Mandelbrot Profiling Data Size vs Computation Size.

Fibonacci. Figure 5.10 shows that the data size grows as the computation size increases. Increasing the computation size by the factor of 2 increases the trace data by 146% on average.

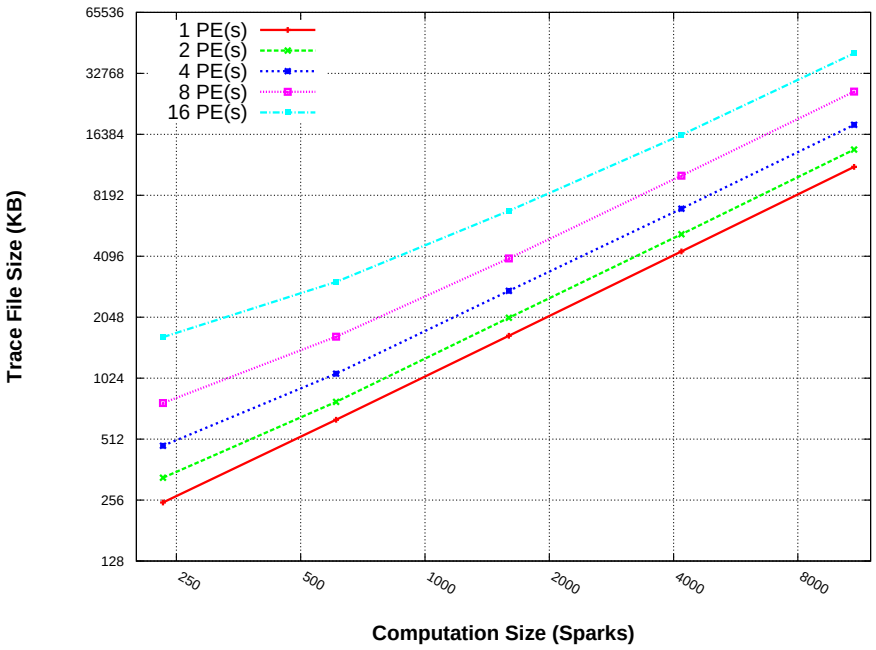


Figure 5.10: Fibonacci Profiling Data Size vs Computation Size.

SumEuler. Figure 5.11 shows that the tracing data size increases as the computation size gets bigger. Doubling the computation size increases the trace data by 108% on average.

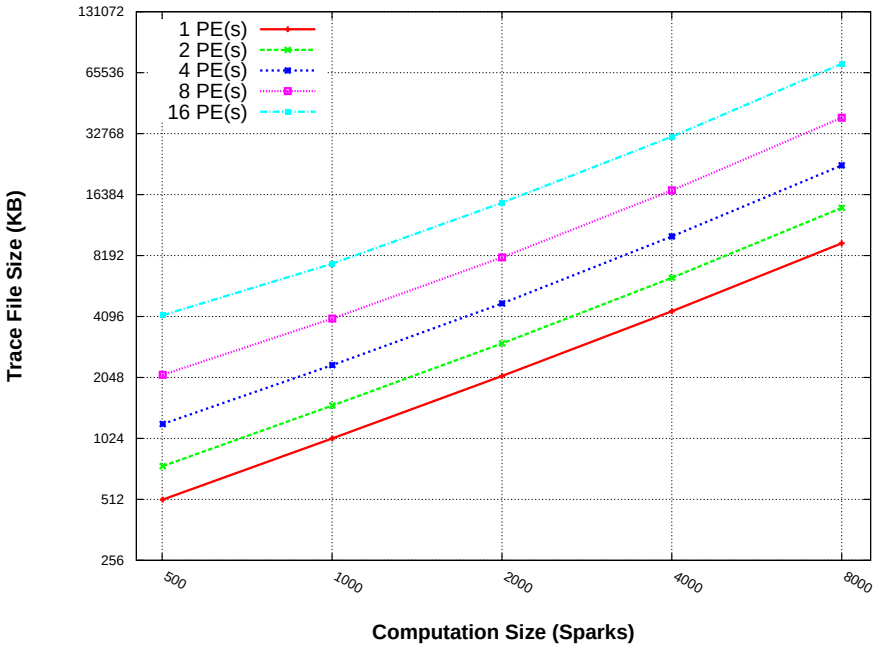


Figure 5.11: SumEuler Profiling Data Size vs Computation Size.

5.4.2 Profiling Data Size vs Number of PEs

Queens. Figure 5.12 shows that increasing the number of PEs increases the tracing data size. Doubling the number of PEs rises the size of tracing data by 164% on average.

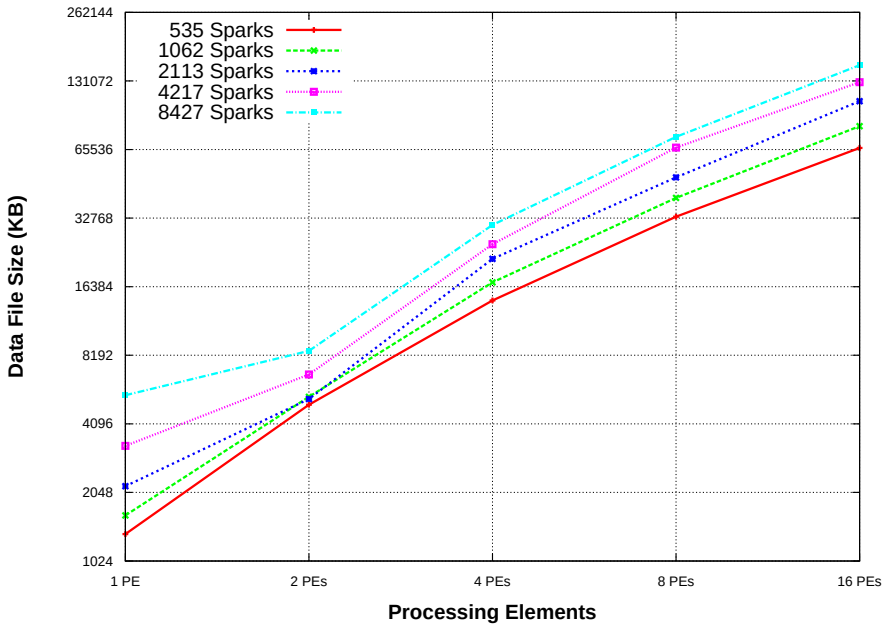


Figure 5.12: Queens Profiling Data Size vs Number of PEs.

Mandelbrot. Figure 5.13 shows that increasing the number of PEs will result in an increase to the tracing data size. Doubling the number of PEs increases the tracing data size by 45% on average.

Fibonacci. Figure 5.14 shows that increasing the number of PEs results in an increase to the tracing data size. Increasing the number of PEs by the factor of 2 increases the tracing data size by 46% on average.

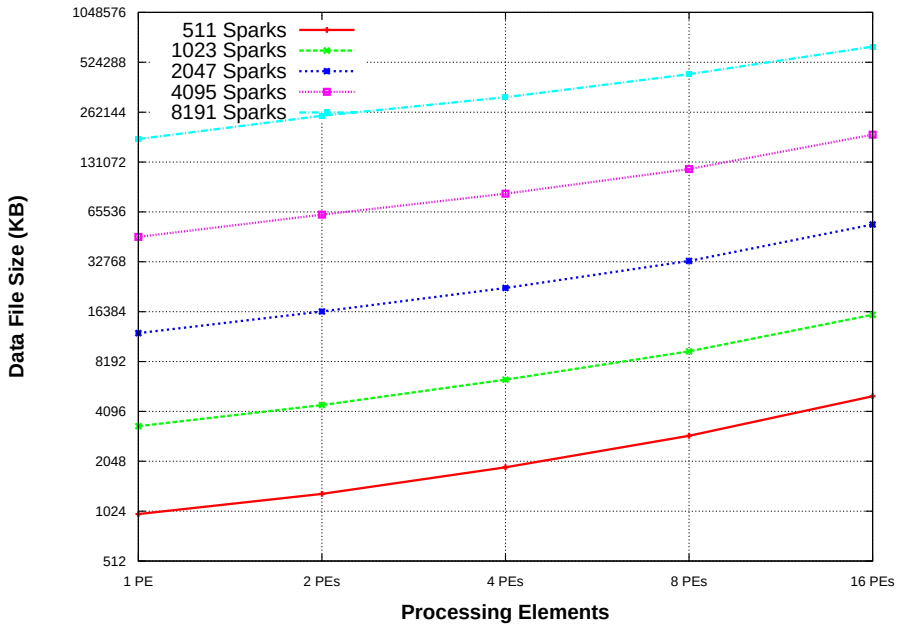


Figure 5.13: Mandelbrot Profiling Data Size vs Number of PEs.

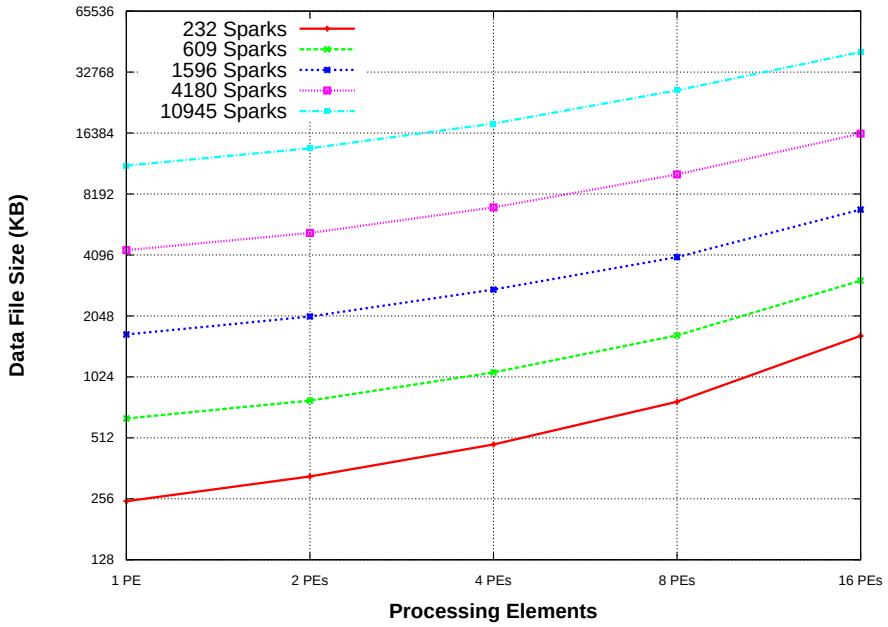


Figure 5.14: Fibonacci Profiling Data Size vs Number of PEs.

SumEuler. Figure 5.15 depicts that the tracing data size increases as the number of PEs increases. Doubling the number of PEs increases the size of tracing data by 66% on average.

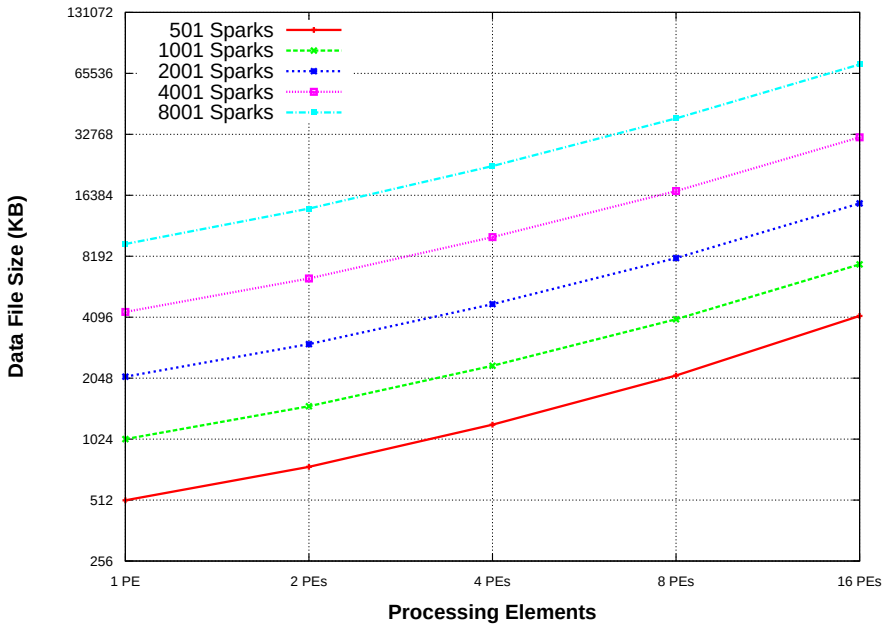


Figure 5.15: SumEuler Profiling Data Size vs Number of PEs.

5.4.3 Profiling Data Size Comparison and Discussion

We have characterised the performance of HdpHProf in terms of profiling data size, and demonstrated how increasing computation size and number of PEs changes the profiling data size. Increasing the computation size makes the profiler produce more profiling data size. Similarly, increasing the number of PEs increases the profiling data size. Noticeably, Queens' results are slightly different from other benchmarks. This is because Queens' performance is poor for reasons that are discussed in Chapter 6.

Knüpfer et al. [71] argue that tracing profilers inevitably introduce overheads that slow down the execution of the application and alter its original behaviour. Tracing tools should try to keep this effect to minimum. Important parts of tracing overheads are the tracing execution time overhead and storage of trace data to disk. Moreover, overall overhead of the monitored application should remain within an acceptable level [71].

To see if HdpHProf overheads stay inside an acceptable range we compared our results characterising HdpHProf performance with the performance results of the studied profilers from Chapter 3 and [4]. HdpHProf and the profilers in our study, were classified according to the same matrices. However, we compared the results of HdpHProf from four different benchmarks because we did not have a concordance for HdpH. Even though it was not the same benchmark, the results showed a pattern that

is more dependent on the profiler than the benchmark. We used four benchmarks and we did not find four different set of behaviours. There are some differences but they are not really major. We report the average from the four benchmarks results. The comparison is based on the results from 1 PE up to 8 PEs, because the other profilers' study was limited to 8 PEs only.

Doubling computation size increases the tracing data size of HdpHProf by an average of 134%. Similarly, Score-P (MPI) increases by 99%, Eden tracing increases by 160%, Score-P (OpenMP) increases by 99% and GHC-PPS increases by 90%. On the other hand, increasing the number of PEs by a factor of two increases HdpHProf tracing data size by an average of 80%. This is compared to 1.7% for Score-P (MPI), 127% for Eden tracing, 1.1% for Score-P (OpenMP), and 78% for GHC-PPS. Moreover, the average trace file size of HdpHProf from all executions is 28 MB compared to 213 MB for Score-P (MPI), 7 MB for Eden tracing, 344 MB for Score-P (OpenMP), and 1.1 MB for GHC-PPS. From this result, we can see that HdpHProf tracing data size is modest and fits within the range of other functional profilers.

5.5 Execution Time Overhead

This section investigates the execution time overhead of HdpHProf. First, we will show how profiling execution runtime overhead changes in respect to the increase in computation size of the profiled application. Then, we will present how overhead changes as the number of PEs increases. The methodology and experimental set-up of this experiment is the exactly same as in Section 5.4. However, we measure the profiling execution time overhead as follows:

$$OH = \frac{(T_P - T_N)}{T_N} * 100 \tag{5.1}$$

where OH is the overhead, T_N is the runtime of the parallel program without profiling, and T_P is the runtime of the parallel program with profiling.

5.5.1 Runtime Overhead vs Computation Size

Queens. Figure 5.16 shows that the runtime overhead decreases as the computation size increases. Increasing the computation size by a factor of 2 declines the runtime overhead by average of 18% points.

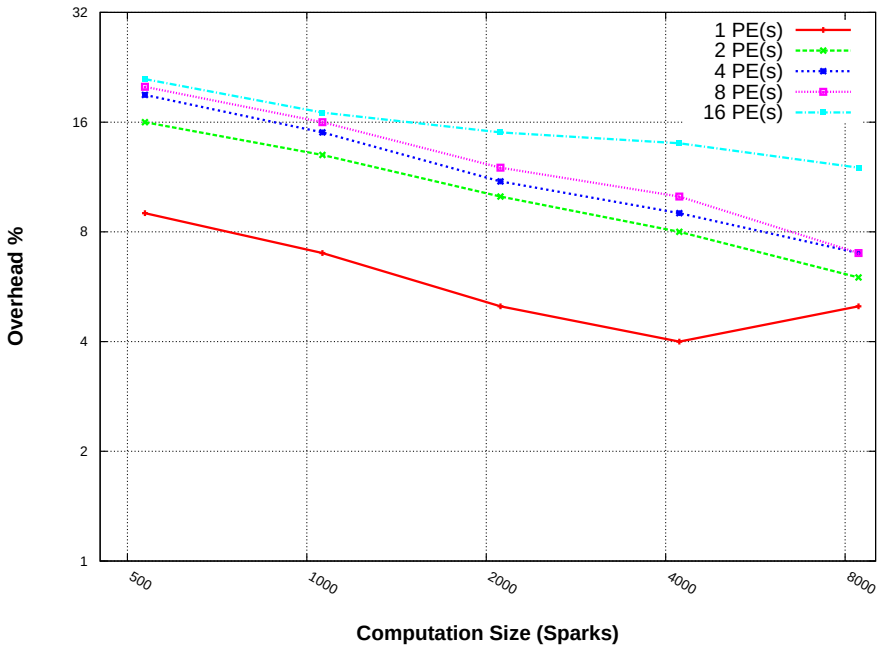


Figure 5.16: Queens Runtime Overhead in Relation to Computation.

Mandelbrot. Figure 5.17 depicts that increasing the computation size decreases runtime overhead. Doubling the computation size decreases the runtime overhead by average of 52% points.

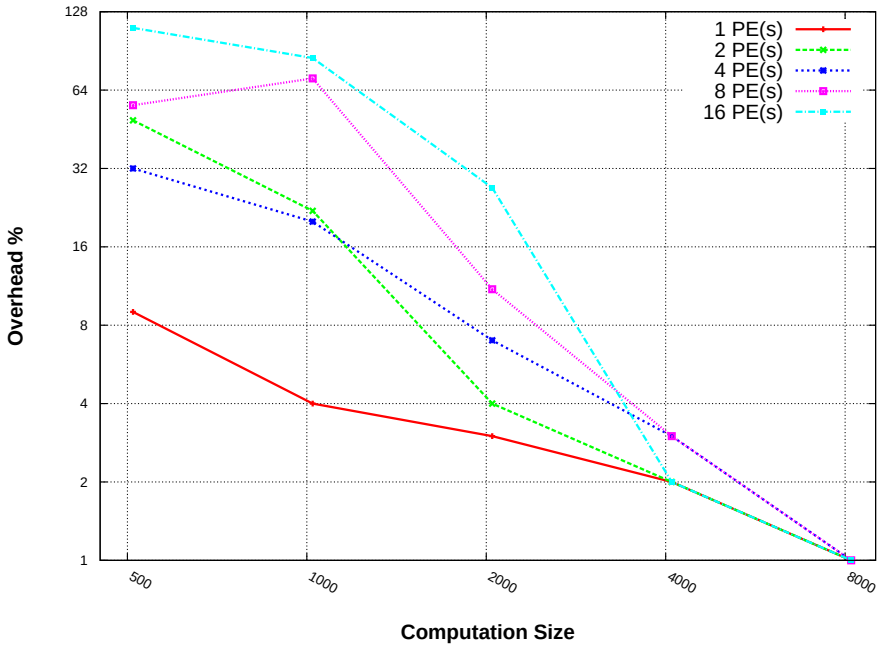


Figure 5.17: Mandelbrot Runtime Overhead in Relation to Computation.

Fibonacci. Figure 5.18 illustrates that the runtime overhead decreases as the computation size increases. Increasing the computation size by a factor of 2 lowers the runtime overhead by 26% points on average.

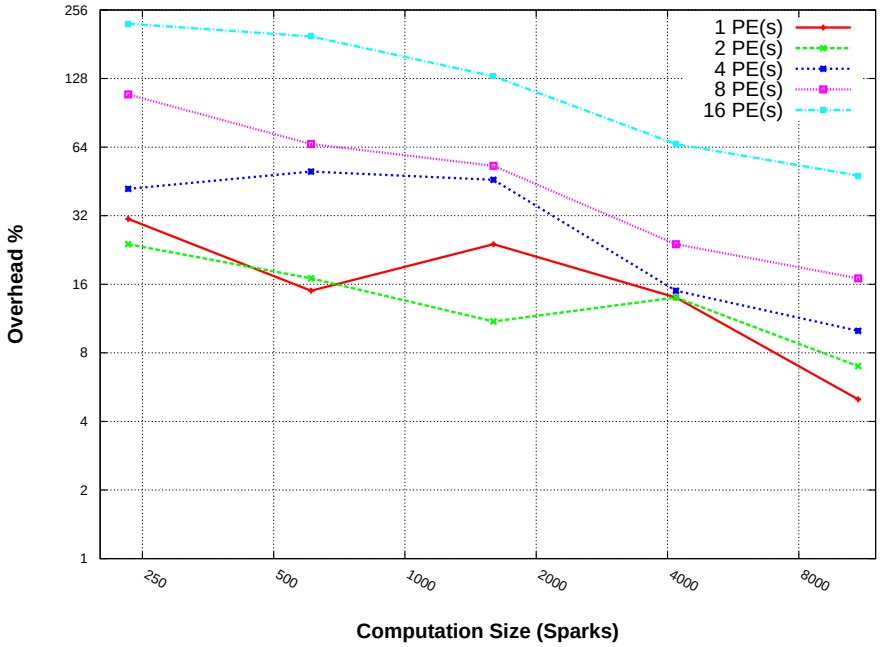


Figure 5.18: Fibonacci Runtime Overhead in Relation to Computation.

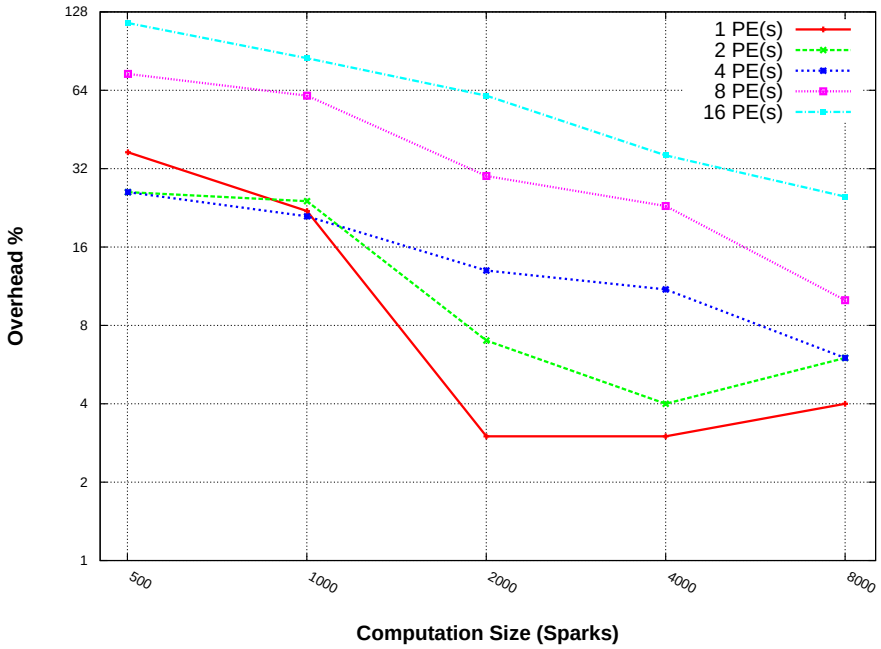


Figure 5.19: SumEuler Runtime Overhead in Relation to Computation.

SumEuler. Figure 5.19 shows that increasing the computation size decreases runtime overhead. Doubling the computation size decreases the runtime overhead by average of 26% points.

5.5.2 Runtime Overhead vs Number PEs

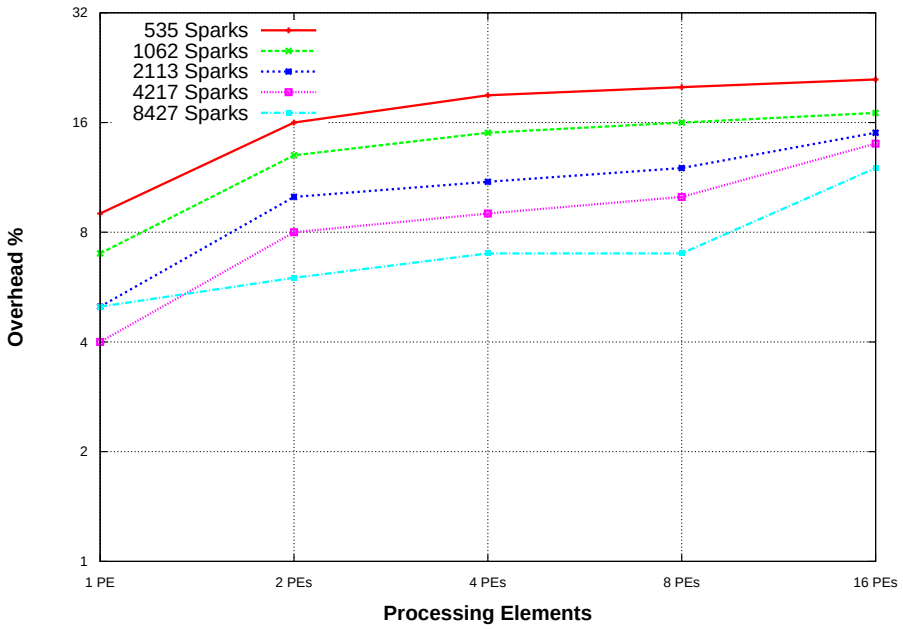


Figure 5.20: Queens Runtime Overhead vs Number of PEs.

Queens. Figure 5.20 shows that the runtime overhead increases as the number of PEs increases. Increasing the number of PEs by a factor of 2 rises the runtime overhead by an average of 31% points.

Mandelbrot. Figure 5.21 illustrates that the majority of the curves show that the runtime overhead increases as the number of PEs increases. Doubling the number of PEs increases the runtime overhead by average of 81% points.

Fibonacci. Figure 5.22 depicts that runtime overhead increases as the number of PEs increases. Increasing the number of PEs by a factor of 2 increases the runtime overhead by average of 87% points.

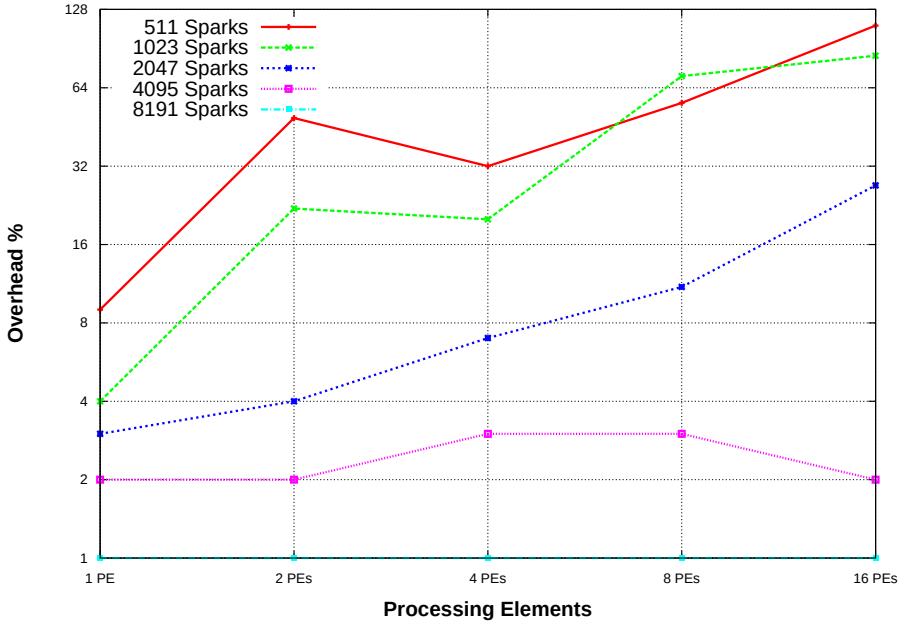


Figure 5.21: Mandelbrot Runtime Overhead vs Number of PEs.

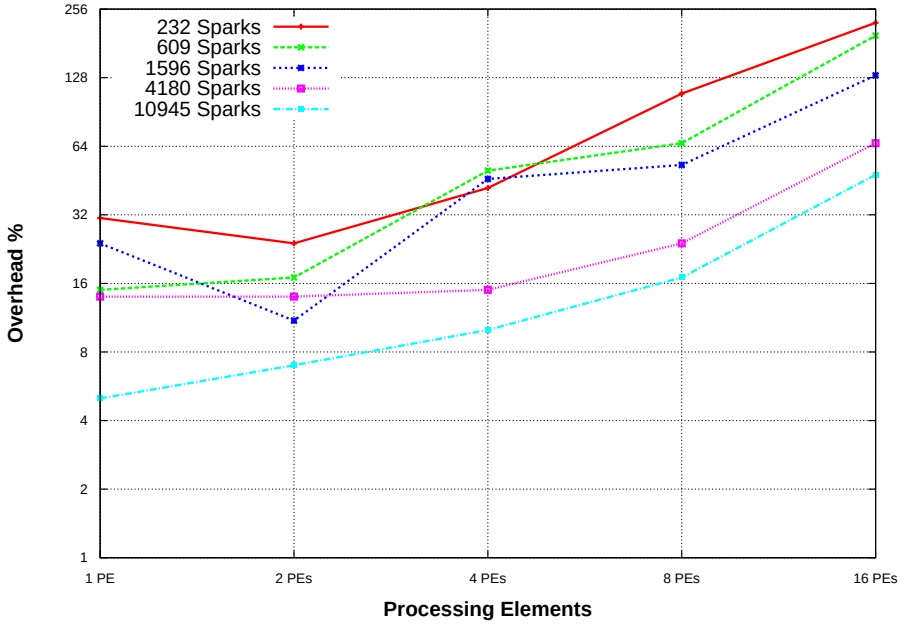


Figure 5.22: Fibonacci Runtime Overhead vs Number of PEs.

SumEuler. Figure 5.23 shows that increasing the number of PEs increases the runtime overhead. Doubling the number of PEs rises the runtime overhead by average of 76% points.

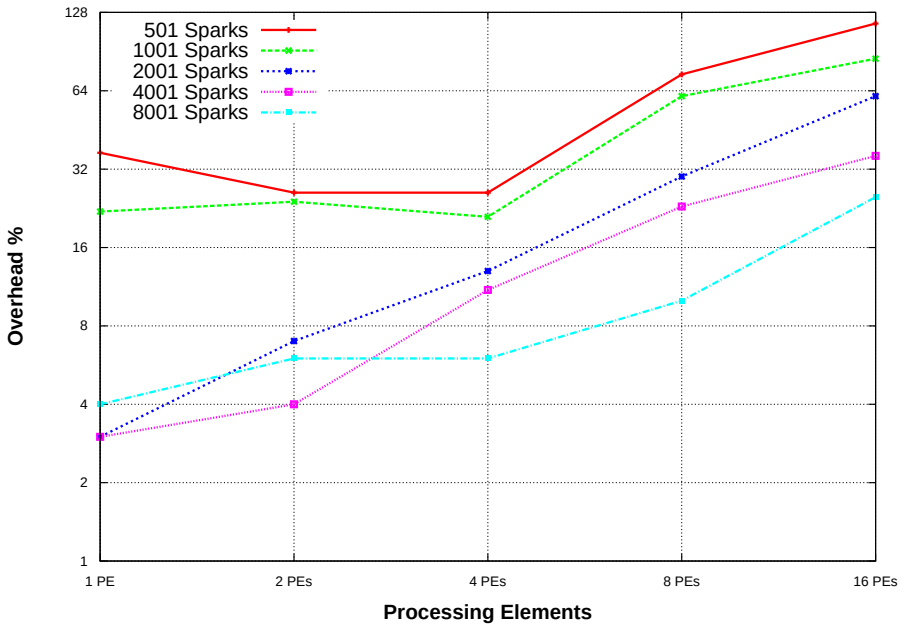


Figure 5.23: SumEuler Runtime Overhead vs Number of PEs.

5.5.3 Runtime Overhead Discussion

We have characterised the performance of HdpHProf in terms profiling execution runtime overhead and illustrated how increasing computation size and number of PEs changes the profiling overhead. Increasing the computation size reduces profiling overhead. However, increasing the number of PEs introduces more profiling overhead. To see if HdpHProf overhead stays inside an acceptable range we compared our results of characterising HdpHProf performance with the performance results of the studied profilers similar to Section 5.4.3.

Doubling the computation size decreases the execution time overhead of HdpHProf by an average of 30% points. Only Score-P (OpenMP) that shows similar behaviour which decreases by 14% points. However, Score-P (MPI) increases by 4% points, Eden tracing increases by 70% points, and GHC-PPS increases by 21% points. Increasing the number of PEs, on the other hand, shows that increasing it by a factor of two HdpHProf data size increases by an average of 68% points. This is compared to 2% increase for Score-P (MPI), 157% points for Eden Tracing, 13% points decrease for Score-P (OpenMP), and 90% points for GHC-PPS.

The average execution time overhead of HdpHProf from all executions is 18% point. In comparison, the execution time overhead for other profilers are 179% for Score-P (MPI), 8% for Eden tracing, 705% for Score-P (OpenMP), and 8% for GHC-

PPS. Again the results show that HdpHProf overhead in terms of execution time is not excessive and stays in the range of other functional profilers. Importantly, execution time overhead decreases as the computations size increases. On the contrary, it increases for other functional profilers.

5.6 HdpH Tracing Overhead

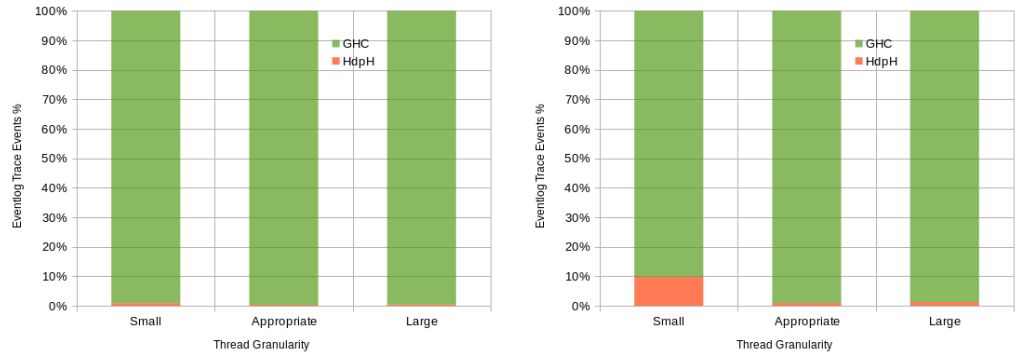
HdpHProf emits HdpH trace events into the GHC-PPS eventlog, and this section compares the overhead of HdpHProf tracing over GHC tracing in the eventlog. This is to ensure that the lightweight tracing of GHC-PPS [67] is preserved and not perturbed with HdpHProf. We used four benchmarks Mandelbrot, Summatory Liouville, Fibonacci, and SumEuler, two divide and conquer, and two data parallel, each with three different thread granularity, i.e. small, appropriate, and large as discussed in Section 6.3. This was to ensure that the measurement represented different profiling scenarios. The hardware set-up and benchmarks were described previously in Section 3.1.1 and Section 5.1 respectively. The parameters and granularity setting of the benchmarks are summarised in Table 5.5.

From the eventlogs we present the overhead of HdpHProf tracing in the form of ratio of HdpH trace events to the GHC trace events. The reported figures are the median of 5 executions with standard deviation around the median [93]. Data was collected by measuring the total number of trace events in the eventlog then separating the HdpH trace events from the GHC trace events. We calculate the ratio of GHC and HdpH trace events as follows:

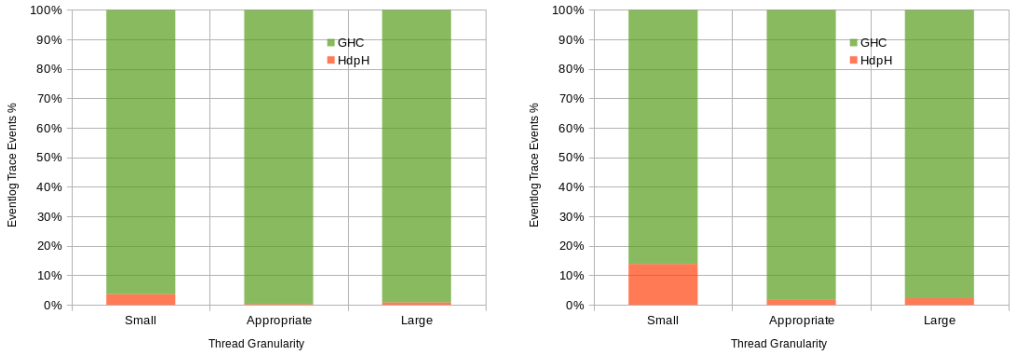
$$Eventlog_{TraceEvents} = GHC_{TraceEvents} + HdpH_{TraceEvents} \tag{5.2}$$

$$HdpH_{Ratio} = \frac{HdpH_{TraceEvents}}{Eventlog_{TraceEvents}} * 100 \tag{5.3}$$

HdpH Tracing Discussion. Table 5.5 shows a a summary of the HdpH tracing overhead experiments. It presents the benchmarks parameters and the granularity settings. Also, it summarises the percentages of HdpH and GHC trace events recorded into the eventlogs. HdpHProf trace events ranges between 0.26% and 1.67% where thread granularity is appropriate. However, we found that small and large granularity



(a) Mandelbrot (b) Summatory Liouville



(c) Fibonacci (d) SumEuler

Figure 5.24: Stacked view GHC vs HdpH Trace Events.

can increase tracing overhead more than appropriate thread granularity. For example, with small thread granularity HdpHProf trace events overheads are high, and at most 13.85 for the SumEuler benchmark.

Benchmarks			Eventlog Trace Events		Standard Deviation (± STDEV)
			GHC%	HdpH%	
Mandelbrot X 4096 Y 4096 D 1024	Thread Granularity (Threshold)	Small (0)	99.14	0.86	0.037
		Appropriate (4)	99.65	0.35	0.163
		Large (10)	99.53	0.47	0.113
Liouville 10000000	Thread Granularity (Chunk Size)	Small (10000)	90.18	9.82	0.040
		Appropriate (100000)	99.06	0.94	0.050
		Large (300000)	98.72	1.28	0.247
Fibonacci 45	Thread Granularity (Threshold)	Small (30)	96.37	3.63	0.048
		Appropriate (35)	99.74	0.26	0.035
		Large (40)	99.18	0.82	0.161
SumEuler [10000-42000]	Thread Granularity (Chunk Size)	Small (1)	86.15	13.85	0.076
		Appropriate (200)	98.33	1.67	0.069
		Large (800)	97.75	2.25	0.094

Table 5.5: Eventlog, GHC vs HdpH Trace Events.

5.7 Summary

This chapter demonstrates that a correct and valid profiler can be constructed using the host language tools to profile a distributed-memory parallel DSL. We validated HdpHProf for functional correctness and profiling performance. We validated HdpHProf's code instrumentation, time synchronisation, and trace file merge. In addition, we validated the functional correctness of the Spark Pool Contention Analysis and the Registry Contention Analysis tools using both hand crafted and real trace files fragments (Section 5.2). Moreover, we have shown that HdpHProf can profile long running programs and programs running on relatively large scale architectures: up to 32 Beowulf cluster nodes and 192 cores (Section 5.3). We also characterised and compared HdpHProf overheads in terms of profiling data size (Section 5.4) and profiling execution runtime overhead (Section 5.5), based on our study in Chapter 3. Finally, we measured the ratio of HdpH trace events in the GHC-PPS eventlog (Section 5.6).

Chapter 6

Evaluating HdpHProf for Applications

This chapter investigates how effectively HdpHProf can be used to profile and tune the behaviour of HdpH applications. We investigate whether HdpHProf can help identify performance problems (Section 6.2). Moreover, we show how HdpHProf can be used to tune the thread granularity of HdpH applications (Section 6.3). Also, we specify how to control shared-memory and distributed-memory thread granularity with two thresholds (Section 6.4).

6.1 Experimental Methodology

This section presents the methodology of collecting the experimental data for this chapter. We present the experimental set-up and experiment measurements.

6.1.1 Experimental Set-up

HdpHProf is used to measure performance of HdpH benchmarks on a Beowulf cluster comprising of 32 nodes with 8 cores each. The hardware set-up and benchmarks were described previously in Section 3.1.1 and Section 5.1 respectively.

6.1.2 Experiments

The investigation regarding the evaluation of HdpHProf for profiling application is divided into two parts. First, HdpHProf will be evaluated by illustrating how it can identify performance problems. Second, HdpHProf will be evaluated by examining how

it can help tuning thread granularity in HdpH applications. For this purpose we used different HdpH benchmarks, Mandelbrot, Liouville, Fibonacci, NBody, Queens, and SumEuler. Performance evaluation was based on visual inspection of profiles and analysis of speed up, efficiency and average task duration. The visual performance profiles were generated by the standard Haskell time-line threads activity browser ThreadScope [134]. The performance graph of ThreadScope is divided into two parts. First, the overall activity graph which shows execution time on the x-axis and the overall activity of multiple Haskell Execution Contexts (HECs) on the y-axis. Second, is a list of per-core HECs status used in the evaluation over the x-axis using colours, i.e. green (active), orange (garbage collecting), and white (idle). In addition, the total run time of the application was used to measure the speed up, efficiency, and average task duration. Speed up presented in this evaluation is the relative speed up and calculated as follows [92]:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} \tag{6.1}$$

Where $S(P)$ is speed up, $T_{total}(1)$ is runtime of the parallel application executed on one core and $T_{total}(P)$ is the run time of the application executed on P cores. Efficiency presented in this evaluation is calculated as follows [92]:

$$E(P) = \frac{S(P)}{P} \tag{6.2}$$

Average task duration shown in this evaluation is calculated as follows:

$$ATD(P) = \frac{T_{total}(1)}{NT_{total}(P)} \tag{6.3}$$

Where $ATD(P)$ is average task duration executed on P cores, $NT_{total}(P)$ is the total number of tasks executed on P cores.

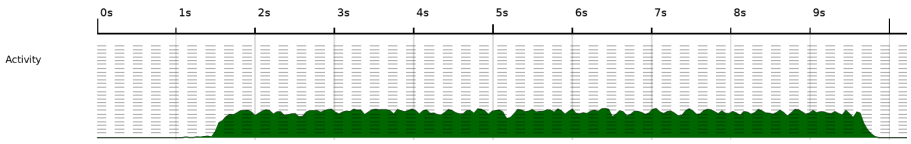
Benchmarks are mostly run on 4 cluster nodes with only 6 cores each to reduce variability in the results (total 24 cores), following common practice [88]. The limit of 4 nodes is used to reduce the size of the graphical profile so it can fit on one page when printed. In addition, we considered 4 nodes to be enough to illustrate the typical performance issues of applications. On the contrary, we had some cases where we had to illustrate some performance issues that need to be run on a large number of nodes, e.g. Section 6.2.4. For these cases the benchmarks were run on 16 cluster nodes with 6 cores (total cores 96).

6.2 Identification of Performance Problems

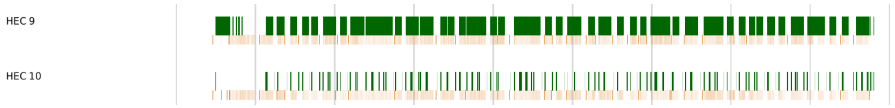
This section investigates how effective HdpHProf is at identifying common parallel performance issues.

6.2.1 Excessively Small Thread Granularity

We started investigating small thread granularity using the Liouville benchmark. HdpH Liouville is a flat data parallel algorithm which gives the user control over thread granularity by a chunk size argument. We profiled the Liouville benchmark with a chunk size deliberately set to produce small thread granularity, to see how HdpHProf identifies this performance problem.



(a) Overall Activity



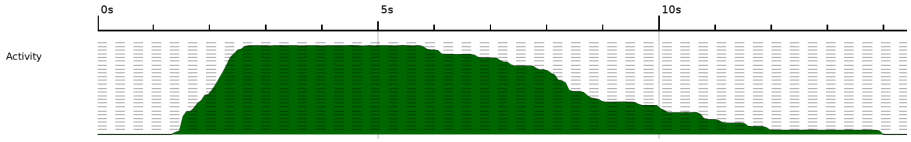
(b) Some HECs

Figure 6.1: Summatory Liouville 10,000,000 Chunk Size 10,000 (24 Cores).

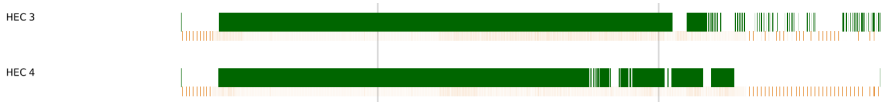
Figure 6.1 presents a performance profile of the Liouville benchmark on 4 nodes with 6 cores. From the overall activity bar (Figure 6.1(a)) it can be seen that the parallel machine was under-utilised. This meant that the program was not performing well and there was a performance problem. For instance, Figure 6.1(b) demonstrates how the majority of the HECs behaved during the program execution. The HECs bars show that there were tiny threads getting active and inactive in small periods of time during execution. If the bar is green this means there is an active thread running; whereas, white means the HEC is idle. What the profiler tells us here is that the task granularity was too small for the program to perform parallel tasks efficiently. Moreover, the run time shows that the application has only a speed up of 6.4 with efficiency 0.26. Also, there are 10,000 tasks with average task duration of 5.67ms. As thread granularity is so small the coordination aspects of the parallel algorithm outweigh the computation of the real problem. This experiment illustrates that HdpHProf can provide suitable information to help programmers identify excessively small thread granularity.

6.2.2 Excessively Large Thread Granularity

We investigated the information HdpHProf provides for programmers with very large thread granularity using the SumEuler benchmark. The HdpH SumEuler benchmark has a flat data parallel algorithm and thread granularity is set by the user before execution. The user can select thread granularity by passing a chunk size argument to the program. Therefore, we profiled the SumEuler benchmark with a deliberately large chunk size to show how HdpHProf visualises the behaviour of HdpH applications with large grain thread granularity.



(a) Overall Activity



(b) Some HECs

Figure 6.2: SumEuler [10000-42000] Chunk Size 800 (24 Cores).

Figure 6.2 shows a performance profile for HdpH SumEuler application. The application was executed with chunk size set to produce large thread granularity. From the figure it can be seen that the activity bar (Figure 6.2(a)) indicates that the parallel machine was utilised efficiently at the first part of execution up to about 6s. After that the activity bar starts to drop down gradually till the machine becomes significantly under-utilised at the end of the program execution. To see exactly what caused this problem we can see the HECs bars. For example, Figure 6.2(b) shows the behaviour of 2 representative HECs; the majority of the HECs used for evaluation in this execution show similar behaviour. From this we can see that each HEC is active at the beginning of evaluation. However, after a certain point this starts to change and HEC bars start showing an idle state, which means that they started running out of work. What the profile shows here is clear sign of the effect of large thread granularity. In addition, the run time shows that the application had a speed up of 12.8 with efficiency 0.53. Also, there are only 41 tasks with average task duration of 4.16s. This phenomenon is called *starvation* which is caused by the large grain thread granularity where each PE has enough work to perform efficiently at the beginning of the evaluation. However, it becomes very hard for HECs to find more work to do toward the end of evaluation.

This experiment illustrates how HdpHProf can show the performance behaviour of HdpH applications with large thread granularity.

6.2.3 Sequentialisation bottleneck

This section investigates how HdpHProf can profile performance behaviour of an HdpH application that has sequentialisation phases during parallel evaluation. For this demonstration we used an NBody HdpH benchmark. The HdpH NBody benchmark runs a simulation of X bodies for Y time steps and uses chunk size Z for parallelising each time step. In other words, the benchmark is phases of data parallel problems where each time step is data parallel. However, time steps sequentially follow each other, which requires sequential synchronisation. The benchmark arguments, including the chunk size, are supplied to the program by the user before execution. We used HdpHProf to profile the benchmark with different combinations of arguments to see how it behaved on our parallel machine and what its performance profile looked like.

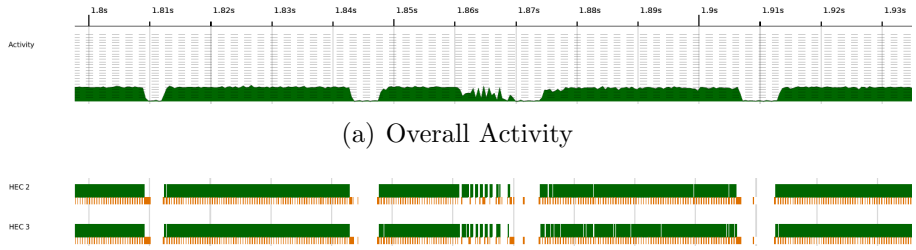
Figure 6.3 shows a performance profile of the NBody benchmark: the number of bodies is 2048, steps are 8, and chunk size is 64. We profiled the application with different inputs. We found that generally all profiles look similar to each other except the execution time increased as the number of steps increased and the middle section of the profile took longer in time. We can see that in each parallel machine one HEC is active before the other HECs until up about 1.5s where HdpH RTS starts, i.e. HEC 0, HEC 6, HEC 12, and HEC 20. This is an artefact of the benchmark; random input data is generated on every node –though it is needed only on the root node– before HdpH even starts. Then other HECs in each machine start evaluating. After that, all the HECs stop together at about 2.2s, then the main node (first one from the top) is the only one who has work until the program terminates. Artefact of the benchmark: main node computes final states.

To investigate this phenomenon further we zoomed in the profile (Figure 6.4) to see what happens during execution in the middle part of the profile. As the figure shows on both the activity bar (Figure 6.4(a)) and the HECs bars (Figure 6.4(b)), there were points in time where the activity bar and all the HECs showed no acting at all. These gaps are the sequential synchronisation phases of the parallel application where all HECs need to stop to synchronise.

Using HdpHProf can help the programmers with such applications by considering the following. Shortening the initialisation and finalisation phases of the parallel

algorithm, e.g. the profile shows that the parallel application spent significant time of the total execution run time on these phases. Furthermore, care must be taken with how often the program should synchronise as the profile shows that during these phases the parallelism was lost. In addition, the parallelism phases should utilise the parallel machine efficiently. Here we have profiled the NBody benchmark to show how sequentialisation bottlenecks can be spotted using HdpHProf.

Figure 6.3: nBody 2048 Steps 8 Chunk Size 64 (24 Cores).



(a) Overall Activity

(b) Some HECs

Figure 6.4: Zoomed in nBody 2048 Steps 8 Chunk Size 64 (24 Cores).

6.2.4 Insufficient Work for Machine Architecture

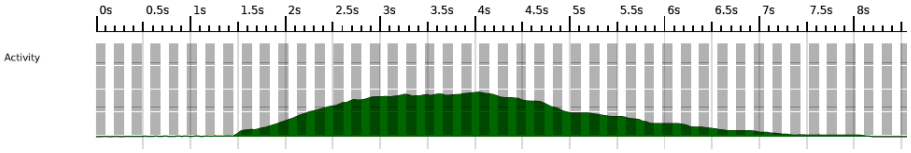
This section presents profiles of HdpH applications where the parallel machine runs with insufficient work. For this experiments we used two benchmarks for demonstration, i.e. SumEuler and Liouville. Both benchmarks are flat data parallel and take as arguments the length of the input interval along with the chunk size for thread granularity. First, we tuned the applications to run on 4 nodes, 24 total cores efficiently with good speed up (Section 6.3). This was because we wanted to be sure that we had the right thread granularity for the problem size. Then we ran the applications with the same inputs on a larger number of cores i.e 16 nodes and 96 total cores; to illustrate how HdpHProf can reveal insufficient work.

SumEuler. The SumEuler application was executed with these inputs [10000-42000] and chunk size of 400. With these settings the application produced 80 tasks with average task duration of 2s. 80 tasks is not a sufficient number to be evaluated on 96 cores. For this run the execution time showed a good speed of 23.4. However, the efficiency with this number of cores was 0.24 which indicates that there was a performance problem.

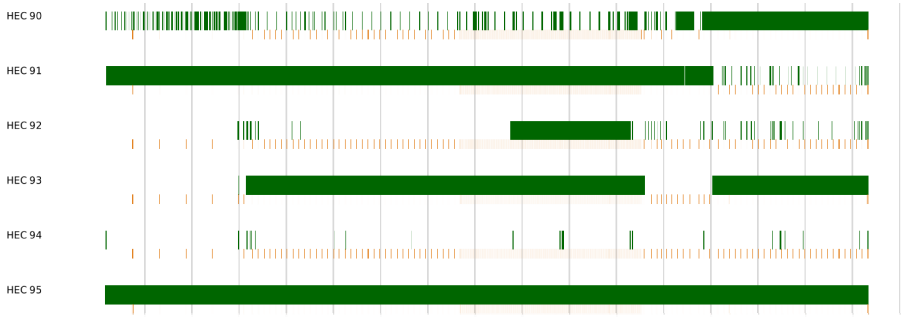
Figure 6.5 demonstrates the profile for SumEuler where the problem size is insufficient for the parallel architecture. As the activity bar shows (Figure 6.5(a)) performance increased incrementally until it reached certain level of utilising –less than half the number of HECs– of the parallel machine. The activity bar remained steady from about 2.2s to 3s. This indicates that the parallel program was not able to utilise the parallel machine efficiently. After that, the activity bar decreased gradually up to the end of the program.

To explore this problem further we examine the per-core HEC bars from the profile (Figure 6.5(b)). This figure shows only 6 HECs of one node out of 16 nodes. This pattern of behaviour is representative for the HECs in other nodes involved in

the evaluation of the program. There are about two to three HECs which have enough work to keep busy most of the time. On the contrary, the rest of the HECs show that they spent significant time of their state as idle, which means they had no work to do. This means that the work was distributed on all machines. However, the work that each machine received was not sufficient to keep all of machine's HECs busy. Such performance may be considered a waste of computing resources. Therefore, the efficiency of the parallel application dropped down to 0.24.



(a) Overall Activity



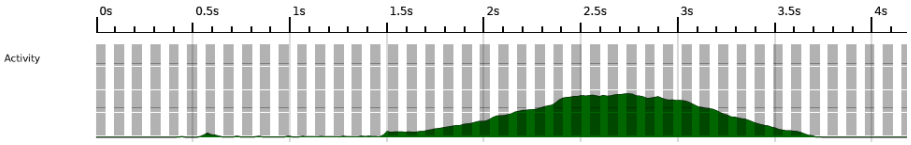
(b) Some HECs

Figure 6.5: SumEuler [10000-42000] Chunk Size 400 (96 Cores).

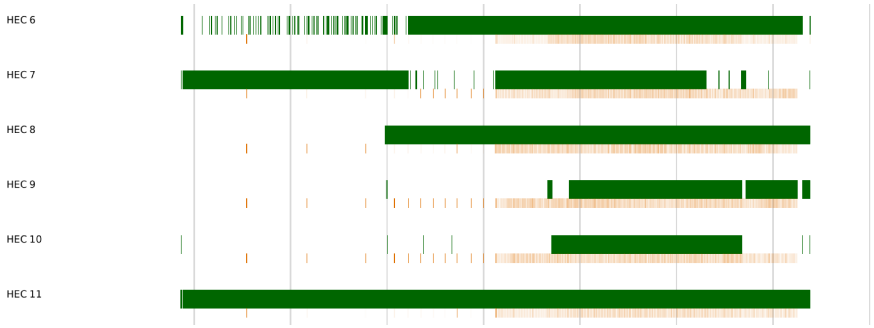
Liouville. The benchmark was executed with these inputs, Liouville 10,000,000 and chunk size 100,000, and it produced 100 tasks with average task duration of 0.56s. This number of tasks is too small to be executed on 96 cores of the parallel machine. As a result, despite the fact that the program has a speed up of 15, the efficiency of this parallelism is 0.16, which is too low. Figure 6.6 presents the performance profile of this execution. We can see that the activity bar (Figure 6.6(a)) shows that the parallel program is trying to utilise the parallel machine gradually. However, the utilisation reaches its peak at about half the number of available HECs. To see what caused this problem we have to see how the HECs on the parallel machine behaved. Figure 6.6(b) is an extract of the profile for this execution that only shows 6 representative HECs on a single node of the parallel machine (16 nodes 96, total cores). By examining the HEC bars from the profile it can be seen that there are two to three HECs that were able to keep busy most of the evaluation time. In contrast, the rest of the HECs show that they spent much of their time in an idle state. This means that the work

was distributed among all nodes. However, the work that each node received was not enough to keep all of its cores busy, leading to the low efficiency of the program.

Insufficient Work for Machine Architecture Discussion. From the two experiments profiling SumEuler and Liouville with insufficient work to be done in parallel we were able to see how such problems can be seen in HdpH profiles. Both experiments showed similar results where the parallel machine was under-utilised, which can be seen form the activity bars from the profiles. In addition, most of the nodes of the parallel machine show that only two or three of its HECs were busy for most of the evaluation time. The other HECs spent a significant amount of their time in an idle state.



(a) Overall Activity



(b) Some HECs

Figure 6.6: Summatory Liouville 10,000,000 Chunk Size 100,000 (96 Cores).

6.2.5 Combination of factors

In this section we demonstrate profiling the Queens, a nested data parallel benchmark. The benchmark tries to solve a chessboard problem by placing N queens on an NxN board such that no queen can attack any other queen. The program takes as arguments the number of queens and a chunk size for thread granularity. We will show one representative profile because ranging input parameters does not significantly alter the performance.

Figure 6.7 shows the performance profile of HdpH Queens application. The activity bar shows the application did not perform well and the parallel machine was under-utilised. Also, the application run time shows a very low speed up of 2 with

an efficiency of 0.08. From the HECs bars we noticed some points. One of the nodes behaved differently from the other nodes in the parallel machine (node 4 last 6 HECs). The other three nodes show similar behaviour with some of the HECs active most of their time. In contrast, some HECs show that they had tiny threads active and inactive over a small period of time which is an indication of small thread granularity, e.g. HEC 3, HEC 12, HEC 13, and HEC 16. Also, some HECs were in idle state most of the time which means that they had no work to do, e.g. HEC5, HEC 8, and HEC 11. The profile apparently shows irregular parallelism with small thread granularity caused by the complex coordination specified by the nested parallelism.

6.3 Tuning Thread Granularity

This section will investigate how effectively HdpHProf can be used to determine an appropriate thread granularity in HdpH applications. The section is divided into two parts. First, we will discuss tuning thread granularity for HdpH applications with flat data parallel algorithms. Then we will present how to tune thread granularity for HdpH applications with divide and conquer algorithms.

6.3.1 Data Parallel Chunk Size

In this section we will present tuning thread granularity for an HdpH flat data parallel benchmark, Liouville. Thread granularity is controlled by a chunk size argument which is passed by the user to the application. We will illustrate how HdpHProf can be used to help identify the best chunk size for parallel performance.

To determine the right thread granularity the application must be executed and profiled to solve a required problem with a preliminary chunk size. Then after examining the profile the user can decide if the thread granularity is too small, appropriate or too large.

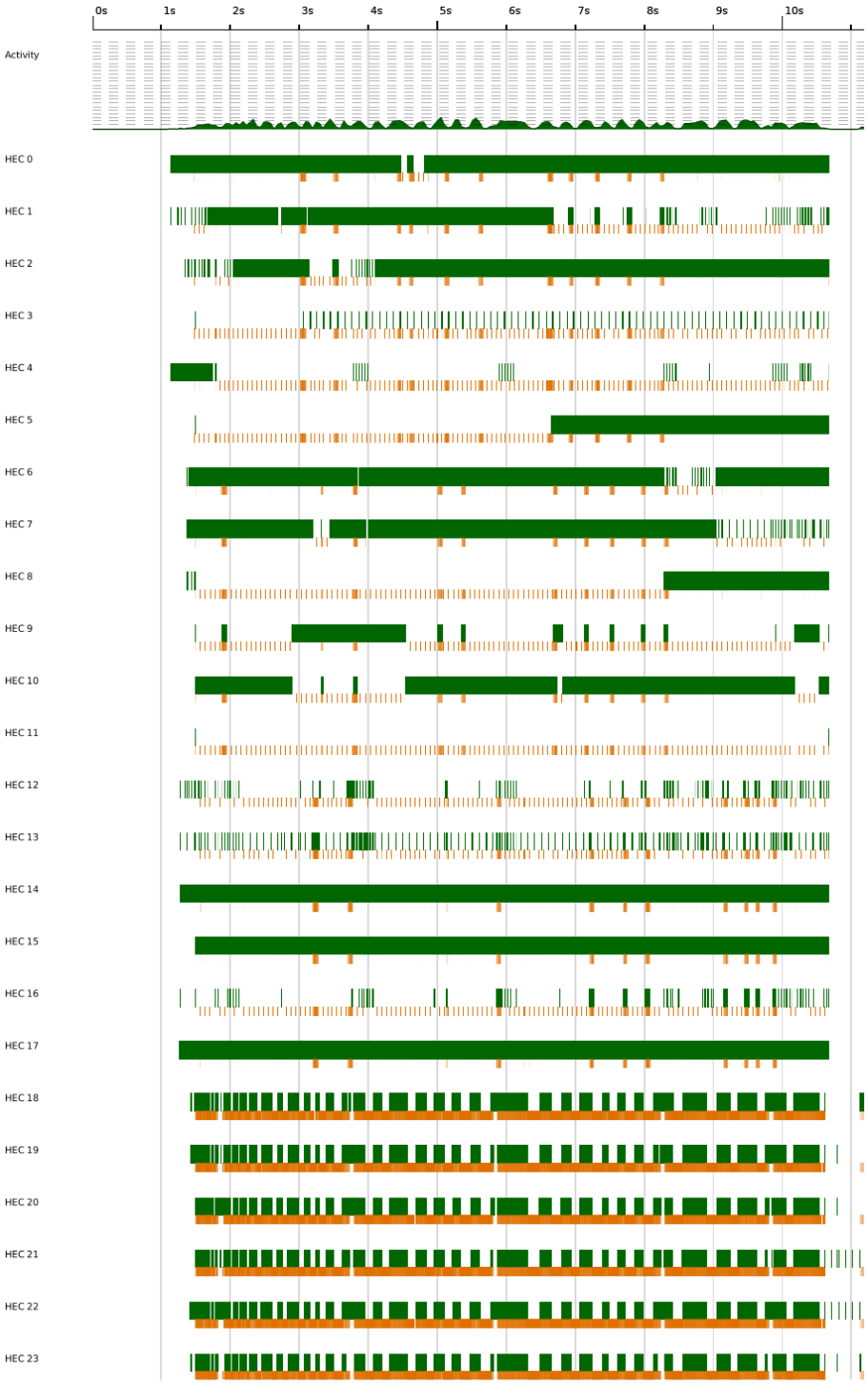
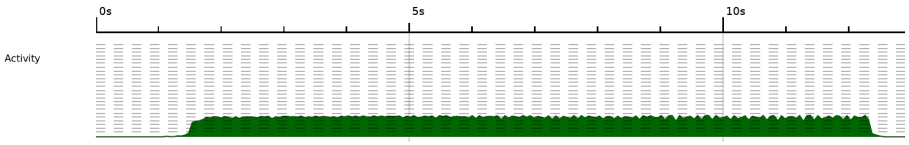


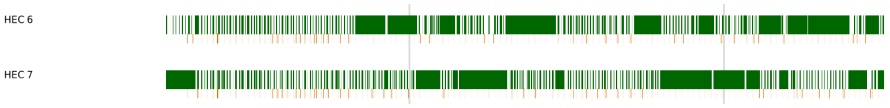
Figure 6.7: Queens 13 Chunk Size 1600 (24 cores).

Chunk Size too Small. Figure 6.8 presents a performance profile of Liouville 10,000,000 with chunk size set to 1000. As the activity bar shows (Figure 6.8(a)) the application did not perform well. It shows only about one fourth of the parallel machine was utilised during evaluation. The application produced 10,000 tasks with an average task duration of 5.67ms. In addition, the run time shows that the application had only a speed up of 4.9 with efficiency 0.2. By examining the HECs bars from the profile

(Figure 6.8(b)) it can be seen that there are tiny threads becoming active and inactive in small periods of time. This is an indication of small grain thread granularity which we discussed earlier in Section 6.2.1. From this performance profile we can see that the chunk size we set was too small for this problem size. As a result, thread granularity needs to be increase by passing a larger chunk size for the application.



(a) Overall Activity

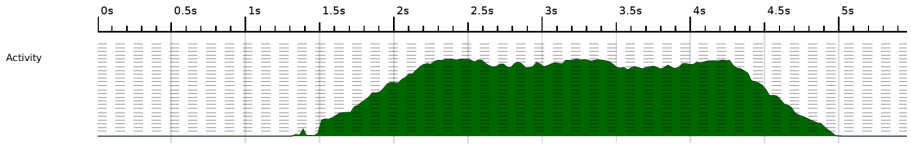


(b) Some HECs

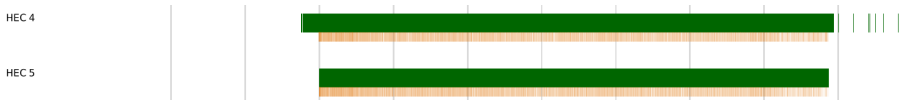
Figure 6.8: Summatory Liouville 10,000,000 Chunk Size 1,000 (24 Cores).

Appropriate Chunk Size. After seeing that a chunk size of 1000 generated small thread granularity with Liouville 10,000,000 we increased the chunk size to 10,000. Figure 6.9 shows the performance profile of Liouville 10,000,000 with the new chunk size. From the activity bar we can see that (Figure 6.9(a)) the performance of the application changed significantly from the previous run when the chunk size was 1000. The application produced 100 tasks with an average task duration of 0.56s. Also, the application run time showed better speed up of 13.9 with efficiency 0.57. Moreover, by examining the HECs bars (Figure 6.9(b)) we can see that most of them were busy working from the beginning of evaluation until the program terminated. This is an indication that the new chunk size gives a appropriate thread granularity; which led to a balance between coordination aspects of the parallel algorithm and its computation aspects. To see if this was best granularity for this problem size we increased the chunk size to see if the performance would improve or get worse because of large thread granularity.

Chunk Size too Large. We have seen that Liouville performed well with a chunk size of 100000. To see if that was the best thread granularity that we could get we needed to run the application with a larger chunk size and see the performance profile. Figure 6.10 demonstrates Liouville 10,000,000 performance profile with larger chunk size of 300,000. As can be seen from the activity bar (Figure 6.10(a)), the performance



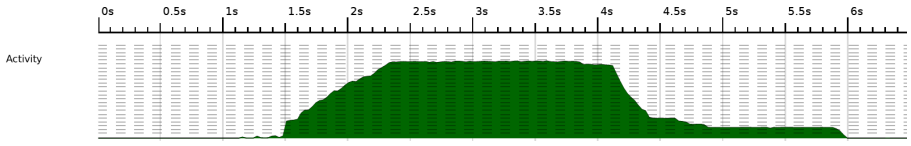
(a) Overall Activity



(b) Some HECs

Figure 6.9: Summatory Liouville 10,000,000 Chunk Size 100,000 (24 Cores).

of the application dropped down noticeably after 4.1s. The application produced 34 tasks with average task duration of 1.6s. The run time shows that the speed up also dropped down to 10.6 with efficiency of 0.44. What the activity bar shows is that after 4.1s the parallel machine started running out of work sharply. Also, the HECs bars (Figure 6.10(b)) illustrate more clearly what is happening. A large number of HECs from the full profile show similar behaviour to HEC 14 from the figure. HEC 14 shows that it was able to evaluate and had work at the beginning of the program. However, it starved and ran out of work after about 4.2s. This is an effect of large thread granularity as we discussed previously in Section 6.2.2.



(a) Overall Activity

(b) Some HECs

Figure 6.10: Summatory Liouville 10,000,000 Chunk Size 300,000 (24 Cores).

Data Parallel Chunk Size Discussion. Overall, in this section we have demonstrated how to tune thread granularity for a flat data parallel HdpH application. The Liouville benchmark was used for demonstration. To tune thread granularity for flat data parallel application we had to profile the application with different chunk sizes for the same problem size. Chunk size is normally a large number which is provided by the user to the application as a cut-off for parallelism. By examining the performance profiles it was possible to identify whether thread granularity was fine grain,

appropriate grain or coarse grain.

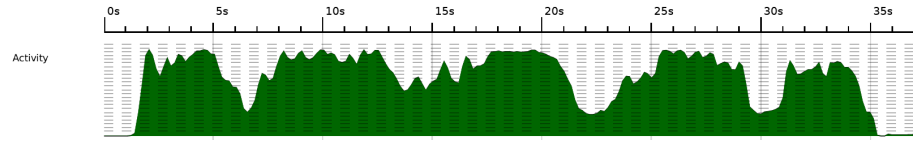
6.3.2 Divide and Conquer Threshold

This section demonstrates tuning thread granularity for HdpH applications with divide and conquer parallel algorithms. For this purpose we used the Mandelbrot benchmark. In divide and conquer programs thread granularity is controlled by a threshold number that is used as a cut-off for parallel evaluation. Providing the application with the right threshold is very important for parallel performance.

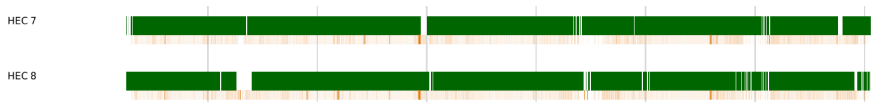
Mandelbrot. The Mandelbrot HdpH benchmark is a divide and conquer parallel application. It takes four arguments, X, Y, Depth, and threshold. The threshold is used by the application to determine when to start to evaluate sequentially. In other word, it is used to control thread granularity. To demonstrate how to tune thread granularity for Mandelbrot we first set the problem size that we would like the application to solve, which is Mandelbrot X=4096 Y=4096 Depth=1024. Then we profiled the application with different thresholds.

Small Thread Granularity. Figure 6.11 presents the performance profile of Mandelbrot with the threshold set to 0. As the activity bar illustrates (Figure 6.11(a)) the application had inconsistent behaviour where the green jagged activity bar went up and down repetitively. Also, by looking at the HECs bars (Figure 6.11(b)) we can see that there are some parts show there were tiny threads becoming active and inactive. However, because the application ran for such a long time these tiny threads are not very clear in the picture. The application generated 4095 tasks with an average task duration of 0.13s. In addition, the application run time shows that it had a speed up of 15.7 with 0.6 of efficiency. We suspect the application behaved in this manner because of the small size thread granularity. To see if this is true we had to increase the threshold then examine the performance profile again.

Appropriate Thread Granularity. Figure 6.12 presents the performance profile of Mandelbrot after increasing the threshold to 4. As can be seen from the activity bar (Figure 6.12(a)) the performance was better and the application behaved more consistently than before. Moreover, most of the HECs bars show they were busy during the program execution, e.g. see Figure 6.12(b). Also, the number of tasks dropped to 1023 and the average task duration increased to 0.53s. The application run



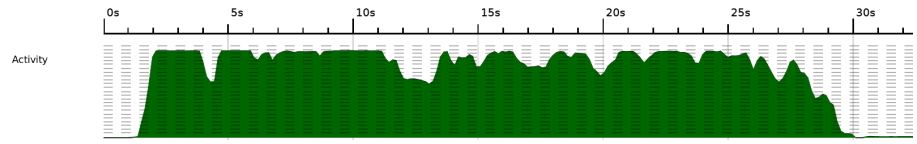
(a) Overall Activity



(b) Some HECs

Figure 6.11: Mandelbrot X=4096 Y=4096 Depth=1024 Threshold 0 (24 Cores).

time shows a speed up of 18.6 with an efficiency of 0.77, which is better than the previous run. From this we can see that increasing the threshold gave appropriate thread granularity, which improved the performance of the parallel application. To see if this is the best thread granularity for this problem size we need to profile the application with a bigger threshold and examine its performance again.

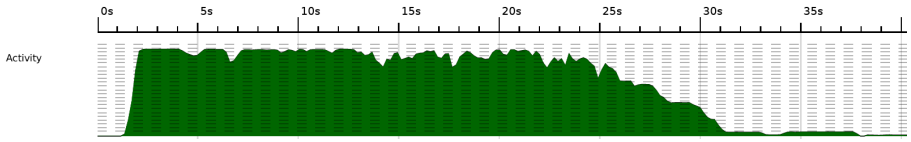


(a) Overall Activity

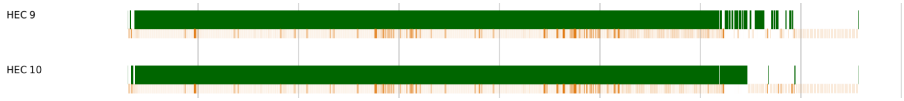
(b) Some HECs

Figure 6.12: Mandelbrot X=4096 Y=4096 Depth=1024 Threshold 4 (24 Cores).

Large Thread Granularity. Figure 6.13 demonstrates performance profile of Mandelbrot after increasing the threshold to 10. The activity bar (Figure 6.13(a)) shows that the application performed well up to 24s. However, after that the activity bar started to drop down dramatically until it reached a very low level, from 31s to 40s. The application generated 511 tasks with average task duration of 1.06s. Also, the run time shows that the speed up dropped to 14.7 with an efficiency of 0.61. By examining some of the HECs bar (Figure 6.13(b)) it can be seen that they were active from the beginning of the program and when they reached 30s they ran out of work. This is a sign of large thread granularity as discussed earlier in Section 6.2.2. As a result, the large thread granularity changed the application performance and speed up dropped down to 14 with an efficiency of 0.6.



(a) Overall Activity



(b) Some HECs

Figure 6.13: Mandelbrot X=4096 Y=4096 Depth=1024 Threshold 10 (24 Cores).

6.4 Co-location and Sequential Granularity

Fibonacci. The Fibonacci benchmark has two cut-off options, i.e. two thresholds to control task granularity. First, is a co-location threshold which is used to determine the granularity of tasks that always execute on the same node. In other words, when reaching this threshold, generated tasks are only distributed to cores of the current node, and they are not allowed to spawn tasks on other nodes. This means that it controls the size of the computation so that it cannot migrate to other nodes. As a result, the threshold controls not just one thread granularity; instead, it controls the granularity for a group of threads, none of which can migrate. Therefore, we call this a co-location thread granularity. Second, is a sequential threshold which is used to determine thread granularity for sequential evaluation.

As a consequence, Fibonacci can be executed with 9 different combinations of granularity settings. We are interested in 8 cases; the case which we are eliminating is where sequential granularity is larger than co-location granularity. This is because the sequential threshold will override the co-location threshold if it is larger. Table 6.1 illustrates these granularity settings which will be used for profiling and demonstration in this section. We reached these settings through multiple experiments until we found the settings that presented small, appropriate, and large granularity.

This section demonstrates how profiling the Fibonacci application can help understand how it behaves with different granularity settings, and how to identify the appropriate granularity for best performance. The section is divided into four sections based on the similarities between results of granularity settings. Each shaded colour of the table represents one of these groups.

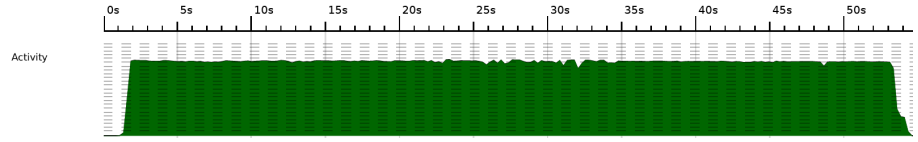
		Sequential Granularity		
		Small	Appropriate	Large
Co-location Granularity	Small	**Co-loc. 30 Seq. 5**	**Co-loc. 30 Seq. 19**	**Co-loc. 30 (Seq. 33)**
	Appropriate	**Co-loc. 35 Seq. 5**	**Co-loc. 35 Seq. 19**	**Co-loc. 35 Seq. 33**
	Large	**Co-loc. 40 Seq. 5**	**Co-loc. 40 Seq. 19**	**Co-loc. 40 Seq. 33**

Table 6.1: Fibonacci Co-loc. and Seq. thresholds settings.

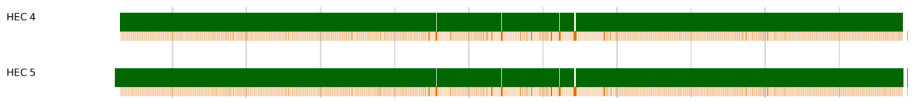
6.4.1 High Co-ordination Overheads

High co-ordination overhead happens when a small sequential threshold is used. The orange shaded combinations of granularity settings in Table 6.1 cause this performance problem. Figures 6.14, 6.15, and 6.16 show the performance profiles where the thread granularity settings introduced excessive overheads creating parallel tasks. From the overall activity of the figures, e.g. Figure 6.14(a), it can be seen that the applications ran for significantly long time. It generated 1596 tasks with an average task duration of 42.9ms. Also, the application showed poor speed up and efficiency is very low, less than 0.05. As the applications ran for between 53s and 74s, it is really hard to see clearly what was happing in the HECs without zooming in. As a result, we zoomed in –about 1ms– to the profiles to see how the HECs behaved with such settings, e.g. see Figure 6.15(b). The figure shows that there were threads becoming active and inactive repeatedly, with many small garbage collection periods in between. We observed that when sequential granularity was too low, the execution time went up. This is because there are high overheads of creating threads and running them for many tiny tasks.

Moreover, when large co-location granularity is combined with small sequential granularity the performance gets worse, e.g. see Figure 6.16. The profile shows that the application ran for a longer time. This was due to the large co-location granularity; the parallel machine started to starve early and less nodes had to complete the computation. To sum up; small sequential granularity introduces high co-ordination overheads and decreases the performance of the parallel application. Moreover, when used with large co-location granularity, this will cause starvation and further degrade the performance.

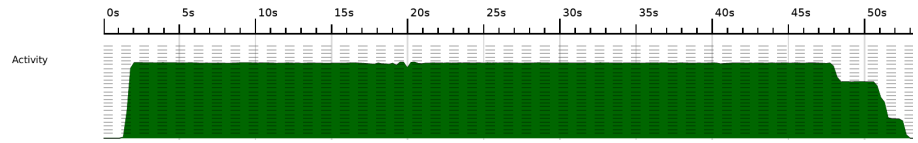


(a) Overall Activity

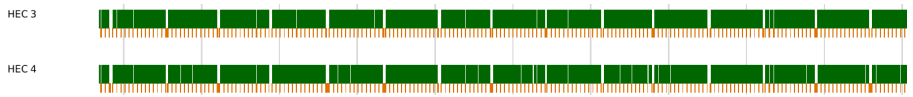


(b) Some HECs

Figure 6.14: Fibonacci 45 Co-loc. 30 Seq. 5 (Small, Small) (24 Cores).

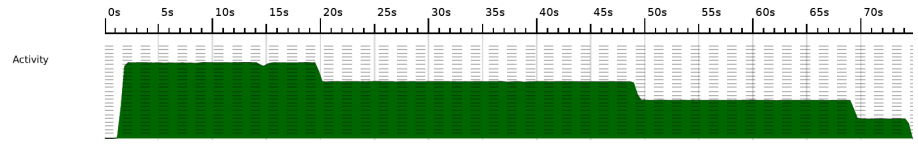


(a) Overall Activity

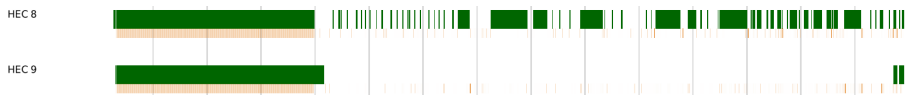


(b) Some HECs (Zoomed 1ms)

Figure 6.15: Fibonacci 45 Co-loc. 35 Seq. 5 (Appr., Small) (24 Cores).



(a) Overall Activity



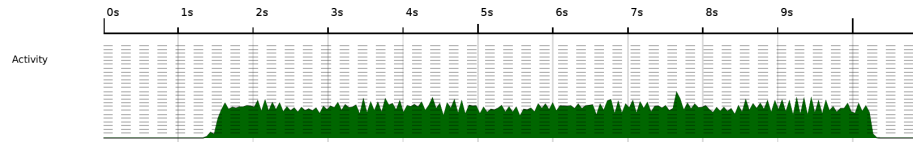
(b) Some HECs

Figure 6.16: Fibonacci 45 Co-loc. 40 Seq. 5 (Large, Small) (24 Cores).

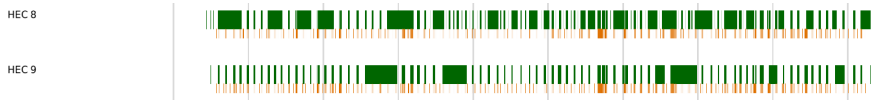
6.4.2 Fine Grained Tasks

Fine grained task parallelism happens when granularity settings are as in the yellow shaded combination of thresholds from Table 6.1. Figure 6.17 shows the performance profile where granularity options were set to be fine grained for co-location granularity and appropriate for sequential granularity. As it can be seen from the overall activity (Figure 6.17(a)) the application did not perform well with these granularity settings. It

generated 1596 tasks with an average task duration of 42.9ms. Also,the runtime shows that the application had a speed up of 7.3 with a low efficiency of 0.3. To investigate further what caused this problem we can examine the HECs activities (Figure 6.17(b)). The figure shows that HECs were not performing optimally as they were idling repeatedly. As discussed earlier in Section 6.2.1, this is a sign of small thread granularity. Consequently, the small co-location granularity caused the machine to coordinate too much.



(a) Overall Activity

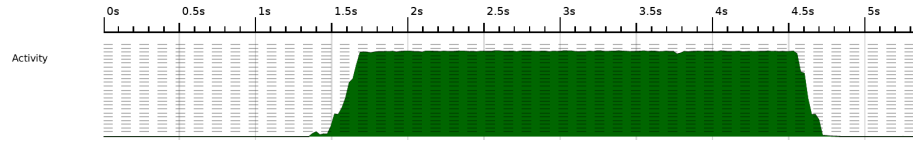


(b) Some HECs

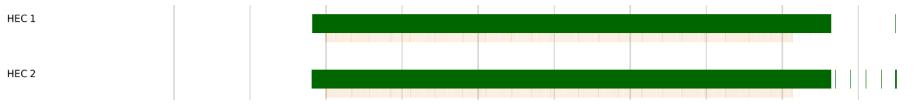
Figure 6.17: Fibonacci 45 Co-loc. 30 Seq. 19 (Small, Appr.) (24 Cores).

6.4.3 Good Performance

Good performance can be produced with appropriate granularity. The result in this section shows the green shaded combination of granularity settings in Table 6.1. Figure 6.18 presents the performance profile of the application with task granularity options set to appropriate co-location and appropriate sequential. The overall activity bar (Figure 6.18(a)) shows that the application performed very well. In addition, the number of tasks decreased to 143, and the average task duration increased to 0.47s. The runtime shows a speed up of 17.6, with 0.73 of efficiency. By examining the HECs from the profile (Figure 6.18(b)) it can be seen that there is no sign of a problem and they evaluated continuously most of the time with no visible gaps or dense garbage collection. This means that the granularity settings were right as the performance profile showed significantly better results than the previous experiments. This is a clear indication that tuning granularity, i.e. co-location and sequential to appropriate values is crucial to a good performance.



(a) Overall Activity

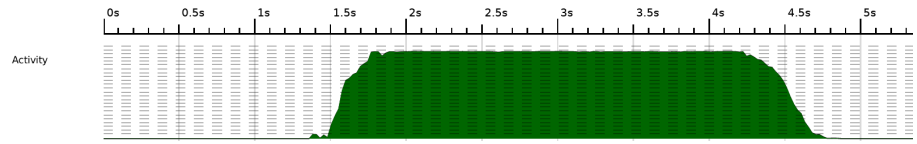


(b) Some HECs

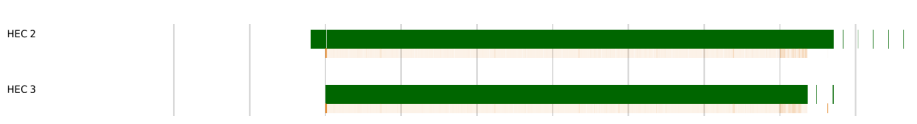
Figure 6.18: Fibonacci 45 Co-loc. 35 Seq. 19 (Appr., Appr.) (24 Cores).

6.4.4 Coarse Grained Tasks

Large thresholds produce coarse grained task parallelism, this section presents the blue shaded cases of thread granularity settings Table 6.1. Figure 6.19 presents a performance profile where granularity settings were set to appropriate co-location and large sequential. From the overall activity (Figure 6.19(a)) it can be seen that the performance was good until before the end of the execution, when the overall activity started to drop gradually. This means that HECs inside each node started to run out of work due to the large sequential granularity. Moreover, the application generated 143 tasks with an average task duration of 0.47s. The runtime shows a speed up of 17, with 0.7 of efficiency. This did not seem to significantly impact on the performance as it all happened at the last moments of the program evaluation. However, for larger problems or longer runs it might have a more significant impact on the performance of the application. Therefore, this granularity setting is not recommended.



(a) Overall Activity

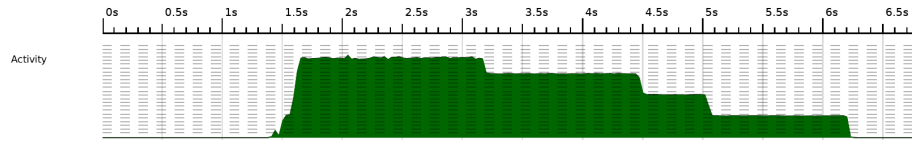


(b) Some HECs

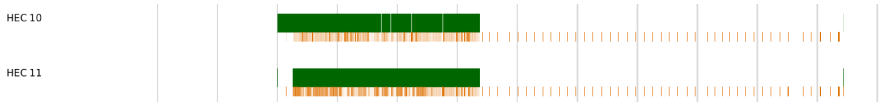
Figure 6.19: Fibonacci 45 Co-loc. 35 Seq. 33 (Appr., Large) (24 Cores).

Figure 6.20 shows the performance profile where granularity options were set to large co-location with appropriate sequential. The overall activity (Figure 6.20(a)) shows that the application performed well at the beginning of the evaluation. After

that, the performance started to drop sharply at certain points, e.g. 3.2s, 4.5s, and 5.1s. Moreover, the number of tasks decreased to 12 tasks with a high average task duration of 5.71s. The speed up dropped down to 12.87, with an efficiency of 0.53. Some HECs from the profile show that they ran out of work at about 3.2s of the evaluation time, then became idle until the program terminated (Figure 6.20(b)). This behaviour was repeated among other HECs in the profile; except some of the HECs ran out of work at different times. What the performance profile shows here is an indication of coarse grained task granularity, as presented earlier in Section 6.2.2. Consequently, the large co-location granularity affected the performance of the application, even though an appropriate sequential granularity was used.



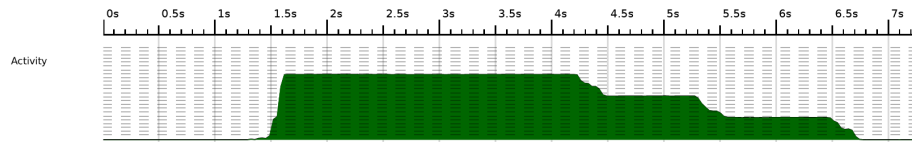
(a) Overall Activity



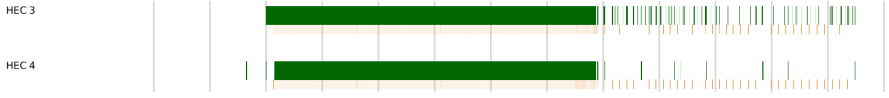
(b) Some HECs

Figure 6.20: Fibonacci 45 Co-loc. 40 Seq. 19 (Large, Appr.) (24 Cores).

Figure 6.21 shows the performance profile where granularity settings were set to large co-location with large sequential. As can be seen from the overall activity (Figure 6.21(a)) there were two stages of starvation. First, nodes of the parallel machine ran out of work, i.e. the nodes starved. Second, cores inside each node ran out of work, i.e. the cores starved. The application produced 12 tasks with an average task duration of 5.71s. The speed up dropped down to 11.95 with an efficiency of 0.49.



(a) Overall Activity



(b) Some HECs

Figure 6.21: Fibonacci 45 Co-loc. 40 Seq. 33 (Large, Large) (24 Cores).

6.4.5 Discussion

Overall, this section demonstrates using HdpHProf to understand the behaviour and tuning thread granularity for a divide and conquer HdpH application, i.e. Fibonacci, which has two thresholds as a mechanism to control thread granularity. We examined all possible scenarios by profiling the application behaviour with all combinations of granularity settings. We found that the performance of the application is very sensitive to these thresholds as changing one of them can change the performance drastically. In addition, our results show that the best performance is only achieved when both granularity settings are appropriate. Without HdpHProf it would be difficult to determine the behaviour of such applications when the run times are close. Table 6.2 summarises the findings from these experiments and illustrates the problems and gains of using different granularity settings.

Granularity Control		Results	Sequential Granularity		
			Small (5)	Appropriate (19)	Large (33)
Co-location Granularity	Small (30)	Speed up	1.27	7.3	
		Efficiency	0.05	0.3	
		Observations	Excessively small tasks High co-ordinations overheads	Fine grained tasks Under utilisation	
	Appropriate (35)	Speed up	1.3	17.6	17.3
		Efficiency	0.05	0.73	0.72
		Observations	Excessively small tasks High co-ordinations overheads	Ideal performance Excellent utilisation	Cores starve at the end Good performance Good utilisation
	Large (40)	Speed up	0.92	12.87	11.95
		Efficiency	0.03	0.53	0.49
		Observations	Excessively small tasks High co-ordinations overheads Under utilisation Nodes starve	Coarse grained tasks Under utilisation Nodes starve	Coarse grained tasks Under utilisation Cores starve at the end Nodes starve

Table 6.2: Analysis of Fibonacci Co-loc. and Seq. Granularity Settings.

6.5 Summary

This chapter shows that a profiler which was constructed from the host language tools is effective and efficient use for performance analysis and tuning of a distributed-memory parallel DSL. We investigated how effectively HdpHProf can be used as an application profiler for the DSL HdpH. The study used several benchmarks, Queens, NBody, Mandelbrot, Liouville, SumEuler, and Fibonacci, to show how HdpHProf is used identify performance problems; for instance, too small/large thread granularity, synchronisa-

tion bottlenecks, and a combination of these factors (Section 6.2). Moreover, it demonstrates using HdpHProf for tuning thread granularity in HdpH applications with flat data parallelism (Section 6.3.1) and applications with divide and conquer parallelism (Section 6.3.2). Lastly, HdpHProf was used to study and tune thread granularity for an HdpH application, with two granularity thresholds. We found that it is crucial to set both thresholds to an appropriate granularity as the application performance is very sensitive to these values (Section 6.4).

Chapter 7

Evaluating HdpHProf for HdpH Internals

This chapter investigates the potential for using host language profiling tools to profile the parallel DSL implementation. It does so by evaluating HdpHProf for profiling the HdpH implementation. HdpH uses mutable data structures in Haskell, e.g. the spark pool (a concurrent deque) and the registry (a concurrent map) [84]. As a lazy functional language, Haskell provides alternative implementations of data structures with different properties of strictness [58]. To understand which functions or data structures perform better for the HdpH RTS implementation we use HdpHProf to investigate the performance of alternative implementations. For this purpose we use the HdpHProf analysis tools from Sections 4.2.6 and 4.3.2, the Spark Pool, and Registry Contention Analysis. Using the Spark Pool Contention Analysis we investigate how the spark pool implementation behaves with highly concurrent access (Section 7.1) and how contention changes in respect to changes in task granularity (Section 7.2). We use the Registry Contention Analysis tool to evaluate the performance of three more or less strict implementations of the HdpH RTS registry to help debug and improve HdpH performance (Section 7.3).

7.1 Spark Pool Contention

HdpH is implemented in a layered fashion to ensure easy maintainability with coordination aspects such as, communication, global references management, sparks management, and scheduling; each realised in an independent module [84]. Figure 7.1 depicts HdpH architecture design in terms of state. HdpH maintains state using mutable data

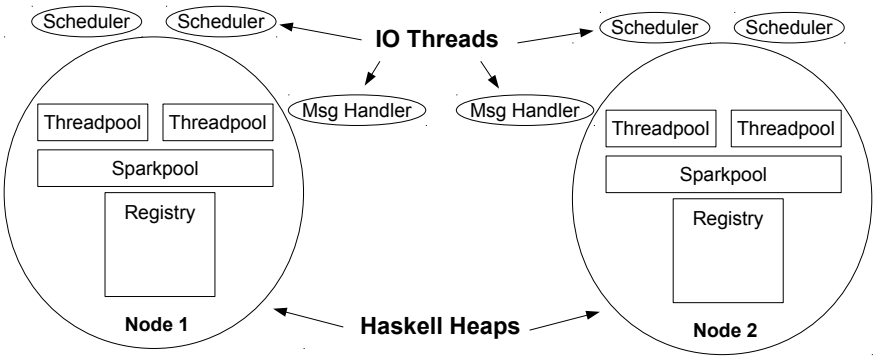


Figure 7.1: HdpH System architecture [84].

structures in Haskell and agents (Haskell IO Threads). Multiple nodes collaborate to do work; each node runs multiple scheduler threads, normally one per core. Each scheduler has a dedicated thread pool that can be accessed concurrently by other schedulers for work stealing. Nodes also have message handlers (one per node) that share access to the spark pool with schedulers on the node. In addition, each node has a registry for global IVars that are shared between the message handler and schedulers.

7.1.1 Spark Management

HdpH manages sparks at the Haskell level, where each HdpH node stores sparks in a spark pool. A spark enters the spark pool, either when being sparked by a scheduler, or when being received by the message handler. Sparks from the spark pool can be transferred to a local thread or can be sent to another node in the form of a SCHEDULE message. When a node's spark pool runs low on sparks it sends a FISH message to a random node, or it directs the message to a node known to have excess sparks.

The spark pool is an essential and important component of the HdpH RTS that is used to coordinate parallelism. A spark represents possible future work and when created it goes to the spark pool. Schedulers running out of work concurrently access the spark pool looking for sparks. Therefore, schedulers that are looking for sparks might compete to access the spark pool and contention becomes a performance problem, as discussed in Section 4.2.6. This section presents how spark pool contention changes in respect to an increase in the number of schedulers. We will use the Spark Pool Contention Analysis tool to produce performance profiles and study the behaviour of HdpH. For this study we are interested in the following aggregated values of the profile:

- Conflict ratio.

$$PCR = \frac{PC_{Total}}{SE_{Total}} * 100 \tag{7.1}$$

  Where PCR is productive conflicts ratio, PC_{Total} is total productive conflict, and SE_{Total} is total spark pool entries.

- Conflict duration.
- Conflict mean duration.
- Conflicts grouped by number of schedulers involved.

7.1.2 Experiments

Using the Spark Pool Contention Analysis tool we studied how increasing the number of schedulers changed contention on the HdpH RTS spark pool. Experiments were executed on a Beowulf cluster of multicores using the Fibonacci and SumEuler benchmarks. The hardware set-up and benchmarks were described previously in Section 3.1.1 and Section 5.1 respectively. We used these benchmarks as they gave us control over task granularity so we could induce contention. These are the benchmarks used to collect performance data:

- A divide and conqueror benchmark, the Fibonacci 35 with Threshold 17.
- A flat data parallel benchmark, the SumEuler 10,000 with Chunk size 1.

We deliberately set low thresholds and chunk sizes for the benchmarks to trigger contention by creating fine grain tasks. The number of schedulers was increased gradually from 1 to 7 on a multicore with 8 cores. We followed standard practice of not using the 8^{th} scheduler to reduce the variability of the results [88]. We analysed the collected data to produce figures that present different aspects of contention on the spark pool, reporting the median of 5 executions, and that use error bars to represent sample standard deviation relative to the median [93].

7.1.3 Conflict Ratio

Figure 7.2 shows how the ratio of productive conflicts changed as the number of schedulers increased. For Fibonacci increasing schedulers linearly introduced more conflicts by an average of 33% points. SumEuler was similar, conflicts increased by an average

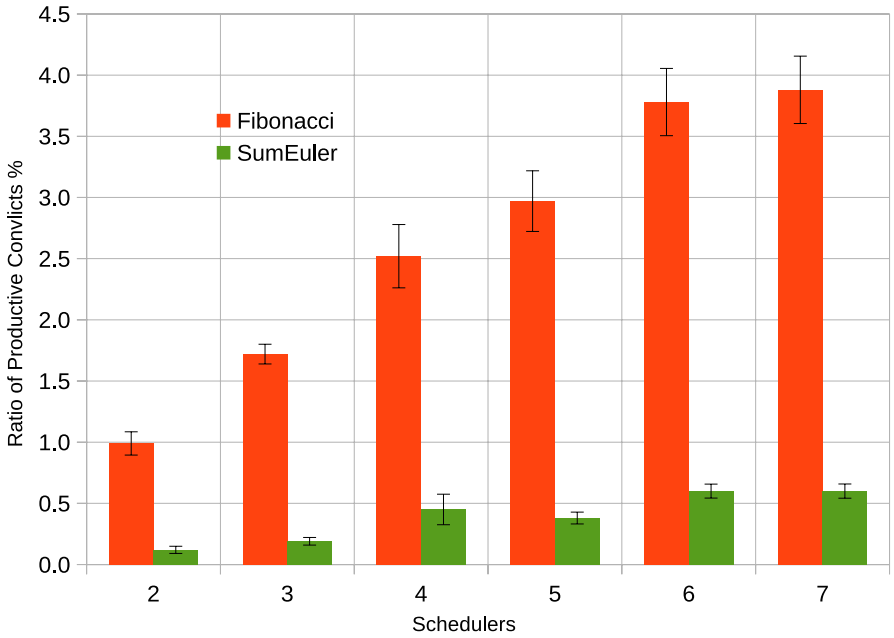


Figure 7.2: Spark Pool Conflict Ratios.

of 30% points with each scheduler. Fibonacci had higher conflict ratios than SumEuler. Overall, increasing the number of schedulers increased spark pool conflicts.

7.1.4 Conflict Duration

Figure 7.3 shows how the durations of productive conflict changed as the number of schedulers increased. The duration presented is the total elapsed waiting time in productive conflicts. For Fibonacci increasing the number of schedulers linearly raised conflict duration by an average of 45%. Similarly, increasing the number of schedulers by 1 increased conflict duration by an average of 23%. Moreover, in both benchmarks increasing the number of schedulers introduced more variability to the results as the error bars show. SumEuler shows less conflict duration than Fibonacci.

7.1.5 Mean Conflict Duration

Figure 7.4 shows how the mean duration of productive conflict changed as the number of schedulers increased. The data is noisy as can be seen from the error bars. The Fibonacci results suggest that increasing the number of schedulers increases the mean conflict duration and increases variability. However, the results for SumEuler are inconclusive.

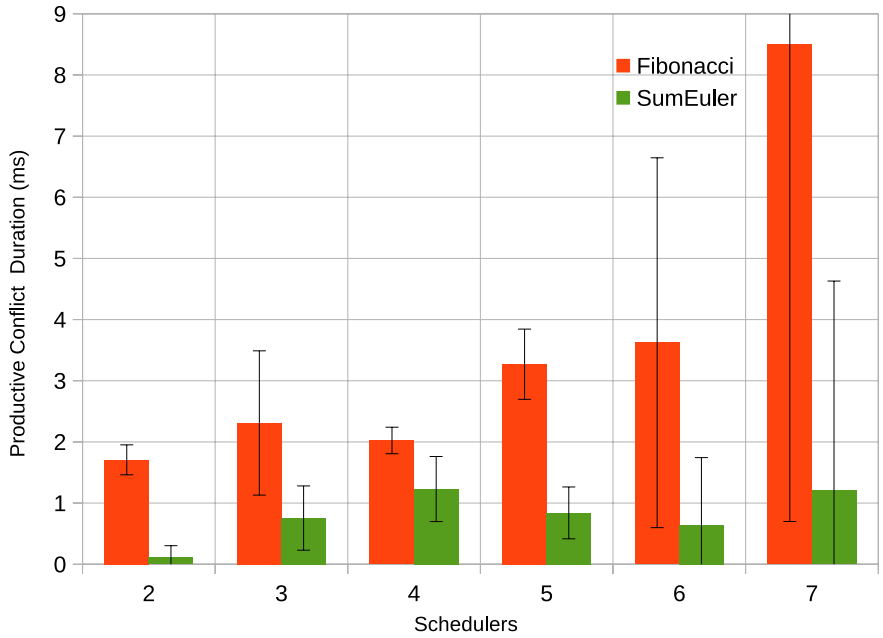


Figure 7.3: Spark Pool Conflict Durations.

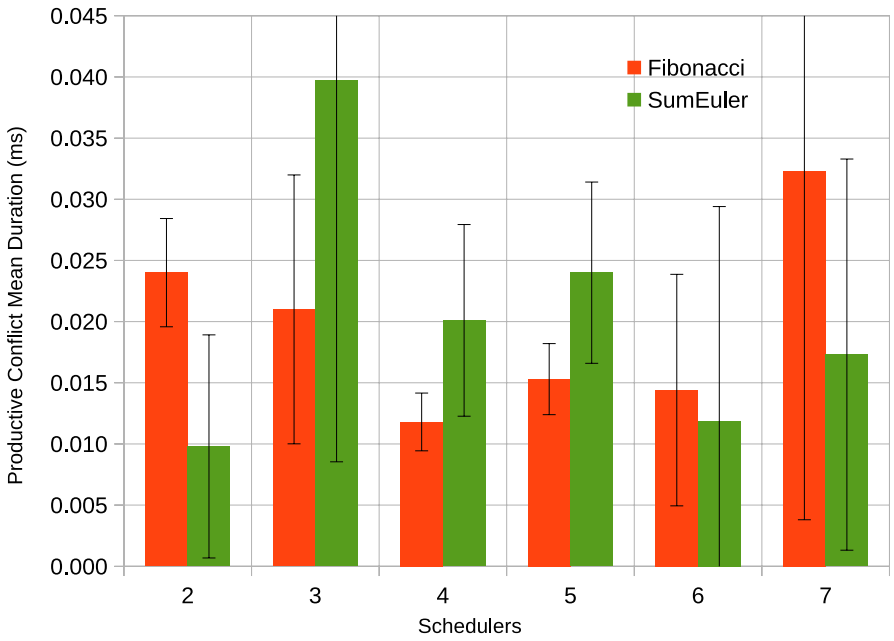


Figure 7.4: Spark Pool Mean Conflict Duration.

7.1.6 Maximum Conflict Duration

Figure 7.5 shows how the maximum conflict duration of a productive conflict changed as the number of schedulers increased. For Fibonacci the maximum conflict duration increased sharply. On the contrary, for SumEuler it increased steadily.

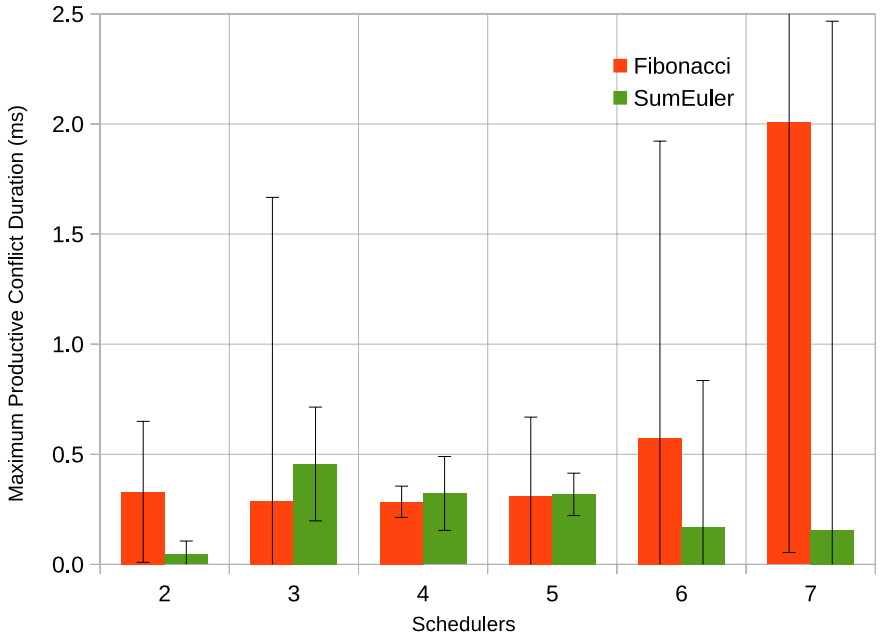
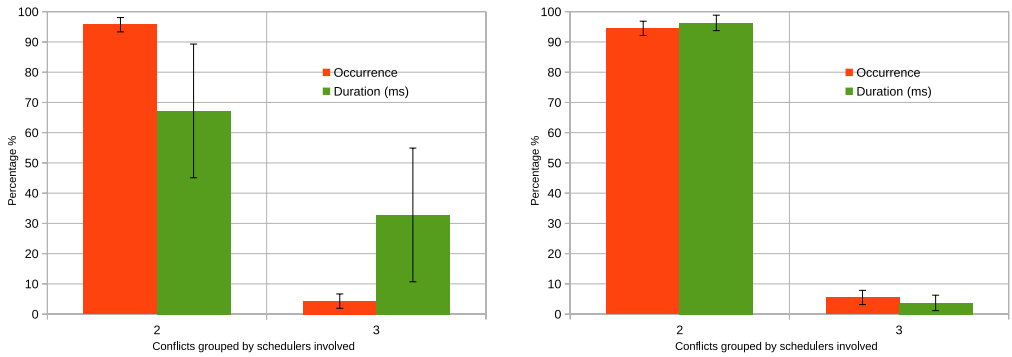


Figure 7.5: Spark Pool Maximum Conflict Duration.

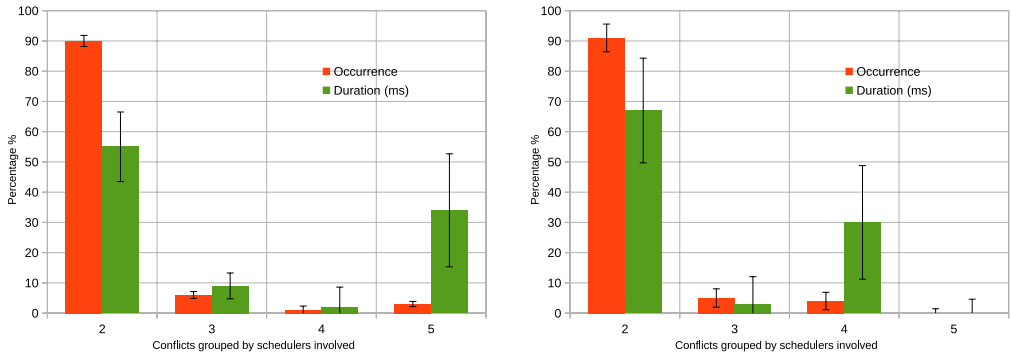
7.1.7 Grouping Conflicts by Schedulers

This section studies how spark pool conflict duration is distributed between groups of conflicting schedulers. We investigated how many schedulers participate in each conflict, and the associated conflict duration. We will only show three cases out of the six due to similarities of the results. Figures 7.6, 7.7 and 7.8 compare conflict occurrence with conflict duration grouped by the number of schedulers involved in a conflict. The date for these figures was derived from the third section of the Spark Pool Contention Analysis tool (Section 4.3.5). This section of the tool shows conflicts grouped by number of schedulers involved in a conflict. Each conflict group has an occurrence percentage and a conflict duration percentage, where occurrence percentage is the ratio of conflicts with a particular number of schedulers to the amount of conflicts on spark pool. Duration percentage is the ratio of total conflict duration for a particular group to the total conflict duration on the spark pool.



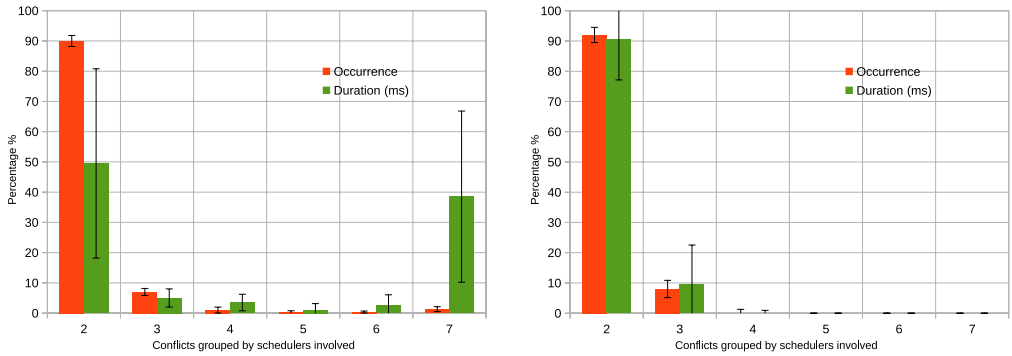
(a) Fibonacci (b) sumEuler

Figure 7.6: Conflicts Grouped by No. Schedulers involved (3 Schedulers).



(a) Fibonacci (b) sumEuler

Figure 7.7: Conflicts Grouped by No. Schedulers involved (5 Schedulers).



(a) Fibonacci (b) sumEuler

Figure 7.8: Conflicts Grouped by No. Schedulers involved (7 Schedulers).

From this study we can see that increasing the number of schedulers for evaluation resulted in more conflicting groups of schedulers within the spark pool. However, the results reveal that conflicts occurrence and conflicts duration were not distributed

evenly between conflicting groups. In the case of conflicts' occurrence, the group of 2 schedulers was the one that dominated the other groups. For instance, in all cases the group of 2 schedulers was never lower than 90% of the conflict occurrence. On the contrary, groups of 3 to 7 schedulers had a small percentage of the conflict occurrence, less than 9%. Also, conflict durations were distributed between groups of schedulers differently as the number of schedulers increased.

Notably, when the number of schedulers is greater than 2, some groups can have significantly longer conflict durations that exceed the occurrence ratio. For example, with Fibonacci on 3 schedulers (Figure 7.6(a)) the 3-scheduler conflicts had an occurrence of 4.3%, with conflict durations of 32.8%; in SumEuler with 5 schedulers (Figure 7.7(b)) the 4-scheduler conflicts had an occurrence of 4%, with a conflict duration of 30%; and Fibonacci on 7 schedulers (Figure 7.8(a)) revealed the 7- scheduler conflicts had an occurrence of 1.3%, with conflict durations of 38.5%. Consequently, increasing the number of schedulers used for evaluation can significantly increase the conflict durations for groups of conflicting schedulers. This is because conflicts with large number of schedulers involved make it more probable that there will be a greater conflicts' duration, even though the occurrence of these conflicts is very low.

7.2 Spark Pool Contention and Granularity

This section presents how contention of the spark pool changes in respect to task granularity. Again we used the Spark Pool Contention Analysis tool to collect the performance data of HdpH. In ordinary settings, where the costs of computation and communication can be determined, granularity can be defined as the ratio of computation in relation to the amount of communication [75].

$$TraditionalGranularity = \frac{ComputationTime}{CommunicationTime}$$

However, in some systems communication time is hard to determine, e.g. GUM [138] or HdpH [84]. To measure granularity for such as systems the communication time is assumed to be the same for all tasks. Therefore, the granularity of a parallel program is defined as "the average computation cost of a sequential unit of computation in the program. [79]" In other words, granularity is measured by computation time only and the communication time is dropped out. Hence, for systems like GUM and HdpH a task granularity is the computation time, which is normally presented in milliseconds.

It is important to choose the right task granularity in parallel execution as it can significantly affect the performance of the parallel program. For instance, fine grain parallelism causes a high communication overhead and is less likely to increase performance because of the low computation to communication ratio. On the other hand, it is easier to get performance gains with coarse grain parallelism, but this may cause load imbalance problems [9]. Therefore, it is important to choose the appropriate task granularity for a parallel program and the choice is dependent on the parallel algorithm and parallel architecture.

7.2.1 Experiments

To study how contention changed in respect to granularity, we needed to know what the task granularity was for the applications that we used as benchmarks for these experiments. We did this by sequentially executing and measuring execution times of the applications. Then we measured the contention by running the applications in parallel with gradually increasing task granularity. After that, we analysed the data to study how contention on spark pool changed as task granularity increased. We executed the experiments on a Beowulf cluster of multicores using two benchmarks, the hardware set-up and benchmarks were described previously in Section 3.1.1 and Section 5.1 respectively:

- Fibonacci, to control task granularity the parameters for this benchmark were varied as shown in Table 7.1.
- SumEuler, similarly, to control task granularity the parameters for this benchmark were varied as shown in Table 7.2.

To measure task granularity both benchmarks were executed sequentially on a single core of a multicore node for 11 runs. The mean value of these runs was used to calculated average task granularity as follows:

$$Granularity = \frac{SequentialRuntime}{NumberOfTasks} \tag{7.2}$$

Contention data was obtained by profiling the benchmarks running on 6 cores of an 8-core Beowulf node. We chose to run on 6 cores to minimise the variability in the result. On the presented figures we report the mean of 51 runs for each input and we do not report error bars here because the figure has a logarithmic scale. We ran

the benchmarks 51 times because the data is noisy so we needed many data points for accuracy.

Fib	Threshold	No. Tasks
35	21	986
37	23	986
39	25	986
41	27	986
43	29	986
45	31	986
47	33	986
49	35	986

Table 7.1: Parameters of Fibonacci Benchmark.

SumEuler	Chunk Size	No. Tasks
[10000 .. 11000]	1	1000
[10000 .. 12000]	2	1000
[10000 .. 14000]	4	1000
[10000 .. 18000]	8	1000
[10000 .. 26000]	16	1000
[10000 .. 42000]	32	1000
[10000 .. 74000]	64	1000
[10000 .. 138000]	128	1000

Table 7.2: Parameters of SumEuler Benchmark.

7.2.2 Conflict Ratio and Granularity

Figure 7.9 shows how the frequency of productive conflicts changed as task granularity increased. The figure includes curves for two benchmarks, Fibonacci and SumEuler. Fibonacci is measured between granularities of 1.09ms and 256ms, and SumEuler between granularities of 2.5ms and 1900ms. The results indicate that low granularity increases conflicts on spark pool in both benchmarks where conflicts are high at granularity below 16ms. However, conflicts fall below 0.03% when granularity is greater than 16ms, and remain stable thereafter.

7.2.3 Conflict Duration and Granularity

Figure 7.10 shows how the duration of productive conflicts changed in respect to increases in task granularity over the same interval as before. Both Fibonacci and SumEuler show that conflict duration declines most between granularity of 30ms and 100ms. This shows that granularity below 16ms increases conflicts duration on the

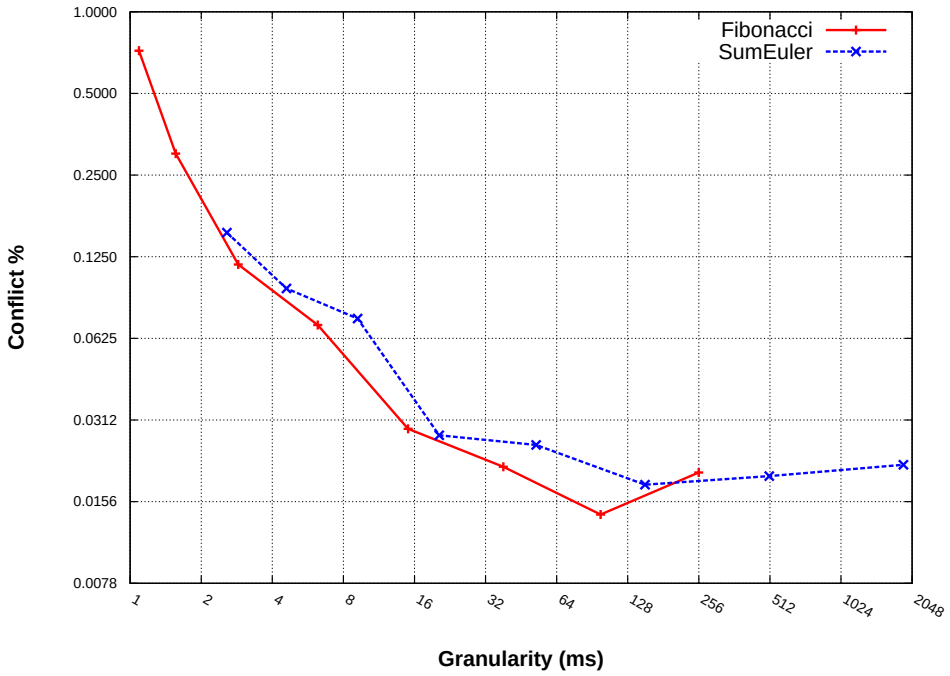


Figure 7.9: Conflict Ratio and Granularity.

spark pool. Moreover, for SumEuler large granularities, i.e. beyond 256ms, there is a slight increase in conflict durations. Therefore, appropriate granularity of between 30ms and 100ms keeps conflicts' duration at the lowest level.

7.2.4 Contention and Granularity Discussion

The experiments demonstrate how task granularity can affect contention within the spark pool. Results of the two benchmarks show that with low task granularity conflict occurrence and duration are high. However, increasing task granularity to an appropriate value, e.g. more than 16ms, significantly reduces both conflict occurrence and duration (by more than 90%).

7.3 Registry Contention

This section presents experimental results of two changes to the registry of the HdpH RTS. Originally, the registry was implemented by using a lazy Map with a lazy globalise operation. To investigate how the performance of HdpH might be improved, we changed these lazy aspects of the registry to be strict. After that, we examined the behaviour of the HdpH RTS in terms of contention on the registry to see whether strictness improved the performance or not.

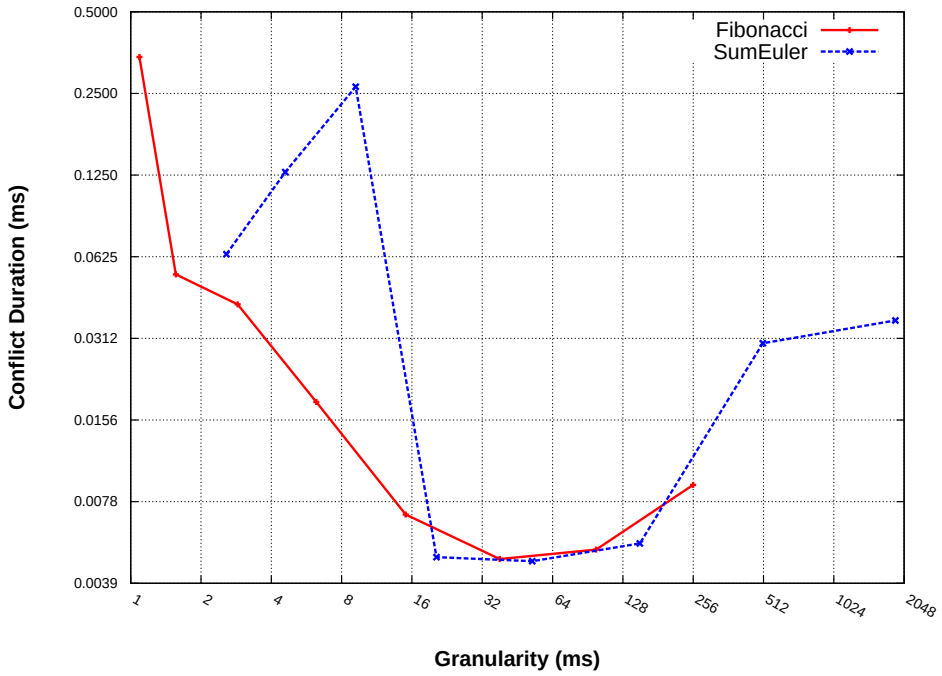


Figure 7.10: Conflict Duration and Granularity.

7.3.1 Global References and Global IVars

HdpH uses global references to access a remotely hosted object in type-safe way. A unique global reference consists of pair, a node ID identifying where the object is hosted, and a unique name for the object on the hosting node that remains unique during the life span of the host. Importantly, the relationship between global references and their referenced objects is kept in the *registry*, a concurrent mutable look-up table (concurrent Map) in Haskell. The operations on global references are as follows.

- Create new global reference (Globalising a local object)
- Dereference an existing global reference
- Free a global reference (To avoid garbage collection)

Conflicts can happen between these operations in the registry when a scheduler holds the registry while one or more other schedulers must wait to access the the registry as discussed in Section 4.2.6.

7.3.2 Experiments

Using the Registry Contention Analysis tool we studied how increasing strictness on the registry implementation in the HdpH RTS changes the behaviour in terms of contention.

We made two changes to the HdpH registry to introduce strictness, A strict function, i.e. `atomicModify`, is used for the globalise operation instead of a non-strict one as was previously used; a strict Map (newly introduced with the GHC 7.6.3 [47]) was used for the registry instead of the standard lazy Map. To understand how these changes to the HdpH RTS might change the behaviour, we took measurements before and after each of these changes. We used the Registry Contention Analysis to gather data and to study the behaviour for the Fibonacci and SumEuler benchmarks from Section 7.1.2. We measured total, mean, and maximum conflict durations and two new measures:

- Conflict ratio.

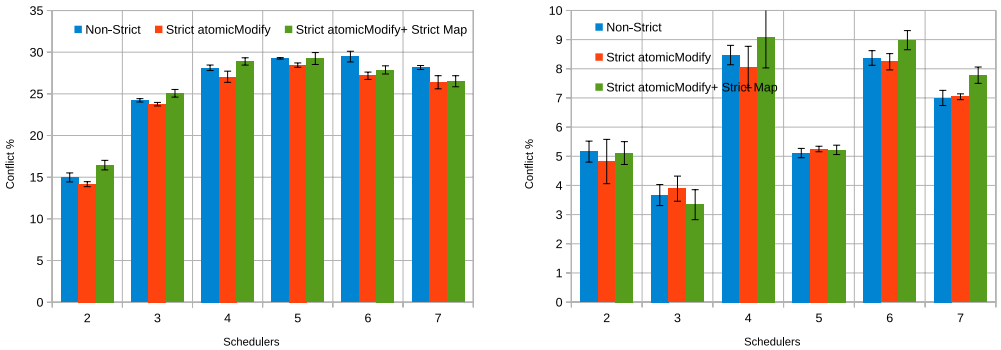
$$CR = \frac{C_{Total}}{RE_{Total}} * 100 \tag{7.3}$$

  Where RC is conflict ration, C_{Total} is total conflicts, and RE_{Total} is total registry entries.

- Ratio of conflict occurrence between operations categorised by type e.g. GG, FF, DD and MIX, where GG is Globalise conflicting with Globalise etc.

We analysed the collected data and compared the results for all alternative implementations of the HdpH RTS registry. On the figures we report the median of five executions and use error bars for the sample standard deviation relative to the median [93]. We increased the number of schedulers gradually from 1 to 7 on a multicore node of 8 cores. We did not use 8 schedulers to reduce variability in the result as we did not use the 8 cores in the node.

7.3.3 Conflict Ratio

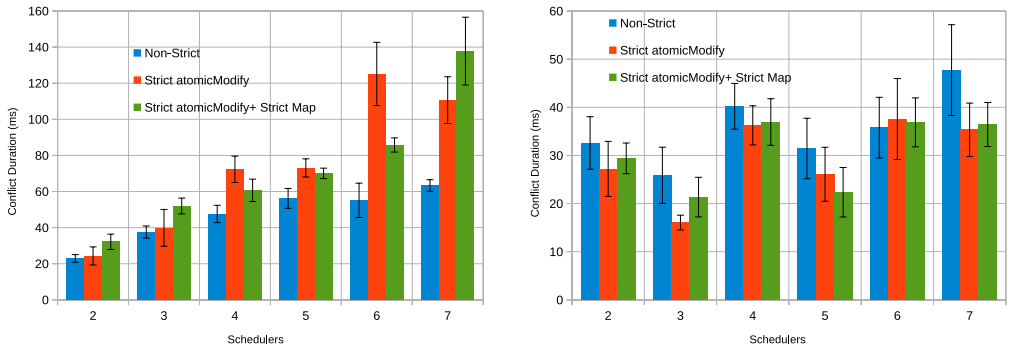


(a) Fibonacci (b) sumEuler

Figure 7.11: Registry Contention Analysis: Conflict Ratio.

This section studies how strictness changes the conflict occurrence between operations on the registry. Figure 7.11 compares conflicts ratio of the three alternative implementations of the registry and shows how conflict ratio changes as the number of schedulers increases. For Fibonacci the Strict atomicModify shows that it has a lower conflict ratio than the Non-Strict. On the other hand, the Strict atomicModify + Strict Map has a higher conflict ratio than the Non-Strict bars. For SumEuler the results fluctuate and the data is noisy. In contrast, the Strict atomicModify + Strict Map shows more conflict ratio than the Non-Strict. Both benchmarks show that using strict globalise operations decreases the conflict occurrence between operations on the registry. However, using the strict Map causes a higher conflict ratio than the lazy Map. Consequently, the Strict atomicModify is a better implementation for the registry to reduce conflict occurrence.

7.3.4 Total Conflict Duration



(a) Fibonacci (b) sumEuler

Figure 7.12: Registry Contention Analysis: Conflict Duration.

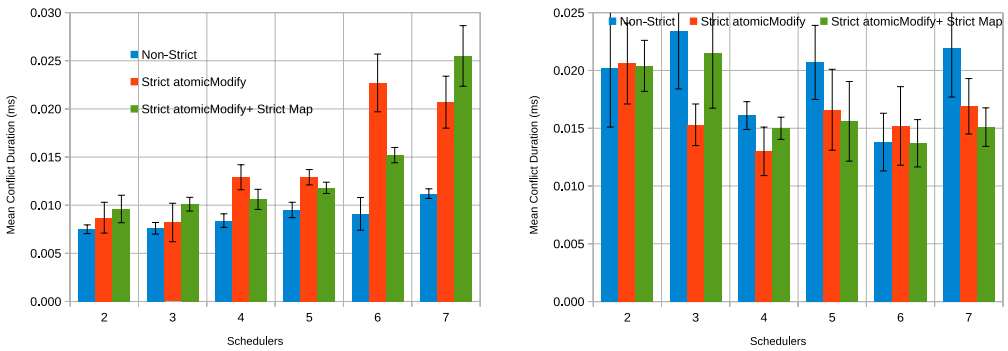
This section studies how strictness changes the conflict duration on the registry. Figure 7.12 compares how total conflict duration changes as the number of schedulers increases between the three alternative implementation of the registry. For Fibonacci conflict duration on the Strict atomicModify and the Strict atomicModify + Strict Map increased more rapidly than the Non-Strict. From the SumEuler conflict duration on the Strict atomicModify and the Strict atomicModify + Strict Map show lower conflict durations than the Non-Strict. The results of the two benchmarks were inconsistent in terms of conflict duration where the Non-Strict appears to have lowest conflict duration on the Fibonacci figure and the highest on the SumEuler figure. We think this is because of the difference between parallel algorithms of these benchmarks and the explanation for this phenomenon is as follows.

For SumEuler all globalise operations are made at the beginning of the computation by one scheduler. This means that there is only one scheduler who accesses the registry so no conflicts with other schedulers will occur. In addition, since all globalise operations were evaluated strictly earlier, none of the proceeding conflicts would have to wait for an unevaluated thunk to be evaluated. Therefore, the conflict duration decreased with strictness.

In contrast, the Fibonacci globalise operations started from the beginning of computation and progressed until the end. This means that globalise operations can be in conflict with other operations at any time during the computation and these conflicts can take longer to bring the registry into normal form. However, with globalising lazily the operation creates a thunk and the real evaluation takes place later when the values are required. With laziness there is only a small time penalty to globalise and skip which reduces the chances that globalis operations will conflict with each other or with other operations on the registry for a longer time. In this case many globalise operations can take place without a long conflict duration unless an unlucky scheduler hits a thunk, then it has to wait until it has been evaluated.

Globalising strictly means no thunks are left to be evaluated later by other operations. This means that globalise operations need more time to fully evaluate and leave the registry. In other words, when globalise operations conflict they will take longer than if they were lazy. It seems that forcing globalise operations has a greater cost than lazy evaluation. We think this is because leaving many globalise operations in the form of thunks, and then evaluating them all at once, is cheaper than strictly evaluating each globalise operation immediately.

7.3.5 Mean Conflict Duration

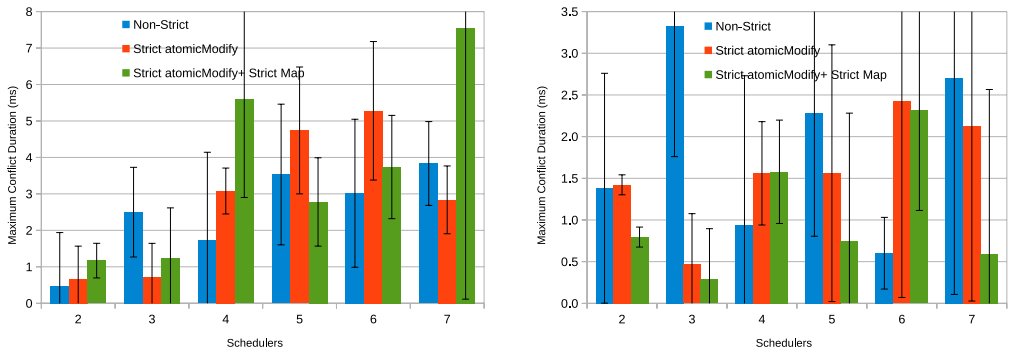


(a) Fibonacci (b) sumEuler

Figure 7.13: Registry Contention Analysis, Mean Conflict Duration.

This section studies how strictness changes the mean conflict duration. Figure 7.13 shows a comparison of how mean conflict duration changes in respect to the increase in the number of schedulers between three implementations of the registry. For Fibonacci the mean conflict duration increases as the number of schedulers increases. The Strict atomicModify and Strict atomicModify + Strict Map increase more dramatically than the Non-Strict. On the other hand, the results from the SumEuler figure show that increasing the number of schedulers does not increase the mean conflict duration but makes it more variable. In addition, the Strict atomicModify and Strict atomicModify + Strict Map show fewer mean conflict durations. Again, the Fibonacci shows an increase in mean conflicts durations with strictness; whereas, SumEuler shows a decrease. The mean conflict duration is derived from the total conflict duration. Therefore, we think what makes these results contradictory are the same reasons we have already stated in the conflict duration section.

7.3.6 Maximum Conflict Duration



(a) Fibonacci (b) sumEuler

Figure 7.14: Registry Contention Analysis, Maximum Conflict Duration.

This section studies how strictness changes maximum conflict duration. Figure 7.14 compares how maximum conflict duration of three implementations changes as the number of schedulers increases. From the Fibonacci figure, it can be seen that increasing the number of schedulers increases the maximum conflict duration. However, the data is noisy and as the number of schedulers increases the more variable it becomes. From the SumEuler figure, data is very noisy and it is hard to determine that increasing the number of schedulers increases the maximum conflict duration, or rather it introduces more variability. Moreover, we think that the noisiness in the results could be due to other HdpH factors, such as non-deterministic aspects in its

implementation. Therefore, in term of the maximum conflict duration we cannot say for sure which implementation is better, nor make any decision based on these results.

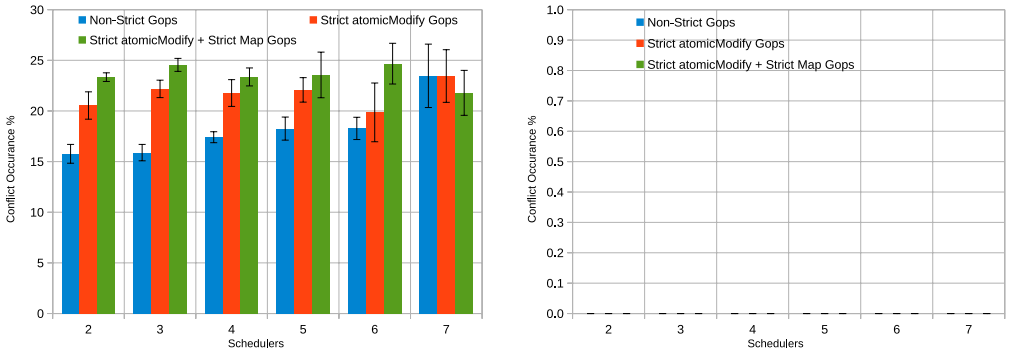
7.3.7 Conflicts By Operation Type

This section compares the three alternative implementations of the registry in terms of conflict types that occur during evaluation. Conflicts are divided into various groups by operation type. We are looking at the ratio of these conflicts and how they differ between the implementations. Performance data of the three versions are collected using the Registry Contention Analysis tool, conflicts grouped by event type (Section 4.3.5). From the profile we study how the occurrence of the grouped events changes as the number of schedulers increases. Data from the three implementations are analysed then results are presented to identify which version performs better in terms of reducing conflict occurrence of certain event types. Conflicts are categorised to four types of conflicts which can occur between operations on the registry as follows:

- Globalise operations (Gops).
- Dereference operations (Dops).
- Free operations (Fops).
- Mixture of operations (Mops).

The Gops group occurrence increases when two or more operations of type Globalise conflict with each other. Similarly, in the Dops and Fops groups occurrence increases when two or more operations of the same event type conflict. However, the Mops group occurrence increases when a conflict has different event types; for example, a globalise operation and free operations.

Globalise operations (Gops)



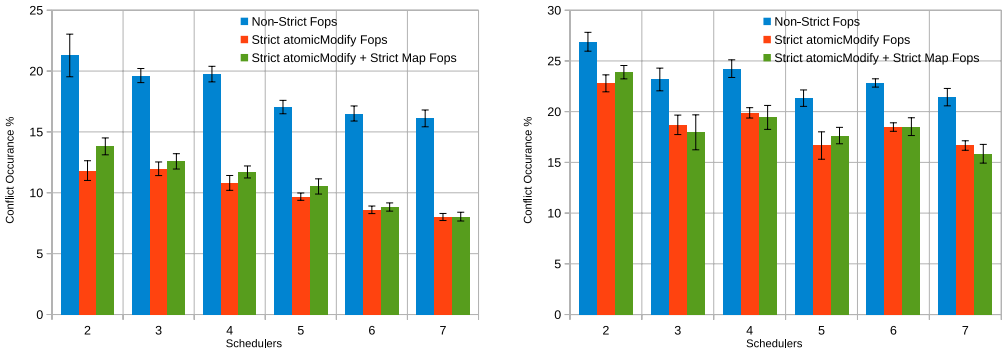
(a) Fibonacci (b) sumEuler

Figure 7.15: Conflicts Between Globalise Operations.

Figure 7.15 compares how conflict occurrence of the group Gops changes as the number of schedulers increases between the three alternative implementations of the registry. From the Fibonacci figure, conflict occurrence on the Strict atomicModify increased by an average of 20% points more than the Non-Strict bars. Similarly, the Strict atomicModify + Strict Map increased by an average of 32% points. From the SumEuler figure, the absence of any Gops conflict is expected. This is because in SumEuler globalisation happens at the beginning of computation by one scheduler only, so no conflict of this type can happen.

Consequently, strictness increases the number of conflicts between events of type Gops. This increase indicates that the globalise operations force evaluation so operations last longer to evaluate. Comparing the results from the two strict versions shows that combining the strict globalise operations with the strict Map has increased Gops conflicts. The increase induced with strictness is justifiable as operations need more time to fully evaluate.

Free operations (Fops)

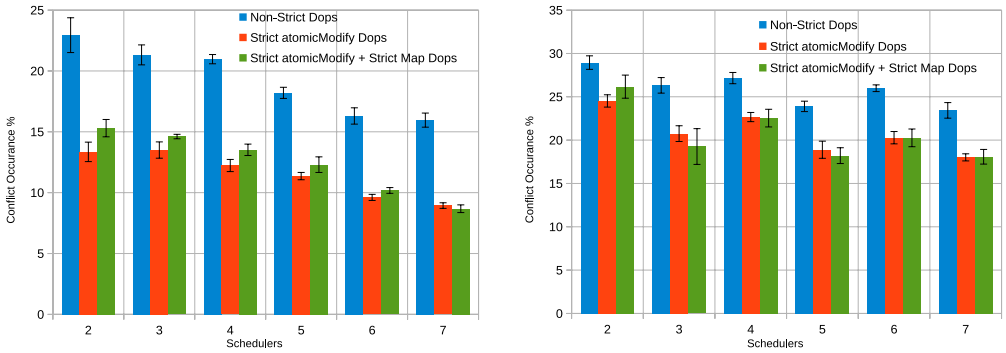


(a) Fibonacci (b) sumEuler

Figure 7.16: Conflicts Between Free Operations.

Figure 7.16 compares the three registry implementations in terms of how Fops conflict occurrence changes as the number of schedulers increases. From the Fibonacci figure, Strict atomicModify bars have less conflict occurrence than the Non-Strict bars by an average of 45% points. Similarly, on the Strict atomicModify + Strict Map bars conflict declined from the Non-Strict by an average of 41% points. From the SumEuler figure, both the Strict atomicModify and Strict atomicModify + Strict Map bars declined by an average of 19% points from the Non-Strict. Therefore, strictness on globalise operations reduces conflict occurrence between Fobs operations. In other words, free operations no longer need to evaluate thunks left by globalise operations and are not involved in more conflicts. Comparing the two strict versions shows that Strict atomicModify reduces Fobs conflicts more than when it is combined with Strict Map. As a consequence, using strict atomicModify is the better alternative as it gives fewer Fobs conflicts.

Dereference operations (Dops)



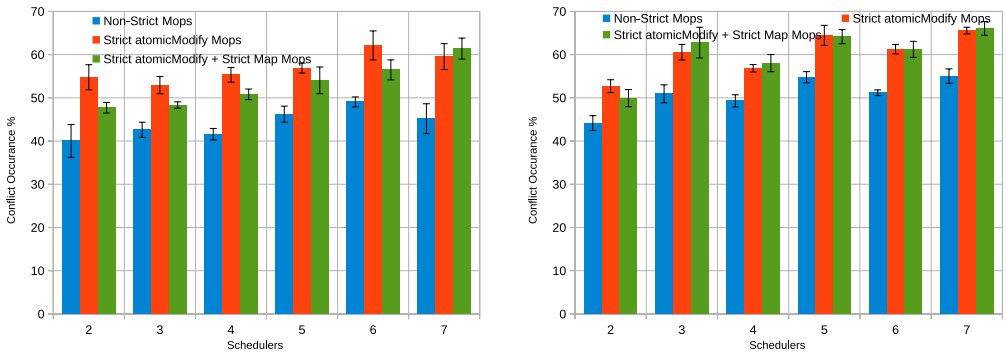
(a) Fibonacci (b) sumEuler

Figure 7.17: Conflicts Between Dereference Operations.

Figure 7.17 compares three registry implementations in terms of how Dops conflict occurrence changes in respect to the increase in number of schedulers. From the Fibonacci figure, the Strict atomicModify introduces lower conflicts than the Non-Strict by an average of 40% points. Likewise, the Strict atomicModify + Strict Map has lower conflicts than the Non-Strict by an average of 35% points. From the SumEuler figure, strictness also reduced conflict occurrence in both versions by an average of 20% points. For example, at 4 schedulers conflicts dropped by 17% points from 27% for Non-Strict, to 22.5% for both Strict atomicModify and Strict atomicModify + Strict Map. Comparing using Strict atomicModify combined with Strict Map shows that using Strict atomicModify alone reduces conflict occurrence more. Consequently, in terms of reducing Dops conflict occurrence the implementation of globalise strictly is the one that performs better for the HdpH RTS.

A Mixture of operations (Mops)

Figure 7.18 compares how Mops conflict occurrence changes in respect to the increase in number of schedulers. From the Fibonacci figure, conflict occurrence increased for the Strict atomicModify by an average of 29% points. Likewise, it increased for the Strict atomicModify + Strict Map by an average of 20% points. From the SumEuler figure, conflicts increased on both strict versions by an average of 18% points. For instance, at 5 schedulers conflicts increased by 17% points from 54% for Non-Strict, to 64% for both Strict atomicModify and Strict atomicModify + Strict Map. Comparing the results of the two strict versions shows that Strict atomicModify, in some cases, increased conflicts more than when it was combined with the Strict Map, and in other



(a) Fibonacci (b) sumEuler

Figure 7.18: Conflicts Between Mixture Operations.

cases it was the opposite. This increase in conflicts occurrence between Mops events is expected because when globalise operations are strict, the chance that they could conflict with other operations increases. As a consequence, globalising strictly increases Mops conflicts.

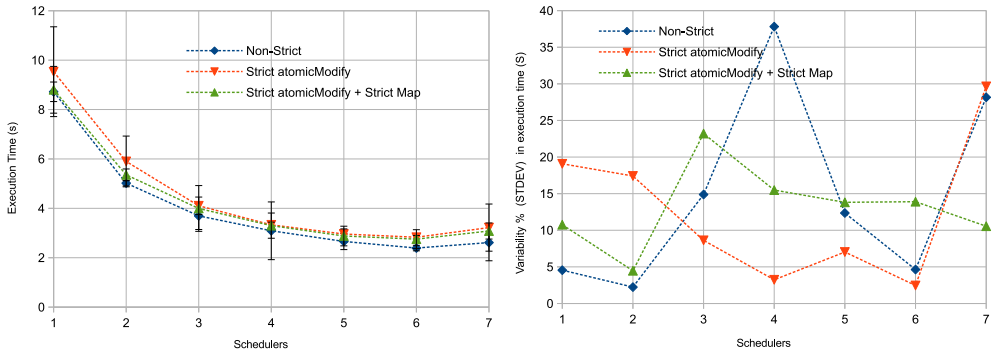
Conflicts by Operations Types Discussion

We compared the results from all operations types Gops, Fops, Dops and Mops to see how overall conflicts occurrence by types change. From the Fibonacci experiments we have seen that strict globalise operations increased Gops and Mops conflict occurrence by an average of 20% points and 29% points respectively. However, it reduced Fops and Dops conflict occurrence by an average of 45% points and 40% points respectively. Similarly, when strict gloabalise is combined with the strict Map, Gops and Mops conflicts increased by an average of 32% points and 20% points respectively. On the contrary, Fops and Dops conflict occurrence declined by an average of 41% points and 35% points respectively. Moreover, from the SumEuler experiments both strict globalise, and strict globalise combined with strict Map, increased only Mops conflict occurrence by an average of 18% points. However, both reduce Fops and Dops conflict occurrence by an average of 19% points and 20% points respectively.

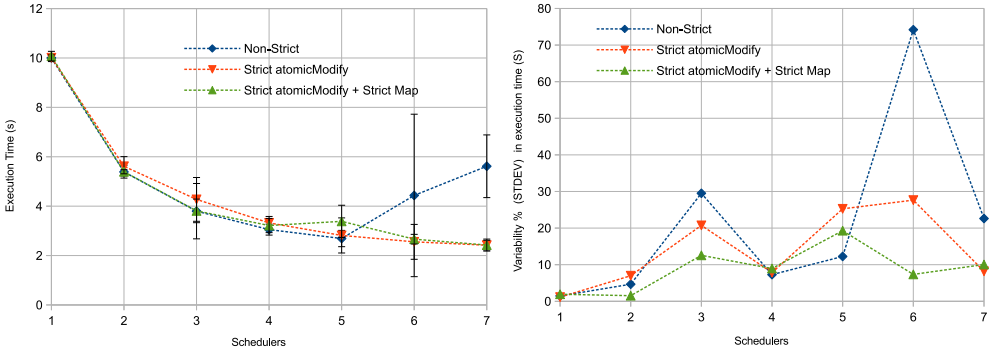
7.3.8 Variability in Execution Time

This section compares the three alternative implementations in terms of variability in execution time. This is to study how the changes that are made to the HdpH RTS, strict globalise operations and strict Map change its execution time behaviour. Variability in execution time data was obtained by executing the same benchmarks

on the same computing architecture introduced in the beginning of Section 7.3. The parameters for Fibonacci were 41 Threshold 19 and the parameters for SumEuler were 10,000 Chunk size 1. With these parameters both benchmarks finished execution in close time, about 10 seconds. On the figures we report the median execution time of eleven runs and variability on execution time. To measure variability on execution time we use the sample standard deviation relative to the median [93].



(a) Fibonacci



(b) sumEuler

Figure 7.19: Variability in Execution Time.

Figure 7.19 compares the three alternative implementations of the HdpH RTS in terms of variability in execution time. Figures on the left show execution time with variability as error bars, whereas, figures on the right show variability as a percentage.

From Figure 7.19(a) Fibonacci (right), on the Non-Strict curve variability increased by 5-fold from 4.5% at 1 scheduler to 28% at 7 schedulers. Also, the curve peaked to 38% at 4 schedulers and most data points of the curve show high variability, between 12% and 37%. On the contrary, variability on the Strict atomicModify curve decreased by 87% points from 19% on 1 scheduler to 2.5% on 6 schedulers, then it increased again by 55% points on 7 schedulers. Importantly, the majority of the curve data points remained low, between 2% and 8%. The variability on the Strict atomicModify + Strict Map curve remained the same, 10.5% on 1 and 7 schedulers. However, most data points of the curve remained high; between 10% and 23%. As a result, the

Fibonacci experiment shows that the implementation of the HdpH RTS registry with the strict atomicModify exhibits less variability in execution time.

From Figure 7.19(b) SumEuler (right), on the Non-Strict curve variability increased by 14-fold from 1.5% at 1 scheduler to 22.5% at 7 scheduler. Moreover, the curve peaked to 74% at 6 schedulers and most data points were high between 12% and 74%. However, variability on the Strict atomicModify curve increased by 621% points from 1.1% on 1 scheduler to 8.2% on 7 schedulers. However, most data points on the curve remained low between 1% and 8%. Similarly, on the Strict atomicModify + Strict Map curve variability grew by 420% points, from 1.9% at 1 scheduler to 10% at 7 schedulers. Again most data points on the curve remained low, between 1% and 8%. Importantly, both strict versions show more consistent results than the non-strict version. Consequently, the SumEuler experiment shows that the both strict implementation of the HdpH RTS registry exhibit less variability than the non-strict version. Although both perform well, comparing the two strict versions together shows that the one with strict globalise operations combined with strict Map has less variability in execution time.

7.3.9 Registry Implementations Discussion

Comparing the registry implementations shows that strictness reduced conflict occurrence. On the other hand, we found the results of conflict duration were inconsistent; the strictness reduced conflict duration for Fibonacci and increased it for SumEuler. Also, we found that strictness increased some registry conflict types, e.g. Gops and Mops, and reduced other conflict types, e.g. Fops and Dops. The results are not clear cut and none of the strict registry implementations perform better than the other in terms of reducing variability in execution time. However, the non-strict version is the worst performer of the three versions in both benchmarks. Based on these results, it is hard to conclude that one of the strict version is better than the other. This is because the registry implementation that used strict globalise operations is the one that performed well for Fibonacci. On the other hand, the version with strict globalise combined with strict map is best for SumEuler. Therefore, we only can say that, generally speaking, with strictness on the registry operations it is more likely that variability in execution will be reduced.

7.4 Summary

This chapter investigates whether a correct and effective profiler can be constructed to tune the parallel DSL implementation (HdpH RTS) using the host language profiling tools (GHC-PPS). HdpHProf analysis tools are used for performance measurement of HdpH internals and data collection. We used the Spark Pool Contention Analysis too to characterise HdpH performance in terms of contention on spark pool. We showed how increasing the number of schedulers in a computing node changes contention between schedulers who try to concurrently access the spark pool (Section 7.1). We found that increasing the number of schedulers can increase the conflict ratio, and conflicts with higher number of schedulers involved, i.e. 3 and more, can have significantly longer conflict durations. Also, we used the tool to study how task granularity affects contention on the spark pool. We showed how increasing the task granularity changes contention on the spark pool, and found that an appropriate task granularity significantly reduced both conflict occurrence and duration (Section 7.2). In addition, the Registry Contention Analysis tool was used to characterise and compare the performance of three alternative implementations of the HdpH RTS registry. We investigated how increasing strictness on the registry implementation can improve the behaviour in terms of contention. Also we compared the implementations for variability in execution time (Section 7.3). We found that strictness can improve some aspects of the implementation, such as reducing conflict occurrence and variability in execution time. However, it can introduce other issues such as, increasing conflict durations and conflicts between some operation types, e.g. Gops and Mops. Moreover, even though it was possible to construct the profiler to tune HdpH implementation, it required the host language profiling tools to be extensible and a took lot of effort to construct the profiler. We had to define appropriate trace events for the DSL, correctly code the DSL implementation to emit the events into an eventlog, build analysis tools to analyse the new events, and validate the correctness of these tools.

Chapter 8

Conclusion

This chapter summarises the thesis, discussing the main issues investigated and the contributions (Section 8.1), gives some limitations of the work (Section 8.2), and gives some suggestions for future work (Section 8.3).

8.1 Summary

A new approach for profiling the performance of parallel DSLs is needed to meet the increasing number of parallel DSLs. This thesis addresses the challenge of using host language profiling tools to provide an effective and efficient profiling for a parallel DSL. We showed that it is possible to construct a profiler using the host language profiling tools, by designing, implementing, and validating HdpHProf; a profiler for the HdpH DSL.

Chapter 3 shows that the host language profiling tools, GHC-PPS and ThreadScope, perform well in terms of overheads and usability, so using them to profile the DSL is feasible and would not have significant impact on the DSL performance. We reported a critical analysis of parallel functional profilers [4, 5], comparing the host language tools with a functional profiler, EdenTV [11], alongside four important imperative profilers, i.e. Vampir [139], Score-P [124], mpiP [141], and ompP [37]. The comparison was based on the SICSA Concordance benchmark [23], covered both shared and distributed-memory parallel languages, and was performed on common parallel architectures. We compared the runtime overheads and amount of profiling data generated by the profilers, analysed whether the parallelism is shared or distributed memory, and whether the profiler is imperative or functional, tracing or summative.

The comparative study showed that summative profilers generated far less profiling data. More interestingly both functional tracing profilers generated one or two orders of magnitude less data than the imperative tracing profilers. While generating so much data risks perturbing program execution, the benefit is that tools like Score-P and Vampir can potentially assist the programmer by providing more detailed information about the program execution (Section 3.2). More work is needed to establish the cost/benefit trade-off between profiling data size and the programmer's understanding of program behaviour.

Both tracing functional profilers induce very low runtime overheads: an order of magnitude less than the imperative tracing profilers. The functional profiler overheads are no more than a factor of two greater than the imperative summative profilers, i.e. 296% for mpiP as compared with 9.4% for EdenTV, and 5.2% for ompP as compared with 10.5% for GHC-PP (Section 3.3).

We systematically compared the profilers for usability and data presentation, and found that the results reflected the design philosophy: summative tools provide key information with minimal intrusion. The functional profilers provide more information and some graphical visualisation, Vampir offers the greatest range of information, and the most sophisticated and usable visualisation tools and maturity of the profilers. Moreover, the results showed that the functional profilers are relatively immature compared with tools like Vampir for popular imperative technologies (Section 3.4).

Chapter 4 shows that it is possible to construct a profiler for a distributed-memory parallel DSL using the host language profiling tools. We presented the design and implementation of HdpHProf which is a post-execution, multi-stage and extensible profiler for the Haskell parallel DSL, HdpH. HdpHProf can profile both the DSL applications and the DSL implementation. HdpHProf can be extended to provide more analysis tools and to present more performance profiles, e.g. how sparks travel between nodes. Importantly, HdpHProf requires no change to the host language (GHC), or the profiled HdpH applications.

HdpHProf uses new GHC platform features to profile the HdpH DSL. The GHC-Events Library was extended to introduce new novel analysis tools to investigate specific performance issue of the DSL implementation, i.e. contention on HdpH internals. It uses ThreadScope, the standard GHC trace browser, to visualise HdpH distributed-memory eventlogs (Section 4.2). The implementation of HdpHProf introduces and modifies different software artefacts to meet its design (Section 4.3). HdpHProf presents

performance information using two ways; summative based profiles and performance graphs. HdpHProf analysis tools produce two summative profiles, the Spark Pool Contention Analysis, and the Registry Contention Analysis. The profiles show a summary of statistical information about how multiple HdpH schedulers access its internals (Section 4.3.5). We found that to introduce new analysis tools for profiling the DSL the host language tool should be extensible like the GHC-PPS [67].

Chapter 5 illustrates that a correct and efficient profiler can be constructed using the host language profiling tools. We validated HdpHProf for functional correctness and profiling performance. We showed that HdpHProf code instrumentation, time synchronisation, and trace file merging, all work correctly and give valid results. Moreover, we validated the functional correctness of the Spark Pool Contention Analysis and the Registry Contention Analysis tools using both hand-crafted and real trace files fragments (Section 5.2). In addition, we demonstrated that HdpHProf can profile long running programs and programs running on relatively large scale architectures: up to 32 Beowulf cluster nodes and 192 cores (Section 5.3). We also characterised and compared HdpHProf overheads in terms of profiling data size (Section 5.4) and profiling execution runtime overhead (Section 5.5), based on the study introduced in Chapter 3. We found that the DSL profiling overheads were within an acceptable level and comparable to other functional profilers, e.g. EdenTV. We measured the ratio of HdpH trace events in the GHC-PPS eventlog and found that the DSL tracing occupied a small percentage of the eventlog, less than 3% on average (Section 5.6).

Chapter 6 shows that a profiler constructed using the host language profiling tools can effectively and efficiently profile a distributed-memory parallel DSL. We investigated how effective HdpHProf is in identifying performance issues as an application profiler for the DSL HdpH. We used several benchmarks, Queens, NBody, Mandelbrot, Liouville, SumEuler, and Fibonacci, to show how HdpHProf is used to identify performance problems; for instance too small/large thread granularity, synchronisation bottlenecks, and a combination of these factors (Section 6.2). We demonstrated using HdpHProf for tuning thread granularity in HdpH applications with flat data parallelism (Section 6.3.1) and applications with divide and conquer parallelism (Section 6.3.2). We used HdpHProf to study and tune thread granularity for an HdpH application with both shared and distributed memory thresholds. We found that it is important to set both thresholds to appropriate values as the application performance is very sensitive to these values (Section 6.4).

Chapter 7 demonstrates that a profiler built using the host language profiling tools is effective and efficient for tuning the parallel DSL implementation (HdpH RTS). We used HdpHProf analysis tools for performance measurement and data collection. The Spark Pool Contention Analysis tool characterises HdpH performance in terms of contention within the spark pool. We showed how increasing the number of schedulers in a computing node changes contention between schedulers who try to access the spark pool concurrently. We found that adding more schedulers can increase conflicts, and conflicts with a higher number of schedulers can have significantly longer conflict durations (Section 7.1). In addition, we used the tool to study how task granularity affects contention on spark pool. We presented how increasing task granularity changes contention on spark pool, and how an appropriate value for task granularity reduces both conflict occurrence and duration (Section 7.2). Also, we used the Registry Contention Analysis tool to characterise and compare the performance of three alternative implementations of the HdpH RTS registry. We studied how increasing strictness on the registry implementation impacts on contention and found that increasing strictness reduces conflict ratio and variability in execution time but increases conflict duration and conflicts between some types of the registry operations (Section 7.3). Moreover, we found that to build the profiler for tuning the DSL implementation, the host language profiling tools required to be extensible.

8.2 Limitations

This section discusses some limitations of the research. Though we found it feasible to produce an effective, usable, and correct profiler for a parallel DSL, using the host language profiling tools, the work has some limitations. First, deciding to use the host language profiling tools restricted us to using those tools available which lacked some important features. ThreadScope is built to profile the fairly simple programming model of GHC shared-memory parallelism and we used it to profile a more elaborate distributed-memory parallel DSL. ThreadScope does not show messages between cores and nodes and hence we were not able to visualise HdpH communications, even though HdpH's messages were traced and emitted into the eventlog. Moreover, user defined trace events in GHC-PPS are user-level messages and hence the DSL trace events are second class citizens in the GHC eventlog. Therefore, extra post-processing is required to analyse the DSL trace events. Second, our profiling approach is limited to profile

applications on a relatively large number of Beowulf cluster nodes. However, we believe that HdpHProf would not scale to large HPC architectures, e.g. HeCTOR [63], because the trace files would become too large to be held and processed in the memory. Finally, for the purpose of evaluating and validating the work of this thesis, we have used a relativity small number of HdpH benchmarks rather than large real applications. This is because HdpH is a new DSL and suitable applications are not available.

8.3 Future Work

It is unclear how much profiling technologies can be shared by the various parallel Haskell DSLs like the Par Monad [89], Cloud Haskell [28], and HdpH [84]. Interesting challenges lie ahead: functional profilers must soon address the issues of scalability and heterogeneity. The scalability challenge is to collect useful information as the number of cores grows exponentially and the bandwidth available to each core shrinks. The challenge of heterogeneity is to profile a program which executes on a range of computing resources, e.g. a combination of multicores and GPUs.

Current functional profilers suffer from immaturity when compared to the more advanced imperative profilers like Score-P and Vampir that provide sophisticated tracing and visualisation tools. Functional profilers could be improved in a number of ways. Currently the data collection and visualisation options are relatively modest, and both could be improved to reflect the advantages of leading tools like Vampir. Functional profiling architectures could better exploit techniques proven by tools like Vampir. For example, instead of different visualisation tools to visualise two variants of parallel Haskell, e.g. ThreadScope for GHC and EdenTV for Eden, one tool could be designed to visualise multiple variants. Similarly, instead of producing different trace formats for each Haskell variant, a standard format is needed which can capture monitoring data from a more generic abstract unit of computation resource. While GHC-PPS represents a move in this direction, it is closely entwined with GHC and has a relatively simple model of computation resources and, crucially for a distributed-memory or heterogeneous system, does not model communication.

It would be interesting to investigate the feasibility of building a shared profiling infrastructure for parallel functional DSLs. Here it would be important to use a standardised tracing data format for trace files. It may be possible to develop a standardised tracing library that can be integrated with the language compilers to produce

performance data. The tracing library can be shared among multiple variants of parallel DSLs. This would allow the development of analysis and visualisation tools that can be shared between different parallel DSLs. The tools should be able to present performance for different models. The tracing library should allow selective events tracing to reduce the overhead of tracing data size and the tracing runtime overhead. The library should use an efficient standard data format to reduce time and space overheads. Such an approach would require a coordinated community effort, including the designers and developers of distributed-memory Haskells like Cloud Haskell and HdpH.

Bibliography

[1] S. Akhter and J. Roberts. *Multi-Core Programming: Increasing Performance Through Software Multi-threading*. Richard Bowles, 2006.

[2] M. Al-Saeed. GitHub repository - Ghc-Events-HdpH Library. `https://github.com/majsaeed/Ghc-events-hdph`.

[3] M. Al-Saeed, P. Maier, P. Trinder, and L. Georgieva. HdpHProf— A Profiler for Haskell Distributed Parallel Haskell. In *The Draft Proceedings of the Symposium on Trends in Functional Programming (TFP'12)*, St Andrews, Scotland, June 2012.

[4] M. Al-Saeed, P. Maier, P. Trinder, and L. Georgieva. A Critical Analysis of Parallel Functional Profilers. In *The Draft Proceedings of The 25th symposium on Implementation and Application of Functional Languages (IFL'13)*, Radboud University Nijmegen, The Netherlands, August 2013.

[5] M. Al-Saeed, P. Trinder, and P. Maier. Critical Analysis of Parallel Functional Profilers. Technical Report HW-MACS-TR-0099, Heriot-Watt University, School of Mathematical & Computer Sciences, Edinburgh, Scotland, EH14 4AS, June 2013.

[6] T. E. Anderson and E. D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '90, pages 115–125, New York, NY, USA, 1990. ACM.

[7] J. Armstrong, S. Virding, M. Williams, and C. Wikstrom. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

[8] R. A. Aydt. The Pablo Self-Defining Data Format. Department of Computer Science, University of Illinois, Urbana, Illinois 61801, 1999.

[9] B. Barney. Introduction to Parallel Computing. Lawrence Livermore National Laboratory. `https://computing.llnl.gov/tutorials/parallel_comp/`, 2010. Accessed on: 17 March 2014.

[10] J. Berthold, M. Dieterle, T. Horstmeyer, B. Pickenbrock, T. Sauerwein, and B. Struckmeier. EdenTV - The Eden Trace Viewer. `http://hackage.haskell.org/package/edentv`. Accessed on: 16 September 2014.

[11] J. Berthold and R. Loogen. Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer. In *In Parallel Computing: Architectures, Algorithms and Applications. Proceedings of the International Conference ParCo 2007*, Jülich, Germany, September 2007.

[12] S. Breitinger, R. Loogen, and Y. Ortega-Mallén. Towards a Declarative Language for Parallel and Concurrent Programming. In *Functional Programming*, Glasgow, 1995. Springer-Verlag Berlin Heidelberg.

[13] H. Brunst. Vampir at Large Scale. In *Lectures of Large-Scale Parallel Profiling with VAMPIR*, Scotland, Edinburgh, April 2013. PRACE Advanced Training Centre, University of Edinburgh.

[14] H. Brunst, D. Kranzlmüller, and W. Nagel. Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. In Z. Juhász, P. Kacsuk, and D. Kranzlmüller, editors, *Distributed and Parallel Systems*, volume 777 of *The Kluwer International Series in Engineering and Computer Science*, pages 93–102. Springer-Verlag Berlin Heidelberg, 2005.

[15] H. Brunst and B. Mohr. Performance Analysis of Large-scale OpenMP and Hybrid MPI/OpenMP Applications with Vampir NG. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, IWOMP'05/IWOMP'06, pages 5–14, Berlin, Heidelberg, 2008. Springer-Verlag Berlin Heidelberg.

[16] H. Brunst, F. Winkler, R. Tschüter, and A. Knüpfer. Introduction to Performance Engineering. In *Lectures of Large-Scale Parallel Profiling with VAMPIR*, Scotland, Edinburgh, April 2013. PRACE Advanced Training Centre, University of Edinburgh.

[17] H. Brunst, F. Winkler, R. Tschüter, and A. Knüpfer. Score-P - A Joint Performance Measurement Run-Time Infrastructure. In *Lectures of Large-Scale Parallel Profiling with VAMPIR*, Scotland, Edinburgh, April 2014. PRACE Advanced Training Centre, University of Edinburgh.

[18] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility . *Future Generation Computer Systems*, 25(6):599–616, 2009.

[19] The CentOS Linux distribution. `http://www.centos.org/`. Accessed on: 15 September 2014.

[20] N. Charles and C. Runciman. An Interactive Approach to Profiling Parallel Functional Programs. In K. Hammond, T. Davie, and C. Clack, editors, *Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 20–37. Springer-Verlag Berlin Heidelberg, 1999.

[21] I.-H. Chung, R. E. Walkup, H.-F. Wen, and H. Yu. MPI Performance Analysis Tools on Blue Gene/L. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, page 16, November 2006.

[22] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *The International Conference on High Performance Computing and Simulation (HPCS'11)*, pages 525–532, July 2011.

[23] Concordance Application - Phase I The SICSA MultiCore Challenge. `http://www.macs.hw.ac.uk/sicsawiki/index.php/Challenge_PhaseI`, December 2010. Accessed on: 14 November 2012.

[24] A. Couch. Locating performance problems in massively parallel executions. *Proceedings of the IEEE*, 81(8):1116–1125, August 1993.

[25] A. L. Couch. *Graphical Representations of Program Performance on Hypercube Message-passing Multiprocessors*. PhD thesis, Medford, MA, USA, 1988. Order No: GAX88-16108.

[26] A. Di Costanzo, M. De Assuncao, and R. Buyya. Harnessing Cloud Technologies for a Virtualized Distributed Computing Infrastructure. *Internet Computing, IEEE*, 13(5):24–33, September 2009.

[27] J. Ekanayake and G. Fox. High Performance Parallel Computing with Clouds and Cloud Technologies. In D. Avresky, M. Diaz, A. Bode, B. Ciciani, and E. Dekel, editors, *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, pages 20–38. Springer-Verlag Berlin Heidelberg, 2010.

[28] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the Cloud. In *Haskell '11, Tokyo, Japan*, pages 118–129. ACM Press, 2011.

[29] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the end of Multicore Scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376, June 2011.

[30] Z. Fan. Research on parallel computing performance visualization based on MPI. In *Advanced Computer Control (ICACC), 2010 2nd International Conference on*, volume 5, pages 323–327, March 2010.

[31] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, September 1972.

[32] M. J. Flynn and K. W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1):67–70, March 1996.

[33] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[34] R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis Onlarge-scale Multiprocessors. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD '88, pages 163–173, New York, NY, USA, 1988. ACM.

[35] K. Fuerlinger, M. Gerndt, and J. Dongarra. On Using Incremental Profiling for the Performance Analysis of Shared Memory Parallel Applications. In A.-M. Kermarrec, L. Boug, and T. Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 62–71. Springer-Verlag Berlin Heidelberg, 2007.

[36] K. Fürlinger. ompP - OpenMP Profiler. `http://www.ompp-tool.com/`. Accessed on: 16 September 2014.

[37] K. Fürlinger and M. Gerndt. ompP: A Profiling Tool for OpenMP. In M. Mueller, B. Chapman, B. de Supinski, A. Malony, and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 15–23. Springer-Verlag Berlin Heidelberg, 2008.

[38] K. Fürlinger and S. Moore. OpenMP-centric Performance Analysis of Hybrid Applications. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing (CLUSTER 2008)*, pages 160–166, Tsukuba, Japan, September 2008.

[39] Q. Gao, X. Zhang, P.-L. P. Rau, A. A. Maciejewski, and H. J. Siegel. Performance Visualization for Large-scale Computing Systems: A Literature Review. In *Proceedings of the 14th International Conference on Human-computer Interaction: Design and Development Approaches - Volume Part I*, HCII'11, pages 450–460, Berlin, Heidelberg, 2011. Springer-Verlag Berlin Heidelberg.

[40] GCC - The GNU Compiler Collection. `https://gcc.gnu.org/`. Accessed on: 16 November 2014.

[41] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. J. N. Wylie. Scalable Collation and Presentation of Call-Path Profile Data with CUBE. In *Proc. of the Conference on Parallel Computing (ParCo), Aachen/Jülich, Germany*, pages 645–652, September 2007. *Minisymposium Scalability and Usability of HPC Programming Tools.*

[42] M. Geimer, F. Wolf, B. J. N. Wylie, E. brahm, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[43] M. Geimer and B. Wylie. Introduction to Parallel Performance Engineering. In *Lectures of Tools for Large-Scale Parallel Debugging and Profiling*, Scotland, Edinburgh, April 2014. PRACE Advanced Training Centre, University of Edinburgh.

[44] G. Geist, M. T. Heath, B. Peyton, and P. H. Worley. PICL: a Portable Instrumented Communication Library. Technical Report ORNL/TM-11130, Oak Ridge National Lab., TN, USA, 1990.

[45] P. Gepner and M. Kowalik. Multi-Core Processors: New Way to Achieve High System Performance. In *Proceedings of the International Symposium on Parallel Computingin Electrical Engineering (PARELEC'06)*, pages 9–13, September 2006.

[46] GHC - The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`. Accessed on: 9 August 2011.

[47] GHC - The Glasgow Haskell Compiler Version 7.6.3. `https://www.haskell.org/ghc/download_ghc_7_6_3`. Released on: 21 April 2013.

[48] Ghc-Eden - The Parallel Haskell Compilation System. `http://www.mathematik.uni-marburg.de/~eden/`. Accessed on: 13 May 2013.

[49] Ghc-Events Library - An Analysing Tool for Haskell Event Logs. `http://www.haskell.org/haskellwiki/Ghc-events`. Accessed on: 12 May 2013.

[50] GHC RTS Events - The Event Log Types. `http://hackage.haskell.org/package/ghc-events-0.4.3.0/docs/GHC-RTS-Events.html`. Accessed on: 16 March 2015.

[51] I. Glendinning, V. S. Getov, S. A. Hellberg, R. W. Hockney, and D. J. Pritchard. Performance Visualisation in a Portable Parallel Programming Environment. In *Performance Measurement and Visualization of Parallel Systems*, pages 251–275. Elsevier Science, 1992.

[52] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[53] S. Hackstadt, A. Malony, and B. Mohr. Scalable performance visualization for data-parallel programs. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 342–349, May 1994.

[54] R. Hall. Call path refinement profiles. *Software Engineering, IEEE Transactions on*, 21(6):481–496, June 1995.

[55] K. Hammond, H.-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *Proceedings of the Glasgow Workshop on Functional Programming*. Springer, July 1995.

[56] K. Hammond, H.-W. Loidl, and P. Trinder. Parallel Cost Centre Profiling. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 1997.

[57] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a Shared-Memory Multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 49–61, New York, NY, USA, 2005. ACM.

[58] The Haskell Programming Language. `http://www.haskell.org`. Accessed on: 18 September 2014.

[59] M. Heath. Visual Animation of Parallel Algorithms for Matrix Computations. In *Distributed Memory Computing Conference, 1990., Proceedings of the Fifth*, volume 2, pages 1213–1222, April 1990.

[60] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *Software, IEEE*, 8(5):29–39, September 1991.

[61] M. Heath and J. E. Finger. ParaGraph: A Tool for Visualizing Performance of Parallel Programs. Technical report, University of Illionis and Oak Ridge National Laboratory, 1994.

[62] M. Heath, A. Malony, and D. Rover. Parallel performance visualization: from practice to theory. *Parallel Distributed Technology: Systems Applications, IEEE*, 3(4):44–60, 1995.

[63] HECToR - UK National Supercomputing Service. `http://www.hector.ac.uk/`. Accessed on: 18 September 2012.

[64] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, February 2010.

[65] G. Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[66] S. Jarvis and R. Morgan. The results of: Profiling large-scale lazy functional programs. In W. Kluge, editor, *Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 200–221. Springer-Verlag Berlin Heidelberg, 1997.

[67] D. Jones Jr., S. Marlow, and S. Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 81–92, New York, NY, USA, 2009. ACM.

[68] M. Kessis and J.-M. Vincent. Performance Monitoring and Visualization of Large-Sized and Multi-Threaded Applications with the Paje Framework. In *Computing in the Global Information Technology, 2006. ICCGI '06. International Multi-Conference on*, page 34, August 2006.

[69] D. J. King, J. Hall, and P. Trinder. A strategic profiler for Glasgow Parallel Haskell. In *The Proceedings of the International Workshop on the Implementation of Functional Languages (IFL'98)*, London, September 1998.

[70] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel. Introducing the Open Trace Format (OTF). In V. Alexandrov, G. Albada, P. Sloot, and J. Dongarra, editors, *Computational Science ICCS 2006*, volume 3992 of *Lecture Notes in Computer Science*, pages 526–533. Springer-Verlag Berlin Heidelberg, 2006.

[71] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller, and W. Nagel. The Vampir Performance Analysis Tool-Set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer-Verlag Berlin Heidelberg, 2008.

[72] A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In H. Brunst, M. S. Mller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91. Springer-Verlag Berlin Heidelberg, 2012.

[73] J. Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. In *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, volume 1, pages 290–299, 1996.

[74] E. Kraemer and J. Stasko. The Visualization of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, 1993.

[75] J. Kwiatkowski. Evaluation of Parallel Programs by Measurement of Its Granularity. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waniewski, editors, *Parallel Processing and Applied Mathematics*, volume 2328 of *Lecture Notes in Computer Science*, pages 145–153. Springer-Verlag Berlin Heidelberg, 2002.

[76] S. Liang and D. Viswanathan. Comprehensive Profiling Support in the Java Virtual Machine. In *COOTS*, pages 229–242, 1999.

[77] M. Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011.

[78] H.-W. Loidl. *GranSim's User Guide*. Department of Computing Science, University of Glasgow, 0.03 edition, July 1996.

[79] H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

[80] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[81] Z. Lu and A. Jantsch. Trends of terascale computing Chips in the next ten years. In *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, pages 62–66, October 2009.

[82] P. Maier. GitHub repository - Haskell distributed parallel Haskell. `https://github.com/PatrickMaier/HdpH`. Accessed on: 15 September 2014.

[83] P. Maier, R. Stewart, and P. Trinder. The HdpH DSLs for Scalable Reliable Computation. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 65–76, New York, NY, USA, 2014. ACM.

[84] P. Maier and P. Trinder. Implementing a High-Level Distributed-Memory Parallel Haskell in Haskell. In A. Gill and J. Hage, editors, *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 35–50. Springer-Verlag Berlin Heidelberg, 2012.

[85] A. D. Malony and D. A. Reed. Visualizing parallel computer system performance. Technical Report UIUCDCS-R-88-1465, Illinois University at Urbana-Champaign, Department of Computer Science, Urbana, IL, USA, September 1988.

[86] A. D. Malony, D. A. Reed, J. W. Arendt, R. A. Aydt, D. Grabas, and B. K. Totty. An integrated performance data collection, analysis, and visualization system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 229–236, California, USA, March 1989. Association for Computing Machinery.

[87] A. D. Malony, S. Shende, A. Morris, and F. Wolf. Compensation of Measurement Overhead in Parallel Performance Profiling. *International Journal of High Performance Computing Applications*, 21(2):174–194, May 2007.

[88] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: Better Strategies for Parallel Haskell. In *Haskell '10, Balore, USA*, pages 91–102. ACM Press, 2010.

[89] S. Marlow, R. Newton, and S. Peyton Jones. A Monad for Deterministic Parallelism. In *Haskell '11, Tokyo, Japan*, pages 71–82. ACM Press, 2011.

[90] S. Marlow, S. Peyton Jones, and S. Singh. Rune Support for Multicore Haskell. In *ICFP '09, Edinburgh, Scotland*, pages 65–78. ACM Press, 2009.

[91] T. Mattson, R. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: the Programmer's View. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, November 2010.

[92] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[93] Measures of variability: the range, inter-quartile range and standard deviation. Leicester, UK, 2009. Student Development, University of Leicester. `http://www2.le.ac.uk/offices/ld/resources/numeracy/variability`, Accessed on: 29 October 2014.

[94] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.

[95] Message Passing Interface (MPI) 2.2 Standard. `http://www.mpi-forum.org/docs/docs.html`, September 2009. Accessed on: 11 March 2013.

[96] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.

[97] K. Nichols, D. Dubois, C. Janczewski, J. Flower, D. Flanagan, J. Yan, A. Malony, D. Reed, N. Saraiya, L. Snyder, D. Krumme, A. Couch, and T. Hideyuki. Performance Tools. *Software, IEEE*, 7(3):21–30, May 1990.

[98] O. Y. Nickolayev, P. C. Roth, Daniel, and D. A. Reed. Real-e Statistical Clustering For Event Trace Reduction. In *International Journal of Supercomputer Applications and High Performance Computing*, pages 144–159, 1997.

[99] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel Distrib. Comput.*, 69(8):696–710, August 2009.

[100] Opari2 - OpenMP instrumenter. `https://silc.zih.tu-dresden.de/opari2-current/html/index.html`. Accessed on: 16 September 2014.

[101] OpenMP - API specification for parallel programming. `http://www.openmp.org`. Accessed on: 16 March 2013.

[102] B. O'Sullivan, J. Goerzen, and D. Stewart. Profiling and optimization. In *Real World Haskell: Code You Can Believe In*. O'Reilly Media, Inc., 1st edition, 2008.

[103] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell: Code You Can Believe In*. O'Reilly Media, Inc., 1st edition, 2008.

[104] OPEN TRACE FORMAT 2 USER MANUAL 1.1, October 2012.

[105] Periscope Performance Measurement Toolkit. `http://www.lrr.in.tum.de/~periscop/`. Accessed on: 22 September 2014.

[106] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL '96, St Petersburg, USA*, pages 295–308. ACM Press, 1996.

[107] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.

[108] R. Pointon, P. Trinder, and H.-W. Loidl. The Design and Implementation of Glasgow Distributed Haskell. In *Implementation of Functional Languages*, volume 2011 of *Lecture Notes in Computer Science*, pages 53–70. Springer-Verlag Berlin Heidelberg, 2001.

[109] Parallel Virtual Machine (PVM). `http://www.csm.ornl.gov/pvm/`. Accessed on: 18 September 2014.

[110] M. J. Quinn. *Parallel programming in C with MPI and OpenMP*. McGraw-Hill, New York, USA, 2004.

[111] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving Time in Large-scale Communication Traces. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 46–55, New York, NY, USA, 2008. ACM.

[112] T. Rauber and G. Rünger. Parallel computer architecture. In *Parallel Programming*, pages 7–91. Springer-Verlag Berlin Heidelberg, 2010.

[113] T. Rauber and G. Rünger. Parallel Programming Models. In *Parallel Programming*, pages 105–167. Springer-Verlag Berlin Heidelberg, 2013.

[114] D. Reed, R. Olson, R. Aydt, T. Madhyastha, T. Birkett, D. Jensen, B. Nazief, and B. Totty. Scalable Performance Environments for Parallel Systems. In *Distributed Memory Computing Conference, 1991. Proceedings., The Sixth*, pages 562–569, May 1991.

[115] D. Reed, P. Roth, R. Aydt, K. Shields, L. Tavera, R. Noe, and B. Schwartz. Scalable performance analysis: the Pablo performance analysis environment. In *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pages 104–113, October 1993.

[116] D. A. Reed, R. A. Aydt, L. DeRose, C. L. Mendes, R. L. Ribler, E. Shaffer, H. Simitci, J. S. Vetter, D. R. Wells, S. Whitmore, et al. Performance analysis of parallel systems: Approaches and open problems. In *Joint Symposium on Parallel Processing*, pages 239–256, 1998.

[117] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz. An Overview of the Pablo Performance Analysis Environment. Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, IL 61801, 1992.

[118] P. Revenga, J. Serot, J. Lazaro, and J. Derutin. A Beowulf-class architecture proposal for real-e embedded vision. In *Computer Architectures for Machine Perception, 2003 IEEE International Workshop on*, pages 7–212, May 2003.

[119] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Proceedings of the Aerospace Conference*, volume 2, pages 79–91. IEEE, February 1997.

[120] C. Runciman and D. Wakeling. Profiling Parallel Functional Computations (Without Parallel Machines). In *Glasgow Workshop on Functional Programming*, pages 236–251. Springer-Verlag Berlin Heidelberg, 1993.

[121] N. Sadashiv and S. Kumar. Cluster, grid and cloud computing: A detailed comparison. In *The 6th International Conference on Computer Science & Education (ICCSE'11)*, pages 477–482, August 2011.

[122] P. M. Sansom and S. L. Peyton Jones. Time and Space Profiling for Non-strict, Higher-order Functional Languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 355–366, New York, NY, USA, 1995. ACM.

[123] The Scalasca Performance Toolset. `http://www.scalasca.org/`. Accessed on: 20 May 2013.

[124] SCORE-P - Scalable Performance Measurement Infrastructure for Parallel Codes. `http://www.vi-hps.org/projects/score-p`. Accessed on: 16 September 2014.

[125] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[126] I. Sommerville. Software process models. In *Software Engineering*, pages 63–91. Pearson Education Limited, 7th edition, 2004.

[127] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A Parallel Workstation For Scientific Computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[128] R. Stewart. GitHub repository - Haskell Distributed Parallel Haskell with Reliable Scheduling. `https://github.com/robstewart57/hdph-rs`. Accessed on: 15 September 2014.

[129] R. Stewart. *Reliable Massively Parallel Symbolic Computing: Fault Tolerance for a Distributed Haskell*. PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, November 2013.

[130] D. Talia. Models and Trends in Parallel Programming. *Parallel Algorithms and Applications*, 16(2):145–180, 2001.

[131] M. Tanaka. A numerical investigation on cumulative sum of the liouville function. *Tokyo Journal of Mathematics*, 03(1):187–189, June 1980.

[132] TAU Performance System. `http://www.cs.uoregon.edu/research/tau/`. Accessed on: 22 September 2014.

[133] S. Thompson. *The Craft of Functional Programming*. Addision-Wesley Longman, Essex, England, 1996.

[134] ThreadScope - A Graphical time line Browser for GHC Trace Files. `http://www.haskell.org/haskellwiki/ThreadScope`, 2013. Accessed on: 3 May 2013.

[135] P. Trinder, H. Loidl, and R. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(5):469–510, July 2002.

[136] P. W. Trinder, E. Barry Jr, M. Davis, K. Hammond, S. Junaidu, U. Klusik, H. Loidl, and S. P. Jones. GPH: an Architecture-independent Functional Language. *IEEE Transactions on Software Engineering*, 1999.

[137] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[138] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: A Portable Parallel Implementation of Haskell. In *Proceedings*

of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, PLDI '96, pages 79–88, New York, NY, USA, 1996. ACM.

[139] Vampir - Performance Optimization. `http://www.vampir.eu/`. Accessed on: 16 September 2014.

[140] VampirTrace 5.14.4 User Manual. `http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace/dateien/VT-UserManual-5.14.4.pdf`. Accessed on: 17 September 2014.

[141] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling. `http://mpip.sourceforge.net/`. Accessed on: 16 September 2014.

[142] J. S. Vetter and F. Mueller. Communication Characteristics of Large-scale Scientific Applications for Contemporary Cluster Architectures. *Journal of Parallel and Distributed Computing*, 63(9):853–865, September 2003.

[143] A. Wikström. *Functional Programming Using Standard ML*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1987.

[144] B. Wilkinson and M. Allen. *Parallel Programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hallall, Inc., Upper Saddle River, New Jersey 07458, 1999.

[145] J. Yan. Performance tuning with AIMS — an Automated Instrumentation and Monitoring System for multicomputers. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 2, pages 625–633, January 1994.

[146] J. C. Yan and S. R. Sarukkai. Analyzing parallel program performance using normalized performance indices and trace transformation techniques. *Parallel Computing*, 22(9):1215–1237, 1996.

[147] C.-Q. Yang and B. Miller. Performance measurement for parallel and distributed programs: a structured and automatic approach. *Software Engineering, IEEE Transactions on*, 15(12):1615–1629, December 1989.