A Critical Analysis of Parallel Functional Profilers

Majed Al Saeed, Patrick Maier, Phil Trinder and Lilia Georgieva

Department of Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, Scotland

{mmaa10,P.Maier,P.W.Trinder,L.Georgieva}@hw.ac.uk

Abstract

We report a critical comparative evaluation of two functional profilers, ThreadScope and EdenTV, alongside four important imperative profilers. The comparison is based on the SICSA Concordance benchmark, covers both shared and distributed-memory parallel languages, and is performed on common parallel architectures.

We compare the runtime overheads of, 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, and find that the results reflect the design philosophy and maturity of the profilers.

Keywords Parallel Profiling, Profiling Overhead, Profiling Data Size, Performance Visualisation.

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.

Profiling has always been a key element of effective parallel programming: it is essential that the programmer understands the parallel behaviour in order to improve it. In languages that provide high-level parallelism, like most parallel functional languages, profiling is especially important. The high level abstractions mean that the conceptual gap between the program and parallel performance is greater.

In general profiling entails data collection, analysis and presentation. Data may be collected by tracing execution against time, or as some summary of the execution. A key challenge is to collect as much useful information for the programmer, with minimal intrusion on the parallel behaviour being observed. A profiler that significantly changes the parallel behaviour is no use. Presentation aims to provide understandable information to the programmer. Summative data is typically reported as text, and tracing data is often visualised as a number of graphical views.

The paper makes the following research contributions.

- We evaluate 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 [30]. The GHC Parallel Profiling System (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 OpenMP/MPI/CUDA and two summative profilers mpiP for MPI and ompP for OpenMP (Sections 2 and 3).
- 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. 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 4).
- 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. The study reveals some interesting results, e.g. both tracing functional profilers induce overheads an order of magnitude less than the imperative tracing profilers. A more complete account of our studies is available as a technical report [1](Section 5).
- We systematically compare the profilers for usability and data presentation, and find that the results reflect the design philosophy of the tools. Summative tools report a small set of key data with minimal intrusion. 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 6).

2. Background

This section briefly outlines the profiling process and introduces the functional and imperative profilers we study.

2.1 Performance Profiling Process

Performance profiling is broken into different stages where the output of one stage is the input for the next [6, 22]. As shown in Figure 1, the process consists of data collection, data analysis and data presentation.

Data Collection. The profiling process starts by collecting performance data from the executing program. The data collection process can take different forms e.g. statistical profiling, sampling or tracing. Statistical profiling counts and times various events during the program execution and derives statistics from these. Sampling

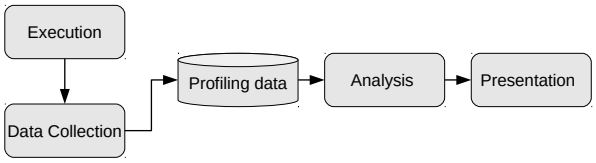


Figure 1. Performance Profiling Process.

takes snapshots at different time intervals of the execution. Tracing records execution events against time. Tracing is the most used approach for collecting more detailed and comprehensive information about the behaviour of the executing program. However, tracing is more intrusive than sampling and profiling. Tracing is implemented by either instrumenting the runtime system or the parallel application to emit trace events during the execution. A trace event consists of time stamp and a string that describes it. The emitted trace events will be stored in what is called a trace file.

Data Analysis. Next in the process is the analysis of performance data. The collected data, in particular trace events can consist of millions of trace events. In order to understand what happened during the execution these trace events need to be analysed. The analysis process starts by reading the raw trace events. After that the trace events can be sorted, categorised or grouped. In addition, counters and statistics can be derived. The results can be shown to the user instantly, saved into a file or feed to a presentation tool for visualisations.

Presentation. Presenting the performance data is about transforming the trace events into a form that reflects the execution behaviour of the parallel program. The data presentation can be graphical or text based. In the graphical case data presentation graphs such as Gantt charts or Kiviat diagrams are used to map trace events to a physical or logical computation resource e.g. a processor or a thread. In contrast, the text based presentation uses summaries, tables and statistics to provide information about the execution behaviour of the program.

2.2 Shared and Distributed Memory Profiling

Parallel performance profiling depends on the programming model of the profiled program. In distributed-memory parallelism, different processes work together to solve a problem communicating by messages passing. In addition, the processes may be located in physically separate places. In contrast, shared-memory parallelism, processes collaboratively solve a problem by reading from and writing to space shared between all the processes. Thus, 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.

2.3 Imperative Profiling

For decades imperative parallel languages have been supported by a variety of performance analysis tools. Early profilers, like ParaGraph [17], Pablo [32] and XPVM [21] provided parallel programmers with useful information about the parallel execution. Nonetheless, the field of parallel profiling is continuing and facing new challenges e.g. larger and highly scaled parallel systems and high-level parallel languages. These challenges encourage the research community to develop advanced performance tools like Vampir [40], Scalasca [11, 34] and TAU [36].

2.3.1 Score-P and Vampir

Score-P. Score-P [35] is a performance measurement infrastructures for parallel programming. Score-P is highly scalable and can support *high-performance computing* (HPC) facilities. It equips its users with tools for profiling, event tracing and online analysis of parallel application. In addition, Score-P can work with a number of performance analysis tools e.g. Vampir [40], Scalasca [11, 34]and TAU [36]. Score-P made this possible by adopting standardised output formats such as the Open Trace Format (OTF) [20], the CUBE4 profiling formats [10] and using instrumentation tools like Opari2 [28].

Vampir. Vampir [4, 29, 40] is a Graphical User Interface (GUI) which provides the capability to read, analyse and present graphically the performance monitoring data (see Figure 2) for different parallel imperative languages e.g. C or Fortran with MPI or OpenMP or CUDA. Vampir provides its users with multiple views in order to help them to understand the execution behaviour of their parallel programs with the capability to work on large scale computing infrastructures e.g. HPC.

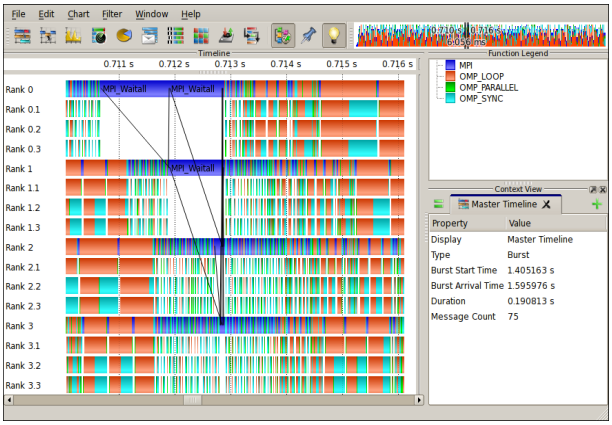


Figure 2. Screenshot of Vampir Visualising a Score-P Trace File.

2.3.2 mpiP

mpiP [41] is a profiler for MPI applications. mpiP monitors the performance of MPI by collecting statistical information about MPI functions from the MPI profiling layer. 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 GUI and does not provide performance graphs but outputs text profiles to show statistical information about the execution of the parallel program.

2.3.3 ompP

ompP [8, 9] is a performance analysis tool for shared-memory programming with OpenMP. ompP monitors the performance of the OpenMP application by collecting statistical data from the parallel execution. ompP is similar in spirit to mpiP [41], it has no GUI and it does not present performance graphs, instead it presents the performance information in a text profile.

2.4 Functional Profilers

Parallel functional languages also have been supported by performance analysis tools. hpcpp [33] was one of the earliest attempts to profile a parallel Haskell. GranSim [14, 23], on the other hand, was the first performance analysis tool for the parallel implementations of the Glasgow Haskell Compiler (GHC) i.e. GpH-GUM [39]

and GpH [31]. In addition, GranSim provided a variety of performance graphs such as overall activity, threads activity and granularity profile. Other variants of GranSim also had been developed, for example, GranSim-CC [15] for cost-centre profiling and GranSim-SP [19] to relate threads to the strategies which they were created with. Influenced by GranSim, more current parallel Haskells profilers have been developed utilising GUI i.e. EdenTV [3] and ThreadScope [38].

2.4.1 Eden Tracing and EdenTV

Eden Tracing. Eden tracing is the performance monitoring tool for the parallel Haskell Eden [24]. The runtime of GHC-Eden [12] 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 [32] and adopting its Self-Defining Data Format (SDDF) [2] for the trace files. On the other hand, Eden currently is adopting the new GHC-PPS Tracing and uses its GHC event log format (GHC-ELF) [18].

EdenTV. The Eden Trace Viewer (EdenTV) [3] 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 [24] from the level of the parallel runtime system. Two versions of the tool have been implemented: the first one was in Java; the current version [37] is implemented in Haskell and includes more features than the previous one (see Figure 3).

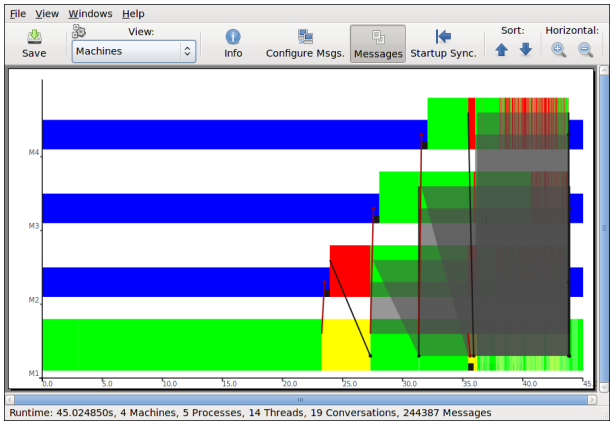


Figure 3. Screenshot of EdenTV Visualising an Eden Trace File.

2.4.2 GHC-PPS and ThreadScope

GHC-PPS. The GHC Parallel Profiling System (GHC-PPS) [18] is the current performance analysis tools for Glasgow Haskell Compiler (GHC) on multicores [16]. The GHC-PPS consists of tracing facility, analysis tools [13] and a GUI for browsing the trace events called ThreadScope [38]. 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 options (known as event logging in the Haskell community). This will produce a trace file (eventlog). The analysis tools are a Haskell library (GHC-Events) for reading and processing eventlogs. The library is extensible, so user of the GHC-PPS can develop custom analysis tools to satisfy their needs.

ThreadScope. ThreadScope [38] is the post-mortem trace analyser for (GHC-SMP) [16]. It is the standard tool to read, analyse and display performance data generated by the GHC-PPS (see Figure 4).

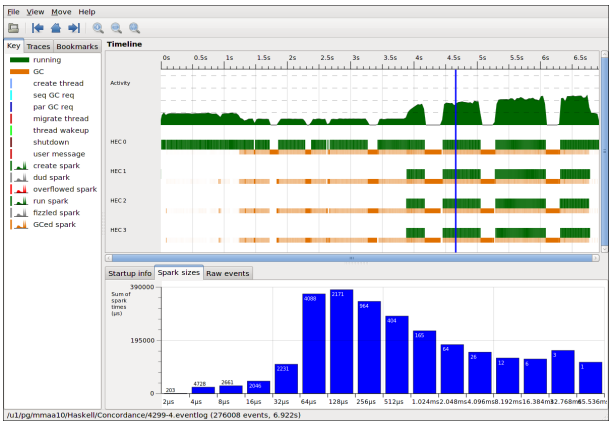


Figure 4. Screenshot of ThreadScope Visualising a GHC Trace File.

3. Experimental Methodology

3.1 Experimental Setup

The profilers are 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 are connected via Gigabit Ethernet and run Linux (CentOS 6.3 x86_64).

3.2 Concordance Benchmark Versions

The profilers are compared using implementations of the same Concordance benchmark that was published as Phase I of the SICSA MultiCore Challenge [30]. 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, developed for the SICSA MultiCore Challenge.

3.3 Measurements

We will use the Concordance benchmarks to measure the profiling data size and the profiling runtime overhead of the profiling tools. This is by profiling the execution of a Concordance application that matches the parallel platform of each profiling tool. We will report the median of 5 executions. In addition, we will use different metrics for measurements. Firstly, we will study how the increase in the computation size changes the performance of the these tools. Secondly, we will evaluate how the increase in the number of PEs affects these profilers.

To increase the computation size, we will use different sizes of input files. The SICSA MultiCore Challenge provides two input files: the smallest files is 35 KB and the largest is 4300 KB. In order 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 increasing the size by a factor of 2 until reaching 3200 KB. Our analysis will be 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 will increase the number of PEs by a factor of 2 starting by 1 PE until reaching 8 PEs as this is the maximum number of cores on our system. However, the MPI Concordance imple-

mentation requires a minimum of 2 PEs in to work: one as master and the other as worker. As consequence, this study will report the results based on the number of workers used in the computation. This is because, the master job is devoted to distributing work and waiting until all the workers have finished their jobs. Therefore, we do not think that it will introduce a significant profiling runtime overhead or a significant increase in the profiling data size. In addition, we will include the 6 PEs to the experiment. This is to validate our measurements for 8 PEs, as using all the available cores on a machine is known to sometimes perturb performance.

4. Profiling Data Size

This section investigates the amount of profiling data generated by the profilers.

4.1 Profiling Data Size in Relation to Computation Size

Score-P (MPI). Figure 5(a) shows how the tracing data size of Score-P changes as the input size increases. On the 1 PE curve the data size grows by 2991% from 23449.6 KB at 100 KB to 724889.6 KB at 3200 KB. Similarly, the data sizes of 2 PEs, 4 PEs, 6 PEs and 8 PEs increase from 100 KB to 3200 KB by about 3000%. As a result, we can say that 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 that ranges between 96% and 103%.

Eden Tracing. Figure 5(b) shows the how tracing data size of Eden tracing changes as the input size increases. All the curves show an increasing data size. On 1 PE the tracing data size grows by 2699% from 69.5 KB at 100 KB to 1945.6 KB at 3200 KB. Similarly, the trace data sizes of 2 PEs, 4 PEs, 6 PEs and 8 PEs increase from 100 KB to 3200 KB by 2946%, 7339%, 13309% and 30876% respectively. Therefore, the tracing data size increases dramatically as the input size gets bigger. Increasing the input size by a factor of 2 causes a change in the size of tracing data. The change ratios caused by the increase range between 39% and 528%. The ratios appear to be uniform on 1 and 2 PEs between 82% and 109% but beyond that the ratios are noisy.

Score-P (OpenMP). The increase in the profiling data size of profiling OpenMP with Score-P is similar to profiling MPI with Score-P. The data size grows by 3000% as the computation size increased from 100 KB to 3200 KB. 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 that ranges between 93% and 103%.

GHC-PPS. Figure 5(c) 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. On 1 PE the tracing data size grows by 7032% from 42.80 KB at 100 KB to 1024.00 KB at 3200 KB. Similarly, the tracing data sizes of 2 PEs, 4 PEs, 6 PEs and 8 PEs increase from 100 KB to 3200 KB by 1659%, 3087%, 4762%, and 2007% respectively. Consequently, we can say that 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 to the trace data size that ranges between 48% and 216%.

mpiP. Figure 5(d) shows that the profiling data size of mpiP does not change as the input size increases.

ompP. Our results show that the profiling data size of ompP does not change as the input size increases. This is expected because ompP is a summative profiler like mpiP; the graph for ompP can be found in the technical report [1].

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

Score-P (MPI). Figure 6(a) shows how the tracing data size of MPI Score-P changes as the number of PEs increases. On the 100 KB curve the tracing data size grows slightly by about 6.6% from 23449.6 KB at 1 PE to 24985.6 KB at 8 PE. Similarly, the tracing data size of the 200 KB, 400 KB, 800 KB, 1600 KB and 3200 curves increase slightly by about 6.1%, 6.5%, 7.6%, 8.2% and 8.2% respectively. Therefore, we can say that increasing the input size will result in a slight increase to the tracing data size when profiling MPI with Score-P. Increasing the number of PEs by a factor of 2 will result in a slight increase to the size of the tracing data that ranges between 1.5% and 3.1%.

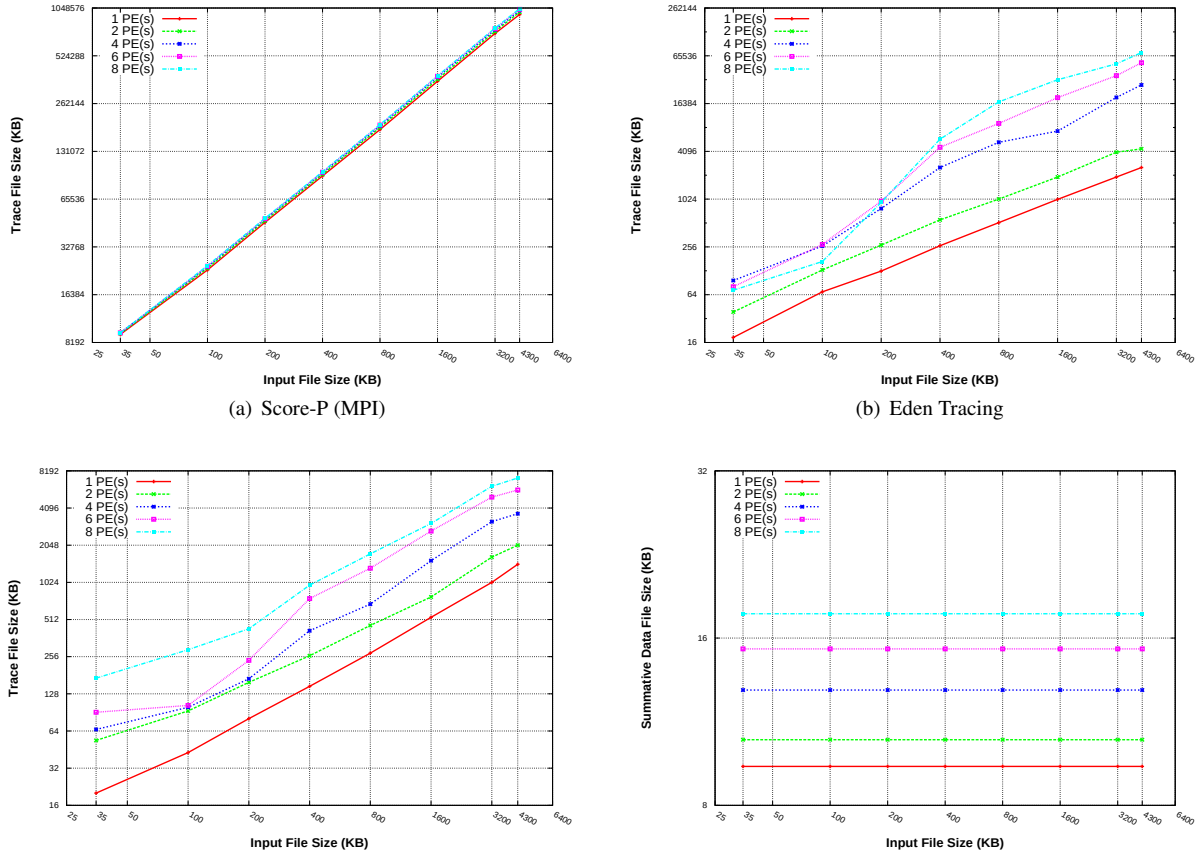
Eden Tracing. Figure 6(b) depicts how the increase in the number of PEs changes the trace data size of Eden tracing. All curves show an increase in data size. On the 100 KB curve the tracing data size increases by about 141% from 69.5 KB at 1 PE to 167.6 KB at 8 PEs. Likewise, the tracing data size of the 200 KB, 400 KB, 800 KB, 1600 KB and 3200 curves increase by about 632%, 2103%, 3222%, 3116% and 2568% respectively. However, we noticed that the 35 KB, 100 KB, and 200 KB curves tail off at 4 PEs, and may even decrease beyond 4 PEs. We think that the input size is too small to effectively use more than 4 PEs, and that the data points at 35 KB, 100 KB and 200 KB beyond 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). The increase in the profiling data size of profiling OpenMP with Score-P is similar to profiling MPI with Score-P. Increasing the number of PEs only results in a slight increase in the profiling data size. 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 0.0% and 4.0%.

GHC-PPS. Figure 6(c) 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. On the 100 KB curve the tracing data size increases by 581.1% from 42.8 KB at 1 PE to 291.5 KB at 8 PEs. Similarly, the tracing data sizes of the 200 KB, 400 KB, 800 KB, 1600 KB and 3200 curves increased by 434%, 559%, 537%, 475% and 500% respectively. 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. As noted above for Eden tracing, this effect is 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 that is ranging between 46% and 155%.

mpiP. Figure 6(d) 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. For all curves the summative data size increases by 88% from 9.4 KB at 1 PE to 17.7 KB at 8 PEs. Increasing the number of PEs by a factor of 2 changes the profiling data by between 11.7% and 37.2%.

ompP. Our results shows that increasing the number of PEs changes the size of the profiling data of ompP. The results of ompP are similar to the results of mpiP; the graph for ompP can be found in the technical report [1]. For all the data sets the profiling data size increased by about 55% from 5.3 KB at 1 PE to 8.2 KB at 8 PEs. Increasing the number of PEs by a factor of 2 changes the profiling data size by between 7.5% and 24.2%.



(a) Score-P (MPI)

(b) Eden Tracing

(c) GHC-PPS

(d) mpiP

Figure 5. Profiling Data Size in Relation to Computation Size.

4.3 Profiling Data Size Discussion

This section compares the profiling data size of the parallel profilers reported in sections 4.1 and 4.2. The goal of this comparison is to see how the functional profilers compare to the imperative profilers in terms of profiling data size. Because of the shortage of space, we will only report the comparison results on increasing the input size with a fixed number of 4 PEs. Comparisons with other fixed numbers of PEs, and comparisons with fixed input sizes show similar results, which can be found in the technical report [1].

Figure 7 compares 6 profiling tools in terms of how increasing the input size changes the profiling data size on a fixed number of PEs. Table 1 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 7.

We base the following comparison on the mean values from Table 1(a) and Table 1(b). We observe that the profiles generated by Score-P are about two orders of magnitude bigger than those generated by the functional profilers, which in turn are two to three orders of magnitude bigger than the data generated by the summative profilers.

Distributed Memory: Imperative vs Functional. We are comparing the profiling data sizes of the distributed-memory profilers Score-P (MPI) and Eden tracing. According to Table 1(a) the mean profiling data size of Score-P (MPI) is 316428.8 KB whereas the

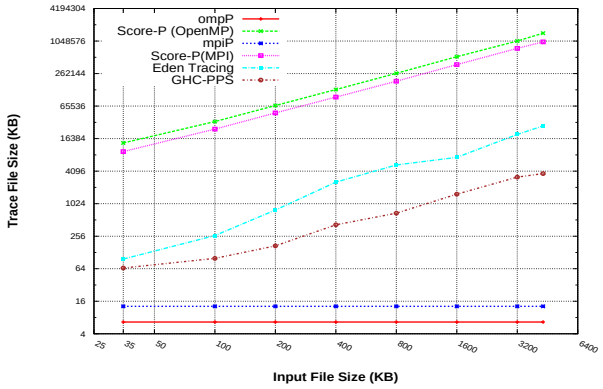
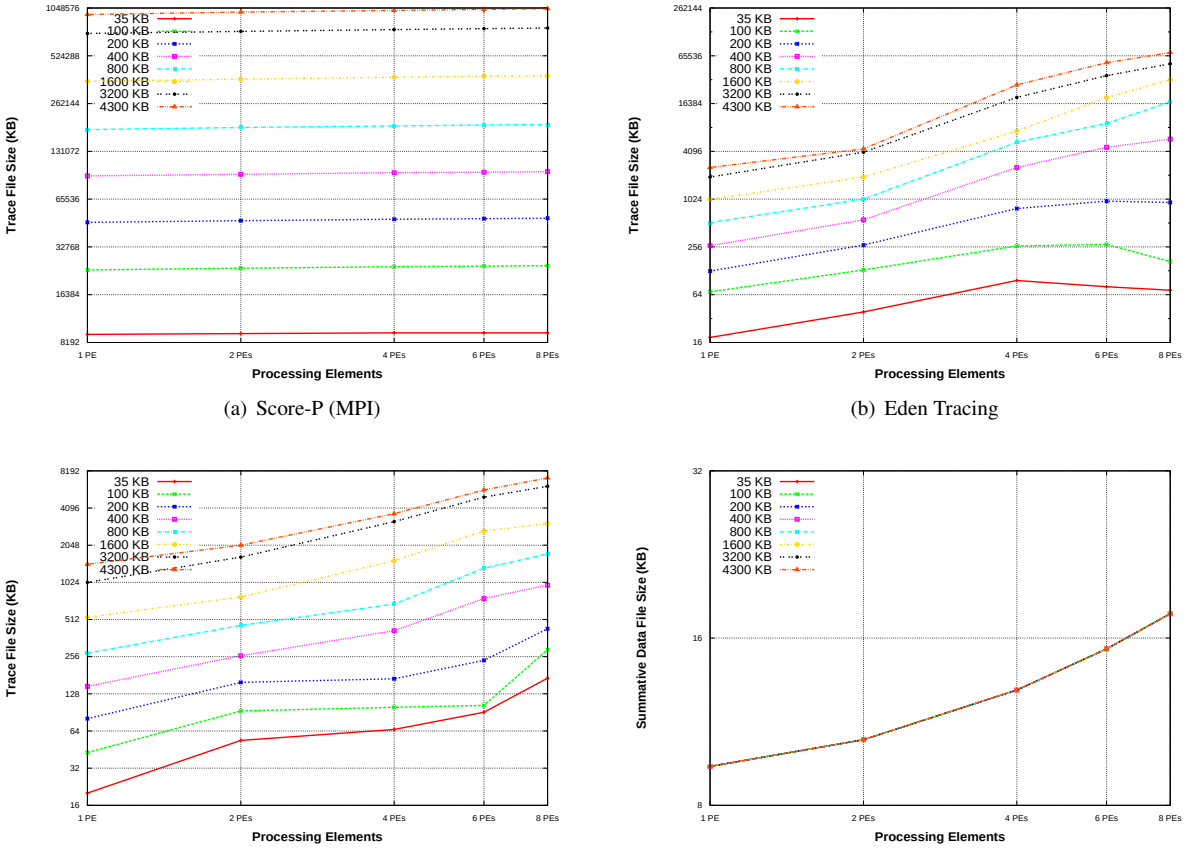


Figure 7. Synopsis of Profiling Data Sizes in Relation to Computation Size (on 4 PEs).

mean profiling data size of Eden tracing is 8014.4 KB. The functional Eden tracing generates significantly smaller profiles than the imperative Score-P (MPI).

Shared Memory: Imperative vs Functional. Comparing the profiling data sizes of the shared-memory profilers Score-P (OpenMP)



(a) Score-P (MPI)

(b) Eden Tracing

(c) GHC-PPS

(d) mpiP

Figure 6. Profiling Data Size in Relation to Number of PEs.

(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 1. Minimum, Mean and Maximum Profiling Data Sizes (on 4 PEs).

and GHC-PPS, we find an even larger difference between the functional and the imperative profilers. The mean profiling data size of GHC-PPS is 1228.7 KB whereas the mean profiling data size of Score-P (OpenMP) is 445952 KB, see Table 1(b).

Trace Based vs Summative. As Figure 7 and Table 1 illustrate, the summative profiling tools require significantly smaller storage space for the profiling data than trace based profiling tools. This is expected since the summative profiling tools only summarise the parallel execution behaviour in a text format. This type of profiling tools does not require large storage space.

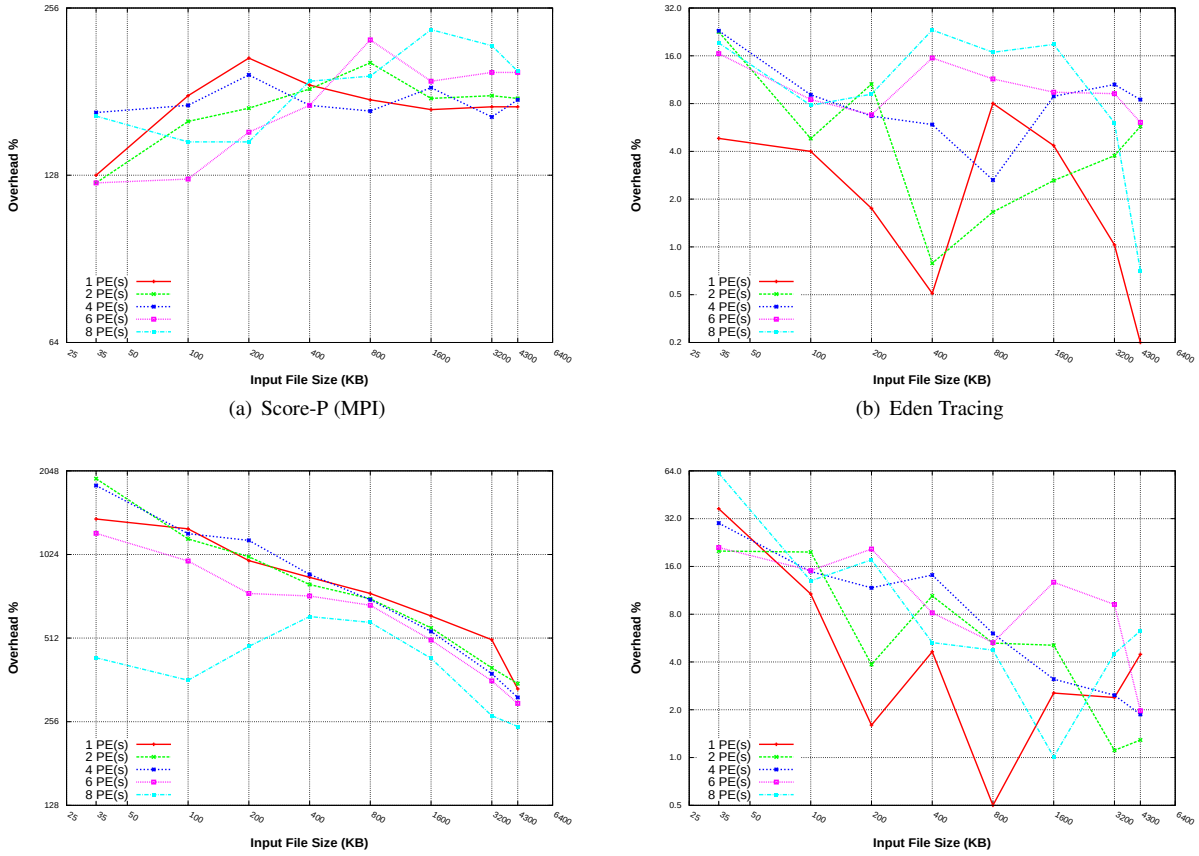
5. Runtime Overheads of Profiling

This section investigates the runtime overheads of the profilers.

5.1 Runtime Overhead in Relation to Computation Size

Score-P (MPI). Figure 8(a) 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 124% and 219%.

Eden Tracing. Figure 8(b) 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 decreases as the input size increases. Overall, the relative runtime overhead of Eden tracing remains between 0.25% and 23%.



(a) Score-P (MPI)

(b) Eden Tracing

(c) Score-P (OpenMP)

(d) GHC-PPS

Figure 8. Relative Runtime Overhead in Relation to Computation Size.

Score-P (OpenMP). Figure 8(c) shows the change of relative runtime overhead of Score-P (OpenMP). There is a drop of all curves as the input size increases. On 1 PE the overhead decreases by about 761% points from 1266% at 100 KB to 505% at 3200 KB. Similarly, the overheads of the 2 PEs, 4 PEs and 6 PEs decrease from 100 KB to 3200 KB by 765% points, 836% points, 611% points and 92% points respectively. However, the 8 PEs curve is an outlier and we think that this is because of 8 is 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 8(d) shows how the relative runtime overhead of GHC-PPS changes as the input size increases. The data is noisy; all curves fluctuate widely. To some degree, 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 remains between 0.50% and 19.70%.

mpiP. The runtime overhead graphs for mpiP and ompP can be found in the technical report [1]. The graphs for mpiP show that the relative runtime overhead of mpiP grows by between 57% and 233% points as the input size increases. Overall, the relative runtime overhead remains between 135% and 378%.

ompP. The graphs for the relative runtime overhead of ompP are noisy. Similar to GHC-PPS, they overhead decreases as the input size increases. Overall, the relative runtime overhead remains between 0.35% and 14.96%.

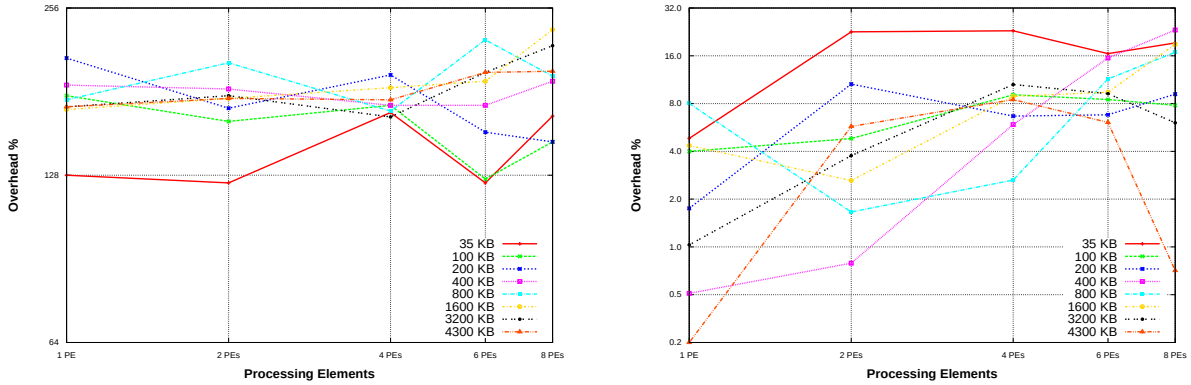
5.2 Profiling Overhead in Relation to Number of PEs

Score-P (MPI). Figure 9(a) shows how the relative runtime overhead of Score-P (MPI) changes as the number of PEs increases. As the Figure illustrates, the data is noisy beyond 4 PEs. It appears that changing the number of PEs does not change the relative overhead but increases variability.

Eden Tracing. Figure 9(b) shows how the relative runtime overhead of Eden tracing [3] 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.

Score-P (OpenMP). Figure 9(c) 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.

GHC-PPS. Figure 9(d) shows the relative runtime overhead of GHC-PPS [18]. The data is noisy; similar to Eden tracing, GHC-



(a) Score-P (MPI)

(b) Eden Tracing

(c) Score-P (OpenMP)

(d) GHC-PPS

Figure 9. Relative Runtime Overhead in Relation to Number of PEs.

PPS appears to contribute significant variability to the overheads. However, unlike with Eden tracing, there is no clear trend showing an increase in relative overheads with increasing number of PEs.

mpiP. The runtime overhead graphs for mpiP and ompP can be found in the technical report [1]. The graphs for mpiP are similar to those for Score-P (MPI) and show that the relative runtime overhead of mpiP [41] does not change as the number of PEs increases. However, like Score-P, mpiP contributes to variability in the overheads.

ompP. The overhead graphs for ompP show a picture similar to the one for GHC-PPS. The data is noisy, with high variability in overheads; there is no clear trend towards an increase in overheads with increasing number of PEs.

5.3 Runtime Overhead Discussion

In this section we will discuss the results presented in sections 5.1 and 5.2. The aim of this discussion is to compare the functional profilers to the imperative profilers in terms of relative runtime overhead. Because of space constraints, we will only report the comparison results on increasing the computation size with a fixed number of 4 PEs. Comparisons with other fixed numbers of PEs, and comparisons with fixed input sizes show similar results, which can be found in the technical report [1].

Figure 10 shows a comparison of the relative runtime overhead between 6 profiling tools. To demonstrate how the overheads of these profiling tools compare to each other we also summarised the minimum, mean, and maximum values of the relative overhead for each curve from Figure 10 into Table 2(a) and Table 2(b). In

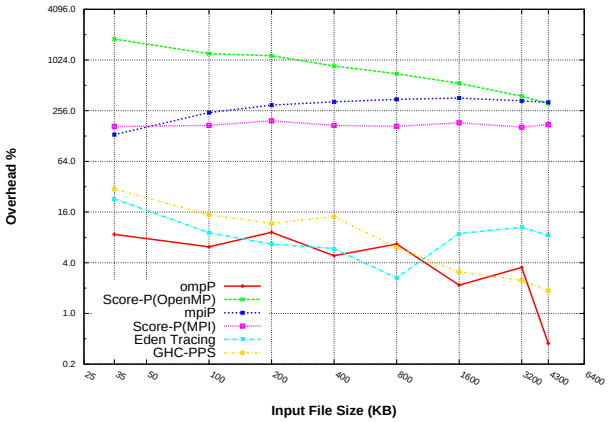


Figure 10. Synopsis of Relative Runtime Overheads in Relation to Computation Size (on 4 PEs.)

(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 2. Minimum, Mean and Maximum Relative Runtime Overheads (on 4 PEs).

this comparison we will use the mean value from the table for each curve.

The figures show that profilers are clearly divided into two groups, one with high overheads (over 100%) and one 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 latter 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 higher variability of overheads.

Distributed Memory: Imperative vs Functional. We first compare the relative runtime overheads of the distributed-memory profilers Score-P (MPI), mpiP and Eden tracing. Table 2(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.

Shared Memory: Imperative vs Functional. We next compare the relative runtime overheads of the shared-memory profilers Score-P (OpenMP), ompP and GHC-PPS. The mean overheads of Score-P (OpenMP), ompP and GHC-PPS are 873.50%, 5.2% and 10.5%, respectively, see Table 2(b). Surprisingly, the overhead of Score-P (OpenMP) is almost two orders of magnitude higher than GHC-PPS and ompP.

Trace Based vs Summative. Finally, we compare the relative runtime overheads of summative profilers to the overheads of trace based profilers. 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 as high as the overheads for the trace bases Score-P (173% on MPI).

6. Data Presentation and Visualisation

In this section we will present a comparison between the performance data presentation tools of functional profilers and imperative profilers i.e. EdenTV [3], ThreadScope [38] Vampir [40], mpiP [41] and ompP [8]. We will critically compare the main features and facilities of these tools. In particular, we are trying to see 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 summaries how visualisation tools of functional profilers compare to imperative profilers.

	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. Synopsis of Visualisation Tools.

6.1 Programming Model

As Table 3 (Programming Model) illustrates, there are differences between these tools. Vampir can visualise the performance data of multiple programming models e.g. MPI, OpenMP, MPI+OpenMP or MPI+Accelerator. mpiP can only support MPI. ompP mainly supports OpenMP but can also profile hybrid applications e.g. MPI+OpenMP. EdenTV only supports the distributed-memory parallel Eden. Likewise, ThreadScope only supports the shared-memory Haskell (GHC-SMP).

In terms of the variety of programming models and parallel languages that these tools support, we can say that Vampir is more flexible. Moreover, we have noticed that 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.

6.2 Presentation of Performance Data

Vampir, EdenTV and ThreadScope provide the user with a browser to visualise graphically the performance data. However, mpiP and ompP do not provide such a facility to the user; instead the performance data is summarised into a text file which can be read by any standard text editor. We think that 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.

We found that Vampir, EdenTV and ThreadScope all provide both types graphical views and textual views to their users. However, mpiP and ompP only present statistical information in the form of text profiles which we think is a shortcoming of these two tools.

6.3 Software Properties

Performance Visualisation tools are Software systems which are used by programmers to tune and improve the behaviour of their parallel programmes. 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 (see Table 3 (Software Properties)).

License. As Table 3 illustrates, the only proprietary visualisation tools is Vampir; mpiP, ompP, EdenTV and ThreadScope are all open source software.

Interoperability. Interoperability is the property of the software system to be able to work with the products (outputs) of other software systems. Here our concern is the ability of the visualisation tool to process trace files from different trace generation tools. We found that Vampir is the only interoperable tool. Vampir can process trace files of the format OTF2 which is the standard trace format adopted by performance monitoring tools e.g. Score-P and VampirTrace. In contrast, EdenTV and ThreadScope are not interoperable visualisation tools. The is because EdenTV can only process trace files generated by GHC-Eden. Similarly, ThreadScope cannot process trace format other than GHC-ELF. Therefore, both EdenTV and ThreadScope are not interoperable tools.

Heterogeneity. Heterogeneity is the ability of the visualisation tools to visualise the performance of heterogeneous parallel applications e.g. MPI+OpenMP+CUDA. We found that, 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. 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. On the other hand, mpiP, EdenTV and ThreadScope are not heterogeneous.

Scalability. Scalability is the ability of the visualisation tool to show the performance of executions on a large number of processors. In particular, scalability is an important issue for visualising the performance of distributed-memory applications since the number of processors on distributed-memory is growing 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. 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. In addition, mpiP can scale up to 65536 processes [41]. 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 will not be able to open it. Therefore, EdenTV is not scalable.

6.4 Usability

We will focus on three main facilities which we think important to help identify performance problems. These are zooming in and out 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. As Table 3 (Usability) illustrates, all profiling tools with GUI provide zooming facilities i.e. Vampir, EdenTV and ThreadScope. However, the graph zooming of 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 (Displays/Views). However, we emphasise that Vampir provides more views than the other profiling tools.

6.5 Summary

We found that functional profilers provide good facilities for identifying parallel performance problems. However, comparing them with imperative profilers shows that functional profilers can benefit from lessons learned from developing imperative profilers. Our study shows that imperative profilers are more advanced, 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. Finally, even though EdenTV and ThreadScope are both visualisation tools for two variants of the functional language Haskell, they have different trace file extentions which make them incompatible systems.

7. Related Work

Chung *et al.* [5] investigate how to reduce the cost of tracing by tracing by selectively recording only certain classes of events using a set of standard HPC profiling tools. They evaluate their approach with an experimental study of the cost of five profiling tools: IBM HPCT, Paraver, KOJAK, TAU and mpiP. Similar to our work, they use 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, [5] is restricted to one programming model, imperative programming with MPI, whereas we cover a range of different programming models (shared vs distributed memory) and paradigms (imperative vs functional). However, [5] uses 4 benchmark applications, whereas we are limited to one because it is difficult to find multiple and similar benchmarks for all the programming models we consider. Finally, [5] investigates the cost of profiling on larger scales than we do. We were limited to small numbers of processors, because we compare profiling tools for both distributed and shared memory, inheriting the low processor limit of shared-memory architectures.

Malony *et al.* [26] investigate overhead compensation in a prototype extension of the TAU [36] 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.* [18] introduce 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.

8. Discussion

We have evaluated two functional profilers, GHC-PPS and EdenTV, alongside four important imperative profilers. The comparison is based on the SICSA Concordance benchmark [30], covers both shared and distributed-memory parallel languages, and is performed on common parallel architectures.

Unsurprisingly the summative profilers generate far less profiling data. More interestingly both functional tracing profilers generate one or two orders of magnitude less data than the imperative tracing profilers. While generating so much data may intrude on the parallel execution, the benefit is that tools like Score-P/Vampir can provide additional information to assist the programmer. (Section 4).

Both tracing functional profilers induce very low runtime overheads: an order of magnitude less than the imperative tracing profilers. Indeed the functional profiler overheads are no more than a factor of two greater than the imperative summative profilers, i.e. 296% for mpiP compared with 9.4% for EdenTV, and 5.2% for ompP compared with 10.5% for GHC-PP (Section 5).

Comparing the profilers for usability and data presentation, we see that the functional profilers are relatively immature 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 6).

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 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 can 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. It is unclear how much profiling technologies can be shared by the various parallel Haskell DSLs like the Par Monad [27], Cloud Haskell [7], and HdpH [25].

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 bandwith available to each core shrinks. The challenge of heterogeneity is to profile a program executing on a range of compute resources, e.g. multicores and GPUs.

References

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

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

[3] 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, Sept. 2007.

[4] 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 US, 2005. ISBN 978-0-387-23096-2.

[5] 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, nov. 2006.

[6] A. Couch. Locating performance problems in massively parallel executions. *Proceedings of the IEEE*, 81(8):1116–1125, aug 1993. ISSN 0018-9219.

[7] 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.

[8] 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 Berlin / Heidelberg, 2008.

[9] 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, Sept. 2008.

[10] 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*.

[11] 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. ISSN 1532-0634.

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

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

[14] 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.

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

[16] T. Harris, S. Marlow, and S. P. 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. ISBN 1-59593-071-X.

[17] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *Software, IEEE*, 8(5):29–39, sept. 1991. ISSN 0740-7459. doi: 10.1109/52.84214.

[18] 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. ISBN 978-1-60558-508-6.

[19] 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, Sept. 1998.

[20] A. Knpfer, 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 Berlin Heidelberg, 2006. ISBN 978-3-540-34381-3.

[21] 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 vol.1, 1996.

[22] E. Kraemer and J. Stasko. The Visualization of Parallel Systems: An Overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, 1993. ISSN 0743-7315.

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

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

[25] 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 Berlin Heidelberg, 2012. ISBN 978-3-642-34406-0.

[26] 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. ISSN 1094-3420.

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

[28] 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.

[29] 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.

[30] Phase I The SICSA MultiCore Challenge. Concordance application. `www.macs.hw.ac.uk/sicsawiki/index.php/Challenge_PhaseI`, 2010. Accessed on: November 2012.

[31] R. F. Pointon, P. W. Trinder, and H.-W. Loidl. The Design and Implementation of Glasgow Distributed Haskell. In *IFL'00 — Intl. Workshop on the Implementation of Functional Languages*, volume 2011 of *Lecture Notes in Computer Science*, pages 53–70, Aachen, Germany, Sept. 2000. Springer. URL `http://www.macs.hw.ac.uk/ dsg/gph/papers/ps/ifl00.ps`.

[32] 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.

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

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

[35] Score-P. Scalable Performance Measurement Infrastructure for Parallel Codes. `http://www.vi-hps.org/projects/score-p`. Accessed on: January 2013.

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

[37] B. Struckmeier. Implementierung eines Werkzeugs zur Visualisierung und Analyse paralleler Programmläufe in Haskell. Master's thesis, Philipps-Universität Marburg, Germany, In German, 2006.

[38] ThreadScope. A Graphical Timeline Browser for GHC Trace Files. `http://www.haskell.org/haskellwiki/ThreadScope`, 2013. Accessed on: May 2013.

[39] 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 *PLDI '96, Philadelphia, USA*, pages 78–88. ACM Press, 1996.

[40] Vampir. Performance Optimization. `http://www.vampir.eu/`. Accessed on: January 2013.

[41] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling. `http://mpip.sourceforge.net/`. Accessed on: September 2012.