Contribution to automatic performance analysis of parallel applications

François Trahay

INSTITUT POLYTECHNIQUE DE PARIS

Habilitation à Diriger les Recherches

TELECOM SudParis

IP PARIS

Contribution to automatic performance analysis of parallel applications

Habilitation à Diriger les Recherches de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité : Informatique

Soutenue à Palaiseau, le 29/06/2021, par

FRANÇOIS TRAHAY

Composition du Jury :

Pascal FELBER Professeur, Université de Neuchâtel	Rapporteur
Brice GOGLIN Directeur de recherche, INRIA Bordeaux Sud-Ouest	Rapporteur
Lionel SEINTURIER Professeur des universités, Université de Lille	Rapporteur
Raymond NAMYST Professeur des universités, Université de Bordeaux	Examinateur
Gaël THOMAS Professeur, Télécom SudParis	Examinateur

Contents

1 Introduction **1**
- 1.1 Hardware resources become complex . . . 1
- 1.2 Applications mix programming models . . . 2
- 1.3 Contributions . . . 2
  - 1.3.1 Collecting performance data . . . 3
  - 1.3.2 Analyzing performance data . . . 3
  - 1.3.3 Remainder of this document . . . 4

2 Collecting performance data **5**
- 2.1 Collecting execution traces with EZTrace . . . 5
  - 2.1.1 Plugin-based tracing tool . . . 6
  - 2.1.2 Instrumentation . . . 7
  - 2.1.3 Trace generation . . . 8
- 2.2 Collecting memory accesses with NumaMMA . . . 9
- 2.3 Conclusion . . . 10

3 Analyzing performance data **13**
- 3.1 Detecting the structure of a trace . . . 14
  - 3.1.1 Related work . . . 14
  - 3.1.2 Contribution . . . 15
- 3.2 Differential execution analysis . . . 17
  - 3.2.1 Related work . . . 17
  - 3.2.2 Contribution . . . 19
- 3.3 Detecting scalability issues with ScalOMP . . . 20
  - 3.3.1 Related work . . . 21
  - 3.3.2 Contribution . . . 21
- 3.4 Conclusion . . . 22

4 Ongoing and future works **23**
- 4.1 Performance analysis of data management . . . 23
- 4.2 Feeding runtime and compilers with performance data . . . 24
- 4.3 Analyzing high performance data analytics applications . . . 25

Personal bibliography **27**

Bibliography **31**

Chapter 1

Introduction

The development of parallel computing over the last decade has changed the landscape of computer science. While parallelism was mostly found in niche domains such as high performance computing, the evolution of hardware has democratized the access to parallel machines. As a result, multiple research domains have faced breakthroughs thanks to the availability of parallel computing power: bioinformatics can now compare genomics efficiently [26], artificial intelligence develop machine learning algorithms capable of inferring model from millions of data [24]. In order to exploit modern parallel architectures, developers face several challenges.

1.1 Hardware resources become complex

For decades, parallelism was mostly found in supercomputers that interconnect many sequential machines through a network. Processing data in parallel then required to distribute the computation and to exchange data over the network. Thanks to the development of multicore processors in the 2000s, parallel programming paradigms that rely on shared memory rapidly grew. Since then, on chip parallelism increase continuously. Multicore processors now commonly include dozens of cores, each core can run several threads with Simultaneous MultiThreading (SMT), and each processing unit implements large vector instructions (such as AVX-512). From the application point of view, this increase in the number of processing units allows for finer grain parallelism. However, exploiting efficiently all the computing resources requires that the program exposes enough parallelism.

In the meantime, accelerators that consist of thousands of simple CPUs have become widespread. 149 of the top 500 supercomputers are now equipped with accelerators, such as NVidia GPUs [5]. Accelerators provide supercomputing center with an energy efficient computing power. However, the transition to heterogeneous architectures makes the development of parallel applications complex since they need to manage data transfers and synchronizations between the host and the device.

Overall, the increase in the number of processing units increases the need for a performant data management subsystem. To mitigate the memory wall, processors now integrate large cache

memories, and multiple memory controllers are distributed to form Non-Uniform Memory Access (NUMA) architectures. To avoid memory bottlenecks, threads and memory objects have to be carefully placed so that most memory accesses hit caches or local memory.

Storage is the other type of data management system that changed greatly over the past decade. While Hard Disk Drives (HDD) take several milliseconds to fetch data, leaving the application stalled for millions of CPU cycles, hardware manufacturers invented new types of permanent storage with lower latencies and higher throughput. Solid State Drives (SSDs) lower the fetch latency to a few dozens of microseconds, or even a few microseconds for NVMe SSDs. This can significantly improve the performance of some applications. This also results in software becoming the main source of overhead of the I/O stack.

Understanding and optimizing the performance of an application running on a modern parallel computer thus requires a fine knowledge of low level details. Many hardware components can become the source of performance problems.

1.2 Applications mix programming models

In addition to the complexification of the hardware, the software stack evolved to adapt to the increasing parallelism of computers. Clusters of single core computers were commonly exploited using a message-passing paradigm such as MPI, where workers are distributed over the compute nodes, and exchange message over the network. The introduction of multicore processors and accelerators has permitted new types of paradigms to appear. Instead of running several MPI ranks per multicore machines, which increases the number of MPI ranks, duplicates memory, and requires to allocate additional network buffers, it is possible to mix a distributed model like MPI with a shared memory model. For instance, each MPI process can spawn OpenMP threads to process parallel region and to exploit multiple cores. Such composition of models exploits all the processing units while limiting the memory overhead of single-model paradigms like MPI. Similarly, clusters of heterogeneous computers can be exploited by mixing MPI with accelerators models. For instance, CUDA or OpenCL allow developers to describe blocks of code that run on an accelerator (such as a GPU), and provide primitives for explicitely transfering data from the main memory to the accelerator memory.

Due to the complex software stack that mixes programming paradigms, developing a parallel application that efficiently exploits the hardware is tedious. Performance problems may be located in multiple pieces of software, or be due to bad interactions between programming models.

1.3 Contributions

Understanding and improving the performance of a parallel application is difficult for a developer who is most of the time expert in a specific application domain. My research activities aim at designing tools that relieve developers from the burden of analyzing performance. Over the last ten years, my research has focused on two phases of performance analysis: collecting

data when an application executes [C3], [R2], [W2], [C10], and analyzing the collected data in order to understand and to improve the performance of applications [W1], [J2], [C6].

This research has been conducted with three PhD students (one of which having already defended), one post-doc researcher, three master students during their internship, and 21 master students who collaborated as part of their master project.

1.3.1 Collecting performance data

Understanding the performance of an application is difficult. It necessitates performance collection tools that are able to capture the application behavior as well as low level metrics that indicate how the hardware resources are exploited.

To capture the execution of a parallel application, we developed a tracing tool named EZTrace [C10]. EZTrace is a convenient way to generate execution traces of parallel applications. It automatically instruments a pre-defined set of functions (that corresponds to the main paradigms used in parallel programming such as MPI, OpenMP, or CUDA) [W2] and records events in a trace each time the application calls the instrumented functions [R2]. In addition to providing plugins for the main parallel programming libraries, EZTrace uses a plugin generation mechanism that allows developers to easily create new plugins to instrument their application functions [C10].

As application memory access pattern may significantly impact performance, we also developed NumaMMA [C3], a lightweight memory profiler that captures the memory accesses of threads and relates them to the memory objects allocated by the application. The collected data is processed in order to measure the impact of memory objects on performance, and to detect how threads access these objects.

1.3.2 Analyzing performance data

The performance data collected during the application execution have to be analyzed in order to spot the cause of performance issues. Users can browse the collected data with text tools, or with visualization techniques [W3]. However, analyzing data automatically simplifies this task.

In order to process large traces that contain millions of events, we designed an algorithm that detects the structure of a program from a sequential trace [C6]. Once the sequences of events that repeat in a trace are detected, we compare them and filter out those with similar duration in order to reduce the quantity of information that users have to manually analyze. In another work [J2], we use a differential analysis technique that compares repetitive sequences of events in order to find several kinds of bottlenecks (lock contention, I/O or network contention, memory contention, etc.) in applications, and to estimate their impact on the application run time.

While the previous works are generic and do not address a particular programming model, we also study in details the OpenMP paradigm. We propose a simple methodology for analyzing OpenMP applications with a tool called ScalOMP [W1]. This tool instruments OpenMP

applications to collect performance data. It identifies various scalability issues (such as load imbalance, lock contention, or task granularity), and suggest optimizations to the application developer.

1.3.3 Remainder of this document

The remainder of this document is organized as follows. Chapter 2 presents our work on performance data collection tools, including EZTrace and NumaMMA. In Chapter 3, we present several works on automatic performance analysis: the detection of a program structure from a trace, the detection of contention that uses differential analysis, and the design of the scalability analysis tool for OpenMP programs. Finally, Chapter 4 concludes this document and discusses ongoing and future work.

Chapter 2

Collecting performance data

Contents

2.1 Collecting execution traces with EZTrace 5

2.1.1 Plugin-based tracing tool 6

2.1.2 Instrumentation 7

2.1.3 Trace generation 8

2.2 Collecting memory accesses with NumaMMA 9

2.3 Conclusion 10

Collecting performance data of an application is the first step of the performance analysis process. This chapter describes several performance collection tools that we designed. These tools were designed with several constraints:

- They must require as little configuration effort as possible from the user. For instance, recompiling the application and its library in order to instrument them should be avoided;
- Observing an application should not affect its behavior;
- The tools should collect enough data to make the analysis useful.

In this chapter, we present several performance data collection tools that we developed over the past ten years. Section 2.1 present a tracing tool named EZTrace. Section 2.2 describes NumaMMA, a memory profiler.

2.1 Collecting execution traces with EZTrace

Understanding precisely the behavior and the performance of a parallel application is tedious. The complexity of a supercomputer hardware as well as the use of various programming models like MPI, OpenMP, MPI+threads or MPI+GPUs makes it more and more difficult to understand

the performance of an application. For decades, various tools have been developed in order to help developers understand their application performance. Two main kinds of tools exist:

Profiling tools. Profiling tools collect performance data at runtime and report the aggregate data [4], [23], [66], [75], [104], [107], [117]. For instance, profilers such as Perf [66], Oprofile [104], GProf [117] report the numbers of calls to a function, or the average time spent in that function. Other tools like PAPI [107], or Likwid [75] collect various performance counters (number of cache misses, number of floating point operations, etc.) Profiling tools usually collect data with a low overhead. However, since the collected data is aggregated over the application lifetime, analyzing precisely a performance problem is complex.

Tracing tools. Tracing tools collect timestamped performance data and generate execution traces that depict the status of threads during the application lifetime [2], [3], [67], [79], [82], [91], [99], [113], [116]. The execution traces can be analyzed *post-mortem* using a visualization tool[3], [30], [W3], [113]–[115], or using an analysis tool that detects typical performance problems [3], [67]. Since the collected events describe precisely how the application behaves, tracing tools allow in-depth analysis of performance problems. However, collecting events can degrade the performance of applications and may generate large trace files that are difficult to analyze.

The use of tracing tools is a great help for understanding the performance of an application. However, the variety of scientific libraries and programming models makes it mandatory for such tools to be generic. Instrumenting an application with these tools can be tedious because it requires to modify the source and to recompile the program. Allowing easy instrumentation of any kind of library or application is crucial in order to work on most modern platforms and to meet the requirements of emerging programming models.

Contribution. We developed EZTrace[1] [C10], a generic framework for performance analysis. EZTrace uses a two phases mechanism based on plugins for tracing applications. This permits to specify easily the functions to analyze and the way they should be represented. Moreover, EZTrace provides an easy to use script language that allows the user to instrument functions without modifying the source code of the application.

The contribution of the EZTrace project are threefold. First, we developed a generic plugin-based tracing tool that allows users to trace multiple libraries simultaneously. Second, we designed a light instrumentation mechanism that does not necessitate to recompile the application and its dependencies. Third, we created a compact trace format for recording events with a low overhead.

2.1.1 Plugin-based tracing tool

When we started developing EZTrace in 2010, most tracing tools were dedicated to a single programming model like MPI [113], [116], or OpenMP [110]. While some tools permitted

[1] Available as open source at `https://gitlab.com/eztrace/eztrace`

to instrument user-defined functions [91], [99], probes had to be manually inserted into the application source code.

EZTrace relies on plugins for selecting the libraries to be analyzed [C10]. Each pre-defined plugin is in charge of analyzing a specific library (*e.g.* MPI, OpenMP, Pthread, or CUDA) by intercepting calls to a set of function (*e.g.* `MPI_Send`, `MPI_Recv`, etc.) and recording events. When running an application with EZTrace, a user can select one or several EZTrace plugins depending on the programming models used by the application.

Additionally, a user can define a new plugin for their application or library. This can be done either by writing the C code of the plugin, or by using a Domain Specific Language that describes the functions to intercept and their meaning. As a result, external projects such as the Plasma linear algebra library [77], or the NewMadeleine communication library [C4], [88] have developed EZTrace plugins for instrumenting their source code.

In order to define a plugin, a user has to provide the prototype of the functions to instrument. EZTrace provides a tool that automates this step by extracting information from the debugging symbols. As a result, EZTrace can automatically generate a plugin for instrumenting all the functions of an application.

2.1.2 Instrumentation

In order to record events, tracing tools need to instrument the application by inserting probes (*e.g.* at the beginning/end of functions of interest). Several instrumentation mechanisms are commonly used.

Manual modification of the application source code. A developer can modify the source code of a program to insert probes in order to analyze specific parts of the program execution. Many tracing tools provide a means to manually insert probes [52], [C10], [91], [99]. This mechanism is typically used in task scheduling systems like StarPU [55], OpenStream [30], or XKaapi [44] for tracing the execution of tasks, or in runtime systems like Marcel [93], or NewMadeleine [88] for analyzing the runtime internals. This instrumentation method makes it possible to control the precise location of probes, at the expense of an implementation complexity.

Instrumentation during the compilation. In order to automate the insertion of probes in applications, some tracing tools rely on compilers for instrumentation [52], [91], [99]. The compiler inserts calls to callbacks at the entry and exit of functions. Compared to the manual instrumentation, this method is less precise as it is limited to the function boundaries. However, the insertion of probes is automatic, which reduces the user burden. Still, the application and its dependencies have to be recompiled with special options.

Runtime instrumentation with `LD_PRELOAD`. When running an application that uses shared libraries, the system loader first loads the required shared libraries and populates a symbol table that indicates the address of functions. Many performance analysis tools use the `LD_PRELOAD`

mechanism to preload wrapper functions in charge of recording events before and after selected functions [52], [79], [82], [91], [99]. This instrumentation method has the same granularity as the compiler-based instrumentation, but it can run the application without recompiling it. However, this method can only intercept calls to functions that are defined in shared libraries.

Binary patching. An alternative solution for instrumentation consists in modifying the program binary for inserting probes. Dyninst [112], PIN [101], MAQAO [63], or DynamoRIO [106] rely on instruction decoding to instrument the code. These tools reverse engineer programs and directly insert opcodes anywhere in the binary. This can be done either at runtime [101], [106], [112], or before the execution by patching the ELF binary file [63]. This fine-grain instrumentation allows fine-grain modifications of the application. Scripting languages can be used for automating the insertion of probes [41]. Since these tools reverse engineer the application binary code, they are heavily dependent on the CPU architecture and require a significant development effort for porting to a new CPU architecture.

Contribution. In order to make the instrumentation as simple as possible for the end user, EZTrace uses two automatic instrumentation methods: runtime instrumentation with `LD_PRELOAD`, and runtime code patching.

Most binary patching tools are heavily architecture dependent. To make binary patching as portable as possible, we designed a lightweight instrumentation method [W2] that requires only a few lines of architecture specific code. Since the expected granularity of the instrumentation is coarse (*i.e.* the function entry/exit points), the proposed method hijacks the application flow by inserting a jump instruction at the function entry. The overwritten opcodes are also moved to preserve the function integrity. As a result, porting this mechanism to a new CPU architecture only requires to write a few dozens of lines of code. Our experiments show that the overhead of the instrumentation at runtime is negligible and does not impact the application performance.

2.1.3 Trace generation

Tracing tools for parallel applications have been developed for decades now, and multiple file formats where developed accordingly [57], [72], [96], [99], [100], [113], [116]. In order to avoid altering the behavior of an analyzed application, the overhead of recording events has to be as low as possible. The scalability of the tracing library when the number of threads grows can be a significant bottleneck if they use a single buffer [100]. Some trace formats rely on ASCII encoding [72], [96] which may generate large trace files, although this can be mitigated by compressing the resulting traces [96]. The semantic of the recorded events also has to be taken into account: some trace formats specify the semantics of events [57], [96], which eases the development of trace analysis tools, but restricts the range of recordable events to the predefined ones. As a result, such trace format is only usable for some libraries (*e.g.* MPI, or OpenMP), but not for others (*e.g.* CUDA, or OpenSHMEM [65]). On the contrary, some trace formats reduce the semantics of their API in order to make the file format more generic [72], [100], [116]. This allows to trace any kind of library. However, trace analysis becomes more complex due to the lack of semantics.

Contribution. We designed a tracing library called LiTL [2] that records events in a compact and scalable way, while being generic [R2]. While LiTL is inspired from FxT [100], it differs in several key points. Each thread records events in a dedicated buffer, which reduces the need for synchronization and improves the cache friendliness. LiTL does not specify a semantics for the recorded events, allowing to trace any kind of library. The trace analysis program is in charge of adding the semantics to the events and to extract, for each event, the parameters that are packed in order to reduce the trace file size. EZTrace uses LiTL for recording events at runtime, and converts the generated trace to several trace formats such as Paje [72] or OTF [96] that can be later analyzed through visualization tools like ViTE [W3].

2.2 Collecting memory accesses with NumaMMA

The performance of processors have significantly increased over the years. The number of processing units on a computer has raised from a handful of cores to dozens of data hungry cores that execute vector instructions. As a result, memory becomes a major bottleneck in parallel applications. While this can be mitigated by *Non-Uniform Memory Access* (NUMA) architectures that integrate several memory nodes, the application has to carefully balance its memory access across the NUMA nodes in order to avoid bottlenecks [42]. In order to help developers, several approaches have been proposed for analyzing an application memory access patterns.

Instrumentation of memory accesses. Executing an application with a simulator such as Simics [109] allows to capture precisely all its memory accesses [56]. Since each instruction of the program is simulated, this approach significantly degrades the application performance. Another possibility is to instrument the application binary at runtime with tools like Pin [101] or Valgrind [92] so that each instruction that reads or writes data is recorded [11], [13], [17], [18], [20], [80]. Instrumentation of the memory accesses can also be performed offline by disassembling the application binary [41], [45] and inserting probes at precise locations of the program.

While the overhead caused by the instrumentation is reduced compared to the simulation approach, it still causes the application to run up to 20 times slower than the non-instrumented version. This prevents from using this solution on large applications. Moreover, in addition to the degradation of performance, collecting all the application memory accesses generates large amounts of data.

Sampling memory accesses. In order to collect the memory accesses of an application with a low overhead, it is necessary to reduce the quantity of collected data. This can be done by leveraging the memory sampling capabilities provided by the hardware. Modern processors implement hardware-based monitoring systems (such as Intel Processor Event Based Sampling, or AMD Instruction Based Sampling) that periodically save information about the instruction being executed. This mechanism can be used for collecting the memory addresses that are

[2] Available as open source at `https://github.com/trahay/LiTL`

accessed [17], [25], [31], [34], [53], [69], [71], [97]. Since only some of the instructions are collected, this approach is less precise than instrumentation. However, since the instrumentation is performed by the hardware, the collection of samples is more efficient, and it only degrades the application performance by a few percents. This makes this approach applicable on large applications. Also, compared to the approach based on binary instrumentation, hardware-based sampling allows to record the level in the memory hierarchy (L1, L2, ...) that served an access along with the latency of the access, which gives additional information on performance.

Analyzing memory access patterns. Once the application memory accesses are collected, several analysis are possible. Some works report the amount of communication between threads in order to improve thread and data locality [20]. TABARNAC [18] detects how threads concurrently access data structures. Finally, the memory accesses of thread can be used for automatically placing or moving memory pages in order to improve data locality [29], [42], [97].

Contribution. We developed NumaMMA [3] [C3], a lightweight memory profiler that collects the memory access pattern of threads. NumaMMA collects information on the application memory allocations, and uses the processor sampling feature to collect memory access. The collected data is then processed in order to detect which memory objects are mostly accessed, and how the threads access these objects.

The contribution of this work is both the coupling of the collected memory accesses with the application memory objects, and the original visualization that allows users to detect how threads use memory objects concurrently. An example of NumaMMA visualization is reported in Figure 2.1.

Overall, this work makes the following contribution:

- NumaMMA collects the application memory accesses with a low overhead by relying on hardware sampling;
- NumaMMA reports how memory access patterns inside objects evolve over time;
- NumaMMA provides developers with original visualization means of these memory access patterns;
- The evaluation shows that this information can be used for defining a placement strategy that improves the performance of applications by up to 28 %.

2.3 Conclusion

The performance collection tools presented in this chapter have been developed over the past ten years. Most of the research and engineering efforts for developing new features have been conducted with the help of a post-doc (Roman Iakymchuk worked on LiTL), or master students

[3] Available as open source at `https://numamma.github.io/numamma/`

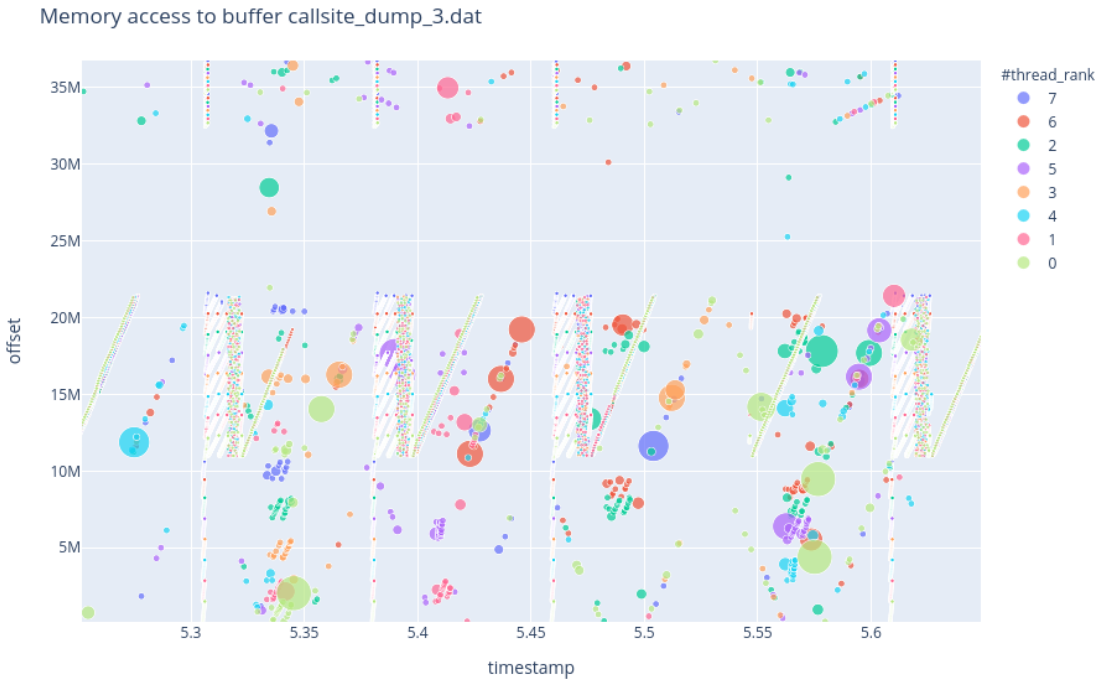


Figure 2.1: *NumaMMA representation of the threads access pattern to the memory object* `cvar_` *in the application LU.A from the NAS Parallel Benchmark OpenMP*

during internship (Gaëtan Bossu worked on EZTrace instrumentation, Pierrick Pamart worked on NumaMMA) or as part of their master project (12 students worked on various EZTrace extensions, 6 students worked on trace visualization, and 3 students worked on NumaMMA). EZTrace is available as open-source at `https://eztrace.gitlab.io/eztrace/`. It has been the building block of several projects: the INRIA ADT EZPerf that aimed at improving the usability of EZTrace for HPC applications, and the ongoing IDIOM FUI project that studies the performance of the whole I/O stack of HPC and Big Data applications. NumaMMA is available as open-source at `https://numamma.github.io/numamma/`.

Chapter 3

Analyzing performance data

Contents

3.1 Detecting the structure of a trace 14

3.1.1 Related work 14

3.1.2 Contribution 15

3.2 Differential execution analysis 17

3.2.1 Related work 17

3.2.2 Contribution 19

3.3 Detecting scalability issues with ScalOMP 20

3.3.1 Related work 21

3.3.2 Contribution 21

3.4 Conclusion 22

Once performance data is collected with an instrumentation tool such as EZTrace or NumaMMA, users have to analyze the data. The analysis of performance data allows users to understand the global behavior of the application, and to detect the source of a performance problem. To do so, users can browse the performance data (eg. execution time, hardware counters, ...), or explore data with a visualization tool (eg. ViTE [W3]). However, the large quantity of collected data may overwhelm users, and make a manual search of performance problems tedious.

In this chapter, we explore several approaches for automatically analyzing performance data. When designing such performance analysis tools, we aim at:

- Detecting problems that would be unnoticed with a manual analysis;
- Processing large quantities of data that would be too tedious to explore manually;
- Providing users with hints on how to optimize their application.

The remainder of this chapter is organized as follows. Section 3.1 describes how the program structure can be extracted from an execution trace. In Section 3.2, we present a generic contention detection mechanism. Finally, Section 3.3 describes ScalOMP, a performance analysis tool that detects the source of scalability issues in OpenMP applications.

3.1 Detecting the structure of a trace

The use of performance analysis tools, such as tracing tools, becomes unavoidable to optimize a parallel application. However, analyzing a trace file composed of millions of events requires a tremendous amount of work in order to spot the cause of the poor performance of an application.

In this Section, we propose mechanisms for assisting application developers in their exploration of trace files [C6]. We propose an algorithm for detecting repetitive sequences of events in trace files. Thanks to this algorithm, the program structure (loops, functions, ...) can be extracted from an execution trace without prior knowledge. We also propose a method to filter traces in order to eliminate duplicated information and to highlight points of interest. These mechanisms allow the performance analysis tool to pre-select the subsets of the trace that are more likely to contain useful information.

3.1.1 Related work

Analyzing an execution trace and detecting the program overall behavior can serve multiple goals.

Detecting inefficient patterns. Some performance analysis tools search of pre-defined patterns of events in execution traces. A database contains a series of sequences of events (such as the *late sender* pattern) that are classical causes of performance problems. By comparing an execution trace with the database, some tools pinpoint inefficient behaviors in applications [105]. This can be combined with compilation techniques to automatically transform the application in order to improve its performance [87]. Similarly, specific communication patterns can also be identified at runtime and be replaced with semantically equivalent but faster communication (such as collective communication primitives) [51].

Another approach consists in detecting the communication patterns based on MPI messages. This can provide users with a high-level understanding of the application [86], and the communication scheme of multiple applications can be compared to detect similarities [81]. The communication pattern can be extrapolated in order to estimate the performance of the application when it is run on a large number of nodes [49], [60].

Prefetching data. Analyzing a program execution permits to predict its future behavior. For example, a program memory access pattern may reveal which data will be accessed in the future. Prefetching this data then improves the cache-hit rate [108]. Memory accesses also provide a profile for applications and permit to model the performance of programs [111].

Access patterns are also used for predicting future disk accesses in file systems, allowing to prefetch blocks of data [89]. A similar mechanism can be used in parallel file systems for predicting future client accesses [83] or disk accesses [47].

Pattern mining techniques. Another way to analyze an application is to use pattern mining techniques. This allows to extract features from applications [98], or to detect the sequences of events that lead to a software bug [54] or a performance problem [35]. However, the performance of the proposed algorithms are prohibitive: detecting patterns in small traces takes dozens of seconds [68].

Reducing the size of execution traces. In order to limit the size of traces files when applications run for a long time, or with a high number of MPI ranks, several approaches have been tested. Scalatrace analyzes the MPI communication patterns and detects common behavior between processes [82]. Since most MPI ranks of an application have similar communication schemes, this allows to efficiently compress the trace files and to achieve near-constant size recording of traces.

Another way to reduce the size of a trace consists of pruning some of its events when the trace becomes too large [50]. In this work, the authors record events in separate buffers based on their callstack level. When the trace becomes too large, the lowest level events are pruned, which limits the trace size while maintaining the coarse view of the application behavior.

3.1.2 Contribution

In this work, we propose a method for assisting users in their search for information in trace files [C6]. This method relies on an algorithm for finding repetitive patterns of events in execution traces. While the usual representation of a trace is a sequential list of events, this algorithm permits to organize the trace by grouping events into loops and sequences, which reflects the program structure.

Detecting the trace structure. The proposed algorithm analyzes the events of a thread in a sequential way. It first searches for a sequence of two consecutive events that appears several time. As illustrated by Figure 3.1, such sequences are replaced with a pattern data structure.

In the next step, we try to form loops with this pattern by searching for pattern instances that repeat. If found, we replace them with loops data structures, as illustrated by Figure 3.2.

We then try to expand a pattern by comparing the event (or pattern) that follows each instance of a pattern. As illustrated by Figure 3.3, this can lead to three cases:

- if a pattern $P\#1$ is always followed by the same event c (case 1), we integrate c to $P\#1$;
- if several occurrences of $P\#1$ are followed by an event c, and some others are not (case 2), we create a new pattern $P\#2$ that consists of $P\#1$ and c;
- if an occurrence of $P\#1$ is followed by c only once, this occurrence cannot be expanded.

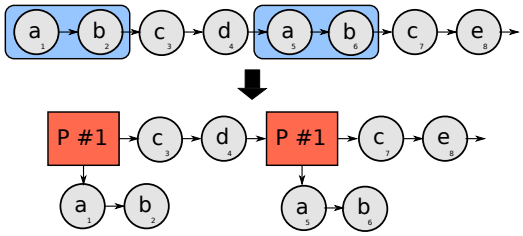


Figure 3.1: *Step 1: find a sequence of two consecutive events that appears several times to form a pattern*

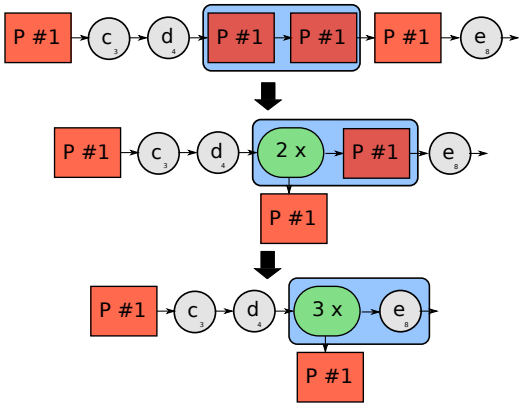
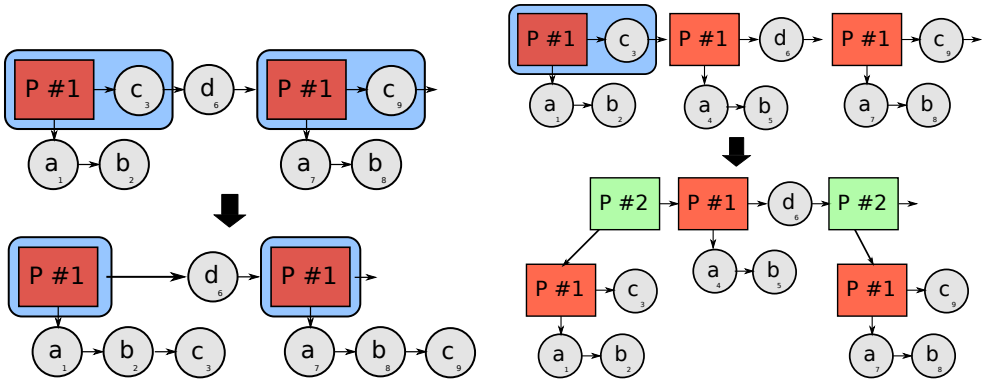


Figure 3.2: *Step 2: compare each occurrence of a pattern with the following event and form loops*



(a) Case 1 (b) Case 2

Figure 3.3: *Step 3: expanding patterns*

Once the third step is complete, the algorithm repeats the first step for the next couple of events in the trace.

Filtering traces. Once the program structure is extracted from the execution trace, and idiomatic sequences of events are detected, we analyze their variation. In order to reduce the quantity of data that user have to analyze, we filter out the sequences that have similar duration.

Evaluation. We evaluate our implementation by analyzing several execution traces of parallel applications. We use EZTrace to collect traces that contain the calls to MPI functions by eight kernels of the NAS Parallel Benchmarks. Analyzing the traces show that:

- The detected patterns of events correspond to the structure of the tested programs: the outermost patterns correspond to the application iterations;
- The pattern detection algorithm processes large traces in a few seconds in the worst case, which allows our implementation to be used in real life applications;
- Filtering out *useless* events significantly reduce the trace size for all the tested applications. Large traces are reduced by up to 99 %.

3.2 Using differential execution analysis to identify thread interference

Once the sequences of events of an execution trace are identified, analyzing the variation of their duration can reveal performance problems. When multiple execution flows access resources concurrently, they may interfere, *i.e.* they slow each other down because one thread has to wait for the other to access the resources, which degrades the performance of the application. This kind of contention problem may affect many different resources (disk, memory, network, locks, ...), and can be difficult to precisely detect.

In this work, we propose to use differential execution to identify the blocks of code that hamper the parallelism [J2]. We define, for a given block of code, its SCI (Slowdown Caused by Interference) score, which gives the theoretical slowdown caused by interference. In order to evaluate the usability of the SCI score, we developed a profiling toolchain, called ISPOT (Interference Spotter), that computes the SCI scores of a set of functions provided by the user. We show that this metric allows to detect several kinds of interference and to assess their impact on the application performance.

3.2.1 Related work

In this Section, we review existing works that aim at detecting interactions between threads and their impact on application performance.

Detecting memory performance issues. Because the performance and the number of processing units in CPUs increase, the memory subsystem may become a major bottleneck. Detecting performance problems that are due to memory contention is tedious.

Several works rely on hardware counters (*e.g.* the *Last Level Cache miss* counters) for detecting cache contention [15], [70], [76], [84], [94], [107], NUMA effects [32], or false sharing [46], [48].

Other approaches capture the application memory access patterns by simulating the memory subsystem [58], [85], [103], by instrumenting the application binary [11], [33], [61], [80], [90], or by using instruction sampling [6], [C3], [14], [25], [42], [53], [73].

Analyzing the memory accesses of threads can be used for detecting cache issues [25], [73], [85], false-sharing [14], [33], [48], [58], [61], [80], [90], [103], or NUMA effects [6], [C3], [42], [53].

Existing work on the detection of memory performance issues do not quantify the impact of the problem on the performance of the application, and the developer is often in charge of spotting which part of their program causes the problem.

Detecting resource contention. Several work have focused on detecting a single type of resource contention. Analyzing the message rate can help detecting network contention [27], [39]. Lock contention can be detected by measuring the time spent waiting for locks [28], [38], [43], [62], [74]. Overall, these projects define metrics that identify a saturation, but they can not tell if the saturation significantly affects the application performance.

Differential execution analysis To identify the root cause of a performance problem, several works use differential execution techniques [37], [95]. By comparing multiple runs of an application while varying the workload [37], or the level of concurrency [95], the variation of functions profiles can indicate the root cause of a bottleneck. VProfile uses a similar approach: it runs an application with the same configuration multiple times, and it compares the functions variance to pinpoint functions that may suffer from contention [12]. This type of differential execution analysis requires to run the application with different configuration, which may be time consuming for long running applications. Moreover, these works help to find the root cause of a performance defect that is already identified by the user.

Coz identifies the code that should be optimized by slowing down the other parts [19]. This differential analysis estimates the relative speedup that could be obtained if a function was optimized. However, Coz does not identify contention problems.

Application logs can be analyzed with data mining techniques in order to identify the interactions between software components in distributed systems [16]. Stitch analyzes logs and extracts a structure graph that describes how software components interact. If a performance bottleneck was previously identified, this can pinpoint the component that is the root cause of the problem. Another work use inference models to build a state machine model that summarizes the application behavior [35]. The state machine model describes how events follow each

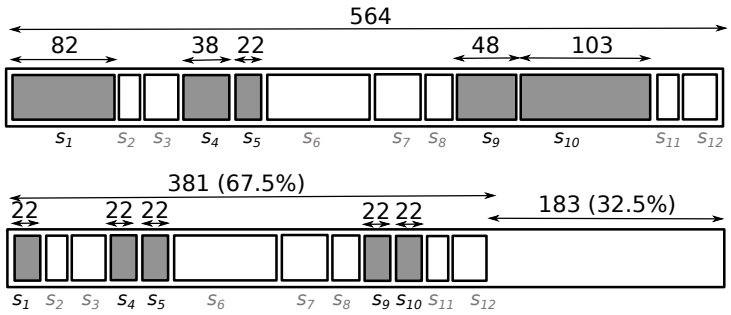


Figure 3.4: *Illustration of the SCI metric: based on the original execution trace (on the top), an ideal improvement without interference is estimated at 32.5% (on the bottom)*

other, and indicates the duration between two events. Due to the high complexity of these algorithms, these approaches fail to apply to real-life applications that generate large size traces. In our experiment [J2], we show that Perfume [35] analyzes a small trace that only consists of 10000 events in 5536 seconds. Thus, processing large traces with millions of events is impractical with this approach.

3.2.2 Contribution

We propose to reuse the ideas behind differential execution analysis, but instead of comparing several executions of an application with different settings that may not be comparable, we compare invocations of blocks of code with each other within a single application run [J2]. In order to detect the blocks of code that hamper the parallelism, we compare their execution time. Instead of focusing on the average execution time, which can hide performance bottleneck, we propose to focus on the fastest one. The intuition is that the fastest execution of a block of code gives a theoretical better execution. Any longer execution is probably caused by the interference from another thread when it accesses the same hardware resource or the same synchronization primitive.

We first identify the structure of the program with the work presented in section 3.1. This structure corresponds to blocks of code in the application. We then analyze the duration of these blocks, and for each block of code, we assume that the fastest execution is contention free. We define the SCI (Slowdown Caused by Interference) metric of a block as the application theoretical speedup if all occurrences of the block were interference-free. Formally, the SCI of a block of code is the sum of all $(d_j - \overline{d_i})$ (where $\overline{d_i}$ is the duration of the fastest execution of this block, d_j is the j^{th} execution occurrence of the block of code) divided by the thread duration. Figure 3.4 illustrates this metric: the gray boxes correspond to multiple invocations of a block of code. If the fastest invocation is considered contention free, applying its duration (here, 22) to all the invocations of the same block approximates the application run time if the contention was removed.

In order to evaluate the usability of the SCI score, we instrument several applications with EZ-Trace, and we analyze the resulting traces to compute the SCI scores of all the instrumented functions. We analyze 27 applications and 4 micro benchmarks, and we found that the SCI

score is able to identify bottlenecks caused by false-sharing, contended locks, saturated networks, saturated hard drives, imbalanced workload, and inefficient NUMA placements.

In detail, we found that:

- Even when an application suffers high contention, the fastest occurrence of the contented function has similar performance as in non-contented cases. This shows that the fastest occurence of a block of code is a good estimate of the performance of the block of code when it does not suffer interference;
- ISPOT detects interference in 14 time consuming functions from 10 of the evaluated applications;
- Among the 14 functions, 2 (13%) are false positives that are not caused by interference. In these cases, the variation of the function duration is caused by the function workload that varies from one occurence to another. We show that a manual inspection of the source code easily identifies these blocks of code as false positives, which allows users to discard them;
- The remaining 12 functions pinpoint actual interference problems caused by false sharing, lock contention, imbalanced load, NUMA memory placement, network stack and disk I/O. 7 interference bottlenecks were previously identified in other works, while 5 are new;
- Based on this analysis, we can correct 8 functions by modifying at most only 25 lines of code, which leads to a performance improvement of up to 9 times

3.3 Detecting scalability issues with ScalOMP

Analyzing and improving the performance of a parallel application requires many skills. A developer has to understand the application domain (*e.g.* molecular dynamics, climate modeling, etc.), the program algorithms, details on the machine architecture (*e.g.* CPU, caches, memory, etc.), and how runtime systems work (*e.g.* how threads are scheduled, how MPI communicates over the network, etc.)

Since this set of required competences is too large for most programmers, we propose to design performance analysis tools that bridge the gap between high-level algorithmic, and low-level architectural details.

We propose a simple methodology for analyzing the scalability of OpenMP applications and for automatically detecting performance bottlenecks [W1]. We implement a performance analysis tool called ScalOMP, which reports the parallel regions with poor scalability, their potential of improvement, and optimization hints for fixing the performance problem.

3.3.1 Related work

Detecting scalability issues in parallel applications has been studied for a long time. However, automatic performance analysis of applications is a more recent field of study. Several works have focused on detecting the root causes of scalability issues in MPI applications. This can be done by modeling the performance of applications to find weak scaling issues [40]. Replaying an execution trace can help identifying the root cause of MPI wait states [64]. ScalAna identifies patterns in execution traces and compares their run time in order to detect the root cause that hampers scalability [7].

Other works have focused on detecting and reporting performance problems in multi-threaded applications. Multiple works target OpenMP applications and detect imbalance in parallel regions that can affect the scalability [59], model OpenMP applications for predicting their performance on a given machine [36], or detect false sharing issues [21]. Intel VTune can estimate the impact of various performance problems such as lock contention, load imbalance, or scheduling overhead [102]. Automated performance modeling can be used to examine the scalability of OpenMP runtime constructs [22], or to analyze the memory access patterns of OpenMP applications [78].

While existing work allow to pinpoint performance problems in OpenMP applications, they do not estimate the impact of the problems on the application performance. Moreover, once a problem is identified, the developer has to find a way to fix the scalability issue.

3.3.2 Contribution

In order to facilitate the detection of scalability issues and their resolution, we present a methodology for analyzing the scalability of OpenMP applications. We implement this approach in ScalOMP, a performance analysis tool that collects performance data, analyze them, and reports the OpenMP constructs that limit the scalability. ScalOMP also suggests optimization strategies to the application developer.

Performance analysis methodology. ScalOMP aims at identifying scalability issues in OpenMP applications. Several problems may reduce the scalability of parallel applications, including load imbalance, lock contention, or tasks granularity. When a scalability problem is identified, several approaches can be used for fixing or mitigating it, from modifying the application algorithms to changing the execution settings.

The proposed methodology consists of running the application with a varying number of threads, and measuring the scalability of each OpenMP construct. The output is the list of parallel regions whose scaling issues affect the most the application, along with optimization hints and their potential time gain.

Detecting OpenMP scaling issues. ScalOMP relies on the OpenMP Tool interface (OMPT) for instrumenting the application. It inserts probes at several OpenMP key points (such as the beginning and end of parallel regions, around barriers, etc.) Based on the collected data,

ScalOMP computes the parallel efficiency of each OpenMP parallel region, the parallel loops imbalance, or the time spent waiting on locks. Each metric estimates the impact of a problem on scaling, and ScalOMP can suggest optimization strategies for each detected performance problem.

Evaluation of ScalOMP. We evaluate ScalOMP on 16 applications running on a 32 cores machine. The evaluation shows that ScalOMP instrumentation does not significantly alter the performance of applications. Moreover, ScalOMP scaling analysis show that:

- ScalOMP detects load imbalance problems in two applications, and estimates their impact on the execution time. Applying the optimization suggested by ScalOMP fixes the problems, and the resulting execution time corresponds to ScalOMP estimate.
- ScalOMP detects locking issues in one application, and one microbenchmark. It correctly distinguishes locking issues that are due to contention, and those that are due to too many uncontended locks. The optimization suggested by ScalOMP improves the application execution time by up to 267 %.

3.4 Conclusion

The works on the detection of the structure of a trace [C6], and the detection of thread interference [J2] were done in collaboration with a PhD student, Mohamed Saïd MOSLI BOUKSIAA, helped by Gaël THOMAS. ScalOMP was designed as part of Anton DAUMEN PhD who I co-advise with Patrick CARRIBAULT and Gaël THOMAS.

Chapter 4

Ongoing and future works

During the last decades, the world of parallel computing has changed due to the evolution of hardware platforms that became highly parallel and heterogeneous. As a result, the software stack evolved and HPC applications now commonly mix paradigms. Developing an application that efficiently exploits a supercomputer becomes more and more difficult. The number of software components required to run a distributed application makes the debugging of performance tedious.

This document presents several works on performance analysis that we conducted over the last decade. As a first contribution, we designed EZTrace, a tracing tool for parallel applications. EZTrace automatically instruments applications and generates execution traces with a low overhead. We also designed NumaMMA, a tracing tool that captures the memory access patterns of applications.

As a second contribution, we developed performance analysis techniques that process performance data and help developers identifying performance problems. We designed algorithms that detect the structure of a program from an execution trace. We used this work and proposed a metric for identifying any type of interference in a parallel application. Finally, we focused on OpenMP applications and designed a methodology for identifying scalability issues and suggesting optimization hints.

This journey leads to new research opportunities that are currently being investigated or that we intend to explore in the future.

4.1 Performance analysis of data management

Recent hardware evolution has bridged the gap between memory technologies and storage systems. On the one hand, the performance of disks have moved from millisecond-latency hard disk drives (HDD), to solid state drives (SSD) that fetch data in a few dozens of microseconds, to NVMe SSDs that achieve latencies of a few microseconds. More recently, non-volatile memory has been used as high performance persistent storage that achieves latencies of a few

hundreds nanoseconds.

On the other hand, memory has evolved from single memory bank systems where all the memory access have similar performance, to NUMA architectures where the data locality significantly impacts the performance of applications. NUMA systems become widespread and are even integrated within multicore processors. More recently, non-volatile memory trades persistence for performance, and the NVDIMM latency becomes significantly slower than traditional RAM [8].

As a result, the performance of memory and I/O becomes more and more non-uniform. Thus, the storage medium may have a significant impact on performance. One research direction to explore is to assess the impact of each type of storage on performance.

This is partially addressed in the ongoing FUI project IDIOM that aims at designing a tool suite for analyzing the performance of the whole I/O stack. As part of this project, we intend to analyze execution traces and to replay them with performance models in order to estimate the performance of an application if it used a particular storage system.

In the future, we plan to investigate the performance of the memory subsystem. As the performance of memory accesses may vary significantly depending on the type of memory (*i.e.* DRAM or NVDIMM), or on the locality, we intend to explore new memory placement strategies that take into account how threads manipulate objects.

Modeling the performance of applications while taking memory into account is another challenging track to explore. Our work on ScalOMP show that analyzing performance data allows to predict how an application will perform with a different setting. However, this performance prediction becomes less accurate as memory effects (such as cache misses, or NUMA effects) become significant. A better understanding of how memory affects the performance of application is needed to improve performance analysis.

4.2 Feeding runtime and compilers with performance data

Our past research works have led us from designing performance analysis tools that allow users to manually explore an application behavior, to automatic performance analysis techniques that pinpoint problems and suggest optimizations. A next step is to automate the optimization once a performance problem is detected. The results of performance analysis could be communicated to a compiler that could adapt the generated application. Another approach would be to feed runtime systems with information on the application behavior so that they improve their decisions.

Alexis COLIN, a PhD student who I co advise with Denis CONAN is currently exploring this approach as part of the ANR JCJC project PYTHIA. This project aims at providing runtime systems with an oracle capable of predicting the application future behavior. While a runtime system makes decisions based on the application current state, knowing the future would permit to anticipate and to choose the best runtime strategy to minimize the application execution time. To design the oracle, we take advantage of the determinism of many HPC applications

that apply the same algorithm on their input data. Collecting an execution trace, and detecting its structure would help the oracle understand the overall behavior of the program. When an application executes, its previous execution profile could be provided to the oracle. The oracle could then follow the application execution and compare it to the previous profile in order to predict how the program will behave in the future.

In the future, a similar approach could be applied to compiler optimization: analyzing the behavior and the performance of a running application could help the compiler apply optimization based on the code that is actually executed.

4.3 Analyzing high performance data analytics applications

For decades, developers used low level programming languages (such as C or Fortran) for designing parallel applications using a few common libraries (eg. MPI, OpenMP, CUDA, etc.) Parallel programming was mainly applied to scientific computing and simulation. The recent development of data analytics has changed the types of applications that run on parallel computers. In addition to traditional HPC simulation, clusters of machines now commonly execute MapReduce jobs running on Hadoop, or distributed machine learning applications running on TensorFlow and Horovod. These new distributed computing frameworks rely on the same building blocks as traditional HPC: executing independent computation on several computers, loading data from a parallel filesystem, offloading computation to accelerators, and communicating through a high speed network. In the future, we intend to explore these new types of parallel applications and their challenges in terms of performance anaysis.

The software stack of data analytics significantly differs from HPC. Data analytics applications are implemented in high-level programming languages such as Python or Java. In order to provide high performance, the high-level frameworks implement routines in low-level languages such as C. This makes the development of performance analysis tools complex as they need to relate low-level events that happen in the C realm to higher-level phases of the application.

The recent shift towards new types of parallel applications creates many research opportunities on performance analysis. While the HPC software stack has been optimized for decades, high performance data analytics framework have a shorter history, which makes rooms for performance improvement. As deep learning models become more and more complex, and training data sets become larger, applications require more computing power and need to be distributed on multiple compute nodes in supercomputers. This raises new problems that need to be addressed. For instance, many deep learning applications read the data set from disk during each epoch [10]. Since data sets may consist of terabytes of data stored in many small files [9], this type of data processing flow may suffer major I/O bottlenecks. In the future, we intend to analyze distributed machine learning applications in order to identify performance problems that arise.

Personal bibliography

International journals

[J1] J. Li, Z. Sha, Z. Cai, F. Trahay, and J. Liao, "Patch-based data management for dual-copy buffers in RAID-enabled SSDs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3956–3967, Nov. 2020. DOI: 10.1109/TCAD.2020.3012252.

[J2] M. S. Mosli Bouksiaa, F. Trahay, A. Lescouet, G. Voron, R. Dulong, A. Guermouche, É. Brunet, and G. Thomas, "Using differential execution analysis to identify thread interference," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2866–2878, Dec. 2019.

[J3] J. Liao, F. Trahay, Z. Cai, S. Chen, Y. Ishikawa, and H. Xiong, "Fine Granularity and Adaptive Cache Update Mechanism for Client Caching," *IEEE systems journal*, 2018.

[J4] J. Liao, Z. Cai, F. Trahay, and X. Peng, "Block Placement in Distributed File Systems based on Block Access Frequency.," in *IEEE Access*, 2018.

[J5] J. Liao, Z. Cai, F. Trahay, J. Zhou, and G. Xiao, "Adaptive Process Migrations in Coupled Applications for Exchanging Data in Local File Cache," in *ACM Transactions on Autonomous and Adaptive Systems*, 2018.

[J6] J. Liao, F. Trahay, G. Xiao, L. Li, and Y. Ishikawa, "Performing initiative data prefetching in distributed file systems for cloud computing," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 550–562, 2017.

[J7] R. Habel, F. Silber-Chaussumier, F. Irigoin, É. Brunet, and F. Trahay, "Combining data and computation distribution directives for hybrid parallel programming: A transformation system," *International Journal of Parallel Programming*, pp. 1–28, 2016.

[J8] J. Liao, F. Trahay, and G. Xiao, "Dynamic process migration based on block access patterns occurring in storage servers," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 2, Jun. 2016.

[J9] J. Liao, F. Trahay, B. Gerofi, and Y. Ishikawa, "Prefetching on storage servers through mining access patterns on blocks," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2015.

International conferences with program committee

[C1] A. Lescouet, É. Brunet, F. Trahay, and G. Thomas, "Transparent Overlapping of Blocking Communication in MPI Applications," in *IEEE International Conference on High-Performance Computing and Communications (HPCC)*, Yanuca Island, Fiji, Dec. 2020.

[C2] P. Sutra, P. Marlier, S. Valerio, and F. Trahay, "A Locality-Aware Software Transactional Memory," in *International Conference on Coordination Models and Languages (COORDINATION)*, 2018.

[C3] F. Trahay, M. Selva, L. Morel, and K. Marquet, "NumaMMA: NUMA MeMory Analyzer," in *International Conference on Parallel Processing (ICPP)*, 2018, pp. 1–10.

[C4] A. Denis and F. Trahay, "MPI Overlap: Benchmark and Analysis," in *International Conference on Parallel Processing (ICPP)*, 2016.

[C5] P. Li, É. Brunet, F. Trahay, C. Parrot, G. Thomas, and R. Namyst, "Automatic OpenCL code generation for multi-device heterogeneous architectures," in *International Conference on Parallel Processing (ICPP)*, 2015.

[C6] F. Trahay, É. Brunet, M. S. Mosli Bouksiaa, and J. Liao, "Selecting points of interest in traces using patterns of events," in *Euromicro International Conference on Parallel, Distributed and Network Based Processing (PDP)*, 2015.

[C7] R. Iakymchuk and F. Trahay, "Performance Analysis on Energy Efficient High-Performance Architectures," in *International Conference on Cluster Computing (CC'13)*, 2013.

[C8] A. Denis, F. Trahay, and Y. Ishikawa, "High performance checksum computation for fault-tolerant mpi over infiniband," in *EuroMPI*, 2012, pp. 183–192.

[C9] É. Brunet, F. Trahay, and A. Denis, "A sampling-based approach for communication libraries auto-tuning," in *International Conference on Cluster Computing (IEEE Cluster)*, Sep. 2011.

[C10] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra, "EZTrace: A generic framework for performance analysis," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, poster session, 2011.

[C11] G. Mercier, F. Trahay, D. Buntinas, and É. Brunet, "NewMadeleine: An Efficient Support for High-Performance Networks in MPICH2," in *International Parallel and Distributed Processing Symposium (IEEE IPDPS)*, May 2009.

[C12] F. Trahay and A. Denis, "A scalable and generic task scheduling system for communication libraries," in *International Conference on Cluster Computing (IEEE Cluster)*, Sep. 2009.

[C13] É. Brunet, F. Trahay, and A. Denis, "A Multicore-enabled Multirail Communication Engine," in *International Conference on Cluster Computing (IEEE Cluster)*, Sep. 2008, pp. 316–321.

International workshops with program committee

[W1] A. Daumen, P. Carribault, F. Trahay, and G. Thomas, "ScalOMP: analyzing the Scalability of OpenMP applications," in *IWOMP 2019: 15th International Workshop on OpenMP*, 2019, pp. 36–49.

[W2] C. Aulagnon, D. Martin-Guillerez, F. Rue, and F. Trahay, "Runtime function instrumentation with EZTrace," in *PROPER - 5th Workshop on Productivity and Performance*, Aug. 2012.

[W3] K. Coulomb, A. Degomme, M. Faverge, and F. Trahay, "An open-source tool-chain for performance analysis," in *Tools for High Performance Computing*, 2011.

[W4] F. Trahay, É. Brunet, and A. Denis, "An analysis of the impact of multi-threading on communication performance," in *CAC 2009: The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*, May 2009.

[W5] F. Trahay, É. Brunet, A. Denis, and R. Namyst, "A multithreaded communication engine for multicore architectures," in *CAC 2008: Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, 2008.

National conference and book chapters

[F1] F. Trahay, M. Selva, L. Morel, and K. Marquet, "NumaMMA: Numa MeMory Analyzer," in *Conférence en Parallélisme, Architecture et Système (COMPAS'2018)*, 2018.

[F2] E.-L. Kern, F. Trahay, and K. Zanin, "Exploration visuelle des traces de calcul haute performance," in *Datalogie : formes et imaginaires du numérique*, Editions Loco, 2016, pp. 152–165.

[F3] M. S. Mosli Bouksiaa, F. Trahay, and G. Thomas, "Détection automatique d'interférences entre threads," in *Conférence en Parallélisme, Architecture et Système (COMPAS'2016)*, 2016.

[F4] ——, "Détection automatique d'anomalies de performance," in *Conférence en Parallélisme, Architecture et Système (COMPAS'2015)*, 2015.

[F5] F. Trahay, "Bibliothèque de communication multi-threadée pour architectures multi-cœurs," in *19ème Rencontres Francophones du Parallélisme*, Sep. 2009.

[F6] ——, "PIOMan : un gestionnaire d'entrées-sorties générique," in *18ème Rencontres Francophones du Parallélisme*, Feb. 2008.

Research reports

[R1] E. André, R. Dulong, A. Guermouche, and F. Trahay, "DUF : Dynamic Uncore Frequency scaling to reduce power consumption," Tech. Rep., Feb. 2020, working paper or preprint. [Online]. Available: `https://hal.archives-ouvertes.fr/hal-02401796`.

[R2] R. Iakymchuk and F. Trahay, "LiTL: Lightweight Trace Library," INF - Département Informatique, Technical Report, Jul. 2013. [Online]. Available: `https://hal.archives-ouvertes.fr/hal-00918733`.

PhD Thesis

[T1] F. Trahay, “De l’interaction des communications et de l’ordonnancement de threads au sein des grappes de machines multi-cœurs,” Ph.D. dissertation, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, Nov. 2009.

Bibliography

References

[2] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in *Ottawa Linux Symposium*, vol. 2006, pp. 209–224.

[3] *Intel Trace Analyzer and Collector*. [Online]. Available: `https://software.intel.com/content/www/us/en/develop/tools/trace-analyzer.html`.

[4] C. Silverstein, *Gperftools*, https://github.com/gperftools/gperftools.

[5] *Top 500 november 2020 list*. [Online]. Available: `https://www.top500.org/lists/top500/2020/11/`.

[6] C. Helm and K. Taura, "Automatic identification and precise attribution of dram bandwidth contention," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

[7] Y. Jin, H. Wang, T. Yu, X. Tang, T. Hoefler, X. Liu, and J. Zhai, "ScalAna: Automating Scaling Loss Detection with Graph Analysis," in *Proceedings of the Conference for High Performance Computing, Networking, Storage and Analysis, SC'20*, ser. SC '20, Atlanta, Georgia: IEEE Press, 2020, ISBN: 9781728199986.

[8] E. A. León, B. Goglin, and A. Rubio Proaño, "M&MMs: Navigating Complex Memory Spaces with hwloc," in *Proceedings of the International Symposium on Memory Systems Proceedings, MEMSYS'19*, Washington, DC, United States, Sep. 2019.

[9] L. Oden, C. Schiffer, H. Spitzer, T. Dickscheid, and D. Pleiter, "IO challenges for human brain atlasing using deep learning methods-an in-depth analysis," in *Proceedings of the International Conference on Parallel, Distributed, and Network-Based Processing, PDP'19*, 2019, pp. 291–298.

[10] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, *et al.*, "Exascale deep learning for climate analytics," in *Proceedings of the Conference for High Performance Computing, Networking, Storage and Analysis, SC'18*, 2018, pp. 649–660.

[11] S. Valat and O. Bouizi, "Numaprof, a numa memory profiler," in *Proceedings of the European conference on Parallel processing, EuroPar'18*, 2018, pp. 159–170.

[12] J. Huang, B. Mozafari, and T. F. Wenisch, "Statistical analysis of latency through semantic profiling," in *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'17*, 2017, pp. 64–79.

[13] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "RTHMS: A tool for data placement on hybrid memory system," *Proceedings of the International Symposium on Memory Management, ISMM'17*, vol. 52, no. 9, pp. 82–91, 2017.

[14] T. Liu and X. Liu, "Cheetah: Detecting false sharing efficiently and effectively," in *Proceedings of the international symposium on Code Generation and Optimization, CGO'16*, 2016, pp. 1–11.

[15] B. Teabe, A. Tchana, and D. Hagimont, "Application-specific quantum for multi-core platform scheduler," in *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'16*, 2016, 3:1–3:14.

[16] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *Proceedings of the conference on Operating Systems Design and Implementation, OSDI'16*, 2016, pp. 603–618.

[17] L. Zhu, H. Jin, and X. Liao, "A tool to detect performance problems of multi-threaded programs on numa systems," in *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 1145–1152.

[18] D. Beniamine, M. Diener, G. Huard, and P. O. Navaux, "TABARNAC: visualizing and resolving memory access issues on NUMA architectures," in *Proceedings of the 2nd Workshop on Visual Performance Analysis*, 2015, pp. 1–9.

[19] C. Curtsinger and E. D. Berger, "Coz: Finding code that counts with causal profiling," in *Proceedings of the Symposium on Operating Systems Principles, SOSP'15*, 2015, pp. 184–197.

[20] M. Diener, E. H. Cruz, L. L. Pilla, F. Dupros, and P. O. Navaux, "Characterizing communication and page usage of parallel applications for thread and data mapping," *Performance Evaluation*, vol. 88, pp. 18–36, 2015.

[21] M. Ghane, A. M. Malik, B. Chapman, and A. Qawasmeh, "False sharing detection in openmp applications using ompt api," in *Proceedings of the International Workshop on OpenMP Applications and Tools15*, 2015, pp. 102–114.

[22] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf, "How many threads will be too many? on the scalability of openmp implementations," in *European Conference on Parallel Processing*, Springer, 2015, pp. 451–463.

[23] C. January, J. Byrd, X. Oró, and M. O'Connor, "Allinea MAP: Adding Energy and OpenMP Profiling Without Increasing Overhead," in *Tools for High Performance Computing 2014*, 2015, pp. 25–35.

[24] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[25] X. Liu and B. Wu, "ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs," in *Proceedings of the Conference for High Performance Computing, Networking, Storage and Analysis, SC'15*, 2015, p. 47.

[26] K. Ocaña and D. de Oliveira, "Parallel computing in genomic research: Advances and applications," *Advances and applications in bioinformatics and chemistry: AABC*, vol. 8, p. 23, 2015.

[27] M. Casas and G. Bronevetsky, "Active measurement of the impact of network switch utilization on application performance," in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'14*, 2014, pp. 165–174.

[28] F. David, G. Thomas, J. Lawall, and G. Muller, "Continuously measuring critical section pressure with the free-lunch profiler," in *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'14*, 2014, pp. 291–307.

[29] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß, "Kmaf: Automatic kernel-level management of thread and data affinity," in *Proceedings of the International Conference on Parallel Architectures and Compilation, PACT'14*, 2014, pp. 277–288.

[30] A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach-Temam, "Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems," in *7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG, associated with HiPEAC)*, 2014.

[31] A. Giménez, T. Gamblin, B. Rountree, A. Bhatele, I. Jusufi, P.-T. Bremer, and B. Hamann, "Dissecting on-node memory access performance: A semantic approach," in *Proceedings of the Conference for High Performance Computing, Networking, Storage and Analysis, SC'14*, 2014.

[32] M. Liu and T. Li, "Optimizing virtual machine consolidation performance on numa server architecture for cloud workloads," in *Proceedings of the International Symposium on Computer Architecture, ISCA'14*, 2014, pp. 325–336.

[33] T. Liu, C. Tian, Z. Hu, and E. D. Berger, "PREDATOR: Predictive false sharing detection," in *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'14*, 2014, pp. 3–14.

[34] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on numa architectures," in *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'14*, 2014.

[35] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun, "Mining precise performance-aware behavioral models from existing instrumentation," in *Proceedings of the International Conference on Software Engineering, ICSE'14*, 2014, pp. 484–487.

[36] B. Putigny, B. Goglin, and D. Barthou, "A benchmark-based performance model for memory-bound hpc applications," in *Proceedings of the 14*, 2014, pp. 943–950.

[37] L. Song and S. Lu, "Statistical debugging for real-world performance problems," in *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'14*, 2014, pp. 561–578.

[38] X. Yu, S. Han, D. Zhang, and T. Xie, "Comprehending performance from real-world execution traces: A device-driver case," in *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14*, 2014, pp. 193–206.

[39] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[40] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proceedings of the Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*, 2013, p. 45.

[41] A. S. Charif-Rubial, D. Barthou, C. Valensi, S. Shende, A. Malony, and W. Jalby, "Mil: A language to build program analysis tools through static binary instrumentation," in *Proceedings of the international conference on High Performance Computing, HiPC'13*, 2013, pp. 206–215.

[42] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on numa systems," in *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'13*, 2013, pp. 381–394.

[43] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proceedings of the International Symposium on Computer Architecture, ISCA'13*, 2013, pp. 511–522.

[44] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'13*, 2013, pp. 1299–1308.

[45] J. Jaeger, P. Philippen, E. Petit, A. C. Rubial, C. Rössel, W. Jalby, and B. Mohr, "Binary instrumentation for scalable performance measurement of openmp applications.," in *PARCO*, 2013, pp. 783–792.

[46] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe, H. De Silva, S. Rathnayake, X. Meng, and Y. Liu, "Detection of false sharing using machine learning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–9.

[47] J. Liao, X. Liu, and Y. Chen, "Dynamical re-striping data on storage servers in parallel file systems," in *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*, IEEE Computer Society, 2013, pp. 65–73.

[48] M. Nanavati, M. Spear, N. Taylor, S. Rajagopalan, D. T. Meyer, W. Aiello, and A. Warfield, "Whose cache line is it anyway?: Operating system support for live detection and repair of false sharing," in *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'13*, 2013, pp. 141–154.

[49] J. Panadero, A. Wong, D. Rexachs del Rosario, and E. Luque Fadón, "Predicting the communication pattern evolution for scalability analysis," in *XVIII Congreso Argentino de Ciencias de la Computación*, 2013.

[50] M. Wagner, A. Knupfer, and W. E. Nagel, "Hierarchical memory buffering techniques for an in-memory event tracing extension to the open trace format 2," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, IEEE, 2013, pp. 970–976.

[51] T. Hoefler and T. Schneider, "Runtime detection and optimization of collective communication patterns," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ACM, 2012, pp. 263–272.

[52] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, *et al.*, "Score-p: A joint performance mea-

surement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*, 2012, pp. 79–91.

[53] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for numa multicore systems," in *Proceedings of the Usenix Annual Technical Conference, USENIX ATC'12*, 2012.

[54] P. López Cueva, A. Bertaux, A. Termier, J. F. Méhaut, and M. Santana, "Debugging embedded multimedia application traces through periodic pattern mining," in *Proceedings of the tenth ACM international conference on Embedded software*, ACM, 2012, pp. 13–22.

[55] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency - Practice & Experience (CP&E)*, vol. 23, no. 2, pp. 187–198, 2011.

[56] E. H. M. da Cruz, M. A. Z. Alves, A. Carissimi, P. O. A. Navaux, C. P. Ribeiro, and J.-F. Méhaut, "Using memory access traces to map threads and data on hierarchical multi-core platforms," in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'11*, 2011, pp. 551–558.

[57] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries.," in *PARCO*, vol. 22, 2011, pp. 481–490.

[58] T. Liu and E. D. Berger, "SHERIFF: Precise detection and automatic mitigation of false sharing," in *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'11*, 2011, pp. 3–18.

[59] M. Woodyard, "An experimental model to analyze openmp applications for system utilization," in *Proceedings of the International Workshop on OpenMP Applications and Tools11*, 2011, pp. 22–36.

[60] X. Wu and F. Mueller, "ScalaExtrap: Trace-based communication extrapolation for SPMD programs," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 113–122, 2011.

[61] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe, "Dynamic cache contention detection in multi-threaded applications," in *Proceedings of the international conference on Virtual Execution Environments, VEE'11*, 2011, pp. 27–38.

[62] E. Altman, M. Arnold, S. Fink, and N. Mitchell, "Performance analysis of idle programs," in *Proceedings of the conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'10*, 2010, pp. 739–753.

[63] D. Barthou, A. Charif Rubial, W. Jalby, S. Koliai, and C. Valensi, "Performance Tuning of x86 OpenMP Codes with MAQAO," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., Springer Berlin Heidelberg, 2010, pp. 95–113, ISBN: 978-3-642-11261-4.

[64] D. Bohme, M. Geimer, F. Wolf, and L. Arnold, "Identifying the root causes of wait states in large-scale parallel applications," in *Proceedings of the International Conference on Parallel Processing, ICPP'10*, 2010, pp. 90–100.

[65] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3.

[66] A. C. De Melo, "The new linux perf tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[67] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency - Practice & Experience (CP&E)*, vol. 22, no. 6, pp. 702–719, 2010.

[68] N. R. Mabroukeh and C. I. Ezeife, "A taxonomy of sequential pattern mining algorithms," *ACM Computing Surveys (CSUR)*, vol. 43, no. 1, p. 3, 2010.

[69] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-directed page placement for ccnuma via hardware-generated memory traces," *J. Parallel Distrib. Comput.*, vol. 70, no. 12, pp. 1204–1219, Dec. 2010, ISSN: 0743-7315.

[70] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: Online contention detection and response," in *Proceedings of the international symposium on Code Generation and Optimization, CGO'10*, 2010, pp. 257–265.

[71] C. McCurdy and J. Vetter, "Memphis: Finding and fixing numa-related performance problems on multi-core platforms," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS'10*, 2010.

[72] B. de Oliveira Stein, J. C. de Kergommeaux, and G. Mounié, "Pajé trace file format," Technical report, ID-IMAG, Grenoble, France, 2002. http://www-id. imag. fr/Logiciels/paje/publications, Tech. Rep., 2010.

[73] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating cache performance bottlenecks using data profiling," in *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'10*, 2010, pp. 335–348.

[74] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," in *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'10*, 2010, pp. 269–280.

[75] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of the International Conference on Parallel Processing, ICPP'10*, 2010, pp. 207–216.

[76] C. Xu, X. Chen, R. Dick, and Z. M. Mao, "Cache contention and application performance prediction for multi-core systems," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software, ISPASS'10*, 2010, pp. 76–86.

[77] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, 2009, p. 012 037.

[78] D. Barthou, A. C. Rubial, W. Jalby, S. Koliai, and C. Valensi, "Performance tuning of x86 openmp codes with maqao," in *Tools for High Performance Computing*, 2009, pp. 95–113.

[79] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–10.

[80] S. M. Günther and J. Weidendorfer, "Assessing cache false sharing effects by dynamic binary instrumentation," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009, pp. 26–33.

[81] C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, and S. See, "An approach for matching communication patterns in parallel applications," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, IEEE, 2009, pp. 1–12.

[82] 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," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 69, no. 8, pp. 696–710, 2009.

[83] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, IEEE, 2008, pp. 1–12.

[84] I.-H. Chung, G. Cong, D. Klepacki, S. Sbaraglia, S. Seelam, and H.-F. Wen, "A framework for automated performance bottleneck detection," in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'08*, 2008, pp. 1–7.

[85] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "Cmp$im: A pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, 2008, pp. 28–36.

[86] R. Preissl, T. Kockerbauer, M. Schulz, D. Kranzlmuller, B. Supinski, and D. J. Quinlan, "Detecting patterns in MPI communication traces," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, IEEE, 2008, pp. 230–237.

[87] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, "Using MPI communication patterns to guide source code transformations," in *Computational Science–ICCS 2008*, Springer, 2008, pp. 253–260.

[88] O. Aumage, E. Brunet, N. Furmento, and R. Namyst, "New Madeleine: A fast communication scheduling engine for high performance networks," in *In Proceedings of the Communication Architecture for Clusters Workshop (CAC 2007), workshop held in conjunction with IPDPS*, 2007, pp. 1–8.

[89] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch.," in *USENIX Annual Technical Conference*, vol. 7, 2007, pp. 261–274.

[90] M. Hobbel, T. Rauber, and C. Scholtes, "Trace-based automatic padding for locality improvement with correlative data visualization interface," in *Proceedings of the International Conference on Parallel Architectures and Compilation, PACT'07*, 2007.

[91] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace.," in *Parallel Computing (PARCO)*, vol. 15, 2007, pp. 637–644.

[92] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[93] S. Thibault, R. Namyst, and P.-A. Wacrenier, "Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework," in *Proceedings of the European conference on Parallel processing, EuroPar'07*, 2007, pp. 42–51.

[94] S. Eranian, "Perfmon2: A flexible performance monitoring interface for linux," in *Proceedings of the 2006 Ottawa Linux Symposium*, 2006, pp. 269–288.

[95] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok, "Operating system profiling via latency analysis," in *Proceedings of the conference on Operating Systems Design and Implementation, OSDI'06*, 2006, pp. 89–102.

[96] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (OTF)," in *International Conference on Computational Science*, 2006, pp. 526–533.

[97] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for ccnuma systems," in *Proceedings of the symposium on Principles and Practices of Parallel Programming, PPoPP'06*, 2006.

[98] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, IEEE, 2006, pp. 84–88.

[99] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[100] V. Danjean, R. Namyst, and P.-A. Wacrenier, "An efficient multi-level trace toolkit for multi-threaded applications," in *European Conference on Parallel Processing*, Springer, 2005, pp. 166–175.

[101] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[102] J. Reinders, "Vtune performance analyzer essentials," 2005.

[103] J. Tao and W. Karl, "Cachein: A toolset for comprehensive cache inspection," in *Proceedings of the International Conference on Computational Science, ICCS'05*, 2005, pp. 174–181.

[104] W. E. Cohen, "Tuning programs with oprofile," *Wide Open Magazine*, vol. 1, pp. 53–62, 2004.

[105] F. Wolf, B. Mohr, J. Dongarra, and S. Moore, "Efficient pattern search in large traces through successive refinement," in *Euro-Par 2004 Parallel Processing*, Springer, 2004, pp. 47–54.

[106] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, IEEE, 2003, pp. 265–275.

[107] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and lessons learned with a portable interface to hardware performance counters," in *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS'03*, 2003, pp. 289.2–.

[108] J. Lee, C. Park, and S. Ha, "Memory access pattern analysis and stream cache design for multimedia applications," in *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, IEEE, 2003, pp. 22–27.

[109] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[110] J. Caubet, J. Gimenez, J. Labarta, L. DeRose, and J. Vetter, "A dynamic tracing mechanism for performance analysis of openmp applications," in *Proceedings of the International Workshop on OpenMP Applications and Tools01*, 2001, pp. 53–67.

[111] A. Snavely, N. Wolter, and L. Carrington, "Modeling application performance by convolving machine signatures with application profiles," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, IEEE, 2001, pp. 149–156.

[112] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[113] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[114] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," 1996.

[115] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, 1995, pp. 17–31.

[116] D. A. Reed, P. C. Roth, R. A. Aydt, K. A. Shields, L. F. Tavera, R. J. Noe, and B. W. Schwartz, "Scalable performance analysis: The pablo performance analysis environment," in *Proceedings of Scalable Parallel Libraries Conference*, 1993, pp. 104–113.

[117] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

ECOLE DOCTORALE

Titre : Contribution à l'analyse automatique de performances d'applications parallèles

Mots clés : Analyse de performance, programmation parallèle, calcul hautes performances

Résumé : Le calcul haute performance est désormais une ressource stratégique car il permet de simuler des phénomènes physiques complexes afin de mieux les comprendre. Alors qu'il y a dix ans, le calcul haute performance était surtout utilisé dans des domaines spécifiques tels que la modélisation du climat, les prévisions météorologiques ou la biologie moléculaire, il s'étend désormais à la plupart des disciplines scientifiques, y compris la génomique et l'intelligence artificielle. Le traitement de grandes quantités de données nécessite d'exploiter efficacement des machines parallèles et distribués. Lors de la conception d'une application parallèle, un développeur doit comprendre comment l'application s'exécute sur un supercalculateur.
Nos activités de recherche visent à concevoir des outils qui soulagent le développeur de ce fardeau. Nos travaux de recherches nécessitent deux phases : collecter des données lors de l'exécution d'une application et analyser les données collectées afin d'améliorer les performances de l'application.
Dans une première contribution, nous présentons plusieurs outils de collecte de performances. Nous présentons EZTrace, un outil permettant de tracer l'exécution d'applications parallèles. Il permet aux utilisateurs de capturer facilement le comportement de leur application. Nous présentons également NumaMMA, un profileur mémoire qui trace les applications et analyse leurs schémas d'accès à la mémoire. Notre deuxième contribution porte sur l'analyse automatique de performances. Nous avons développé des algorithmes qui détectent la structure globale d'une application à partir d'une trace d'exécution et qui éliminent les information superflues. Nous avons également conçu une métrique capable de détecter tout type de problème de contention en utilisant une technique d'analyse d'exécution différentielle. Enfin, nous avons conçu une méthodologie pour détecter des problèmes de scalabilité dans les applications OpenMP.

Title : Contribution to automatic performance analysis of parallel applications

Keywords : Performance analysis, parallel programming, high performance computing

Abstract : High Performance Computing is now a strategic resource as it allows to simulate complex phenomena in order to better understand them. While ten years ago, HPC was mostly used in specific domains such as climate research, weather forecasting, or molecular modeling, it now spreads to most scientific disciplines including genomics and artificial intelligence. Processing large amounts of data requires to efficiently exploit parallel and distributed computers. When designing a parallel application, a developer has to understand how the application executes on a supercomputer.
Our research activities aim at designing tools that relieve the developer from this burden. Our research necessitates two phases : to collect data when an application executes, and to analyze the collected data in order to improve the application performance.
Our first contribution consists of several performance collection tools. We present EZTrace, a tracing framework for parallel applications that allows users to easily capture the behavior of their application. We also present NumaMMA, a memory profiler that traces applications and analyzes their memory access patterns.
Our second contribution focuses on automatic performance analysis. We developed algorithms that detects the overal structure of an application from an execution trace and that filters out duplicate information. We also design a versatile metric that detects any kind of contention problem by using a differential execution analysis technique. Finally, we designed a methodology for detecting scalability issues in OpenMP applications.

Institut Polytechnique de Paris
91120 Palaiseau, France