POLITECNICO DI TORINO

Master's Degree in Computer Engineering
Data Science

- & -

UNIVERSITAT POLITÈCNICA
DE CATALUNYA (UPC) - BarcelonaTech

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

Master in Innovation and Research in Informatics
High Performance Computing

Related to an Internship at

Barcelona Supercomputing Center (BSC)

Master Thesis

Tracing methodologies and tools for Artificial Intelligence and Data Mining Java applications

Supervisors

Prof. Maria Luisa GIL GOMEZ[†]

Prof. Paolo GARZA[††]

[†]Department of Computer Architecture (DAC), UPC
[††]DAUIN, Politecnico di Torino

Author

Roberto STAGI

July 2020

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida

placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra

metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

AOP

Aspect Oriented Programming

API

Application Programming Interface

BSC

Barcelona Supercomputing Center

HPC

High Performance Computing

JNI

Java Native Interface

JVM

Java Virtual Machine

JVMTI

Java Virtual Machine Tool Interface

Chapter 1

Introduction

1.1 Context: High Performance Computing, Artificial Intelligence and Java

Supercomputing and Artificial Intelligence are among the most important outcomes of the last decades. Both of them have been behind the scenes of many recent discoveries, correctly credited to other classes of instrumentation (e.g. the Hubble telescope), but that required supercomputing and AI as the enabling tools for large datasets processing—usually referred to as "Big Data" [1] [2].

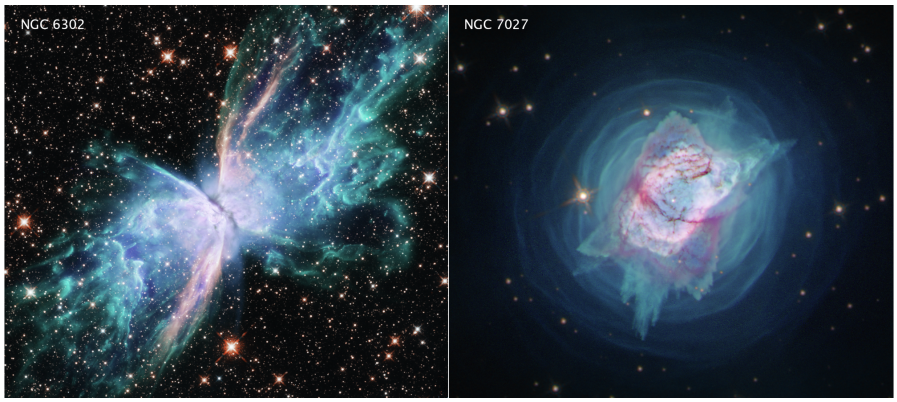

Figure 1.1: Two Planetary Nebulas photographed by the Hubble Telescope

AI applications, from Deep Learning to Data Mining, going through Neural Networks and Clustering algorithms, together with most of the applications in general,

have been switching from a sequential paradigm to parallel and distributed approaches, that best fit the new hardware. The High Performance Computing (HPC) discipline is at the heart of these developments.

HPC is a field of endeavor that relates to all facets of technology, methodology, and application associated with achieving the greatest computing capability possible at any point in time and technology. The action of performing an application on a supercomputer is widely termed "supercomputing" and is synonymous with HPC (T. Sterling *et al* [1, p. 3]).

In this context, the Java programming language plays a marginal role. Languages such as R and Python are much more common when manipulation of Big Data and statistic analysis are the primary goals [3]. However, Java is still in high demand, it is employed in AI and runs effectively on supercomputers. Even if a smaller set of programmers use it for HPC applications, its influence in the AI world is not negligible and it deserves a larger attention to the tools that support its development in such environment.

1.1.1 Java-powered AI and Data Mining

The high and always increasing demand of AI features has affected almost all the programming languages. Research institutions and companies started to invest on AI and Machine Learning [4]. Java, as one of the most common languages, got a bunch of new libraries to enable the developers to access this various world, made of statistics and algorithms. Among all the frameworks for AI, Machine Learning and Data Mining, the ones listed below are probably the most common ones employed with Java. Worthy to be in a resume and capable of figuring in the skills requirement of some tech careers.

Weka The Waikato Environment for Knowledge Analysis (Weka) is an open source software developed at the University of Waikato, in New Zealand. The Weka workbench is a collection of machine learning algorithms and data preprocessing tools, providing a Java library and a graphical User Interface to train and validate data models. It is among the most common Machine Learning frameworks for Java, since it was one of the first ones and it is still maintained [5].

Apache Spark MLlib Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Spark Core provides distributed task dispatching, scheduling, and basic I/O functionalities, exposed through an application programming interface (for Java, Python, Scala, and R) centered on the Resilient Distributed Dataset (RDD) abstraction. RDD is

a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way [6]. Spark MLlib is a distributed machine-learning framework on top of Spark Core that implements many machine learning and statistical algorithms, simplifying large scale machine learning pipelines [7].

Apache Mahout Apache Mahout is a distributed linear algebra framework, written in Java and Scala, whose architecture is built atop a scalable distributed platform. Although Apache Spark is the recommended one, Mahout supports multiple distributed back-ends. The framework features console interface and Java API, that give access to scalable algorithms for clustering, classification, and collaborative filtering [8].

1.1.2 Distributed Java in HPC

The above frameworks are not thought for an HPC environment. The standard implementation of Weka, for example, is designed to run on standard machines (like PCs, laptops or small servers), with most of the algorithms implemented sequentially. This makes it difficult to gain advantage of a strongly parallel architecture like a supercomputer. Spark and Mahout, instead, run both on a distributed platform, which means that they're designed to run on a cluster of different machines instead of a unique system. Java is indeed perfectly suitable to work on a distributed environment, usually using frameworks like MapReduce[1], whose most common implementation is Apache Hadoop, written in Java.

Spark has its own core that work in a similar fashion, Mahout runs on a distributed backend and Weka too can go distributed with some packages, running on frameworks such as Spark or Hadoop [10]. All of them rely directly or indirectly on the map-reduce framework.

Although a supercomputer and a distributed environment, made of nodes connected over a network, are similar from the physical perspective, they're much different when speaking about logic organization. Nevertheless, a distributed system can be deployed on a supercomputer, by keeping the logic organization and taking advantage of its computational power. Such emulated environment can be reached with the use of software containers.

Software containers are a form of OS-level virtualization introduced by Docker. Even if an emulated distributed system can be made of Docker containers, the BSC

[1]MapReduce is a programming model for processing big data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a map procedure, which performs filtering and sorting, and a reduce method, which performs a summary operation [9].

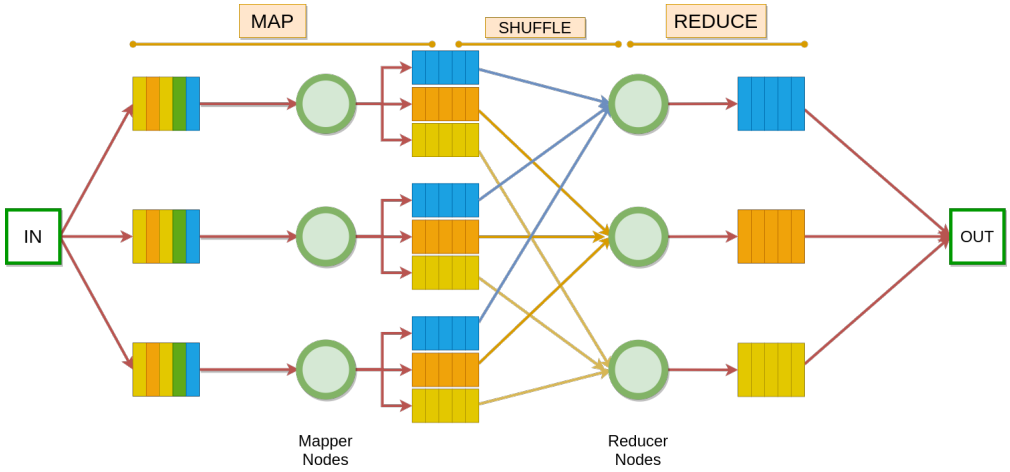

Figure 1.2: How the map-reduce framework works. The nodes can be virtualized using containers.

choice for MareNostrum is Singularity, another type of software containers created as an alternative to Docker for Clusters HPC environments[2].

Singularity enables users to have full control of their environment. Singularity containers can be used to package entire scientific workflows, software and libraries, and even data [11]. Distributed applications based on Hadoop or Spark can run on supercomputers, by using virtualized clusters made of Singularity containers.

1.1.3 Performance analysis

Quoting prof. Jesus Labarta, one of my professors at the UPC, «measurement techniques are "enablers of science". They are present everywhere, and so they are in Computer Science. In the specific case of HPC, they are represented by the performance analysis tools.»

Parallel program performance analysis and tuning is concerned with achieving efficient utilisation of system resources. One common technique is to collect trace data and then analyse it for possible causes of poor performance. The objective is to gather insights, both qualitative and quantitative, in order to increase predictability, build confidence and properly suggest improvements in the software [12].

A great way to describe the observed behaviour is through performance models, which take in consideration multiple model factors based on specific metrics (load

[2]Indeed, that is a common choice in most of the supercomputers worldwide.

balance, communication efficiency, etc.), making it easier to understand where to act.

At the BSC, performance analysis methodologies are mainly based on "traces", generated by instrumenting the libraries and taking several kinds of data from hardware counters, system calls or other low level mechanisms.

1.2 MareNostrum Tools Environment

The work in the present thesis is carried out as an intern at the Barcelona Supercomputing Center (BSC), that has one of the most powerful supercomputers in the world. It is situated right next to the Campus North of the UPC, and the work in the present thesis is carried out as an intern at the BSC.

The BSC supercomputer is called MareNostrum, and it is at its 4th version. With 6,470.8 TFlop/s of max performance, at the time of writing the MareNostrum occupies the 30th position among the most powerful supercomputers in the world, according to the top500 list [13]. It has 165,888 cores, divided on 3,456 nodes, and counts more than 300 TB of memory [14].

Figure 1.3: MareNostrum 4

The MareNostrum software environment is based on the SUSE Linux Enterprise Server Operating System. There are several available modules ready to be loaded, and many of them are developed within the BSC research departments. Among these modules, we find the performance analysis tools developed at the Performance Tools Department of the BSC: Extrae and Paraver.

1.2.1 Extrae

Extrae is a tool that uses different interposition mechanisms to inject probes into the target application, in order to gather information regarding the application behaviour and performance. It uses different interposition mechanisms to inject the probes, and generates trace-files for an analysis carried after the termination of the program [15].

The most common interposition mechanism is the Linker preload "trick". It consists in setting the `LD_PRELOAD` environment variable to the path of the Extrae tracing library, before executing the target program. In this way, if the preloaded library contains the same symbols of other libraries loaded later, it can implement some wrappers for the functions valuable to get data from. This is what Extrae does for most of the instrumented programming models. Another mechanism consists in manually inserting some probes. On top of the instrumented frameworks, Extrae provides an API which gives the user the possibility to manually instrument the application and emit its own events.

Extrae is subjected to different settings, configured through an XML file specified in the `EXTRAE_CONFIG_FILE` environment variable. These settings have the control on almost everything in the instrumentation process. They can enable the instrumentation of some libraries instead of others, the hardware counters, the sampling mode, etc. The tracefiles generated by Extrae are formatted to comply with the Paraver trace format.

1.2.2 Paraver

Paraver is developed to visually inspect the behaviour of an application and then to perform detailed quantitative analysis. It has a clear and modular structure, which gives to the user expressive power and flexibility.

The Paraver trace format has no semantics. Thanks to that, supporting new performance data or new programming models requires only capturing such data in a Paraver trace. Moreover, the metrics are not hardwired on the tool, but programmed. To compute them, the tool offers a large set of time functions, a filter module, and a mechanism to combine two time lines.

Preserving experts knowledge is made very simple. In fact, any view or set of views can be saved as a Paraver configuration file, which reduces the re-computing of the view with new data to a simple loading of a file [16].

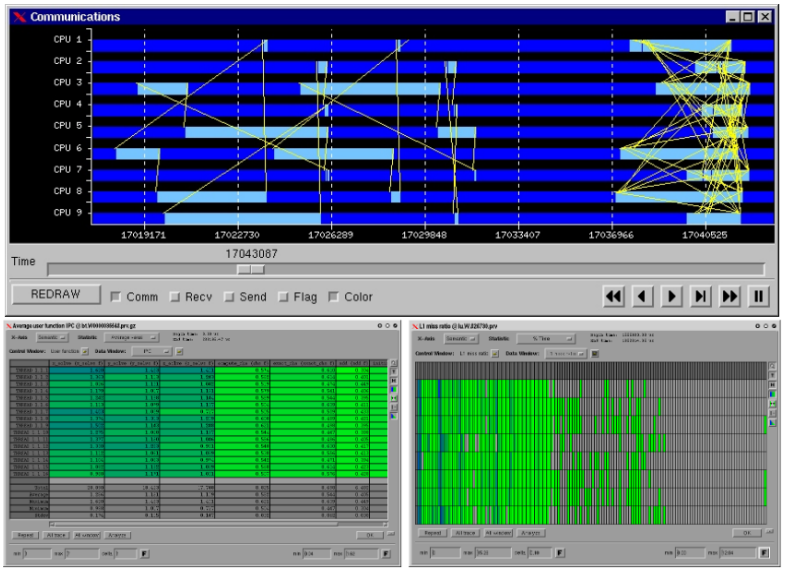

Figure 1.4: Example of traces analysis using Paraver

1.3 Problem Statement and Goal

The final purpose of this thesis was to find some patterns and methodologies that can help analysing the applications based on an HPC Java implementation, by instrumenting the frameworks they rely on. The focus was on the AI frameworks, which represent a large portion of the Java applications that typically run on supercomputers.

Since I developed this thesis as an intern at the Barcelona Supercomputing Center (BSC), the starting basis for the work will be the BSC performance tools, that are not suited for analyzing Java applications. The interest for the company—and so my job as an intern—was to improve such tools (Extrae, mainly) and to gain the right experience in analyzing the typical HPC applications implemented in Java.

1.4 Materials and Methods

If not specified, all the executions reported in this thesis ran on a Dell XPS15 9570, with an Intel i7-8750H CPU at 2.20GHz and 32GB of DDR4 RAM. The programming tools employed in the work have been the following:

- Visual Studio Code for all the `C/C++` code
- Jetbrains IDEA IntelliJ for all the Java/AspectJ code

- Docker to virtualize the OS and the tools environment
- GitHub to store all the code and manage Extrae's pull requests and versioning

All the examples are easily reproducible thanks to the virtualization offered by the Docker image. The image is based on OpenSUSE, provides the BSC tools (Extrae and Paraver) and all the other dependencies (AspectJ, Java, etc.) are installed and ready to be used. The reason behind choosing OpenSUSE is because of the MareNostrum 4 Operating System (SUSE Linux Enterprise Server), of which OpenSUSE—being its free version—is the most similar OS that could be virtualized with a Docker image.

An explanation of the usage of the environment, including the steps to build and setup the image, as well as the instructions on how to run the examples, is present in the Appendix A.

Chapter 2

Extrae for JAVA: State of the Art

If it is true that Extrae is not suitable to analyze Java applications, this does not mean that no features were implemented at all. Some basic instrumentation was implemented years ago, but due to its scarce use, it ended to be unmaintained.

This chapter will go through the state of the art of Extrae's instrumentation for Java. It will do that by looking at one at the Extrae's Java example, provided in the Extrae package.

2.1 The example program

The program that is going to be analyzed is a simple algorithm to calculate π. It does so 5 times: the first time with a sequential algorithm, the other four with a parallel implementation, respectively with 1, 2, 4 and 8 threads.

The main class is shown in the following Code 2.1[1]:

Code 2.1: PiExample.java

```
1 public class PiExample
2 {
3   public static void main (String [] args)
4   {
5     PiSerial pis = new PiSerial (steps);
```

[1]In reality, that is just an extract of the real main class. The real one contains some time measurements and print messages. The whole code is available in the Appendix B

```
6       pis.calculate ();
7
8       PiThreaded pit1 = new PiThreaded (steps, 1);
9       pit1.calculate ();
10
11      PiThreaded pit2 = new PiThreaded (steps, 2);
12      pit2.calculate ();
13
14      PiThreaded pit4 = new PiThreaded (steps, 4);
15      pit4.calculate ();
16
17      PiThreaded pit8 = new PiThreaded (steps, 8);
18      pit8.calculate ();
19    }
20  }
```

The `PiThreaded` class is responsible to create and run the threads:

Code 2.2: PiThreaded.java

```
1   public class PiThreaded
2   {
3     long m_n;
4     double m_h;
5     Vector<PiThread> m_threads;
6
7     public PiThreaded (long n, int nthreads)
8     {
9       m_n = n;
10      m_h = 1.0 / (double) n;
11
12      m_threads = new Vector<PiThread>(nthreads);
13      for (long i = 0; i < nthreads; i++)
14      {
15        m_threads.addElement (
16          new PiThread (m_h,
17            (n/nthreads)*i,
18            (n/nthreads)*(i+1)-1)
19        );
20      }
21    }
22
23    public void calculate()
24    {
25      /* Let the threads run */
26      for (int i = 0; i < m_threads.size(); i++)
27        (m_threads.get(i)).start();
28
```

```
29      /* Wait for their work */
30      for (int i = 0; i < m_threads.size(); i++)
31      {
32        try { (m_threads.get(i)).join(); }
33        catch (InterruptedException ignore) { }
34      }
35    }
36
37    public double result()
38    {
39      double res = 0.0;
40      for (int i = 0; i < m_threads.size(); i++)
41      {
42        /* reduce the value to result */
43        res += (m_threads.get(i)).result();
44      }
45
46      return res;
47    }
48  }
```

The classes `PiThread` and `PiSerial`, here omitted for brevity, are both responsible for the actual calculation, but the former extends Java's `Thread` class. The whole code is available in the Appendix B.

2.2 Generate the traces

Generating the traces for a Java program is done using a launcher script installed by Extrae, named `extraej`. Assuming to use the Extrae XML file contained in the examples, and assuming to have the Java class to test (named `PiExample` in this case) in the Class Path[2], the following command will launch the program and generate the traces:

```
EXTRAE_CONFIG_FILE=extrae.xml extraej -- PiExample
```

This command will generate a file called `PiExample.prv` (the name is specified in the `extrae.xml` file). By analyzing the file using Paraver, the result is shown

[2]The class path is the path where the applications, including the JDK tools, look for user classes. The default value of the class path is " . " (dot), meaning that only the current directory is searched. Specifying either the `CLASSPATH` variable or the `-cp` command line switch overrides this value [17]. By default, `extraej` does not override it, and so the compiled `.class` file must be in the current directory, or the `CLASSPATH` variable should be set accordingly.

in Figure 2.1. As it can be seen, there are not much states shown, but there are several threads detected and traced.

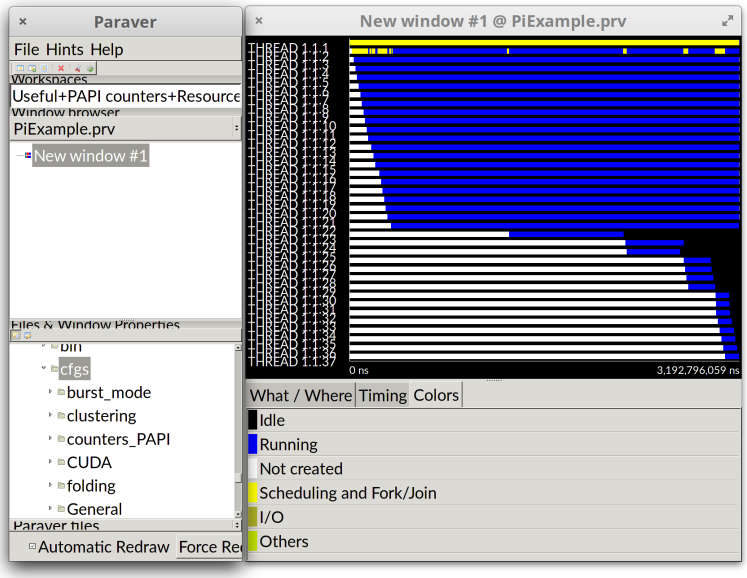

Figure 2.1: PiExample resulting traces

As described in the Introduction, the most common way to instrument an application using Extrae is by using the `LD_PRELOAD` environment variable. This mechanism can be used for any program running "directly" on Linux. For this reason, even if it does not directly affect the Java program, it can be used to instrument the Java Virtual Machine (JVM).

Having a look at the `extraej` script, focusing on the part responsible to launch the program with instrumentation (Code 2.3), it can be seen how it actually uses the `LD_PRELOAD` method. The complete `extraej` code is in Appendix B.

Code 2.3: Extract of extraej showing the launching command

```
191     LD_PRELOAD=${preload} \
192     CLASSPATH=${cp} \
193       ${JAVA} ${@}
```

However, Extrae does not provide any direct instrumentation of the JVM. So how can it extract such data, without any ad-hoc instrumentation probes? The answer can be found in a mechanism that is not strictly related to Java, but to `C` and Linux: pthreads.

2.3 Pthread instrumentation

Pthread is one of the libraries instrumented by Extrae. The probes injection to trace the states of the Linux threads is done by implementing the wrappers of the valuable pthread functions like `pthread_create` or `pthread_join`.

The Java Virtual Machine (JVM) is mostly implemented in C, and for its Linux implementation each Java Thread is mapped on a POSIX thread (pthread) [18]. It is for this reason that Extrae's pthread instrumentation is effective with Java programs too.

However, this instrumentation is not enough to show all the details of a Java execution. Apparently, out of the pthread creation and termination, no other pthread calls are normally employed in the JVM. That can be easily seen by the previous example, where the synchronization operations present in the Java implementation are not traced.

To let Extrae use pthread instrumentation, it must be enabled in the `extrae.xml` file as in the following piece of code (the full XML file can be found in **??**).

Code 2.4: extrae.xml

```
1  <pthread enabled="yes">
2    <locks enabled="no" />
3    <counters enabled="yes" />
4  </pthread>
```

2.4 Traces analysis

By looking at the traces, several threads can be seen running throughout the process. The reported events include just "running", "join" and "I/O" events. Without knowing anything about the code and with such poor data gathered by the performance analysis, it would have been almost impossible to understand what each thread is doing. Recalling what the program does, it seems that the number of threads should be lower than the ones appearing on the traces. Indeed, the total should be $1 + 1 + 2 + 4 + 8 = 16$ threads, but on the traces the number of threads is 37.

To understand what is going on, one could try to execute the program with a different number of threads, to see how this additional number of threads behaves. By trying to instrument the sequential single-threaded program, the resulting traces are the one in Figure 2.2.

The first thread looks exactly the same, while in the second one there are some

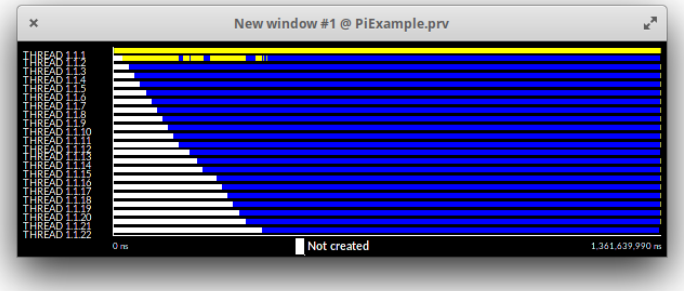

Figure 2.2: Trace of serial PiExample execution

differences related to the absence of the other threads (the yellow chunks are missing). The threads numbered from 3 to 22, instead, look the same in both of the tracefiles. A guess[3] that can be made is that such threads are inherently launched by the JVM for each execution (Figure 2.3).

Figure 2.3: PiExample trace explained

As we will see in the following chapters, this guess will appear to be mostly correct.

[3]The methodology behind performance analysis expects some level of "guessing" when looking at the traces.

2.5 Extrae Java API through JNI implementations

When installing Extrae with Java enabled, this will install a basic instrumentation library based on JNI bindings[4]. The purpose of this library is to inject probes manually, by putting events in certain positions of the code during the development phase. This kind of events, however, do not provide any way to signal new threads, but rather they are used as "markers" for specific portions of code, relying on other mechanisms to discover on which thread they're being called—like the pthreads identifiers, in relation to the pthread instrumentation seen earlier.

The work in this thesis is focused on the mechanisms described above, and to the instrumentation of a framework. These JNI bindings will play a useful role in the later stages of the work.

2.6 Experimental features

Besides the features presented in this chapter, Extrae's reference presented a couple described as "experimental".

2.6.1 Java Virtual Machine Tool Interface

The first one is a tracing platform based on the the JVM Tool Interface (JVM TI), a native programming interface thought for tools development. It provides both a way to inspect the state and to control the execution of applications running in the JVM. JVM TI supports the full breadth of tools that need access to JVM state, including but not limited to: profiling, debugging, monitoring, thread analysis, and coverage analysis tools. It is available as an API for C/C++, and once the library is implemented, it can be used as an agent for the JVM [20].

Comes naturally to think that it would be a useful tool to be used to gather data for the traces—and it actually is. The idea was to implement it in Extrae, and inject it as an agent when executing the program with `extraej`.

However, despite being present among the features, it was never fully implemented. It was thought to be used as a threads-tracing mechanism instead of pthread. The problem in this solution was about identification, because the JVM does not

[4]JNI stands for *Java Native Interface*. The JNI is a native programming interface. It allows Java code that runs inside a JVM to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. [19].

provide the JVM TI with a unique ID for the threads, which is instead required by Extrae architecture. Indeed, this one was one of the issues addressed (and solved) in this thesis.

2.6.2 AspectJ for User Functions

The second experimental feature sees AspectJ, an Aspect Oriented Programming[5] extension for Java, as the tool to generate the events for the user functions. In this context, the "user functions" are those functions that the user would like to see on the traces. It should basically do what the JNI bindings were designed for, but in an automatic way by wrapping the target functions with some events, traced by Extrae—and so, placing custom events without modifying the code. Indeed, wrapping the methods with some code is basically the main purpose of AspectJ—and more in general of the Aspect Oriented Programming paradigm. The wrapper code calls the JNI implemented Extrae API library introduced in the previous section to trace the user functions events. The target user functions are listed in a file, whose path is passed to Extrae through the XML configuration file (i.e. `extrae.xml`).

This feature was also not working, but this time due to some bugs in the installation process. Once fixed, it was possible to use the "User functions" configuration of Paraver to look at them (Figure 2.4).

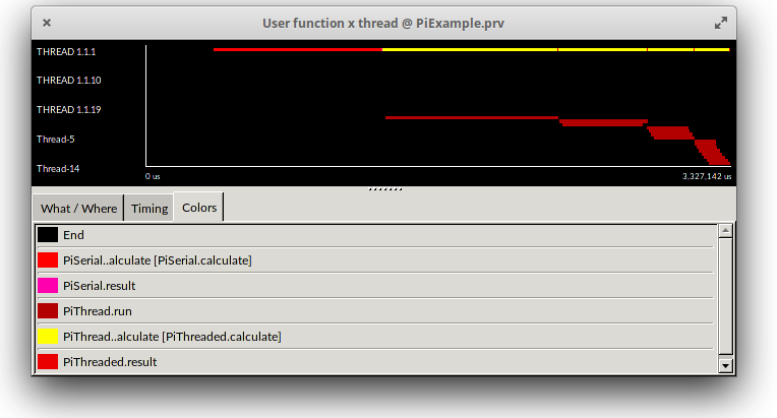

Figure 2.4: Trace showing the user functions for the PiExample program

[5]An introduction to AOP can be found in the next chapter.

2.7 Where to go from here

In this overview there are some specific issues that emerged.

The first issue to be addressed is the lack of specific events related to Java. The only ones traced by Extrae totally rely on the pthreads instrumentation.

The second one is the presence of some bugs on the already present Java instrumentation, that make Extrae not working properly with Java programs. Although this process has been carried out in parallel to the rest of the work, the bug-fixing operations will be omitted in the thesis discussion—unless they are particularly related to the objective, or some interesting cause and solution were found.

The final one is the absence of a mechanism to trace distributed applications. Since it is probably the most common way of executing Java on HPC, it would be interesting to inspect this field and try to find a solution for this problem.

Chapter 3

Java Tracing Methodologies

The main focus of this thesis is on Extrae, because the main issue is on extracting the application performance details from Java program, and not on how to visualize them. Indeed, as it has been said in the Introduction, Paraver is quite flexible and does not need any change in the code to effectively depict the events for a new instrumented framework.

Since Extrae is implemented in `C`, generating probes and wrappers would not be an issue for other C-implemented programs. Unfortunately, generating traces for a Java program cannot be so straight forward, but there are some approaches that could be tried out to extract the data needed to generate an effective trace. In this chapter there is an overview of the approaches studied in this thesis.

NB: all the discussed approaches, in order to trace the events, need to interface with Extrae's functions at some point. Such events must be coded and globally identified through the use of some constants. However, besides this small clarification, all the implementation details of such approaches are left for the next chapters.

3.1 Linker Preload approach

Once got used to Extrae, the first approach that comes to mind is the one of instrumenting the JVM using the linker preload. This kind of instrumentation expects to find the valuable functions inside the JVM, in order to define some wrappers for them, with the purpose of injecting the probes responsible for the tracing activity.

Although valuable for many frameworks, in this case this approach has not been analyzed at all. JVM internals are not standardized, and so they are not defined and immune to changes. There is no way to know that some function used in one version will be maintained in successive releases. For this reason, despite being worth a mention because of its relatedness with Extrae's standard approach, it will not be discussed further—except for possible comparison purposes.

3.2 Event-driven instrumentation

A more convenient way to trace the JVM states would be by catching the JVM events. This kind of work can be done thanks to the interface provided by the Java language: the JVM Tool Interface (JVM TI).

This approach expects to use an event-driven platform, in which the events launched by the JVM are catched by some functions containing the tracing instructions. The JVM TI allows to do that by setting user-defined callbacks, able to catch several JVM events—like thread start and end, method entry and exit, garbage collection, object allocation, etc. [21]

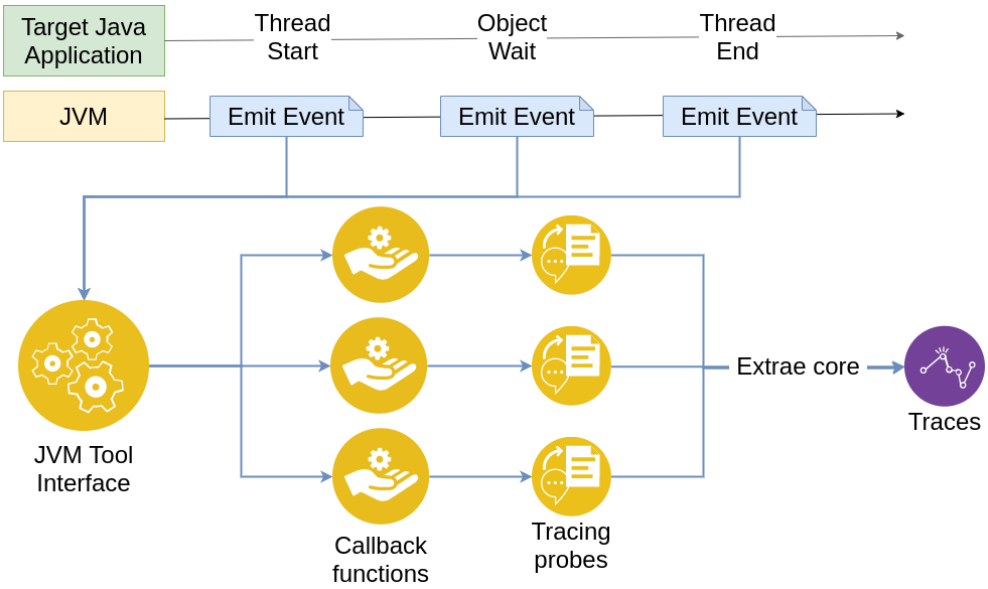

Figure 3.1: Visual explanation of event-driven approach

JVM events are essential for profiling programs, but basing the whole instrumentation on them would be somehow limiting. There are many different events catchable, but there may be interesting data depending on specific functions that do not raise any event. For example, many frameworks use busy-waiting loops implemented

in custom functions to wait for something, instead of the native `Object.wait()` method—that raises an event.

The JVM TI provides two events that may be useful in this case: method entry and method exit events. Such events are raised for each method executed by the JVM, respectively at entry and exit point. To trace some interesting methods it would be enough to monitor all these events, waiting for the interesting ones to trace the related events. Using these events is highly discouraged by the JVM TI reference guide, because of its large impact on performances. Speaking of traces in general, if the retrieved information is valuable, the great overhead generated by a callback function for each method can be acceptable. However, the problem of instrumenting specific functions can be solved in other ways. These methods are explained in the following sections.

3.3 Bytecode and Native Instrumentation

Another approach is by instrumenting the specific Java methods, using a technique called "bytecode instrumentation".

Bytecode instrumentation is a way of injecting custom instructions inside other classes, without directly modifying the code. The JVM provides some events and control functions to transform the bytecode of classes and methods. The JVM—if enabled to do it—fires an event when a class or a method are loaded. In both cases, it provides the possibility to catch the event and to read the loaded bytecode, in order to modify it[1] and re-load it again. This possibility is given through the Instrumentation API, available for both the Java language, in the package `java.lang.instrument`, and for the JVM TI, through a specific set of events and functions.

Since the injected instructions are in form of bytecodes, the instrumentation probes need to be in a form of callable Java methods. In the case of Extrae, since it is written in `C` language, such methods must rely on JNI implementations of the probes, either directly (the injected bytecodes call a JNI implemented method) or indirectly (the injected bytecodes call a Java method, which in the end calls calls one or more JNI implemented methods). The JNI implementations need to be written in `C` language, and these must rely on the Extrae's functions to trace the events.

[1] According to JVM TI reference guide [21], the modifications inserted through bytecode instrumentation can only be purely additive. Moreover, it is not possible to add new methods inside a class o to modify the signature of the other methods.

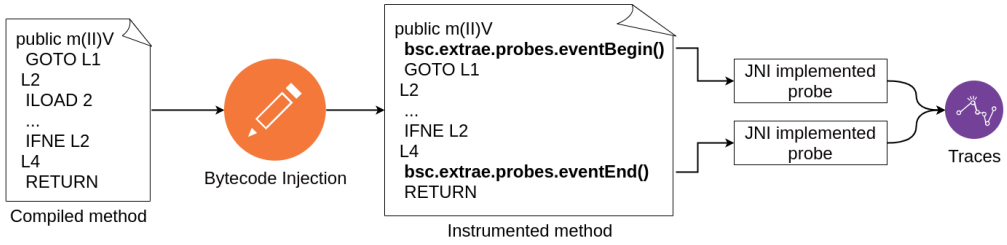

Figure 3.2: Visual explanation of the bytecode instrumentation approach

3.3.1 Bytecode manipulation in `C` and Java

Injecting bytecode is conceptually easy to understand, but the implementation can be tricky. The JVM provides all the tools to catch when the compiled binary is loaded and to retrieve the related bytecodes. However, these bytecodes are returned in a form of nothing more than long arrays of bytes. Injecting the bytecodes in this binary, requires essentially an intelligent array manipulation work. There are several frameworks implemented in Java for bytecode manipulation[2], but none is available for `C` language[3].

A nontrivial problem is represented by exceptions handling. For its nature, bytecode instrumentation is simply an addition in given program places. For this reason, if the program raises an exception before reaching the end of the method, it would never execute the bytecodes responsible to trace the end of the event. A possible solution would be to catch the event—available for the JVMTI—of the Exception raise, by using the agent, and to develop a mechanism to trace the end of the currently traced method.

3.3.2 Native methods instrumentation

Native methods are those methods available to be called in Java, but implemented natively in `C` language. As for the probes discussed in the previous section, this can be done thanks to the JNI. Having a C-implementation, these methods are not compiled in bytecodes, and so are not suitable to be instrumented by injecting bytecodes. This problem is solved by the JVM TI, which provides a way to change the name of natively implemented methods, in order to define a wrapper function (similar to what the Linker Preload method does).

[2]Among the most popular, there are ASM and Javassist.

[3]To be fair, I could find one for `C++`, but since Extrae is written in C it resulted to be quite complex to cross compile it effectively (I tried, but I had many problems at running time with the linked functions).

For the instrumentation in Java language, this problem cannot be solved by using the Java instrument API[4]. Instead, this, and other problems, can be solved thanks to the approach explained in the next section: AspectJ.

3.4 Aspect Oriented Programming approach

Aspect Oriented programming (AOP) is a programming paradigm first introduced by G. Kiczales *et al* [22], that aims to add additional behavior to existing code without modifying the code itself. AspectJ, as one of the most popular aspect-oriented frameworks, has been chosen for this thesis.

The way of using it for tracing purposes is similar to the bytecode instrumentation approach, with the only difference in the practical implementation. Instead of using JVM events on loaded classes and inject custom bytecodes, with AspectJ it is possible to define a "pointcut" for each interesting method, and then defining the behaviour of the application before and/or after the execution [23]. This behaviour is implemented as simple Java code and, as it was for the bytecode instrumentation, it would call the JNI methods to trace the specific events.

For tracing purposes AspectJ looks very powerful, because it combines an easy Java implementation of the probes (without bytecode manipulation) and the solution to many problems given by the previous approach, like native methods instrumentation or exception handling.

3.5 Discussion on the methodology to adopt

Before trying there are no definitive conclusions to make. However, there are some approaches more promising than others. The solution proposed in this thesis will not rely on one single approach, but rather a combination of them.

Considering the needs of tracing threads behaviour and specific methods (either native or standard Java methods), the following configurations are viable solutions.

JVM TI based tracing The JVM TI is responsible to trace the events fired by the JVM, but also for the bytecode and native methods instrumentation at each

[4]The reason can be found in how the bytecode injection is designed. Indeed, it consists in actually inserting instructions in specific points of the bytecodes, almost like inserting instructions to specific lines of codes. For this reason, it is impossible to inject bytecode in a native method, which is compiled into a binary executable and not into bytecodes.

class load time. Exception handling must be managed to keep coherence among the states, and bytecode needs to be manipulated in native language.

AspectJ based tracing The tracing software would rely totally on AspectJ and JNI implemented probes. Being based on methods and not JVM events, it needs some care when defining when a new thread is created, since it could happen in different ways.

JVM TI with the aid of Java instrumentation API The JVM TI is responsible to catch and trace the JVM events, while the Java instrumentation API is responsible to trace the specific methods. Exception handling must be managed to keep coherence among the states, and the native methods must be instrumented using the JVM TI. Java instrumentation needs JNI methods for the natively implemented probes.

JVM TI with the aid of AspectJ As the previous case, but employing AspectJ to trace the Java and native methods. Moreover, exception handling can be omitted, since adding behaviour at the end of a method can be done even when an exception is raised, without extra development effort. The drawback, with respect to the previous method, would be the required dependency of AspectJ, that needs to be installed and supported by the OS on which Extrae is going to install.

All of the above solutions require the tracing platform to be compiled as a shared library, in the case of `C`/`C++`, or a Java archive (JAR), in case of Java and AspectJ. In any case, they need to run as a Java agent during the target application execution.

This thesis considers the JVM TI events as tools of undoubted value. For this reason, the solution studied in the following chapters is based on the last configuration. The JVM TI will be employed for basic events tracing, but also for some native methods instrumentation. For methods tracing instead, AspectJ was preferred to bytecode instrumentation because of its flexibility and ease of use (and maintainability)[5].

[5]As a side note, it would be fair to say that the first solution based totally on JVM TI would be much appreciated, if Extrae was implemented in `C++`. Being implemented in `C`, it makes bytecode manipulation hard to manage. In `C++`, instead, there are some frameworks available that would have made it much simpler. I made an attempt to implement it and try to compile Extrae by mixing `C` and `C++`, but I couldn't make the linking at run-time to work properly. It would have been nice to create it externally to Extrae and compare it to the AspectJ solution, in terms of usability and generated overhead.

Chapter 4

Tracing threads on a single JVM

4.1 JVM Tool Interface library

4.2 Generate the Aspects

Chapter 5

Tracing in a distributed environment

5.1 HPC and Distributed Systems

5.2 Singularity containers

5.3 Shared resources for tracing

Chapter 6

Case Study: Hadoop MapReduce

6.1 What methods to instrument

6.2 Probes implementation

6.3 Aspects generation

6.4 Distributed execution

Chapter 7

Discussion of Results

7.1 What to Look for

7.2 Tracing overhead analysis

7.3 Further Improvements

Chapter 8

Conclusions

Appendix A

Environment set-up

Appendix B

Extrae State of the Art complete code (with the Example)

Bibliography

[1] T. Sterling, M. Anderson, M. Brodowicz. *High Performance Computing. Modern systems and practices*. Cambridge, MA, USA: Morgan Kauffman, 2018 (cit. on pp. 1, 2).

[2] A. Heck, F. Murtagh. «Artificial intelligence applications for Hubble Space Telescope operations». In: *Knowledge-Based Systems in Astronomy*. Heidelberg, Germany: Springer, 1989. Chap. 1, pp. 3–31 (cit. on p. 1).

[3] Kaggle. *2019 Kaggle ML DS Survey*. Tech. rep. Kaggle, 2019 (cit. on p. 2).

[4] International Data Corporation (IDC). «Worldwide Spending on Artificial Intelligence Systems Will Grow to Nearly $35.8 Billion in 2019». In: *Worldwide Semiannual Artificial Intelligence Systems Spending Guide* (2019). URL: https://www.idc.com/getdoc.jsp?containerId=prUS44911419 (cit. on p. 2).

[5] I.H. Witten, E. Frank, M.A. Hall, C.J. Pal. *Data Mining. Practical Machine Learning Tools and Techniques*. Fourth. Cambridge, MA, USA: Morgan Kauffman, 2017. Chap. Appendix B (cit. on p. 2).

[6] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica. «Spark: Cluster Computing with Working Sets». In: UC Berkley, California, USA, June 2011, p. 1. URL: https://amplab.cs.berkeley.edu/wp-content/uploads/2011/06/Spark-Cluster-Computing-with-Working-Sets.pdf (cit. on p. 3).

[7] E. Sparks, A. Talwalkar. «Spark Meetup: MLbase, Distributed Machine Learning with Spark». In: San Francisco, California, USA, June 2013. URL: http://www.slideshare.net/chaochen5496/mlllib-sparkmeetup8613finalreduced (cit. on p. 3).

[8] G. Ingersoll. «Introducing Apache Mahout». In: *IBM developers Archives* (2009). URL: https://www.ibm.com/developerworks/java/library/j-mahout/ (cit. on p. 3).

[9] *MapReduce*. Wikipedia.org, 2020. URL: https://en.wikipedia.org/wiki/MapReduce (cit. on p. 3).

[10] A. Koliopoulos, P. Yiapanis. F. Tekiner, G. Nenadic, J. Keane. «A Parallel Distributed Weka Framework for Big Data Mining Using Spark». In: *2015 IEEE International Congress on Big Data*. New York, NY, USA, June 2015 (cit. on p. 3).

[11] *Singularity*. URL: `https://singularity.lbl.gov` (cit. on p. 4).

[12] A. Hondroudakis, R. Procter. *The Design of a Tool for Parallel Program Performance Analysis and Tuning*. Edinburgh Parallel Computing Centre. The University of Edinburgh, 1998 (cit. on p. 4).

[13] *TOP500*. URL: `https://www.top500.org` (cit. on p. 5).

[14] *MareNostrum technical information*. URL: `https://www.bsc.es/marenostrum/marenostrum/technical-information` (cit. on p. 5).

[15] *Extrae*. URL: `https://tools.bsc.es/extrae` (cit. on p. 6).

[16] *Paraver: a flexible performance analysis tool*. URL: `https://tools.bsc.es/paraver` (cit. on p. 6).

[17] *PATH and CLASSPATH*. The Java Tutorials, Oracle. URL: `https://docs.oracle.com/javase/tutorial/essential/environment/paths.html` (cit. on p. 11).

[18] U. Joshi. *How Java thread maps to OS thread?* URL: `https://medium.com/@unmeshvjoshi/how-java-thread-maps-to-os-thread-e280a9fb2e06` (cit. on p. 13).

[19] *Java Native Interface Overview*. The Java Native Interface Programmer's Guide and Specification. URL: `https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html#java-native-interface-overview` (cit. on p. 15).

[20] *Java Virtual Machine Tool Interface (JVM TI)*. Oracle Docs. URL: `https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/` (cit. on p. 15).

[21] *Java Virtual Machine Tool Interface (JVM TI) Reference Guide*. Oracle Docs. URL: `https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html` (cit. on pp. 19, 20).

[22] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. «Aspect-Oriented Programming». In: Palo Alto, CA, USA, 1997 (cit. on p. 22).

[23] *The Anatomy of an Aspect. The AspectJ Language*. Eclipse Foundation. URL: `https://www.eclipse.org/aspectj/doc/released/progguide/language-anatomy.html#pointcuts` (cit. on p. 22).