Universidade do Minho
Escola de Engenharia
Departamento de Informática

Vítor Domingos Araújo Gomes

Profiling Tools for Java

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Vítor Domingos Araújo Gomes

Profiling Tools for Java

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
João Luís Ferreira Sobral

December 2021

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RESUMO

Atualmente, Java é uma das linguagens de programação mais populares. Esta popularidade é parcialmente devida à sua portabilidade que advém do facto do código Java ser compilado para bytecode que poderá ser executado por uma máquina virtual Java (JVM) compatível em qualquer sistema. A JVM pode depois interpretar diretamente ou compilar para código máquina a aplicação Java. No entanto, esta execução sobre uma máquina virtual cria alguns obstáculos à obtenção do perfil de execução de aplicações.

Perfis de execução são valiosos para quem procura compreender o comportamento de uma aplicação pela recolha de métricas sobre a sua execução. A obtenção de perfis corretos é importante, mas a sua obtenção e análise pode ser desafiante, particularmente para aplicações paralelas.

Esta dissertação sugere um fluxo de trabalho de otimização a aplicar na procura de aumentos na escalabilidade de aplicações Java paralelas. Este fluxo sugerido foi concebido para facilitar a descoberta dos problemas de desempenho que afetam uma dada aplicação paralela e sugerir ações a tomar para os investigar a fundo.

O fluxo de trabalho utiliza a noção de *possible speedups* para quantificar o impacto de problemas de desempenho diferentes. A ideia de *possible speedups* passa por estimar o speedup que uma aplicação poderia atingir se um problema de desempenho específico fosse completamente removido. Esta estimativa é calculada utilizando as métricas recolhidas durante uma análise ao perfil de uma aplicação paralela e de uma versão sequencial da mesma aplicação.

O conjunto de problemas de desempenho considerados incluem o desequilíbrio da carga de trabalho, sobrecarga de paralelismo devido ao aumento no número de instruções executadas, sobrecarga de sincronização, gargalos de desempenho no acesso à memória e a fração de trabalho sequencial. Estes problemas foram considerados as causas mais comuns de limitações à escalabilidade de aplicações paralelas. Para investigar mais a fundo o efeito destes problemas numa aplicação paralela, são sugeridos alguns modos de visualização do perfil de execução de uma aplicação dependendo do problema que mais limita a sua escalabilidade. As visualizações sugeridas consistem maioritariamente de diferentes tipos de flame graphs do perfil de uma aplicação.

Duas ferramentas foram desenvolvidas para ajudar a aplicar este fluxo de trabalho na otimização de aplicações Java paralelas. Uma destas ferramentas utiliza o async-profiler para recolher perfis de execução de uma dada aplicação Java. A outra ferramenta utiliza os perfis recolhidos pela primeira ferramenta para estimar *possible speedups* e produzir todas as visualizações mencionadas no fluxo de trabalho sugerido.

Por fim, o fluxo de trabalho foi validado com alguns casos de estudo. O caso de estudo principal consistiu na otimização iterativa de um algoritmo K-means, partindo de uma implementação sequencial e resultando no aumento gradual da escalabilidade da aplicação. Casos de estudo adicionais também foram apresentados para ilustrar possibilidades não abordadas no caso de estudo principal.

PALAVRAS-CHAVE escalabilidade, Java, paralelo, perfil de execução

ABSTRACT

Java is currently one of the most popular programming languages. This popularity is, in part, due to the portability it offers which comes from the fact that Java source code is compiled into bytecode which can be executed by a compatible Java Virtual Machine (JVM) in a different system. The JVM can then directly interpret or compile into machine code the Java application. However, this execution on top of a virtual machine creates some obstacles to developers looking to profile their applications.

Profilers are precious tools for developers who seek to understand an application's behaviour by collecting metrics about its execution. Obtaining accurate profiles of an application is important, but they can also be challenging to obtain and to analyse, particularly for parallel applications.

This dissertation suggests an optimisation workflow to employ in the pursuit of reducing scalability bottlenecks of parallel Java applications. The workflow is designed to simplify the discovery of the performance problems affecting a given parallel application and suggest possible actions to investigate them further.

The suggested workflow relies on *possible speedups* to quantify the impact of different performance problems. The idea of *possible speedups* is to estimate the speedup an application could achieve if a specific performance problem were to completely disappear. This estimation is performed using metrics collected during the profile of the parallel application and its sequential version.

The set of performance problems considered include workload imbalance, parallelism overhead due to an increase in the number of instructions, synchronisation overhead, memory bottlenecks and the fraction of sequential workloads. These were deemed to be the most common causes for scalability issues in parallel applications. To further investigate the effect of these problems on a parallel application, some visualisations of the application's behaviour are suggested depending on which problem limits scalability the most. The suggested visualisations mostly consist of different flame graphs of the application's profile.

Two tools were also developed to help in the application of this optimisation workflow for parallel Java applications. One of these tools relies on async-profiler to collect profiles of a given Java application. The other tool uses the profiles collected by the first tool to estimate *possible speedups* and also produce all visualisations mentioned in the suggested workflow.

Finally, the workflow was validated on multiple case studies. The main case study was the iterative optimisation of a K-means algorithm, starting from a sequential implementation and resulting in the gradual increase of the application's scalability. Additional case studies were also presented in order to highlight additional paths not covered in the main case study.

KEYWORDS Java, parallel, profiling, scalability.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

INTRODUCTORY MATERIAL

1

INTRODUCTION

Parallel applications seek to take advantage of every processing element (PE) available to increase their performance. However, many of these applications cannot use all available PEs effectively which could be caused by multiple common performance issues found in parallel applications. So, improving the scalability of these applications requires identifying the performance limitations affecting performance. These limitations could be caused by either the parallel algorithm or its implementation. This dissertation is focused on optimising a given implementation.

The process of optimising a parallel application takes place after the implementation of a given algorithm. The optimisation of an algorithm's parallel implementation consists of 3 main steps: identifying the performance problems affecting its scalability, locating them in space/time (e.g., line in the source code, timeline), and taking optimisation decisions accordingly to counteract them. The profiling of an application is useful in this process to reveal the existing performance problems as well as locate them. When optimising an application, we look to identify the hottest methods of an application to optimise them and obtain the largest performance gains. Profiling allows a developer to collect information about where the computing resources are spent during the execution of an application. To avoid wasting time optimising away cold methods, the profiles produced should provide a representation of the program profile as accurately as possible.

The Java programming language is one of the most popular programming languages, in part due to its portability. This portability is achieved because of its execution on top of a Java Virtual Machine, however, this presents some unique challenges in obtaining an accurate profile of its execution. These challenges are addressed in different ways, but each solution has its own drawbacks.

This dissertation seeks to accomplish three goals. The first goal is to study current profiling tools for Java applications, describing and comparing the approaches they use to profile Java applications. The second goal is to develop a set of performance metrics, obtained from an application's profile, capable of identifying or quantifying specific performance problems found commonly in parallel applications. As the final goal, this work should result in the development of tools with the intent of aiding a programmer in the pursuit of scalability improvements of parallel applications. These tools will implement the performance metrics developed and are accompanied by an optimisation workflow to guide the optimisation process.

This document is organised into four chapters. The next chapter presents how the performance of parallel applications will be evaluated throughout this work and highlights common performance problems affecting the scalability of parallel applications. The chapter also gives a brief overview of application profiling methods, and more specifically, methods for profiling Java applications and current Java profilers. Chapter three presents the developed optimisation workflow which suggests optimisation paths for a parallel application based on metrics collected through profiling which determine the biggest performance limitations. The chapter also validates the suggested workflow with some use cases and presents the tools developed to support the workflow. The fourth chapter is dedicated to the conclusions made after the work produced in this dissertation and prospects for future work.

2

STATE OF THE ART

2.1 PERFORMANCE EVALUATION

Performance evaluation is an important process of application development to determine how an application can fulfil its goals in the most optimal way. Application performance can be defined by multiple factors depending on the studied application. Here, performance is defined by the execution time of the application, so an application has better performance when its execution time is shorter.

For parallel applications, it is important to also measure performance improvements because there is an obvious improvement target to achieve. An ideal parallelisation with N processing elements should result in an execution time that is N times smaller than its sequential execution time.

Speedup is a metric used to measure the impact of performance improvements. This metric measures the improvement in execution time of an application over another. The speedup achieved by application B over application A can be evaluated with equation 1. For an ideal parallelisation, the speedup of a parallel application over its sequential counterpart should be equal to the number of processing elements running the parallel application.

$$\text{Speedup}_\text{B} = \frac{\text{Execution Time}_\text{A}}{\text{Execution Time}_\text{B}} \tag{1}$$

Efficiency is a metric, ranging from 0 to 1, that measures how well the available resources are being used to improve an application's performance. An ideal use of the available resources would see an efficiency equal to 1. For parallel applications, an ideal use of the available resources, or processing elements, would observe a speedup equal to the number of processing elements available. The efficiency of a parallel application can be measured with equation 2.

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of processing elements}} \tag{2}$$

The scalability of a parallel application is defined by the speedup or efficiency they achieve. An application with good scalability is able to consistently increase its performance as the number of PEs running the application grows. So, to assess the scalability of applications, it is important to observe the evolution of speedup or efficiency as the number of PEs of the parallel application increases. This can be seen in speedup or efficiency graphs as in figures 1 and 2 where the scalability of a parallel application is compared to the ideal scalability. For speedup

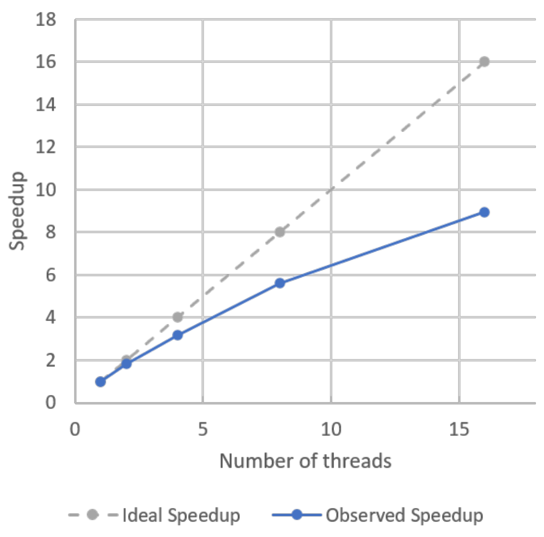


Figure 1.: Example of a speedup graph

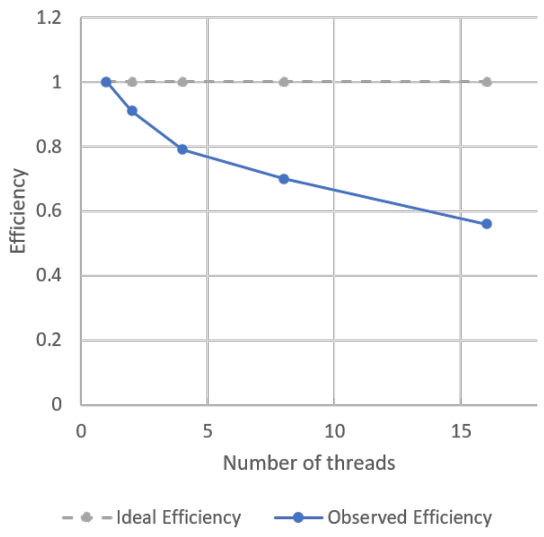


Figure 2.: Example of an efficiency graph

graphs, ideal scalability would see speedup always equal to the number of threads, while for efficiency graphs it would see the efficiency always equal to 1.

2.2 COMMON PERFORMANCE PROBLEMS

Many parallel applications cannot achieve an efficiency close to 1. In general, the efficiency of these applications decreases as the number of processing elements increases. The reason for this lack of efficiency is due to the existence of limited resources and intrinsic limitations of the algorithms used [8]. For example, the application may need to share resources between the threads which limits their availability or the algorithm may contain computational work that cannot be distributed evenly between the threads or be made parallel at all. This section classifies and describes common problems found in parallel applications into 5 categories.

2.2.1 *Amdahl's Law*

A parallel application's workload can be split into 2 fractions, the sequential fraction and the parallel, or enhanced, fraction. A common performance problem in applications arises from Amdahl's law which states that the speedup of an application is limited by the fraction of the application that is not enhanced. For parallel applications, this means the sequential fraction of the application will be a constraint on its scalability. Since all of the available processing elements can only be used during the parallel fraction of the application, we can only expect increased performance during this fraction and thus the sequential fraction limits the scalability of the whole application.

Figure 3 illustrates this problem, where the sequential fraction goes from being the cause of a small portion of application time when running on 1 thread, to being the cause of half of its execution time when running on 4 threads. This performance problem becomes more noticeable as the time spent processing parallel workloads is reduced and for larger sequential fractions.

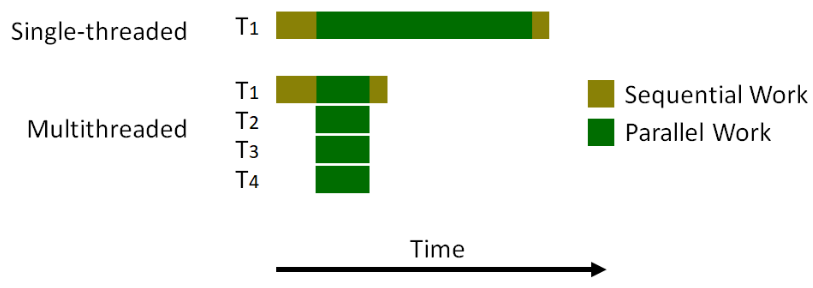


Figure 3.: Effect of Amdahl's law on parallel applications

2.2.2 *Memory Wall*

The scalability of an application can also be greatly impacted by the memory accesses of the application. If a sequential version of the application already spent a large chunk of its time with memory accesses, the fact that memory accesses are now made concurrently in the parallel application could affect scalability if the memory bandwidth isn't big enough to concurrently satisfy all memory accesses. Additionally, the parallelisation of an application may sometimes destroy the cache locality that was present in the sequential version or introduce false sharing. This will impact scalability by increasing the miss rate on cache accesses leading to an increase in the number of memory accesses and, consequentially, by increasing the strain on memory bandwidth. This increase would raise the time spent accessing the memory, ultimately increasing the application's execution time.

2.2.3 *Parallelism Overhead*

Another common performance problem of parallel applications comes from parallelism overhead. This problem emerges from the additional workload required to introduce parallelism to the application such as the creation of execution threads or redundant computations. Since this additional workload needs to be processed, the execution time of the application will not decrease as much as expected. This overhead increases as the number of parallel tasks increases, so the creation of parallel workloads as coarse as possible reduces the impact of parallelism overhead.

2.2.4 *Resource Contention*

To ensure correctness of the parallel implementation, it may be necessary to guarantee exclusive access to some resources. This could result in processing elements having to wait for the release of a resource held by another processing element. When this happens, it creates resource contention overhead, essentially serialising what should be parallel work. This leads to inefficient use of the processing elements available which translates into a lack of scalability.

2.2.5 *Workload Imbalance*

Parallelisation requires the distribution of the parallel workload among the available processing elements. Ideally, this distribution results in an equal workload for all processing elements. However, this is often not the case, especially in coarse-grained parallelism where the parallel tasks are large and few and it becomes harder to split them evenly resulting in load imbalance. This leads to inactive processing elements whilst some parallel workloads are yet to be processed. This could mean the parallel workloads should be more fine-grained to reduce workload imbalance, even though this could increase parallelism overhead. Figure 4 illustrates the effects of load imbalance delaying the end of parallel fraction execution.

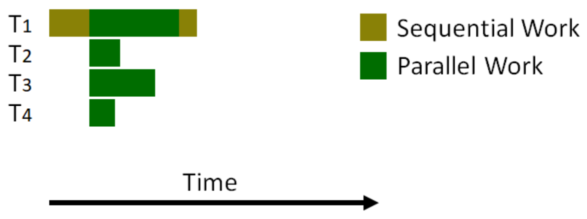


Figure 4.: Effect of load imbalance on parallel applications

2.3 PROFILING

Profiling is a form of analysing a program by recording information about its use of the system's resources. Profiling can be performed while the program is running in the PE's or not, and during all of its lifetime or any fraction of it. The recorded information can be related to various performance metrics, for example, a profile can show where a program spends time when executing in a PE or where it performs more memory allocations. Profiling is a useful technique for developers to find performance bottlenecks in their applications and support them during program optimisation.

When profiling an application, there are several things to keep in mind. Profiling will change a program's behaviour and we should seek to minimise its impact. The profile obtained may not be representative of an application's real profile, so it is important to be aware of the obtained profile's quality to avoid being misled. Profiling can also generate a lot of data which can be problematic when profiling applications running for a long time.

Sections 2.3.1 and 2.3.2 discuss the two main methods of obtaining a program's profile, instrumentation and sampling. Section 2.3.3 introduces techniques for displaying profiles.

2.3.1 *Instrumentation*

Instrumentation-based profilers measure occurrences or time inside functions within some instrumented section of the code. To accomplish this, instructions need to be added to the code to identify when functions are called and exited.

This method of profiling can be very detailed as it is capable of counting the exact number of function calls and the time spent in these. However, this can cause a big overhead due to the added instructions, which is most noticeable when calling small functions at a high frequency. Additionally, the introduced code may influence optimisation decisions done by compilers which can fundamentally change the profile of an application. Because of these main drawbacks, instrumentation-based profiling may produce very inaccurate profiles of an application, however, it can still be very useful and report information other methods cannot (e.g. sampling).

2.3.2 *Sampling*

Sampling-based profiling operates by collecting the profile information (for example, a call stack) of the inspected program at a given frequency or period, halting the program in this process. This results in a collection of locations the inspected program has been spotted at, from which can be deduced a general profile of the application. This means that we can only obtain an approximation to the real profile of the application. However, sampling-based profiling allows the target program to run with little overhead introduced by the profiler and since the profiler is not as intrusive as in an instrumentation based method, the generated profile may even be more accurate.

For this strategy to be reliable, the profiler must collect a significant number of samples to portray an accurate representation of a program's profile. The number of collected samples can be controlled by the duration of time the profiler inspects the application or the frequency the profiler collects samples of the execution, keeping in mind that higher frequencies incur in higher overhead. In addition, we should also make sure to avoid any bias that may arise from the sampling strategy causing a misrepresentation of the program's profile. Biases may appear for many different reasons, for example, the frequency of sampling may line up with some periodic process or the method used to collect samples may be biased itself.

Some common issues found with sampling-based profiling are the stack reconstruction and identification of inlined methods. Stack reconstruction can be problematic when execution is halted from outside the application and without knowledge of how to walk the stack. If the conventional method of using the base pointer register to identify the boundaries of different functions in the stack is not used by the application, this may result in broken stack traces. These issues will be more detailed for the case of Java applications in section 2.4.

2.3.3 *Profile Visualisations*

The profile of the execution of a program can contain a large amount of information. For it to be of any use, it is important to filter this information and present it in an easy to read format.

The following profile of an application will serve as an example of the information contained in a profile which will be used to demonstrate the different visualisations of a profile. The explanation for the different visualisation options treat this profile as if it was obtained via sampling, but they also serve for profiles obtained via instrumentation. This profile contains:

- 2 occurrences of function A called inside the main function
- 2 occurrences of function B called inside function A called inside the main function
- 3 occurrences of function C called inside function A called inside the main function
- 1 occurrence of function E called inside function A called inside the main function
- 2 occurrences of function B called inside function D called inside the main function

Flat Profile View

Flat views only take into account the innermost functions of a call stack, essentially ignoring the context of where these functions were called from, by not taking the rest of the call stack into consideration. This means that if, for example, the profile contains a sample with a call stack composed of function B called inside function A called inside the main function, that sample will only account for an occurrence of the function B, regardless of where this function was called from. This visualisation loses some information contained in the profile, such as the prevalence of outer functions during the execution of the application, but it can highlight the presence of functions being used inside multiple other functions.

A flat view of the example profile would only give us the following information:

- 4 occurrences of function B
- 3 occurrences of function C
- 2 occurrences of function A
- 1 occurrence of function E

This view summarised the profile to the innermost functions of each stack. This visualisation option can be especially useful for finding functions with a big number of samples spread between deep and different call stacks. However, it cannot exhibit the impact of outer functions. In the previous example, the flat profile shows that 40% of the samples were found in function B, so optimising this function has the potential of providing the greatest performance improvements, but it fails to show that, for example, function A is found in the call stack of 80% of the samples, so there is no clue as to what performance improvements may be achieved if function A is optimised by reducing its calls to other functions.

Call Tree

As opposed to the flat profile, a call tree displays the full information of the call stacks in the profile. This type of visualisation can be interpreted as, basically, a tree where a path from the root to the leaves represents a call stack. This visualisation can bring light to the impact that outer functions may have in the application's execution. On the other hand, it can represent too much information which can easily overwhelm when profiles become too large and diverse.

The profile used as an example previously can be presented in the format of a call tree as follows:

```
main function - Total samples:  10(100%); Self samples:  0(0%)
 |__function A - Total samples:  8(80%); Self samples:  2(20%)
 |   |__function B - Total samples:  2(20%); Self samples:  2(20%)
 |   |__function C - Total samples:  3(30%); Self samples:  3(30%)
 |   |__function E - Total samples:  1(10%); Self samples:  1(10%)
 |__function D - Total samples:  2(20%); Self samples:  0(0%)
     |__function B - Total samples:  2(20%); Self samples:  2(20%)
```

This presentation aggregates call stacks with common outer functions. The number of self samples of a function is the number of samples in which such function was the innermost function of the call stack. The number of total samples includes the number of times the function was present in any level of the call stack. In this presentation, it can now be clearly seen the presence of function A in the profile as it is present in 80% of the call stacks.

Flame Graph

Flame graphs [7] are a specific visualisation method for observing call trees. They represent a call stack as a pile of boxes on top of each other, where each box represents a function. The width of these boxes is proportional to the number of occurrences of its function, which gives emphasis to functions with a higher number of occurrences. These graphs are interactive, so that less prominent methods can still be seen by zooming into them and their number of occurrences be obtained by mousing over them, making these graphs capable of relaying all the information of a call tree.

Figure 5 displays the flame graph of the example profile presented previously. The call stacks are represented here from the base to the top. This type of visualisation does not offer many advantages when displaying smaller profiles. Figure 6 shows the flame graph for a larger profile in which the advantages of using flame graphs are more clear, the most prominent call stacks stand out from the rest and are quickly identified.

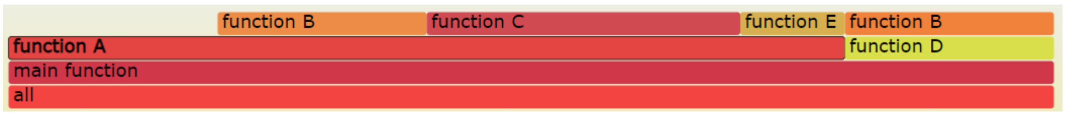


Figure 5.: Resulting flame graph generated from a small profile

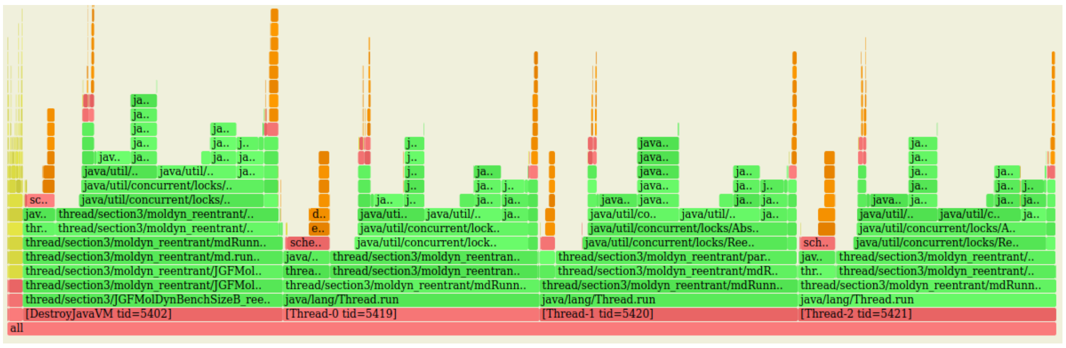


Figure 6.: Resulting flame graph generated from a larger profile

2.4 PROFILING JAVA

The Java programming language is currently one of the most popular programming languages. This popularity is partly due to the security and portability it offers. To achieve this portability, the programs developed in this language are compiled into bytecode, which is then interpreted or compiled and optimised on top of a Java Virtual Machine (JVM). However, the introduction of this middle step introduces some issues in the process of analysing and optimising Java applications.

In the case of a Java application, the profiling process gets tricky due to its execution on top of a JVM. This process gets even more complex when we intend to profile applications with various threads, where the performance problems are usually connected to the hardware properties.

In addition to the problems generally found in sampling-based profiling such as stack reconstruction and inlining, some common issues with sampling-based profiling Java applications are the bias caused by safepoint restrictions, tracking native, JVM and kernel calls, identifying interpreted methods and identifying symbols for compiled methods.

This section will present the most common techniques for profiling Java applications along with their main advantages and drawbacks.

2.4.1 *Profiling with GetAllStackTraces*

To profile a Java application we can make use of the class `Thread` offered by the package `java.lang` containing the function `GetAllStackTraces` which returns a map of the java stack traces for all live threads. A better option may be the native function `GetAllStackTraces` offered by the JVM Tool Interface (JVMTI) to collect these stack traces.

Both of these functions allow us to easily generate a profile for the execution of a Java application. One of the main advantages of using this technique is the guaranteed support from any JVM. However, there are a few drawbacks to this approach.

The function `GetAllStackTraces` can only operate when all the Java threads are at a safepoint. When the JVM requires the system to be at a safepoint for a VM operation, all threads must stop at the next safepoint in its execution and can only resume execution after the VM operation has been performed [5]. When a safepoint is requested, a thread may be in a few different states [4]. These are the ways a thread can reach a safepoint depending on their state for the HotSpot JVM [12]:

- When running interpreted code, the thread can reach a safepoint when executing branching/returning byte codes.
- When running in native code, the thread does not need to stop its execution and must only stop when returning from the native code if a safepoint state is still required.
- When running compiled code, the thread stops when it finds itself in a safepoint poll. These are usually located in method exits and uncounted loops.
- When blocked, the thread will not be allowed to return from the block condition until the safepoint operation is complete.

This requirement of bringing the threads to a safepoint means we can only sample the threads at specific points in its execution, which may lead to inaccurate samples. For example, when a thread is running compiled code, if the thread is inside an inlined method, the safepoint poll when exiting this method would cease to exist and the thread would stop its execution already outside of this method.

Another problem safepoints introduce is the added overhead of bringing the system to a safepoint. Every time the system wants to reach a safepoint, it will take as long as the slowest thread to reach a safepoint. This overhead will be more significant as the number of live threads increases.

In conclusion, this method of profiling will collect samples from all threads, whether they're currently running in the PEs or not. However, this type of sampling may be inaccurate due to the safepoints introducing additional overhead and bias to the profile. This sampling method is also unable to profile kernel, native and JVM code.

2.4.2 *Profiling with AsyncGetCallTrace*

The JVM call `AsyncGetCallTrace` was initially implemented on OpenJDK JVMs and may not be available in other JVMs. With this call, it's possible to collect stack traces of Java applications outside of safepoints avoiding the issues present in the `GetAllStackTraces` method. With this method, a collection of samples can be obtained by sending Linux signals (for example, SIGPROF on a regular interval) to the Java process being profiled with a signal handler registered. This signal would then be caught and handled by a random thread running our Java application on the PEs. While handling the received signal, a call to `AsyncGetCallTrace` can be safely made to record stack traces.

It's advisable to set the flag `-XX:+DebugNonSafepoints` to generate extra debugging info for non-safepoints since, by default, JVMs usually only generate debug information for safepoints.

Using this function, we are no longer restricted to collecting samples from safepoints and as a result, we're able to generate a more accurate profile. On the downside, this function is not part of the JVM specification and is unsupported and undocumented. Using `AsyncGetCallTrace` we can only profile threads running, meaning the time a thread spends waiting for a contended resource cannot be seen as in when using the previous method. As in the previous method, this method is also unable to profile native and JVM code.

2.4.3 *Profiling with perf_event_open*

The system call `perf_event_open` is a Linux method for collecting stack traces which can also be used to obtain Java stack traces asynchronously. This method allows the capture of native and kernel functions along with the java functions. This function also allows sampling according to various hardware performance event counters, allowing us to obtain profiles displaying various metrics. Hardware performance counters can record information such as instructions retired, the number of clock cycles elapsed, cache misses, etc..

In order to traverse the stack to collect stack traces, this method uses the frame pointer to find the chain of functions forming a call stack. However, by default, JVMs use the frame pointer as a general purpose register as an optimisation. This means that this method will create samples of broken stack traces. To fix this, it's necessary to include the flag `-XX:+PreserveFramePointer` when running a Java application. This flag overrides this optimisation, allowing for stack traces to be built.

Using this method, we're collecting stack traces from outside the JVM, and as a result, we cannot differentiate between the different interpreted methods or find the method names of the JIT compiled code.

Overall, using `perf_event_open` can give us a wider scope of what our application is doing, by allowing us to profile JVM, native and kernel functions. On the other hand, this method has some limitations in profiling interpreted Java code and JIT compiled code.

2.5 JAVA PROFILING TOOLS

In this section, a set of profilers usable with Java applications is presented along with the strategies used by these to profile Java applications, their main features, drawbacks and some visualisation options. To display the profilers in action, a sequential implementation of an array sorting algorithm, radix sort, was chosen to be inspected by the presented profilers.

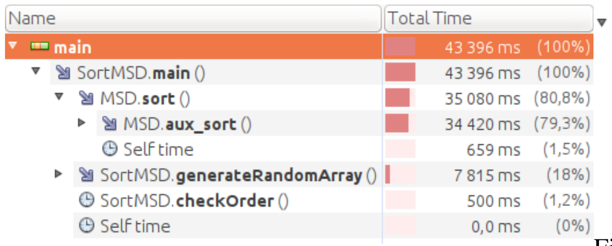
2.5.1 *VisualVM*

VisualVM[1] is one of the most popular Java profilers. This profiler allows for memory and CPU profiling via two methods, sampling based profiling and instrumentation based profiling.

Sampling

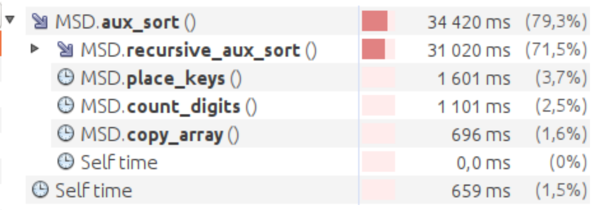
The sampling option of VisualVM uses the method `GetAllStackTraces` from JVM Tool Interface. For this reason, this method suffers from the safepoint bias problem and it is also unable to profile native or kernel code.

The output of this tool can be visualised in a call stack tree format as shown in figure 7.



Name	Total Time	
main	43 396 ms	(100%)
SortMSD.**main** ()	43 396 ms	(100%)
MSD.**sort** ()	35 080 ms	(80,8%)
MSD.**aux_sort** ()	34 420 ms	(79,3%)
Self time	659 ms	(1,5%)
SortMSD.**generateRandomArray** ()	7 815 ms	(18%)
SortMSD.**checkOrder** ()	500 ms	(1,2%)
Self time	0,0 ms	(0%)

Figure 7.: Resulting call tree obtained with VisualVM by sampling a sequential version of radix sort

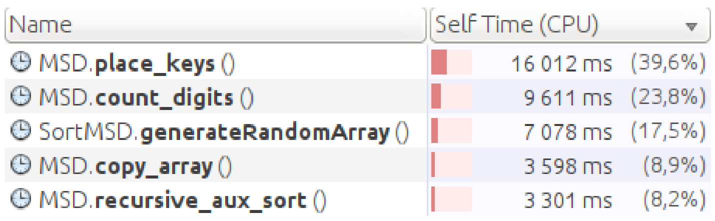


MSD.**aux_sort** ()	34 420 ms	(79,3%)
MSD.**recursive_aux_sort** ()	31 020 ms	(71,5%)
MSD.**place_keys** ()	1 601 ms	(3,7%)
MSD.**count_digits** ()	1 101 ms	(2,5%)
MSD.**copy_array** ()	696 ms	(1,6%)
Self time	0,0 ms	(0%)
Self time	659 ms	(1,5%)

Figure 8.: Resulting call tree obtained with VisualVM by sampling a sequential version of radix sort, expanded on `aux_sort`

In figure 8, we can see that this implementation of radix sort contains the methods named `count_digits` and `place_keys`. What we cannot see in this profile, however, is that these two methods call the method `digitAt`. This method tends to get inlined by the JIT compiler, and for this reason, it cannot be seen inside the calling methods in figure 8 because the safepoint polls when exiting this method are lost after inlining.

In figure 9 we see a flat profile obtained with VisualVM. This visualisation is displaying the 5 hottest methods, the methods with the most total self time, during the program execution. While this visualisation option is simpler, especially when the call is tree extremely deep or complex, it does not provide any context of where the methods were called. So optimising these methods would result in increased performance, but optimising the methods calling these may be a more viable strategy.



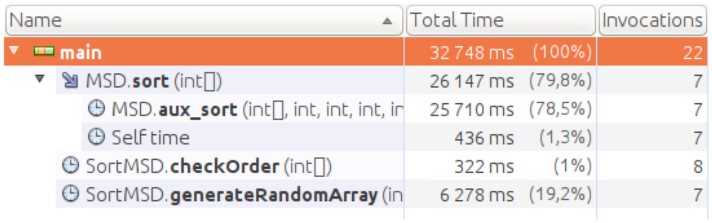
Name	Self Time (CPU)	
MSD.**place_keys** ()	16 012 ms	(39,6%)
MSD.**count_digits** ()	9 611 ms	(23,8%)
SortMSD.**generateRandomArray** ()	7 078 ms	(17,5%)
MSD.**copy_array** ()	3 598 ms	(8,9%)
MSD.**recursive_aux_sort** ()	3 301 ms	(8,2%)

Figure 9.: Resulting flat profile from sampling a sequential version of radix sort with VisualVM

Instrumentation

The *Profiler* option of VisualVM works by instrumentation. By providing the classes we wish to profile, the profiler will count the number of invocations of each method along with their duration. While this method of profiling returns a more detailed profile, the overhead introduced by the instrumentation may skew the obtained results.

In figure 10, the output of profiling our sequential implementation of radix sort is shown. With this profiling method, the method `digitAt` would be visible even if inlined. However, profiling this method would incur in a a huge overhead due to the large number of times this method is invoked.



Name	Total Time	Invocations
main	32 748 ms (100%)	22
MSD.**sort** (int[])	26 147 ms (79,8%)	7
MSD.**aux_sort** (int[], int, int, int, ir	25 710 ms (78,5%)	7
Self time	436 ms (1,3%)	7
SortMSD.**checkOrder** (int[])	322 ms (1%)	8
SortMSD.**generateRandomArray** (in	6 278 ms (19,2%)	7

Figure 10.: Result of profiling, by instrumentation, a sequential version of radix sort with VisualVM

2.5.2 *Perf*

Perf is a Linux profiler making use of the perf_event_open system call to obtain event counts and record events for later processing. Since it uses this system call, it inherits all the problems mentioned in the previous section. This means the JVM must run with the flag `-XX:+PreserveFramePointer` to be able to build stack traces, incurring in a small overhead, and interpreted methods will not be differentiated or identified. However, method names of JIT compiled code can be deciphered by providing a file mapping memory locations to symbol names for Perf to read. This file can be created by using the tool `perf-map-agent`[3] as a JVMTI agent to map addresses to java methods.

In figure 11, we see the profile collected by Perf represented in a flame graph. In this visualisation, green represents Java methods, blue are inlined Java methods, yellow methods are C++, native code is in red and kernel functions are coloured orange. In order to see inlined methods, the flag `-XX:+DebugNonSafepoints` is necessary to provide more accurate information about inlined methods. We can also see that interpreted methods are not identified and simply show up as "Interpreter". This problem should mostly affect short-running programs spending much of their time running in the interpreter.

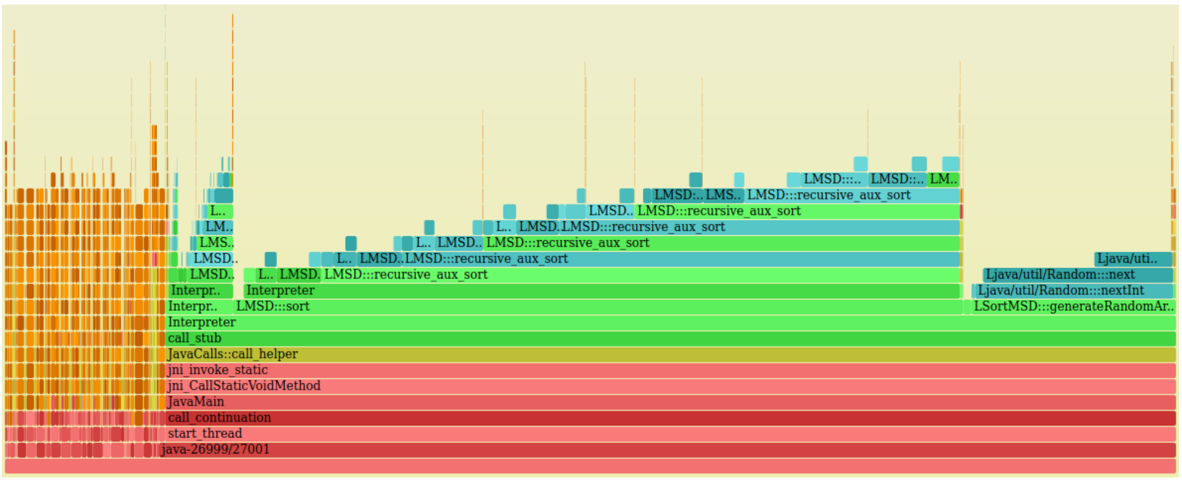


Figure 11.: Flame graph output out of profiling the sequential radix sort with Perf

The recursiveness of the `recursive_aux_sort` method makes the analysis of this application's profile a trickier process. Figure 12 zooms in on the innermost `recursive_aux_sort` call, where we can find samples taken while the application was executing the method `digitAt`. In fact, this method can also be found inside the remaining outer `recursive_aux_sort` stacks, meaning it actually takes a considerable amount of application time which was not possible to identify in the profile provided by VisualVM.

In the profile provided by Perf, we can also see that multiple other methods were also inlined, however they were sampled by VisualVM. This happens because these methods contain uncounted loops, which introduce a safepoint poll inside the method, allowing the program execution to stop inside this inlined methods during sampling.

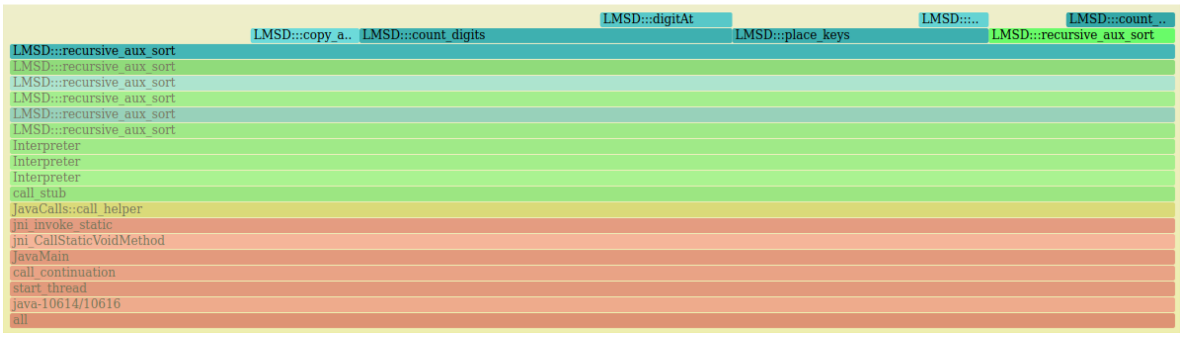
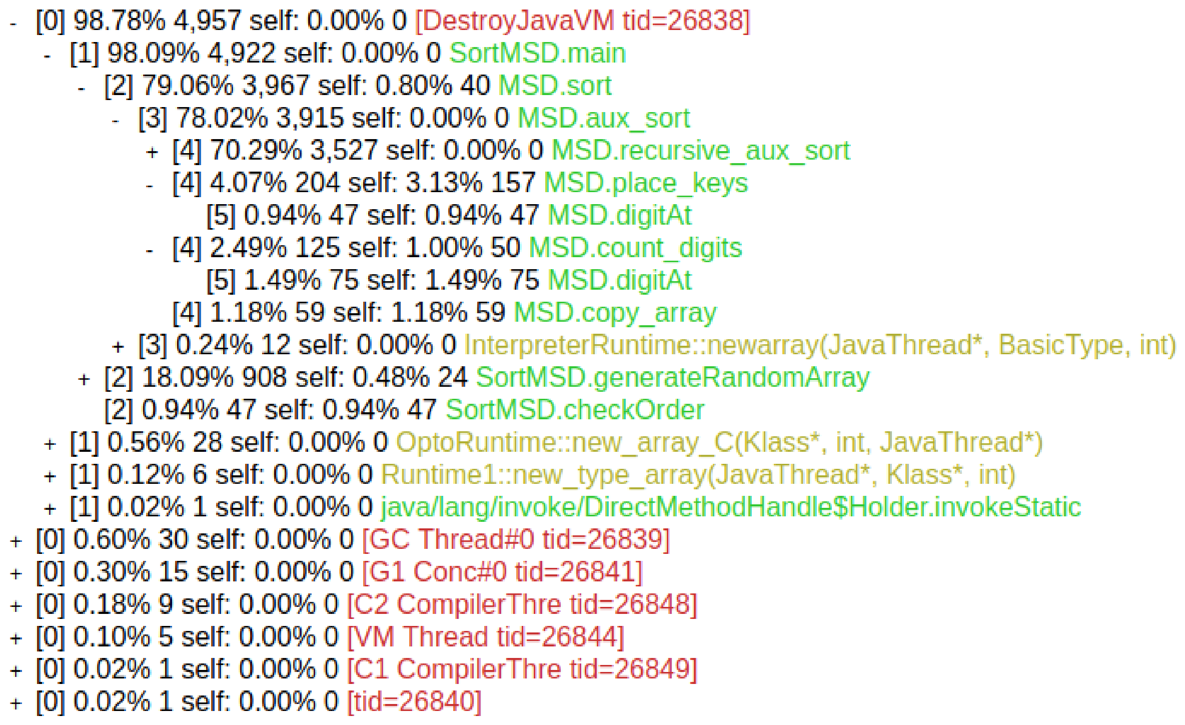


Figure 12.: Flame graph output out of profiling the sequential radix sort with Perf

2.5.3 *Async-profiler*

Async-profiler[2], as the name suggests, is a tool using the `AsyncGetCallTrace` call, present in some JVMs, to retrieve java stack traces avoiding the safepoint bias problem. This profiler is also capable of obtaining kernel and native stack traces by using the `perf_event_open` system call. Using this call also allows it to take advantage of the wide range of available hardware performance events. To obtain the full stack profile, this tool basically collects a call trace of the program when a hardware counter overflows and calls `AsyncGetCallTrace` to collect a second call trace and then merges these two obtained call traces.

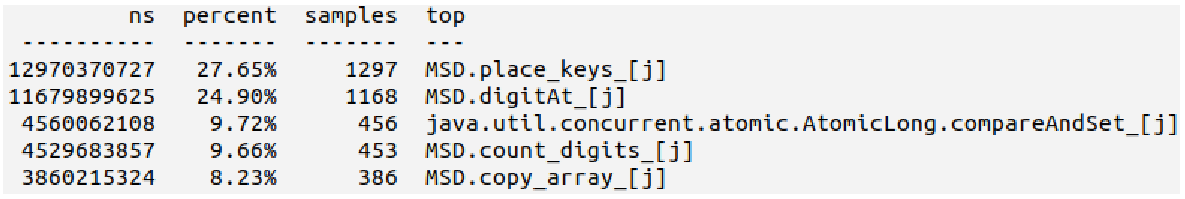
Figure 13 shows a tree view of the profile collected by Async-profiler when profiling the sequential radix sort. In this profile we can identify, if the JVM flag `-XX:+DebugNonSafepoints` is set, the inlined method `digitAt` we couldn't before with VisualVM using the `GetAllStackTraces` method. Here we can also see threads used for managing the JVM such as compiler and garbage collector threads along with native code, e.g. `new_array_C`. These segments are obtained through the use of the `perf_event_open` system call.



```
- [0] 98.78% 4,957 self: 0.00% 0 [DestroyJavaVM tid=26838]
   - [1] 98.09% 4,922 self: 0.00% 0 SortMSD.main
      - [2] 79.06% 3,967 self: 0.80% 40 MSD.sort
         - [3] 78.02% 3,915 self: 0.00% 0 MSD.aux_sort
            + [4] 70.29% 3,527 self: 0.00% 0 MSD.recursive_aux_sort
            - [4] 4.07% 204 self: 3.13% 157 MSD.place_keys
                 [5] 0.94% 47 self: 0.94% 47 MSD.digitAt
            - [4] 2.49% 125 self: 1.00% 50 MSD.count_digits
                 [5] 1.49% 75 self: 1.49% 75 MSD.digitAt
              [4] 1.18% 59 self: 1.18% 59 MSD.copy_array
         + [3] 0.24% 12 self: 0.00% 0 InterpreterRuntime::newarray(JavaThread*, BasicType, int)
      + [2] 18.09% 908 self: 0.48% 24 SortMSD.generateRandomArray
        [2] 0.94% 47 self: 0.94% 47 SortMSD.checkOrder
   + [1] 0.56% 28 self: 0.00% 0 OptoRuntime::new_array_C(Klass*, int, JavaThread*)
   + [1] 0.12% 6 self: 0.00% 0 Runtime1::new_type_array(JavaThread*, Klass*, int)
   + [1] 0.02% 1 self: 0.00% 0 java/lang/invoke/DirectMethodHandle$Holder.invokeStatic
+ [0] 0.60% 30 self: 0.00% 0 [GC Thread#0 tid=26839]
+ [0] 0.30% 15 self: 0.00% 0 [G1 Conc#0 tid=26841]
+ [0] 0.18% 9 self: 0.00% 0 [C2 CompilerThre tid=26848]
+ [0] 0.10% 5 self: 0.00% 0 [VM Thread tid=26844]
+ [0] 0.02% 1 self: 0.00% 0 [C1 CompilerThre tid=26849]
+ [0] 0.02% 1 self: 0.00% 0 [tid=26840]
```

Figure 13.: Resulting tree view of profiling the sequential radix sort with Async-profiler

Comparing the flat profile produced by VisualVM to the one generated by Async-profiler, in figure 14, we can see that much of the time is actually spent in `digitAt`, which could not be identified previously. We also see that the time spent in `generateRandomArray` disappears because it contains more inner methods than VisualVM is capable of detecting, and so, much of the time is spent in the JVM method `compareAndSet` and not in itself.



```
         ns  percent  samples  top
 ----------  -------  -------  ---
12970370727   27.65%     1297  MSD.place_keys_[j]
11679899625   24.90%     1168  MSD.digitAt_[j]
 4560062108    9.72%      456  java.util.concurrent.atomic.AtomicLong.compareAndSet_[j]
 4529683857    9.66%      453  MSD.count_digits_[j]
 3860215324    8.23%      386  MSD.copy_array_[j]
```

Figure 14.: Resulting flat profile of the sequential radix sort with Async-profiler

Figure 15 presents the flame graph generated by Async-profiler. Here, we see that Async-profiler does not distinguish between interpreted and compiled methods and it's able to identify the interpreted methods, contrary to Perf.

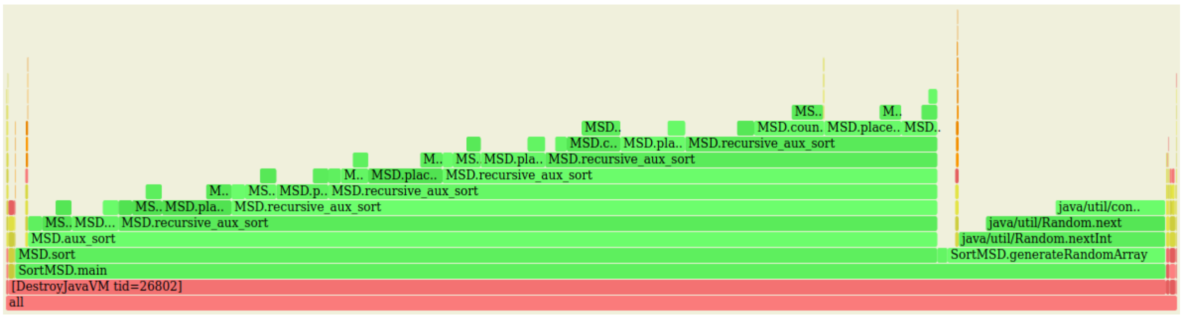


Figure 15.: Resulting flame graph of profiling the sequential radix sort with Async-profiler

2.5.4 *Summary*

Table 1 presents a summary of the characteristics of each profiling tool. All of these tools support profiling by sampling as well as offer support for instrumentation in some form.

Interpreted methods are the methods that were executed by the Java interpreter. These methods can be captured and identified by VisualVM and Async-profiler. Perf, however, is only capable of recognising the existence of interpreted methods, it cannot identify or distinguish between different interpreted methods.

Inlined methods cannot be caught by VisualVM if they do not contain safepoints. Perf and Async-profiler can always capture these methods if the JVM receives the flag `+DebugNonSafepoints` which generates debug information for locations without safepoints.

Both Perf and Async-profiler can capture JVM, native and kernel functions as a result of using the system call `perf_event_open`. The use of this system call also allows the collection of various metrics through the use of hardware performance counters.

Perf requires the definition of the `+PreserveFramePointer` flag when running the Java application to be able to reconstruct the stack, which induces an overhead ranging from 0 to 5% [6] [13].

Profiler	VisualVM	Perf + perf-map-agent	Async-profiler
Profiling Method	Sampling and Instrumentation	Sampling and Instrumentation	Sampling and Instrumentation
Named interpreted methods	Yes	No	Yes
Inlined methods	If they contain safepoints	With `+DebugNonSafepoints` flag	With `+DebugNonSafepoints` flag
JVM code	No	Yes	Yes
Native and kernel functions	No	Yes	Yes
Hardware Performance Counters	No	Yes	Some
Safepoint bias	Yes	No	No
Others	-	Requires `+PreserveFramePointer`	-

Table 1.: Summary of different Java profiling tools

In this work, Async-profiler was chosen over VisualVM to profile Java applications due to its support for hardware performance counters, which allows a collection of various performance metrics. Async-profiler was also chosen over Perf due to its better suitability for profiling Java applications, as the use of `AsyncGetCallTrace` allows the collection of a more detailed Java stack. Nisbet et al. [11] also highlight the importance and other advantages of this hybrid approach for collecting a Java profile.

Testing the Profiling Tools

To test the sampling accuracy of these tools, they were used to profile a simple application which runs two equal methods but with different names, `cicloJ` and `cicloJ2`. These methods are called inside `cicloI`, which calls `cicloJ` at double the frequency of `cicloJ2`. This test was created to check if the relation between the time spent in each of these methods can be observed with these profilers.

Figure 16 displays the flame graph obtained with perf for the test application. Here, `cicloJ` takes 64.8% of application time and `cicloJ2` takes 26%. There is also an interpreted method with 6.4% of application time which can be confirmed to be the method `cicloJ2` because it spends all its time inside a function only called inside `cicloJ2`. So, the sum of the time spent between the call identified as `cicloJ2` and `Interpreter` is half of the time spent in `cicloJ` and the ratio between these is verified.

Figure 17 displays the flame graph obtained with async-profiler, which observes 66.5% for `cicloJ` and 33.3% for `cicloJ2`.

Figure 18 displays the call tree obtained by profiling the test application with VisualVM. The results are similar to the results obtained with the other profilers, detecting the same expected ratio between the two methods.

This simple test fails in revealing the inaccuracies these profiles may have. Mytkowicz et al.[10] evaluate the accuracy of some Java profilers which can only sample at safepoints and one profiler which is free from this restriction. They observe that profilers dependent on safepoints often produce, not only incorrect profiles, but also different profiles compared to each other. They conclude that the reasons for incorrect and different profiles obtained are due to the bias introduced from sampling only at safepoints and due to the unique observer effect of each profiler, which affects the optimisations and placement of safepoints of the observed application.

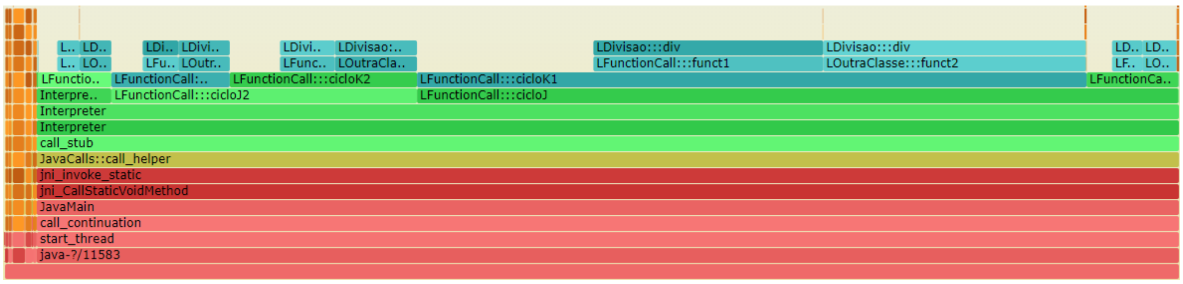


Figure 16.: Resulting flame graph obtained with perf

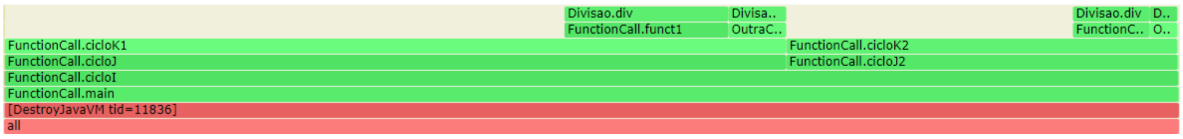
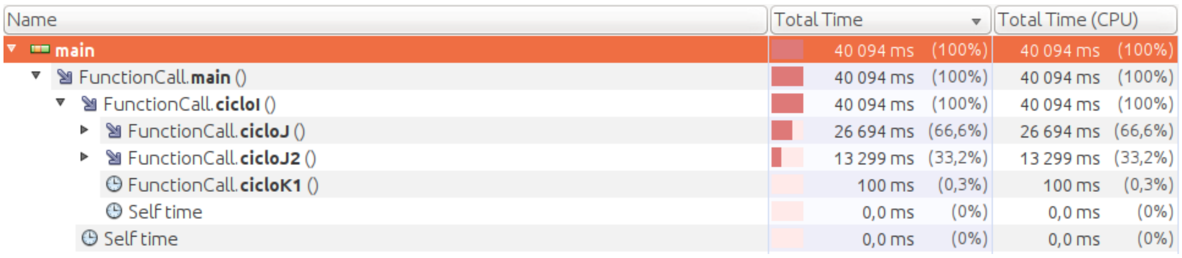


Figure 17.: Resulting flame graph obtained with async-profiler



Name	Total Time		Total Time (CPU)	
main	40 094 ms	(100%)	40 094 ms	(100%)
FunctionCall.**main** ()	40 094 ms	(100%)	40 094 ms	(100%)
FunctionCall.**cicloI** ()	40 094 ms	(100%)	40 094 ms	(100%)
FunctionCall.**cicloJ** ()	26 694 ms	(66,6%)	26 694 ms	(66,6%)
FunctionCall.**cicloJ2** ()	13 299 ms	(33,2%)	13 299 ms	(33,2%)
FunctionCall.**cicloK1** ()	100 ms	(0,3%)	100 ms	(0,3%)
Self time	0,0 ms	(0%)	0,0 ms	(0%)
Self time	0,0 ms	(0%)	0,0 ms	(0%)

Figure 18.: Resulting call tree obtained with VisualVM

Part II

CORE OF THE DISSERTATION

3 DEVELOPMENT

Parallel applications often do not achieve ideal speedup due to multiple common limitations that can affect their scalability. The objective of the work developed in this dissertation is to guide programmers in the optimisation of parallel applications, specifically Java applications. This is achieved through the conception of an optimisation workflow and supplementary tools that aid them in quantifying the impact of the limitations affecting a parallel application's scalability and directing their optimisation efforts accordingly, in order to maximise performance gains.

To accomplish this, some performance limitations, deemed to be the most commonly observed performance limitations for parallel applications, are defined in section 3.1. The workflow suggested in section 3.2 is based around the impact of these limitations to identify which one imposes the biggest bottleneck on performance. The workflow consists of a decision tree of actions to perform according to the performance problem that is deemed to be the biggest constraint on the scalability of the application. Depending on which limitation is identified as the biggest bottleneck, it suggests different actions in order to investigate the problem further. The usage of this workflow requires the collection of metrics from the execution profile of the parallel application as well as its sequential counterpart.

The impact of the performance limitations on scalability can be estimated using metrics collected by profiling parallel applications and their sequential versions. Their impact is measured using the formulas presented in section 3.3. While the values given by the performance metrics collected may not be fully accurate due to multiple factors, they can still be useful in identifying performance limitations by comparison of their estimated impact.

Section 3.4 exemplifies and validates the usage of the suggested workflow in the iterative optimisation process of a given case study. Some extra case studies are also presented in order to highlight additional paths not covered in the main case study.

The suggested workflow is supplemented by the tools detailed in 3.5. These tools aid in the collection of the metrics required to follow the workflow for Java applications and offer the visualisation options mentioned in the workflow.

Section 3.6 describes some of the existing limitations of the workflow and tools suggested and highlights some scenarios in which these may be inaccurate or unreliable.

3.1 PERFORMANCE LIMITATIONS

Parallel applications seek to use the multiple processing elements available to them in order to increase their performance. An application with good scalability can use its available processing elements to increase its performance, ideally increasing it by a factor equal to the number of processing elements. However, applications might not have ideal scalability due to common performance problems found in parallel applications. There are 5 groups of performance limitations addressed in this work:

1. Sequential fraction
2. Parallelism overhead
3. Memory bottleneck
4. Load imbalance
5. Lock contention

The sequential fraction limitation consists of the limitation imposed by Amdahl's law, this is, the limitation caused by the fraction of the application that is not enhanced, the sequential fraction. This fraction will represent a larger part of the application time as the parallel fraction gets more optimised meaning that further optimisations to the parallel fraction will result in lower performance gains.

The parallelism overhead limitation consists of the increase in the number of instructions executed when running the parallel application when compared to the sequential application. This increase in the number of instructions is due to the additional logic necessary to create parallelism and it may appear in the sequential or parallel fractions of the application. The increase in the number of instructions translates into an increase in execution time which will affect the scalability of the application.

The memory bottleneck limitation is the limitation caused by an increase in total time waiting for the memory. This increase can be caused by multiple factors. For example, an increase in memory accesses due to lack of cache locality in the parallel application or an increase in concurrent memory accesses resulting in a limitation by the memory bandwidth. The increase in time spent waiting for the memory results in an increase of application time, affecting the scalability of the application.

The limitation caused by load imbalance is the limitation imposed by the distribution of an uneven workload to the different processing elements. Since the parallel fraction of an application is only finished when all processing elements finish executing it, a balanced workload will maximise performance gains by lowering the time it takes for the last processor to finish their execution. On the other hand, an unbalanced workload will increase the execution time of the parallel fraction and limit the scalability of the application

The lock contention limitation is caused by locks introduced in a parallel application. These locks may be added to force serialised execution of some portion of the parallel fraction. This serialised execution delays the execution of the parallel fraction on some processing elements, resulting in an increase of the execution time of application and affecting its scalability.

3.2 WORKFLOW

As mentioned before, parallel applications may suffer poor scalability due to various limitations. To fix scalability issues, it is helpful to have an idea of which limitations are ruining scalability so they can be looked into and mitigated. Identifying these limitations could be achieved by analysing the application's source code or profiling data which could be a daunting task. The suggested workflow tries to simplify this task by implementing guidelines for optimising parallel applications. The presented workflow helps in identifying application's major performance problems and suggests an action in accordance to locate the specific problem causing the bottleneck.

The suggested workflow is presented in figure 19 and starts with the profiling of an initial sequential application with the objective of identifying its hottest methods and optimise them in order to obtain the largest performance gains. There are multiple ways of displaying the profile of an application, but the flame graph view is suggested because of its readability, giving a larger visibility to the hottest call stacks, the ones we seek to optimise. Figure 20 displays the clock cycles flame graph of a sequential application where the number of clock cycles spent inside the method `m1` represents about 85% of the total number of clock cycles of the whole application. The fraction of clock cycles spent on a call stack has a direct correlation to the fraction of time spent on it, so the number of clock cycles is a good metric for finding the hottest methods of an application.

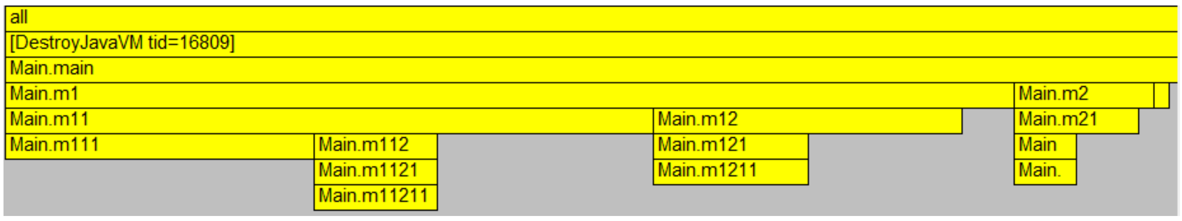


Figure 20.: Clock cycles flame graph of a sequential application

After identifying the application's hot spots, we should look into how these can be optimised. In the case of figure 20, method `m1` should be one of the methods looked into for optimisations since it represents a large chunk of the application, and so, the optimisation of this method has the possibility of returning the greatest performance increase.

If we choose to optimise the application without resorting to parallelisation, we can then profile the optimised application to look for its hot spots and repeat this process again. However, if we decide to turn to parallelisation to improve the application's performance, we can then profile this new parallel application and compare it to the old sequential application to obtain information about the scalability of the parallelisation. The scalability of a parallel application may be affected by different problems, and to mitigate the impact of these problems, different actions can be taken.

Figure 21 exemplifies the comparison among speedup lines of different groups. This comparison is used to quickly identify the biggest limitations to an application's speedup over a different number of threads. In this case, the speedup of the application can reach the highest values if the sequential fraction of the application is reduced (speedup with no sequential fraction line in the figure). The current speedup line shows the speedup obtained with the current parallelisation and is obtained by comparing the parallel application with its sequential version.

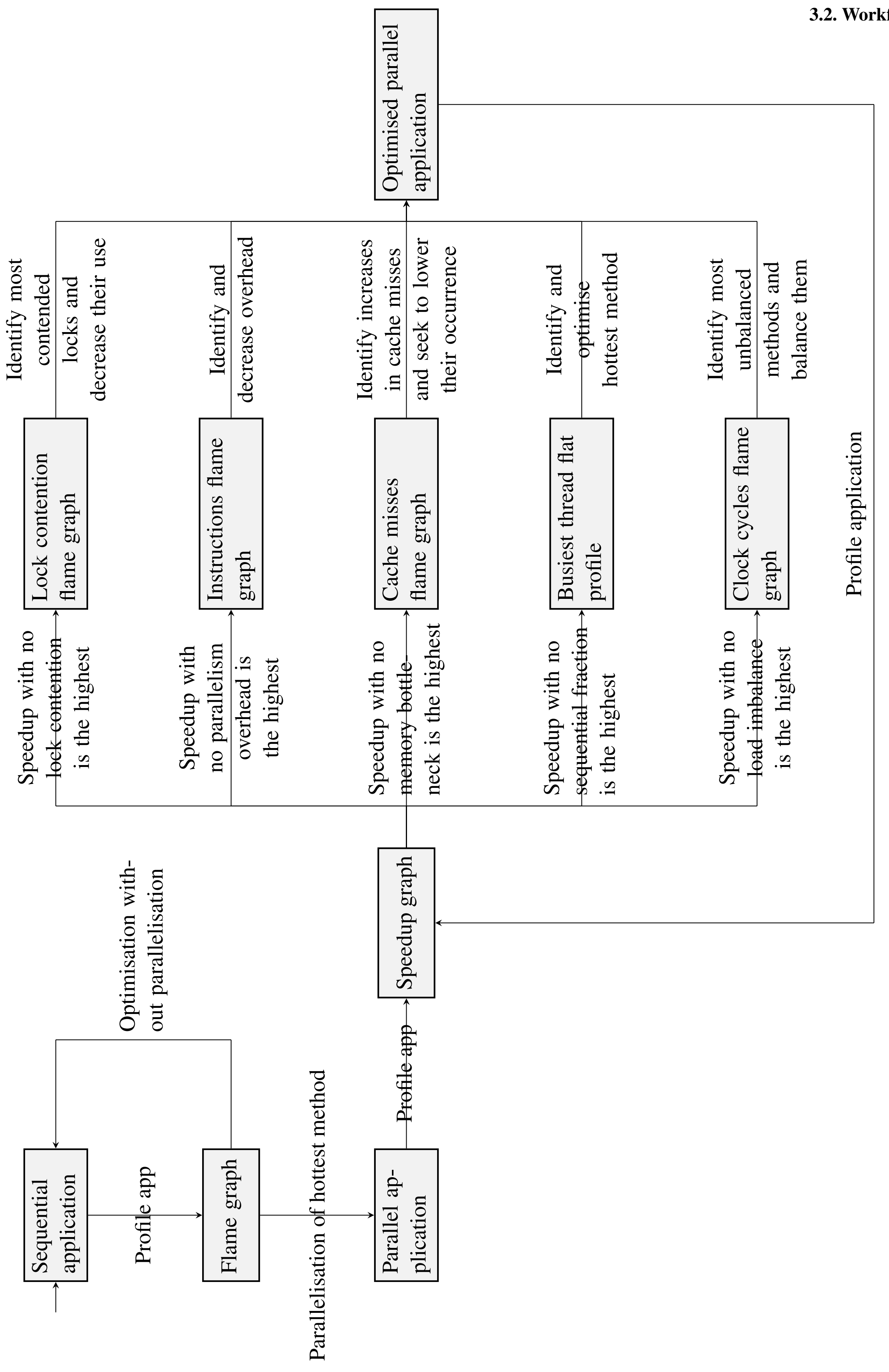

Sequential application
Profile app
Flame graph
Optimisation without parallelisation
Parallelisation of hottest method
Parallel application
Profile app
Speedup graph
Speedup with no lock contention is the highest
Speedup with no parallelism overhead is the highest
Speedup with no memory bottleneck is the highest
Speedup with no sequential fraction is the highest
Speedup with no load imbalance is the highest
Lock contention flame graph
Instructions flame graph
Cache misses flame graph
Busiest thread flat profile
Clock cycles flame graph
Identify most contended locks and decrease their use
Identify and decrease overhead
Identify increases in cache misses and seek to lower their occurrence
Identify and optimise hottest method
Identify most unbalanced methods and balance them
Optimised parallel application
Profile application

Figure 19.: General application workflow

Since the remaining speedup lines mostly overlap the current speedup line, solving any remaining performance limitation won't improve the application's speedup as much.

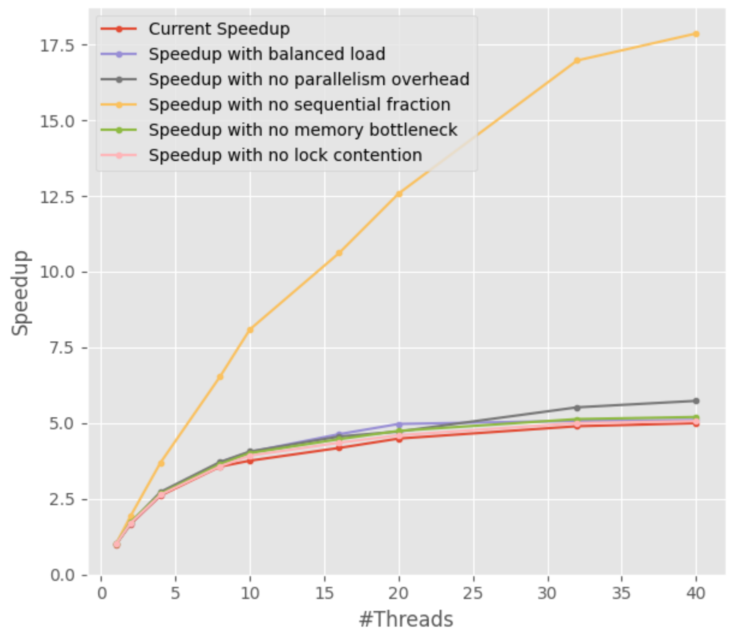


Figure 21.: Comparison of possible speedups

If the speedup graph implies that the sequential fraction is the most limiting factor of the application's global speedup, as in figure 21, then it means the sequential portion of the application started to take a bigger chunk of the application's time as the parallelised portion's execution time shrunk. In this case, the workflow suggests we should look at the busiest thread flat profile (figure 22). This graph allows us to have a better understanding of the impact of the sequential fraction of the application by comparing the number of clock cycles spent on innermost methods of the sequential fraction of the application and the parallel fraction of the busiest thread over the execution of the application for different numbers of threads. This graph essentially displays the innermost methods with the highest number of clock cycles, suggesting that these should be the ones to be optimised. In the example of figure 22, the number of cycles spent on the method `m1` in the busiest thread gets reduced to the point that further optimisations to it will have little impact in the performance of the whole application as the number of threads increases. In this case, `m2` becomes the most attractive method to optimise given its higher number of clock cycles when running the application on more than 10 threads.

In the case that the innermost methods don't reveal much, as in the case of figure 23a, a clock cycles flame graph of the sequential fraction of the application can be used to have a better idea of the hottest methods stack. Figure 23b allows us to see that optimising `m2` could provide a good performance improvement, while the flat profile can't show us this detail.

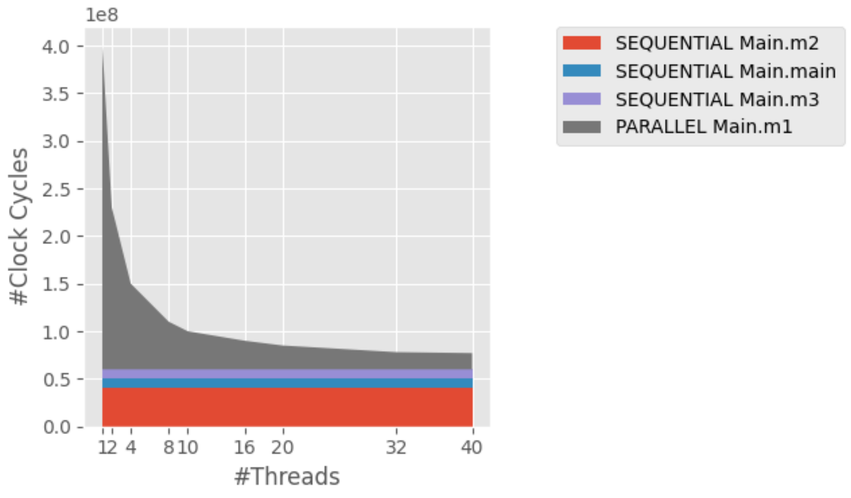
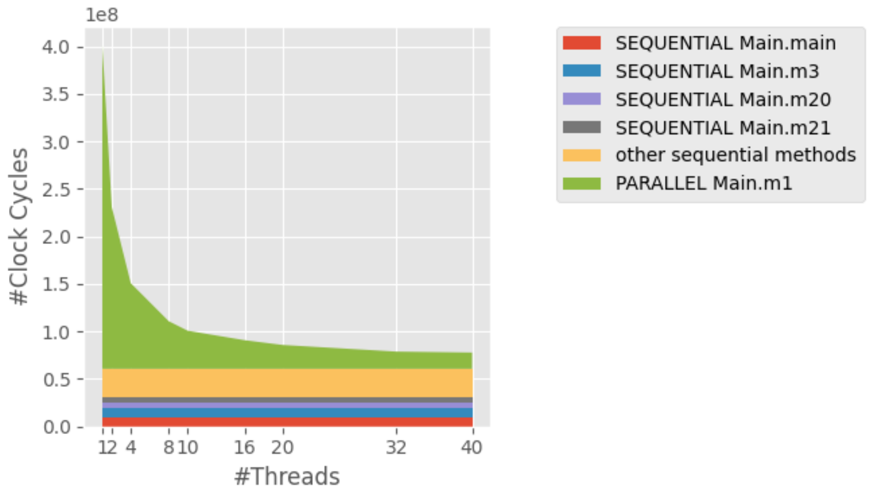
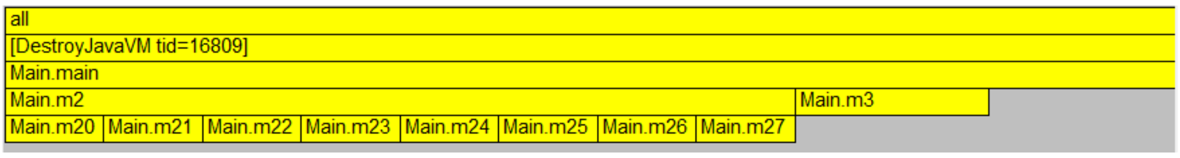


Figure 22.: Flat profile of the busiest thread



(a) Flat profile of the busiest thread



(b) Clock cycles flame graph of the sequential portion of an application

Figure 23.: In this application, the number of clock cycles of innermost methods is well distributed between different methods, as seen in 23a. Looking at 23b, however, we can see that most clock cycles originate from methods called inside `m2`, making this method a prime candidate for optimisation.

After optimising this parallelisation by decreasing the sequential fraction of the application, we can look into the speedup graph of the new application to discover its biggest performance limitations and seek to optimise accordingly.

To address other performance limitations, flame graphs are also suggested to look deeper into each limitation, changing only the metrics presented in the flame graph. For applications where load imbalance is the bottleneck, there should be a visible difference in the number of clock cycles of the unbalanced methods between the different threads in a clock cycles flame graph. Comparing cache misses flame graphs for different levels of parallelism, displaying in which call stacks memory accesses increase, helps in identifying where memory access limitations may occur from. To identify the sources of parallelism overhead, a comparison between instructions flame graphs is necessary to pin down increases in instruction count of different methods. If lock contention is the culprit of bad scalability, a lock contention flame graph displays the time spent waiting on different locks by each thread, revealing where contention mostly occurs.

3.3 METRICS

For each group of performance limitations presented previously, an efficiency metric was developed to allow the estimation of the possible speedup lines used in the workflow. If the efficiency of a group equals 1, that means such group doesn't create any inefficiency on the application's performance. However, we cannot simply compare these efficiencies in order to find which group is the most harmful to the overall speedup because some groups may be more impactful than others, regardless of having a higher efficiency. To measure each efficiency group's impact on the overall speedup of the application, we can estimate the achievable speedups of the application if such group caused no inefficiencies. By comparing these possible speedups without inefficiencies we can determine which group is being the biggest bottleneck on the application's performance and optimise accordingly to obtain the largest performance increases. In this section we present the implementation of the efficiency groups used in the suggested workflow, how their efficiencies are computed and how their impact on the speedup of an application can be compared.

The efficiencies presented in this section require profiling the number of clock cycles, $\#CC$, and instructions executed, $\#I$, of a sequential application and its parallelised version, identified by the subscripts $_S$ and $_P$, respectively. The number of clock cycles and instructions of these profiles must also be divided into the ones ran or to be ran in parallel, identified with the subscript p, and the ones ran sequentially in both applications, identified by the subscript s. For example, $\#I_{Sp}$ is the number of instructions executed in the sequential application of methods to be made parallel. The profile also must be able to differentiate the number of clock cycles and instructions of different threads. To obtain results regarding lock contention, the profile of the time spent waiting on locks should also be available for the parallel application, since the impact of this limitation could not be accurately estimated with the number of instructions or clock cycles. All metrics used to compute the efficiencies are described in table 2.

This section also assumes that both applications, sequential and parallel, run on the same fixed processor clock rate. Another assumption is that the machine running the parallel application has at least one core available for each thread of the parallel application, all threads suffer the same degree of parallelism overhead and lock contention times and that the busiest thread is always the busiest thread whenever any parallel work is distributed.

Primitive Metric	Description
$\#I_{Ss}$	Number of instructions executed on the sequential fraction of the sequential application
$\#I_{Sp}$	Number of instructions executed on the parallel fraction of the sequential application
$\#I_{Ps}$	Number of instructions executed on the sequential fraction of the parallel application
$\#I_{Pp}$	Sum of instructions executed on the parallel fraction of the parallel application by all threads
$\#CC_{Sp}$	Number of clock cycles spent on the parallel fraction of the sequential application
$\#CC_{Pp}$	Sum of clock cycles spent on the parallel fraction of the parallel application by all threads
$\#CC_{PpM}$	Number of clock cycles spent on the parallel fraction of the parallel application by the busiest thread
$T_{Pp_{lock}}$	Total time spent waiting on locks by all threads on the parallel fraction of the parallel application
$T_{Pp_{exec}}$	Total time spent by all threads on the parallel fraction of the parallel application
Derived Metric	**Description**
$\#CC_S$	Number of clock cycles spent on the whole sequential application
CPI_{Ss}	Average CPI of the sequential fraction of the sequential application
CPI_{Ps}	Average CPI of the sequential fraction of the parallel application
CPI_{Sp}	Average CPI of the parallel fraction of the sequential application
CPI_{Pp}	Average CPI of the parallel fraction of the parallel application

Table 2.: Metrics used to compute efficiencies

Sequential Fraction - Parallel Fraction Efficiency

The parallel fraction efficiency aims to quantify the performance limitation that arises from Amdahl's law stating that the maximum speedup achievable with parallelisation is limited by the sequential fraction of the application. This efficiency, or enhanced fraction, is the proportion of execution time that the methods to be made parallel occupy on the sequential application.

Since we're assuming these applications are running on a fixed processor frequency, the fraction of execution time will be equal to the fraction of clock cycles. Equation 3 evaluates the parallel fraction efficiency (E_{PF}) as the fraction of clock cycles spent executing instructions to be made parallel ($\#CC_{Sp}$) out of the clock cycles spent during the execution of the whole sequential application ($\#CC_S$).

$$E_{PF} = \frac{\#CC_{Sp}}{\#CC_S} \tag{3}$$

Parallelism Overhead - Instruction Overhead Efficiency

The parallelisation of an application may introduce some noticeable overhead due to added instructions necessary for parallelisation. These additional instructions may appear in the sequential and parallel fractions of the application and depending on where these instructions appear, they'll have a different impact on the overall speedup of the application due to other factors such as the parallel fraction efficiency and the number of processors used. The instruction overhead efficiency aims to quantify the parallelism overhead by measuring the increase in the total number of instructions.

Since the increase in the number of instructions has a different impact on the scalability of an application depending on which fraction of the application this increase occurs, the instruction overhead efficiency is split into two efficiencies, the instruction overhead efficiency on the sequential fraction and on the parallel fraction.

The instruction overhead on the sequential fraction efficiency, E_{SO_I}, in equation 4 is obtained by dividing the number of instructions executed in the sequential fraction of the sequential application, $\#I_{Ss}$, by the number of instructions executed in the sequential fraction of the parallel application, $\#I_{Ps}$.

$$E_{SO_I} = \frac{\#I_{Ss}}{\#I_{Ps}} \tag{4}$$

The instruction overhead on the parallel fraction, E_{PO_I}, shown in equation 5 is the ratio between the number of instructions executed in the parallel fraction of the sequential application, $\#I_{Sp}$, by the number of instructions executed in the parallel fraction of the parallel application, $\#I_{Pp}$.

$$E_{PO_I} = \frac{\#I_{Sp}}{\#I_{Pp}} \tag{5}$$

The efficiency of the instruction overhead on the parallel fraction assumes that the increase in the number of instructions is distributed evenly between all threads.

Memory overhead - CPI Overhead Efficiency

Memory access overhead is a performance limitation that may arise in a parallel application. The increase in the latency or number of memory accesses translates into an increase of the average CPI of the application. CPI is a metric that measures the average number of clock cycles required to execute an instruction, as shown in equation 6. The CPI overhead efficiency aims to quantify the memory access overhead by measuring the increase in the average CPI of the application.

$$CPI = \frac{\#CC}{\#I} \tag{6}$$

The CPI overhead efficiency is also split into two efficiencies, the CPI overhead efficiency on the sequential fraction and parallel fraction, due to the different impact on scalability depending on which fraction the CPI increase occurs.

The average CPI overhead efficiency on the sequential fraction, $E_{SO_{CPI}}$, is defined in equation 7 as the ratio between the average CPI of the sequential fraction of the sequential, CPI_{Ss}, and parallel programs, CPI_{Ps}.

$$E_{SO_{CPI}} = \frac{CPI_{Ss}}{CPI_{Ps}} \tag{7}$$

The average CPI overhead efficiency on the parallel fraction, $E_{PO_{CPI}}$, in equation 8 is the ratio between the average CPI of the parallel fraction of the sequential, CPI_{Sp}, and parallel programs,CPI_{Pp}.

$$E_{PO_{CPI}} = \frac{CPI_{Sp}}{CPI_{Pp}} \tag{8}$$

The use of average CPI to quantify memory access overhead assumes that increases in CPI are caused mostly due to memory access overhead. The efficiency of the average CPI overhead on the parallel fraction assumes that the increase of the number of instructions is distributed evenly between all threads.

Load Imbalance - Load Balance Efficiency

The parallel fraction of the application is distributed between the multiple threads running the parallel application and load imbalance happens when the time spent performing this work is not evenly distributed between the threads. The load balance efficiency seeks to evaluate the balance of the workload between all the threads of the parallel application. This efficiency can be evaluated as the ratio between the average time a thread spends on the parallel fraction of the code and the total time the busiest thread spends on the parallel fraction of the code.

Since we're assuming that the clock cycle frequency is constant, we can evaluate the load balance efficiency with clock cycles, as in equation 9, where the load balance efficiency, E_{LB} is evaluated as the average number of clock cycles a thread spends on the parallel fraction, $\frac{\#CC_{Pp}}{\#T}$, divided by the number of clock cycles the busiest thread spends on the parallel fraction of the code, $\#CC_{PpM}$.

$$E_{LB} = \frac{\#CC_{Pp}}{\#CC_{PpM} \times \#T} \tag{9}$$

This efficiency formula is limited by the assumptions that the clock rate is constant and that the busiest thread is consistently the thread with the most work if parallel work is issued multiple times after a sequential section.

Lock Contention - Lock Contention Efficiency

The parallelisation of an application may sometimes require the introduction of locks to guarantee exclusive access to some resource. The introduction of these locks can reduce scalability by forcing serial execution on the parallel fraction of the program. Lock contention efficiency evaluates how the introduced locks affect the application's performance.

Lock contention efficiency, E_{LC}, is defined by the parallel time of the application that was not spent waiting on locks divided by the total parallel time of the application as presented in equation 10, where $T_{Pp_{lock}}$ is the total time threads spent waiting on locks and $T_{Pp_{exec}}$ is the total time spent executing the parallel fraction of the code. This efficiency is also assuming that each thread spends a similar amount of time waiting on locks.

$$E_{LC} = 1 - \frac{T_{Pp_{lock}}}{T_{Pp_{exec}}} \tag{10}$$

3.3.1 *Evaluating the impact of efficiencies*

Most of the efficiencies presented previously cannot be compared directly to determine which one is the biggest limitation on the speedup of a parallel application. One way to visualise the real impact each one of these efficiencies has on the overall speedup of an application is to estimate the achievable speedup if we consider these efficiencies to equal 1 and compare it to the real speedup of the application. This would give an indication of possible performance returns by solving all the problems related to one of these efficiencies.

Equation 11 evaluates the speedup of an application if we take into consideration all the assumptions made when presenting these efficiencies. The left side of the denominator represents the inefficiencies introduced on the sequential fraction and the right side represents the inefficiencies introduced on the parallel fraction of the application. Using this speedup equation, we can easily obtain the maximum speedup achievable if we eliminate one type of inefficiency. For example, if we want to find the speedup of the parallel application if we were to remove all parallelism overhead caused by the increase in number of instructions of the application, we can evaluate equation 11 assuming E_{SO_I} and E_{PO_I} equal 1. This would only be the maximum attainable speedup because lowering one inefficiency may cause other inefficiencies to grow.

$$\text{Speedup} = \frac{1}{\frac{1-E_{PF}}{E_{SO_I} \times E_{SO_{CPI}}} + \frac{E_{PF}}{E_{PO_I} \times E_{PO_{CPI}} \times E_{LB} \times E_{LC} \times \#T}} \tag{11}$$

3.4 CASE STUDIES

This section presents some case studies to illustrate the use of the suggested workflow to analyse applications and the results obtained. First, the optimisation workflow is tested in the iterative optimisation of a k-means algorithm starting from a sequential implementation. Afterwards, the possible speedup analysis is made for some additional algorithms to discover the major scalability issues in their implementations.

All the case studies presented in this section were performed on a Linux NUMA machine with two Xeon E5-2670v2 processors (i.e., 10+10 physical cores, with HT enabled), and frequency fixed to 2.5 GHz. To run and profile the Java applications, OpenJDK 11 (build 11+28) and async-profiler v2.0 were used.

3.4.1 *K-means*

The k-means case study is an algorithm that aims to classify a set of data points into a predefined number of clusters based on the closest cluster centroid to the data point. The implementation used is based on the implementation from [9]. It consists of 2 main steps, computing the centroid of each cluster and reassigning each point to the cluster with the closest centroid. These steps are repeated until there is no change in the classification of the data points.

The initial implementation of the k-means algorithm is a sequential one. We can look into possible optimisations of this sequential version by profiling this implementation and identifying its hottest methods. In figure 24 we have the clock cycles flame graph for this sequential application. Here we can easily identify the hottest methods and select which one might be the most adequate target for parallelisation. In this case, the highlighted reassign method, the method responsible for the whole step of reassigning points to clusters, encompasses most of the application's cycles and is a great target for parallelisation.

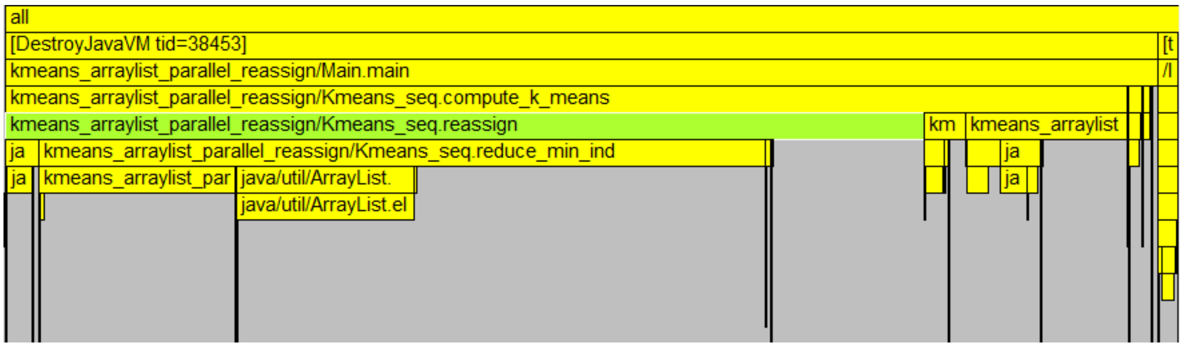


Figure 24.: Clock cycles flame graph of the initial k-means implementation

After creating the first parallel version of the algorithm, we can look into the possible speedups of this application in figure 25. Here we see that the overall speedup is the highest if there was no sequential fraction, meaning that the sequential fraction of the code is the most limiting factor of the application's scalability. Figure 26 displays the innermost methods found in the stack samples for the busiest thread and while it may hide some relevant information, we can use it to gain some clues into what might be the next optimisation target and, in this case, we clearly see methods related to the `ArrayList` class and the *tally* step of the algorithm are some of the hottest points of the application.

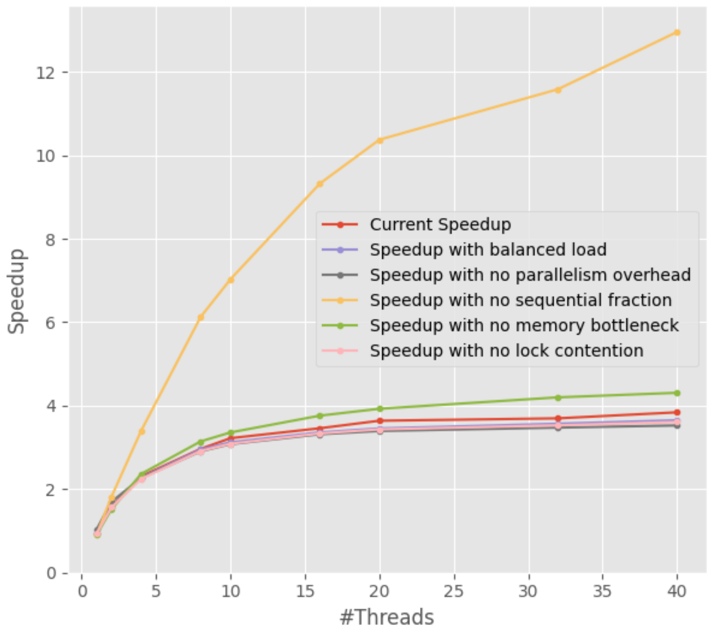


Figure 25.: Possible speedup of the first optimisation to the k-means algorithm

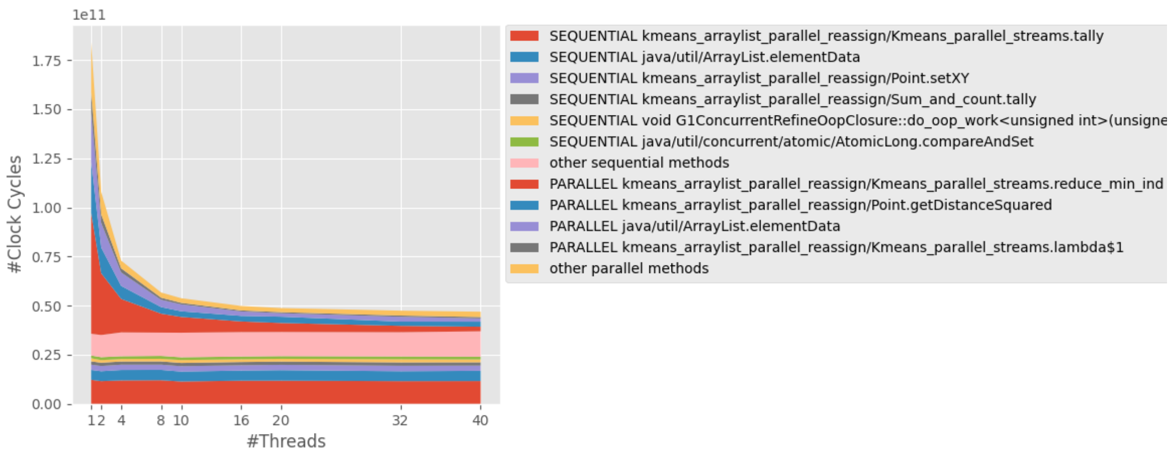


Figure 26.: Flat profile of the busiest thread of the first optimisation to the k-means algorithm

The second optimisation to the application sought to improve performance by removing the `ArrayList` structure and replacing it with arrays. Since this optimisation could have been done to the initial sequential application, the speedup of this parallel version is compared against a sequential application subject to the same optimisation. After this optimisation, the most limiting factor was still the sequential fraction of the application (figure 27) and looking at figure 28 we see the hottest innermost sequential methods are related to the *tally* step of the algorithm. This information can be confirmed in the sequential fraction of the clock cycles flame graph of the application (figure 29).

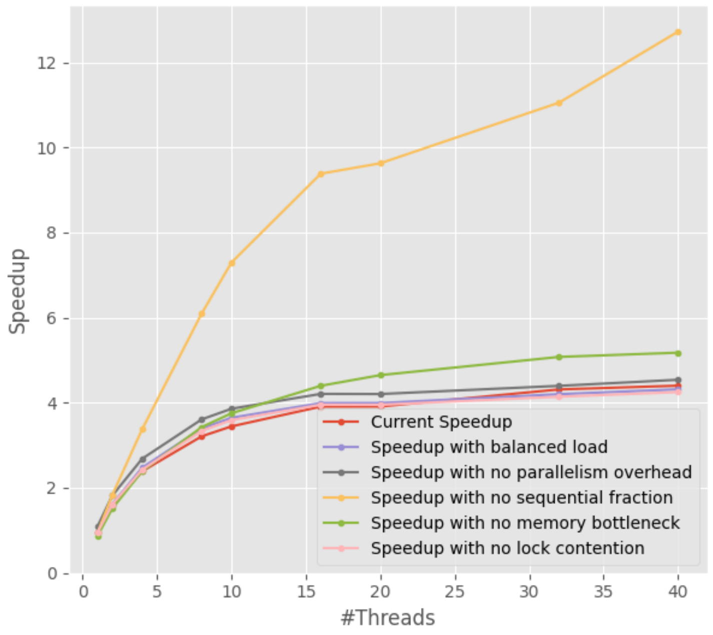


Figure 27.: Possible speedup of the second optimisation to the k-means algorithm

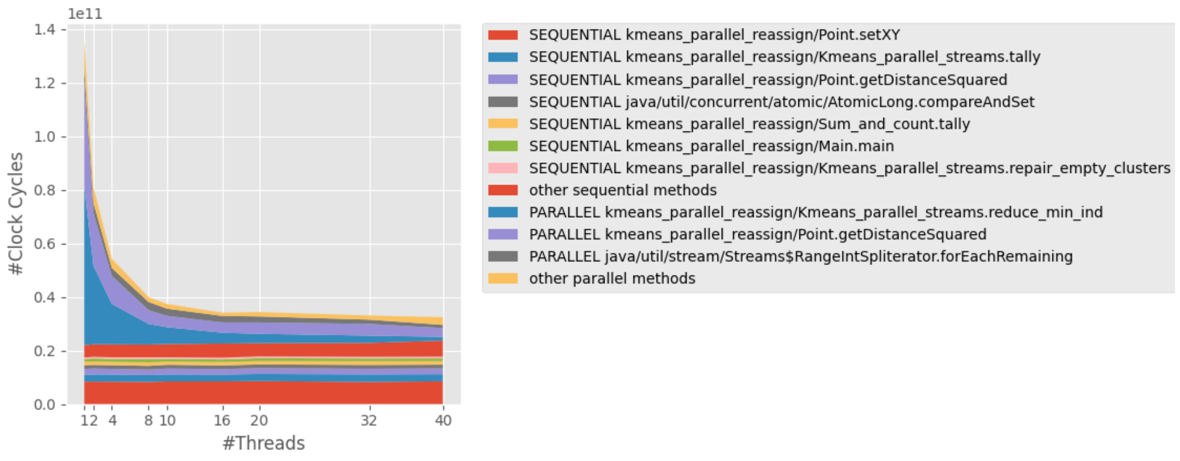


Figure 28.: Flat profile of the busiest thread of the second optimisation to the k-means algorithm

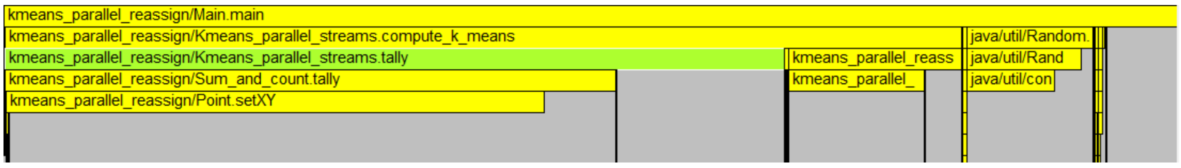


Figure 29.: Clock cycles flame graph of the sequential fraction of the second optimisation to the k-means algorithm

The tally step aims to reduce the values of the data points in each cluster as a part of the step responsible for computing the centroid of each cluster. The parallelisation of the tally step was carried out resorting to locks resulting in a speedup below 1. Figure 30 displays the speedup lines for this version of the application and it suggests memory bottleneck and lock contention create the biggest constraints on the speedup of the application. It is important to note that a big discrepancy is observed between the real speedup and the calculated speedup of the application, pictured in figure 31, which affects the calculations of the possible speedups. The reason for this may be due to the fact that profiling lock contention can induce a large overhead on the execution time of the application. The application is profiled multiple times to profile different metrics in separate executions, the execution times for this version of the algorithm when profiling different metrics is shown in figure 32, which confirms the large overhead caused by the lock profiling when compared to the profiling of other metrics. One effort to mitigate this issue is discussed in the section describing the implementation of the analysis tool. But while these speedup lines are obtained with huge overhead and the comparison between these lines may not be accurate, it can still be seen that lock contention is one of the main causes this version of the application has a terrible speedup. Checking the flame graph displaying how much time each thread spends on each lock reveals that all threads spend a large amount of time waiting for the lock of objects of the same class. Digging deeper into the memory bottleneck issue, this parallel version of the algorithm can reach up to 30 times more total cache misses, as seen in figure 33, suggesting a big decrease in cache locality.

In order to remove the limitation imposed by locks introduced in the previous parallelisation, the following optimisation assigns for each thread, an instance of the data structure that was previously being shared. This approach has improved the speedup of the application, as seen in figure 34, and it changed the main limitation of the application's speedup, which is now the parallelism overhead caused by the instruction increase imposed by the implemented parallelisation, at least up until 16 threads. The flame graph in figure 35 reveals this problem by showing the distribution of instructions inside the parallel tally step, where methods related to the `Map` class represent overhead of this parallelisation since they were not present in the sequential application.

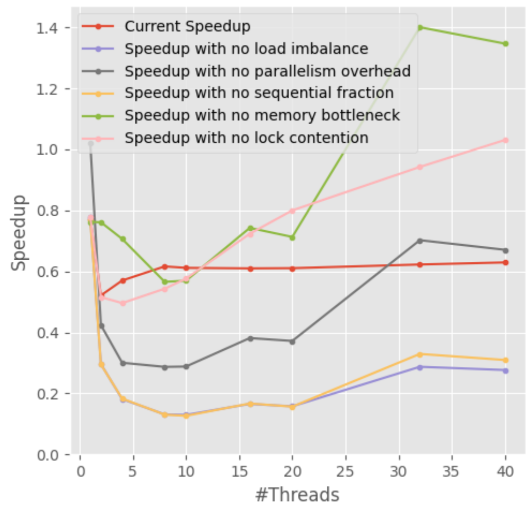


Figure 30.: Possible speedup of the third optimisation to the k-means algorithm

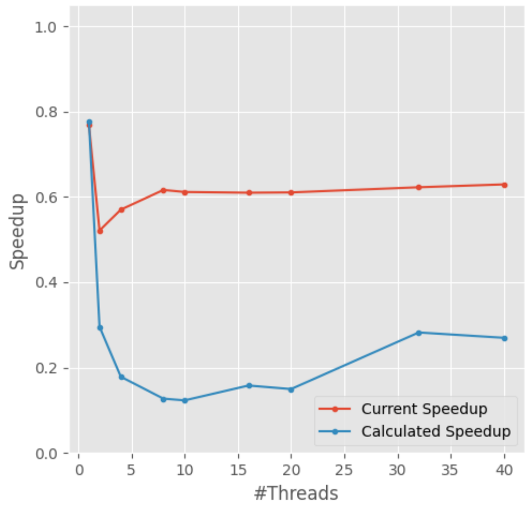


Figure 31.: Comparison between the real and calculated speedups of the third optimisation to the k-means algorithm

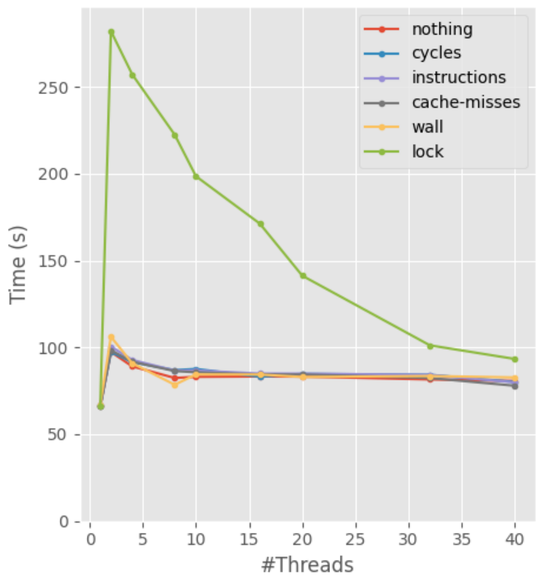


Figure 32.: Profiling overheads of the third optimisation to the k-means algorithm

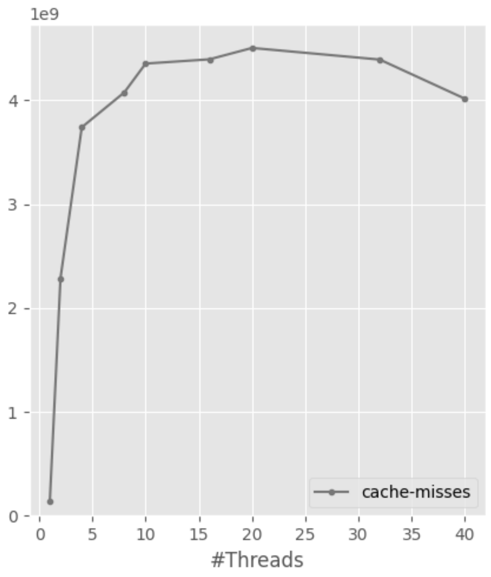


Figure 33.: Number of cache misses of the third optimisation to the k-means algorithm

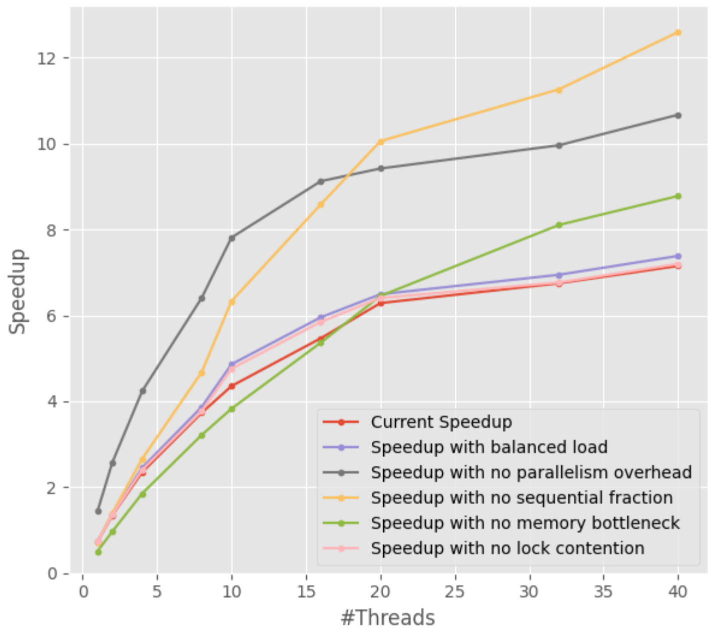


Figure 34.: Possible speedup of the fourth optimisation to the k-means algorithm

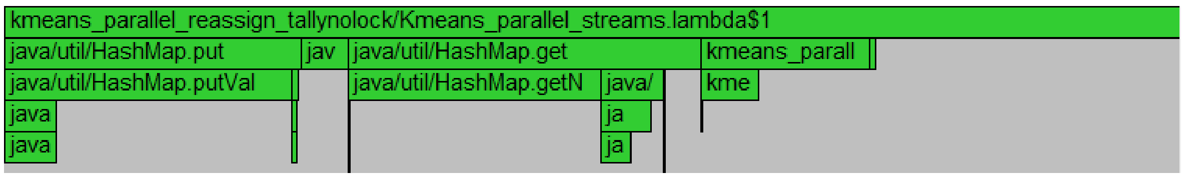


Figure 35.: Instruction flame graph of the fourth optimisation to the k-means algorithm

Figure 36 shows the speedup lines for an improvement on the previous application in which some of the parallelism overhead was reduced by removing some unnecessary management of data structures (removing the need for the *Map* structure). This results in the sequential fraction being, again, the main limitation on the scalability of the application, meaning we should look, again, into the sequential fraction of the application to improve the scalability of the application.

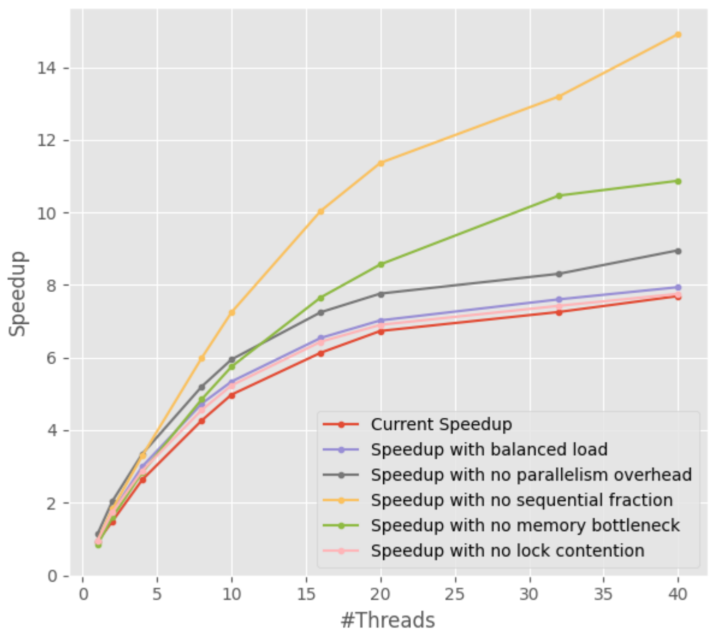


Figure 36.: Possible speedup of the fifth optimisation to the k-means algorithm

3.4.2 *Other case studies*

This section presents some other case studies with the intention of covering some paths in the decision tree of the workflow not observed in the k-means case study.

Figure 37 presents the possible speedups for a molecular dynamics simulation based on an implementation from [15]. This implementation has its scalability limited mostly by the memory bottleneck. This is because the number of cache misses in the parallel application can reach 100 times more misses as the number of threads increases, which could be seen by comparing cache misses counts of the cache misses flame graph of the sequential and parallel applications, but can also be seen with a wider perspective in a graph displaying the total count of cache misses over different levels of parallelism as shown in figure 38.

Figure 39 shows the possible speedups for the same application with a modification to the algorithm that causes the uneven distribution of the parallel workload. In this case, the application still suffers a limitation due to memory bottleneck but also due to load imbalance. The load imbalance can be spotted in figure 40, a clock cycles flame graph of the application running with 4 threads, where the busiest thread runs for almost 10 times more clock cycles than the thread with the smallest workload.

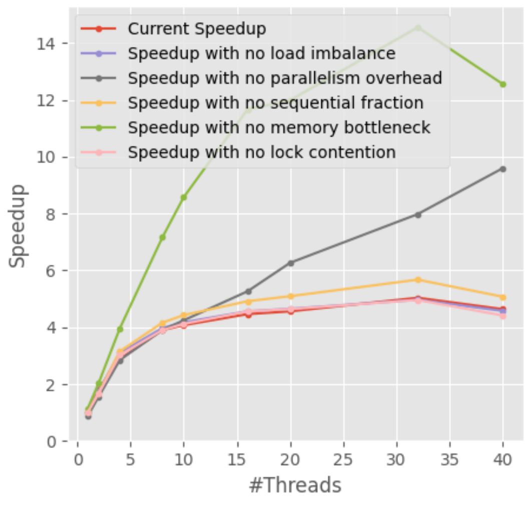


Figure 37.: Possible speedup of the MD algorithm

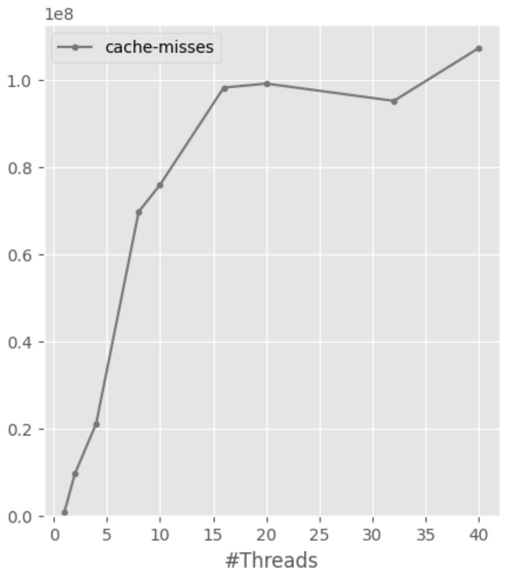


Figure 38.: Cache misses of the MD algorithm

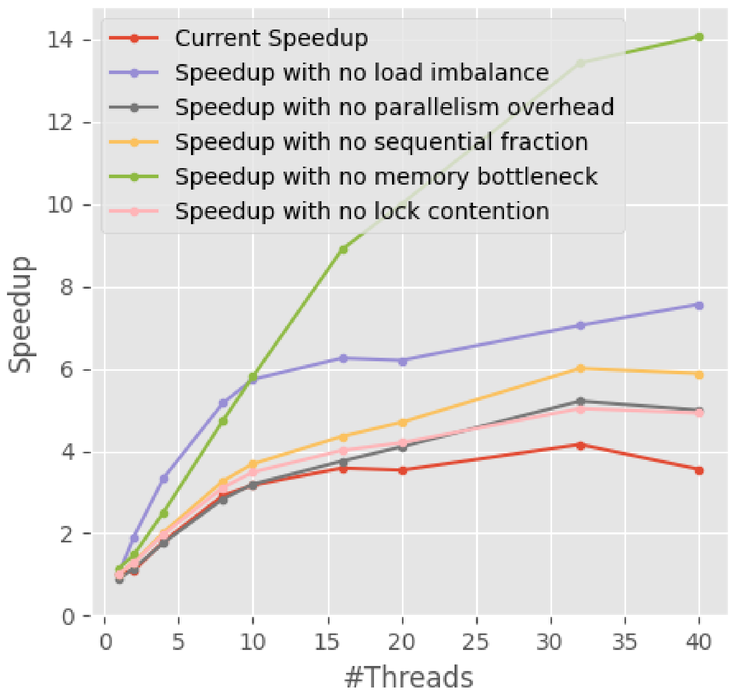


Figure 39.: Possible speedup of the unbalanced MD algorithm

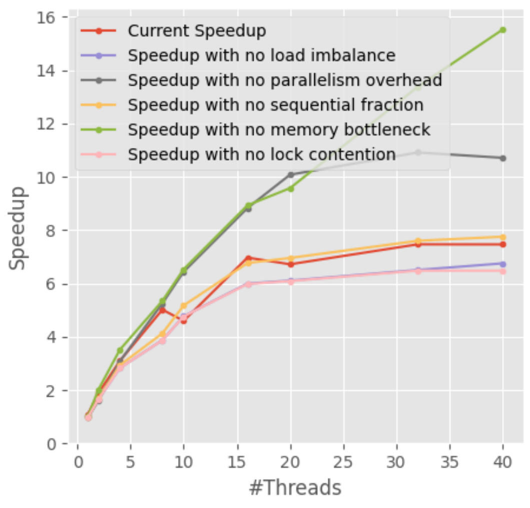


Figure 41.: Possible speedup of the first ray tracing algorithm

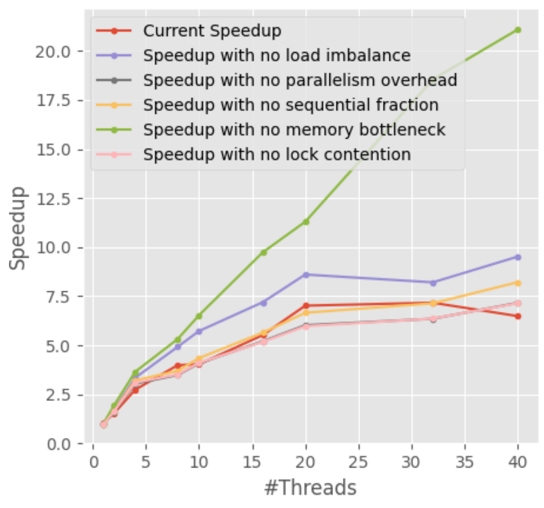


Figure 42.: Possible speedup of the second ray tracing algorithm

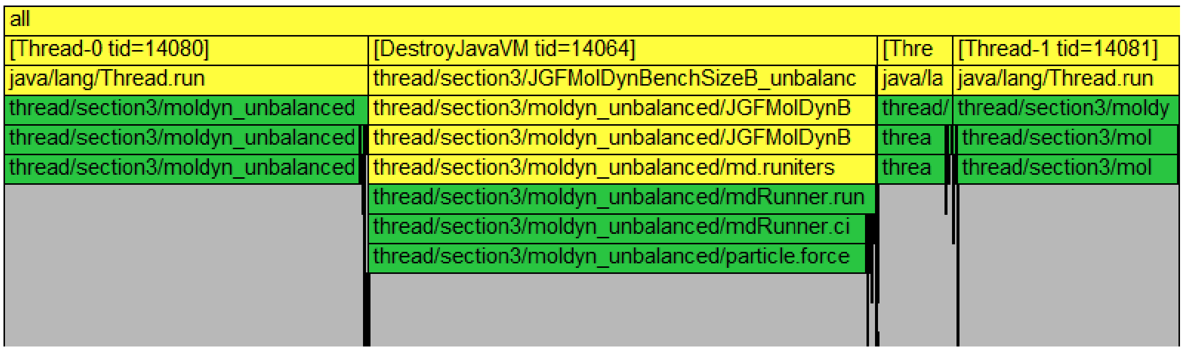


Figure 40.: Clock cycles flame graph of the unbalanced MD algorithm

The following two applications implement a ray tracer algorithm also based on the implementation from [15]. The first one implements its own barrier for synchronisation. The second application implements synchronisation by using *CyclicBarrier* found in Java's library. The possible speedups for these applications can be found in figures 41 and 42, respectively. These graphs suggest both applications seem to be limited by memory accesses. However, the first implementation seems to also be limited by overhead introduced with the parallelisation, while the second implementation sees load imbalance as the second biggest limitation of scalability.

Inspecting a portion of the instructions flame graph of the first application, in figure 43, we see the additional instructions introduced by the barrier. For each thread, there are 2 notable stacks, one corresponds to the ray tracer algorithm and the other to the barrier. The stack related to the barrier creates the instruction overhead due to busy waiting and hide the load imbalance. This flame graph also shows an instruction imbalance, seen by the fact that the stacks of the different threads have different widths, which did not translate into a workload imbalance.

The profile of the second application does not display stack samples referring to the barrier. This makes it so that there is no apparent instruction overhead and reveals the workload imbalance, as seen in its clock cycle flame graph, figure 44. This workload imbalance is not due to an imbalance in the number of instructions of each workload, but due to an imbalance in CPI of the workloads, as it would be seen in a cache misses flame graph.

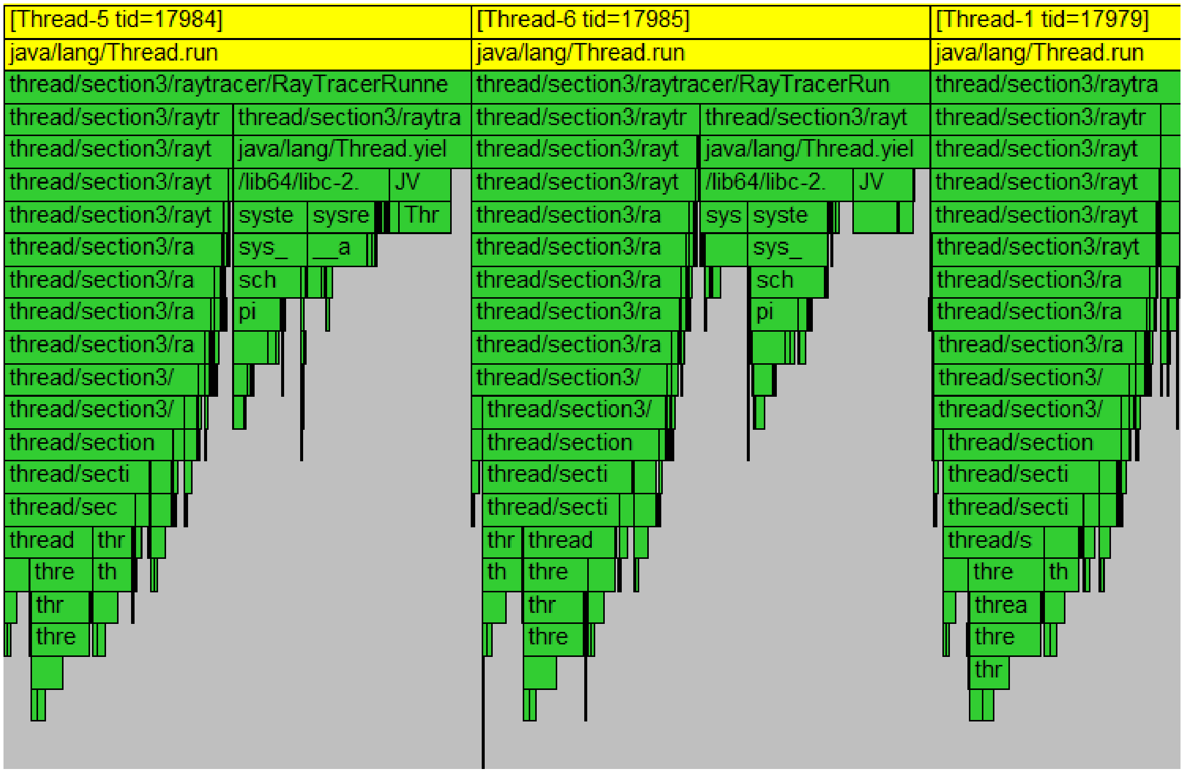


Figure 43.: Fraction of the instructions flame graph of the first ray tracing algorithm

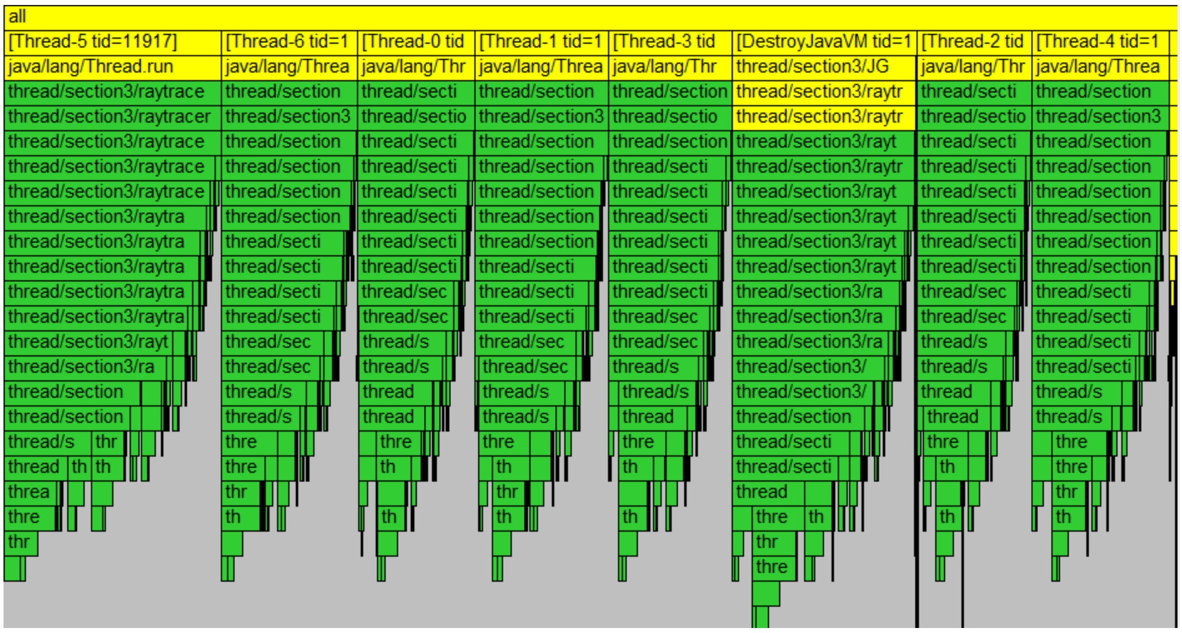


Figure 44.: Clock cycles flame graph of the second ray tracing algorithm

3.5 TOOLS IMPLEMENTATION

To support the suggested workflow, two tools were built, a profile collector and a profile visualiser. The first consists of a python script making use of async-profiler to collect stack trace samples of sequential and parallel versions of a program. The reason for choosing this profiler was due to its capability of profiling every required metric, easy attachment to the application, identification of Java, JVM, native and kernel functions and the low overhead it introduces. The path for async-profiler should be specified inside the script. This script runs the given sequential and parallel programs multiple times to collect multiple metrics separately. It profiles cycles, instructions, cache-misses, wall time and time spent on locks in different executions for each number of threads and the number of iterations intended (used to create an average profile). The result of this script is a file with the profile of every metric and for every number of threads and iterations. Each profile consists of a list of stack traces accompanied by a number which represents the estimated count of a metric. The usage of the profile collector is shown in appendix A.

The profile visualiser can be used to generate visualisations to follow the presented workflow. This tool, also implemented in python, takes as input the output of the profile collector as well as 2 regular expressions to identify the parallel stack samples of the sequential and parallel applications. The definition of these 2 regular expressions is important to classify samples as sequential or parallel and thus be able to perform the possible speedup calculations, which are to be performed using the equations presented previously.

The visualiser starts by parsing the given file and storing the profile of each program execution in a tree data structure. This structure was chosen to simplify the drawing of flame graphs and speed up the process of looking up which methods are called inside other methods and how many total samples this equates to. Each node of the tree identifies the name of a function, its sample count, if it belongs to the parallel fraction of the application and the nodes of functions called inside this function. The sample count is the number of samples collected in which the function is sampled as being the innermost function.

After the file is parsed, resulting in multiple profiles of different metrics for different numbers of threads stored in trees, the efficiencies used to measure the impact of the different limitations are calculated using the formulas presented previously, except for the lock contention efficiency. Due to big overheads when profiling the application for locks, sometimes resulting in lock waiting times per thread higher than the total application time when it's not being profiled, the time spent waiting on locks is multiplied by the ratio between the execution time of the application when it's being profiled for wall time and lock time. This is an attempt of trying to find what the time spent on locks would be if lock profiling had the same overhead as wall time profiling. This attempt is flawed because the overhead is not consistent throughout the application. But while it may be a flawed solution, it does bring the calculated speedup of the applications closer to their real speedup.

After calculating possible speedups, the application provides the options shown in figure 45. General metrics displays a graph showing the evolution of the number of clock cycles, instructions, CPI and cache-misses with the increase in the number of threads. Execution times displays the execution times of the application with and without profiling, allowing the comparison of profiling overheads. Possible speedup gains displays the possible scalability gains by completely removing specific limitations. Flat profile of the busiest thread option displays a graph useful for comparing the fraction of clock cycles spent on the sequential and parallel fractions as the number of threads increases. The flame graphs option can display flame graphs for clock cycles, instructions, cache misses, wall time and time spent on locks for any number of threads.

3.6 LIMITATIONS OF THE WORK DEVELOPED

The aim of the work developed during this dissertation was to create a methodology and tools capable of aiding programmers in the process of increasing a parallel application's scalability. The resulting work aims to accomplish this by pointing the biggest scalability bottlenecks of a given application, and with this information, help the programmer in deciding where to seek optimisations. While the presented work is able to achieve its goals, especially for simpler applications, it tends to fail in some specific scenarios. This section exposes some of the issues that may affect the effectiveness of the solution suggested.

3.6.1 *Possible Speedups*

There is no guarantee that the estimated possible speedup values can be achieved, or even that aiming to reduce the performance problems responsible for the biggest scalability bottlenecks achieves the largest performance gains. For example, an application may suffer performance limitations mostly due to delays in memory access. This limitation is usually impossible to overcome completely. If, in practice, only a small fraction of the possible speedup can realistically be achieved, seeking to make optimisations against this performance limitation may be futile compared to other limitations.

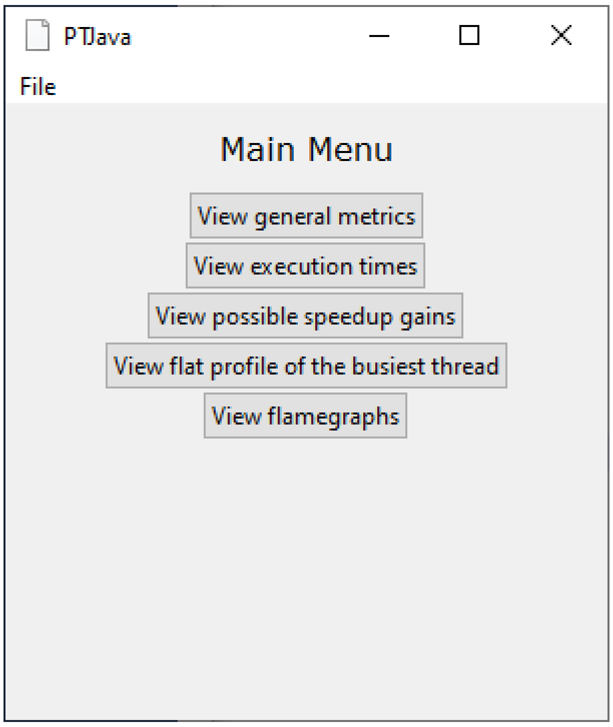


Figure 45.: Profile visualiser's main menu

Another issue is that fixing some performance problems, may exacerbate the impact of others. For example, the sequential fraction of an application could always be considered to be a part of the parallel fraction but with maximum imbalance (only one thread executing all the parallel work). This would result in the total removal of the sequential fraction, but no increase the application's performance due to the imbalance increase of the parallel fraction of the application.

To predict the speedup improvements from removing multiple limitations, multiple efficiencies need to be considered equal to 1. Trying to predict the possible speedup increases from removing multiple limitations by merely looking at the possible speedups obtained from removing single limitations is challenging due to the existent cross impact of the limitations. In a similar work to the possible speedups approach which uses speedup stacks [14], the speedup losses associated with each limitation can be summed. Speedup stacks essentially attribute fractions of the difference between ideal speedup and real speedup to specific problems by measuring specific time overheads.

3.6.2 *Profiling Inaccuracies and Overheads*

This methodology will always be affected by the profiling inaccuracies and overheads introduced by the profiler used. Currently, the biggest impact caused by the profiler used is the overhead introduced by lock profiling which is based on instrumentation to capture Java object monitors and `ReentrantLocks`. This profile is used to estimate the impact of lock contention, so applications whose scalability is affected noticeably by lock contention suffer from a bigger inaccuracy in the estimation of possible speedups. This inaccuracy is more pronounced when locks are acquired more frequently in the application since this results in higher profiling overheads. To be aware of the possible inaccuracies in the estimation of possible speedups, it can be helpful to observe overheads introduced by profiling.

3.6.3 *Fixed Frequency*

This work assumes the PEs all run at the same fixed clock frequency throughout the whole application's execution. If an equal fixed frequency cannot be guaranteed, then the estimated calculated speedup becomes more inaccurate, however the overall methodology can still be applied. This problem affects mostly calculations involving the impact of lock contention because these are estimated using execution times, which are dependent on clock cycle frequency, instead of number of clock cycles or instructions.

Variable frequency can also affect the difference between the observed speedup of the application and the estimated speedup which should be equal to the observed speedup. The estimated speedup is the basis from which all possible speedups are calculated, so it is important for it to reflect the real speedup of the application.

3.6.4 *Spinning*

When threads are forced to wait for each other due to resource contention or load imbalances, they can either spin or be scheduled out of execution. Spinning results in the increase of the instruction count of the application and the incorrect attribution of a fraction of the scalability issues to parallelisation overhead. To overcome this issue, it is possible to remove stack samples referring to spin locks from the profile to obtain similar profiles to the applications that schedule their threads out of execution. However, this approach was not used due to the fact that removing these stack samples reduces the profile's information and also because when there is a significant amount of spinning for a particular application, it can easily be detected by inspecting the instructions flame graph.

3.6.5 *Load Imbalance*

Load imbalance is currently estimated by comparing the number of clock cycles of parallel workloads executed by the busiest thread with the total number of clock cycles of parallel workloads executed by all threads. This method of calculating load imbalance can be inaccurate if parallel work is issued multiple times. If there exists more than one parallel section in an application and the busiest thread is not always the busiest for all parallel sections, then the estimated load imbalance will be lower than its real impact. This happens because the busiest thread is used as a reference to the moments any thread is executing the parallel workload, which is only valid if the busiest thread is also busy whenever any other thread is busy.

To overcome this problem, there should be some reliable method of obtaining the number of elapsed clock cycles whenever any thread is executing parallel work. As it stands, the overall busiest thread is used as a reference, but if the collected profile was capable of distinguishing between different parallel sections, some of this inaccuracy could be removed by using the busiest thread for each parallel section as the reference.

3.6.6 *Multiple Parallel Sections*

The developed work does not have any option to specify more than one parallel section. While this does not make it unusable for applications with multiple parallel sections, it becomes less useful in locating performance limitations for these applications. Different parallel sections can often suffer from different bottlenecks, so multiple parallel fractions suffering from different performance problems would result in an application suffering from a mix of multiple performance limitations and no way to attribute these limitations to specific parallel sections.

4

CONCLUSIONS AND FUTURE WORK

4.1 CONCLUSIONS

Modern multicore computers have an increasing number of processing elements in order to raise their computational power in the face of limitations found for further increases in single core performance. Parallel applications seek to increase performance by taking advantage of these multiple processing elements available to them on modern multicore machines.

To take advantage of the increase in the number of PEs available, parallel applications must present good scalability, which means they should be capable of using an increasing number of processing elements efficiently. However, applications often do not achieve good scalability due to some performance problems commonly found in parallel applications, these problems were presented in section 2.2.

A profile can reveal information about the execution of an application. This information can be related to various metrics and can be used to obtain an insight into how the application uses its resources and find its performance bottlenecks. For Java applications, there are multiple state of the art profilers employing different strategies to collect an application's profile, each with their own distinctive characteristics. Async-profiler was the profiler chosen to collect metrics for the work produced in this dissertation due to its support for hardware performance counters to collect every metric that was deemed necessary, easy attachment to an application, low overhead and capability of identifying relevant Java, JVM, native or kernel functions.

The methodology suggested to aid in improving the scalability of Java parallel applications consists of identifying which performance problems are the most harmful to the application's scalability and, accordingly, recommending actions to further investigate and locate the sources of poor scalability. To identify the performance problems that most affect the scalability of an application, a comparison between the impact of different performance problems is made using the concept of *possible speedups*, which is a metric that attempts to predict the attainable speedup if a specific performance problem were to be completely removed. *Possible speedups* can be calculated from a set of some primitive metrics obtained from profiling the sequential and parallel versions of an application. The tools developed in conjunction with this methodology aim to facilitate the collection of metrics as well as create helpful visualisations required to effectively apply the suggested methodology.

The accuracy of the *possible speedups* approach to observe the impact caused by different performance limitations on the scalability of an application can usually be observed by comparing the real speedup with the calculated speedup. A bigger difference between these values results in more inaccuracy in the calculations of *possible speedups*. These differences can arise for multiple reasons, for example, non-fixed clock frequency, distinct parallel regions or lock profiling overhead.

The methodology was validated with the optimisation of an implementation of the K-means algorithm. The starting point of this process was a sequential implementation of the algorithm, from which the first parallel implementation was obtained. The parallel implementation was then subject to an iterative optimisation, which saw each iteration overcome different performance limitations in order to achieve better scalability.

While the workflow and tools presented intend to guide the analysis of Java applications, the idea behind them could also be used for any other application, as long as there is the possibility of collecting the necessary metrics to calculate the *possible speedups*.

4.2 PROSPECT FOR FUTURE WORK

The proposed workflow and tools could be improved in multiple ways. In its current state, inaccuracies may arise in various circumstances as discussed in section 3.6. Future work should seek to overcome some of these issues, starting with the possibility of defining multiple parallel regions with multiple regular expressions. This would be a worthwhile improvement, so that different parallel regions could be analysed separately as they could have different performance limitations. This could be better accomplished with the implementation of time-range profiles, to perform the analysis on specific time windows.

This work can also be expanded upon by considering additional and more specific performance limitations. With more specific performance limitations, the scalability bottlenecks can be more accurately located. For example, workload imbalance could be divided into instruction and CPI imbalance which would ease the identification of the reasons for a workload imbalance.

BIBLIOGRAPHY

[1] VisualVM - All-in-One Java Troubleshooting Tool. URL https://visualvm.github.io. [Online; accessed 12-December-2021].

[2] async-profiler - Sampling CPU and HEAP profiler for Java featuring AsyncGetCallTrace + perf_events. URL https://github.com/jvm-profiling-tools/async-profiler. [Online; accessed 12-December-2021].

[3] perf-map-agent - A java agent to generate method mappings to use with the linux 'perf' tool. URL https://github.com/jvm-profiling-tools/perf-map-agent. [Online; accessed 12-December-2021].

[4] Safepoints: Meaning, Side Effects and Overheads. URL http://psy-lob-saw.blogspot.com/2015/12/safepoints.html. [Online; accessed 23-December-2021].

[5] HotSpot Runtime Overview - Thread Management. URL http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#Thread%20Management|outline. [Online; accessed 12-December-2021].

[6] B. Gregg and M. Spier. Java in Flames, 2015. URL https://netflixtechblog.com/java-in-flames-e763b3d32166. [Online; accessed 12-December-2021].

[7] B. Gregg. The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging. *Queue*, 14(2):91–110, mar 2016. ISSN 1542-7730. doi: 10.1145/2927299.2927301.

[8] M. McCool, A. D. Robison, and J. Reinders. Chapter 2 - Background. In *Structured Parallel Programming*, pages 39–75. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-415993-8. doi: 10.1016/B978-0-12-415993-8.00002-5.

[9] M. McCool, A. D. Robison, and J. Reinders. Chapter 11 - K-Means Clustering. In *Structured Parallel Programming*, pages 279–289. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-415993-8. doi: 10.1016/B978-0-12-415993-8.00011-6.

[10] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 187–197, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300193. doi: 10.1145/1806596.1806618.

[11] A. Nisbet, N. M. Nobre, G. Riley, and M. Luján. Profiling and tracing support for java applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 119–126, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362399. doi: 10.1145/3297663.3309677.

[12] openjdk. JDK 17 development - safepoint.cpp. URL https://github.com/openjdk/jdk17/blob/4afbcaf55383ec2f5da53282a1547bac3d099e9d/src/hotspot/share/runtime/safepoint.cpp#L285. [Online; accessed 12-December-2021].

[13] A. Pangin and V. Tsesko. The art of JVM profiling. JPoint International Java Conference, 2017. URL https://2017.jpoint.ru/en/talks/the-art-of-jvm-profiling/.

[14] J. B. Sartor, K. D. Bois, S. Eyerman, and L. Eeckhout. Analyzing the scalability of managed language applications with speedup stacks. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–32, 2017. doi: 10.1109/ISPASS.2017.7975267.

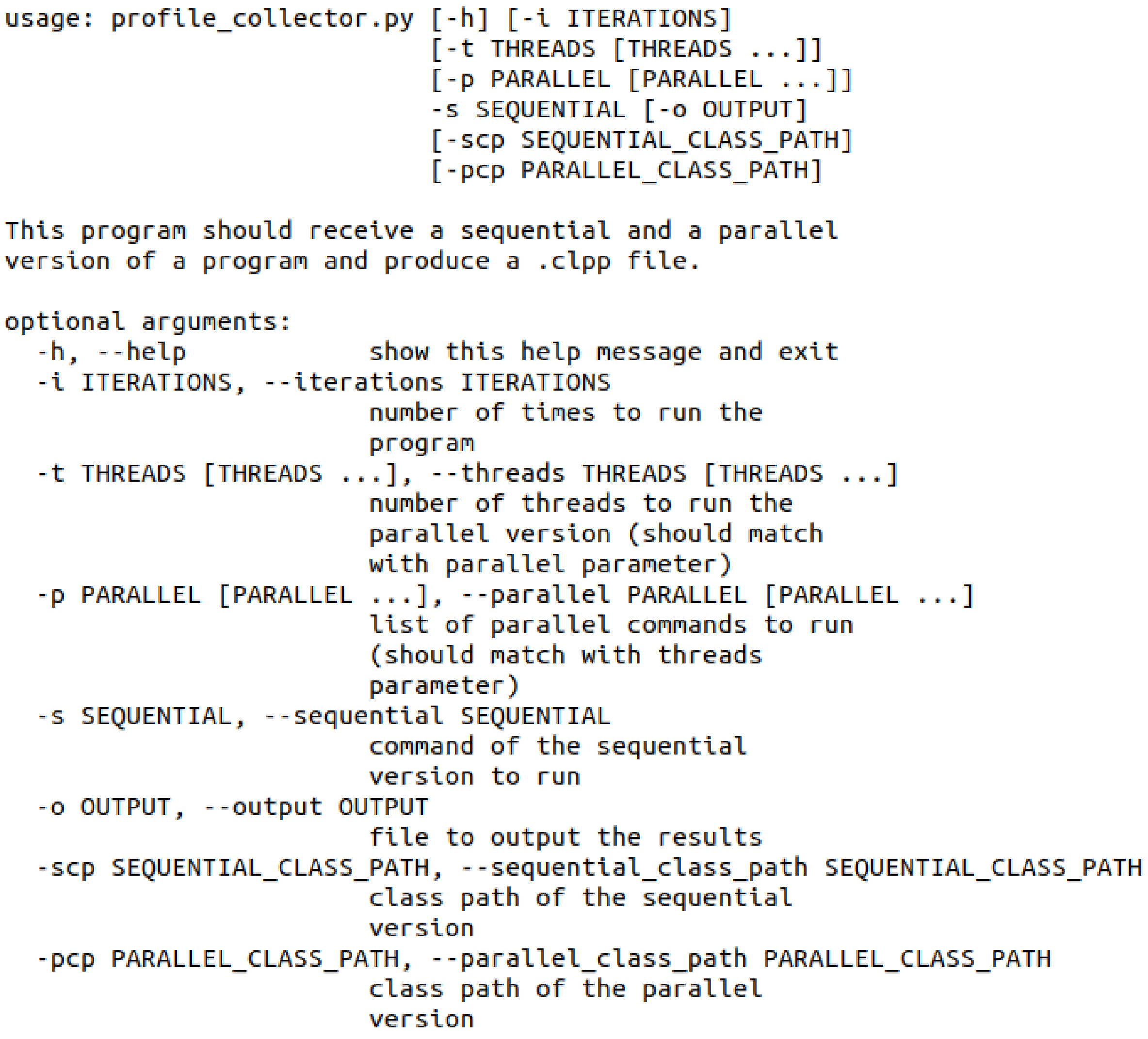
[15] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, page 8, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 158113293X. doi: 10.1145/582034.582042.

Part III

APPENDICES

A

TOOLS



```
usage: profile_collector.py [-h] [-i ITERATIONS]
                            [-t THREADS [THREADS ...]]
                            [-p PARALLEL [PARALLEL ...]]
                            -s SEQUENTIAL [-o OUTPUT]
                            [-scp SEQUENTIAL_CLASS_PATH]
                            [-pcp PARALLEL_CLASS_PATH]

This program should receive a sequential and a parallel
version of a program and produce a .clpp file.

optional arguments:
  -h, --help            show this help message and exit
  -i ITERATIONS, --iterations ITERATIONS
                        number of times to run the
                        program
  -t THREADS [THREADS ...], --threads THREADS [THREADS ...]
                        number of threads to run the
                        parallel version (should match
                        with parallel parameter)
  -p PARALLEL [PARALLEL ...], --parallel PARALLEL [PARALLEL ...]
                        list of parallel commands to run
                        (should match with threads
                        parameter)
  -s SEQUENTIAL, --sequential SEQUENTIAL
                        command of the sequential
                        version to run
  -o OUTPUT, --output OUTPUT
                        file to output the results
  -scp SEQUENTIAL_CLASS_PATH, --sequential_class_path SEQUENTIAL_CLASS_PATH
                        class path of the sequential
                        version
  -pcp PARALLEL_CLASS_PATH, --parallel_class_path PARALLEL_CLASS_PATH
                        class path of the parallel
                        version
```

Figure 46.: Profile collector's usage