Optimization Coaching for Fork/Join Applications on the Java Virtual Machine

Extended abstract

Eduardo Rosales
Università della Svizzera italiana
Lugano, Switzerland
rosale@usi.ch

Walter Binder
Università della Svizzera italiana
Lugano, Switzerland
walter.binder@usi.ch

ABSTRACT

In the multicore era, developers are increasingly writing parallel applications to leverage the available computing resources and to achieve speedups. Still, developing and tuning parallel applications to exploit the hardware resources remains a major challenge. We tackle this issue for fork/join applications running in a single Java Virtual Machine (JVM) on a shared-memory multicore.

Developing fork/join applications is often challenging since failing in balancing the tradeoff between maximizing parallelism and minimizing overheads, along with maximizing locality and minimizing contention, may lead to applications suffering from several performance problems (e.g., excessive object creation/reclaiming, load imbalance, inappropriate synchronization).

In contrast to the manual experimentation commonly required to tune fork/join parallelism on the JVM, we propose coaching developers towards optimizing fork/join applications by automatically diagnosing performance issues on such applications and further suggest concrete code modifications to fix them.

CCS CONCEPTS

• General and reference → Metrics; • Software and its engineering → Software performance;

KEYWORDS

Performance analysis, fork/join applications, optimization coaching, software optimization, Java Virtual Machine

ACM Reference Format:


1 INTRODUCTION

Recent hardware advances have brought shared-memory multicore to the mainstream, increasingly encouraging developers to write parallel applications able to utilize the available computing resources. Still, developing and tuning parallel applications to exploit the hardware resources remains challenging.

We tackle this issue for fork/join applications, i.e., parallel versions of divide-and-conquer algorithms recursively splitting (fork) a work into tasks that are executed in parallel, waiting for them to

complete, and then typically merging (join) results; running in a single Java Virtual Machine (JVM) on a shared-memory multicore.

The Java fork/join framework [27] is the implementation enabling fork/join applications on the JVM. It uses the work-stealing [57] scheduling strategy for parallel programs, using an implementation where each worker thread has a deque (i.e., a double-ended queue) of tasks and processes them one by one until the deque is empty, in which case the thread “steals” by taking out tasks from deque belonging to other active worker threads. Since the release of Java 8, the Java fork/join framework has been supporting the Java library in the java.util.Arrays [37] class on methods performing parallel sorting, in the java.util.streams package [40], which provides a sequence of elements supporting sequential and parallel aggregate operations, and in the java.util.concurrent.CompletableFuture<T> [38] class which extends the functionalities provided by Java’s Future API [39] to develop asynchronous programs in Java. Moreover, the Java fork/join framework supports thread management for other JVM languages [4, 16, 46] and has been increasingly used in supporting fork/join applications based on Actors [28, 42] and MapReduce [5, 20, 53], to mention some.

Despite the features offered by the Java fork/join framework to developers targeting the JVM, writing efficient fork/join applications remains challenging. While design patterns have been proposed [26] to guide developers in declaring fork/join parallelism, there is no unique optimal implementation that best resolves the tradeoff between maximizing parallelism and minimizing overheads, along with maximizing locality and minimizing contention. Failing in balancing such conflicting goals in the development process may lead to fork/join applications suffering from several performance issues (e.g., excessive deque accesses and object creation/reclaiming, load imbalance, inappropriate synchronization) that may overshadow the performance gained by parallel execution.

Inspired by Optimization Coaching [51, 52], a technique focused on automatically providing the developer with information to facilitate both detecting issues and achieving potential optimizations on a program, we propose coaching developers towards optimizing fork/join applications by automatically diagnosing performance issues on such applications and further suggest concrete code refactoring to solve them. To this end, we devise a tool generating recommendations that should only require source-level knowledge (i.e., knowledge of the high-level programming language in which the target application is implemented) to be straightforwardly interpreted and implemented by the developer.

Although several works and tools address the assisted optimization of sequential applications [15, 51, 52], parallelism discovery [14, 17, 19, 25, 45, 49, 50], parallelism profiling [1, 2, 11, 23, 36, 54, 56, 59],
or have analyzed the use of concurrency on the JVM [29–31, 44],
with some even targeting fork/join parallelism [9, 43], we are not
aware of previous works on coaching a developer towards optimiz-
ing a fork/join application running in the JVM.

The remainder of this proposal is organized as follows. In Sec-
tion 2, we discuss our research directions. In Section 3, we analyze
the feasibility of our proposal by reviewing related work.

2 OPTIMIZATION COACHING FOR
FORK/JOIN APPLICATIONS ON THE JVM

We propose coaching developers towards the optimization of fork/
join applications on the JVM. To achieve our goal it is imperative
to address the following research directions.

Diagnosing performance issues. We plan to build a methodo-
yology aimed at diagnosing issues and understanding its impact on
the performance of fork/join applications on the JVM. To this end,
first we plan to define a new model to characterize fork/join tasks.

Second, we plan to determine the entities and metrics worth
to consider to automatically detect performance drawbacks on
fork/join applications. We plan to use static analysis to inspect
the source code in the search of fork/join anti patterns [9, 43].
Complementary, we plan to use dynamic analysis to deal with
polymorphism and reflection, along with collecting performance
metrics enabling diagnosing performance issues noticeable at run-
time (e.g., excessive deque accesses and object creation/reclaiming,
load imbalance, inappropriate synchronization). To this end, we
devis a vertical profiler [18] collecting information from the full
system stack, including metrics at the application level (e.g., tasks
and threads), Java fork/join framework level (e.g., task submission,
the number of already queued tasks), JVM level (e.g., garbage collec-
tions, allocations in Java Heap), operating system level (e.g., CPU
usage, memory load) and hardware level (e.g., reference cycles,
machine instructions, context switches, cache misses).

Third, we plan to profile and characterize all tasks spawned by a
fork/join application to reveal performance drawbacks related to
suboptimal task granularity, a key factor when implementing
fork/join parallelism on the JVM as pointed out by Lea [26]. This
is challenging since recursion, fine-grained parallelism, and sched-
uling affecting the life-cycle of a fork/join task (e.g., the steal
operation), complicate task granularity profiling and may lead to
incorrect measurements when not handled properly. Our profiler
will rely on DiSL [33] to ensure full coverage of the tasks spawned
by a fork/join application, including those in the Java class library.

Finally, the metrics collected can be susceptible to perturbations
caused by the inserted instrumentation code, which may bias the
analysis. Therefore, we plan to lower the perturbation of dynamic
metrics by using efficient profiling data structures avoiding heap
allocations in the target application [32].

In previous studies [47, 48] we analyzed the tradeoff between
maximizing parallelism and minimizing overheads, mainly focusing
on the size of the tasks spawned by task-parallel applications on
the JVM. We introduced an early model to define a task as every
instance of the Java interfaces java.lang.Runnable and java.util.
- concurrent.Callable, focusing on the study of Java thread pools [41].
Furthermore, we presented tgp [48], a task-granularity profiler for
the JVM. Relying on bytecode instrumentation [33] and resorting to
Hardware Performance Counters (HPC) [22], tgp allowed us finding
suboptimal task granularities causing performance issues in the
DaCapo [6] benchmark suite. We plan to extend this work to study
suboptimal task granularity on fork/join applications on the JVM.

Suggesting optimizations. We plan to develop a methodology
for automatically generating (and reporting to the developer) spe-
cific recommendations aimed at fixing detected performance issues
on fork/join applications. Inspired by the works of De Wael et al. [9]
and Pinto et al. [43], we plan to develop a tool able to recognize
fork/join anti-patterns, but it is also of paramount importance in
our aim to automatically match such anti-patterns to concrete rec-
nomendations that a developer can use to avoid them. We plan to
use calling context profiling [3] to pinpoint the developer to appli-
cation code where tasks classified as problematic were created,
submitted, or executed, guiding optimizations by means of sug-
gest the use of fork/join design patterns [26] and concrete code
modifications to enable potential missed parallelization opportuni-
ties. This is challenging, since automatically matching performance
issues with concrete code suggestions to fix them will require the
use of advance data analysis techniques. In a previous study [47]
we used calling contexts to locate tasks previously identified as of
suboptimal granularity in DaCapo, guiding optimizations through
code refactoring that led to noticeable speedups. Based on this work,
we plan to automate and specialize such techniques to specifically
coach developers in optimizing fork/join applications on the JVM.

Validating optimizations. As previously pointed out by De
Wael et al. [9], there is a lack of standard benchmarks suitable to
perform empirical studies focused on fork/join parallelism on the
JVM. Indeed, our early work mentioned above relied on DaCapo
to study suboptimal task granularities mainly in Java thread pools,
however, this suite does not have any fork/join application. In this
direction, we plan to discover workloads exhibiting high diversity
in their task-parallel behavior by plugging tgp to AutoBench [60], a
toolchain combining massive code repository crawling, pluggable
hybrid analyses, and workload characterization techniques to dis-
cover candidate workloads satisfying the needs of domain-specific
benchmarking. By fully integrating tgp and AutoBench, we plan to
discover workloads exhibiting diverse fork/join parallelism, suitable
for validating both aforementioned methodologies.

3 RELATED WORK

This section first introduces related studies on assisted optimization
of applications. Next, we review previous work analyzing the use of
concurrency on the JVM. Then, we focus on parallelism discovers.
Finally, we review profilers for parallel applications.

Assisted optimization of applications. Several studies have
focused on providing the developer with feedback to assist the
optimization of several aspects of an application. Here, we relate
three of them pioneering an approach that is very close to our goal
in providing recommendations which require only source-level
knowledge to be implemented by a developer.

St-Amour et al. [51, 52] first coined the term Optimization Coach-
ing, introducing a technique modifying the compiler’s optimizer to
inform the developer which optimizations it performs, which opti-
izations it misses, and suggesting specific changes to the source
code that could trigger additional optimizations. The goal is reporting to the developer precise recommendations which interpretation do not require expertise about the compiler’s internals and which implementation enables a missed or failed optimization attempted by the compiler. The technique was first prototyped in a work [52] modifying Racket’s [12] simple ahead-of-time byte-compiler which performs basic optimizations on Racket applications. In a second work [51], the authors extended this approach to consider dynamic object oriented just-in-time compiled programming languages, prototyping assisted optimizations in the SpiderMonkey JavaScript [34] engine.

Gong et al. [15] present JITProf, a prototype profiling framework that detects code patterns prohibiting optimizations on JavaScript Just-In-Time (JIT) compilers and reports their occurrences to the developer in the goal of achieving optimizations. JITProf profiles the target application at runtime implementing different strategies to detect seven JIT-unfriendly patterns. The goal is reporting to the developer JIT-unfriendly locations ranked by their relevance on preventing optimizations to facilitate the developer fixing them.

The above works shed light in the goal of mentoring developers in locating performance drawbacks and in providing concrete recommendations to optimize an application. Nonetheless, they notably differ from our goal because the related tools were not designed for diagnosing and further recommending modifications specifically aimed at optimizing parallel applications. Indeed, the techniques proposed focus on feedback generated by specific compiler’s optimizers, falling short in considering other metrics from the full system stack which may enable revealing issues specific to parallel applications, including fork/join applications on the JVM.

Analyses of the use of concurrency on the JVM. Numerous studies have focused on the use of concurrency on the JVM. Here, we highlight two works closely related to our aim.

De Wael et al. [9] analyzed 120 open-source Java projects to study the use of the Java fork/join framework, confirming manually the frequent use of four best coding practices and three recurring anti-patterns that potentially limit parallel performance. Their aim is providing practical conclusions to guide language or framework designers to propose new or improved abstractions that steer programmers toward using the right patterns and away from using the anti-patterns in developing fork/join applications for the JVM.

Pinto et al. [43] study parallelism bottlenecks in fork/join applications on the JVM, with a unique focus on how they interact with underlying system-level features, such as work-stealing scheduling and memory management. They identify six bottlenecks impacting performance and energy efficiency on the AKKA [28] actor framework and conduct an in-depth refactoring on it to improve its core messaging engine. They also present FJDETECTOR, a plugin for the Eclipse IDE [55] prototyping the detection of some bottlenecks identified by the authors by means of code inspection.

Overall, these works bring meaningful insight in methodologies focused on detecting bad coding practices and bottlenecks in real fork/join applications on the JVM. Complementary to such studies, we plan to focus on both performance metrics collected from the full system stack via dynamic analysis to automatically diagnose performance issues noticeable only at runtime and in characterizing all the tasks spawned by the fork/join application to study performance drawbacks related to suboptimal task granularity. Moreover, we specifically devise coaching developers towards optimizing fork/join applications by providing them with code refactoring suggestions which implementation may enable missed parallelization opportunities.

Parallelism discovers. A number of tools has been developed to discover parallelization opportunities. Most notably, tools focusing on parallelism discovery include Kremlin [14], which tracks loops and dependencies to pinpoint sequential parts of a program that can be parallelized. Kismet [25] builds on Kremlin and enhances the prediction of potential speedups after parallelization, given a target machine and runtime system. Similarly, Hammacher et al. [17], Rountev et al. [49], and Dig et al. [10] focus on refactoring sequential legacy Java programs identifying independent computation paths that could have been parallelized and recommend locations to the programmer with the highest potential for parallelization. Although the above works aim at guiding the developer towards modifications enabling parallelization, they only target the optimization of sequential applications.

In addition, several parallelism profilers based on the work-span model [7, 24] target parallel applications. Cilkview [19] builds upon this model to predict achievable speedup bounds for a Cilk [13] application when the number of used cores increases. CilkProf [50] extends Cilkview by measuring work and span on each call site to determine which of them constitutes a bottleneck towards parallelization. TaskProf [58] computes work, span, and the asymptotic parallelism of C++ applications using the Intel Threading Building Blocks (TBB) [45] task-parallel library, relying on causal profiling [8] techniques to estimate parallelism improvements. Despite these tools allow detecting bottlenecks on parallel applications and can estimate potential speedups, they do not focus on coaching a developer towards optimizing a fork/join application. Moreover, none of these tools supports the JVM.

Parallelism profilers. Researchers from industry and academia have developed several tools to analyze various parallel characteristics of an application. HPCToolkit [1] is a suite of tools using statistical sampling of timers and HPC collection to analyze the performance of an application, resource consumption, and inefficiency, attributing them to the full calling context in which they occur. THOR [54] is a tool for performance analysis of Java applications on multicore systems. THOR relies on vertical profiling to collect fine-grained events, such as OS context switches and Java lock contention from multiple layers of the execution stack. Each event is subsequently associated with a thread, providing a detailed graphical report on the behavior of threads spawned in the execution of a parallel Java application.

Finally, a number of tools exist to support the analysis of parallel Java applications through the collection of metrics at the application layer, most notably including VisualVM [56], IBM HealthCenter [21], Oracle Developer Studio Performance Analyzer [36], Java Mission Control [35], JProfiler [11], and YourKit [59], or at the hardware layer resorting to HPCs, including Intel VTune [23] and AMD Code Analyst [2], among others.

Overall, despite the aforementioned tools provide insight in characterizing processes or threads over time, none of them specializes on fork/join applications, and thus they fall short in diagnosing performance problems specific to this type of parallel applications.