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

Eduardo Rosales

Advisor: Prof. Walter Binder
Research area: Parallel applications, performance analysis
PhD stage: Planner
The problem: despite the complexities associated with developing and tuning fork/join applications, *there is little work focused on assisting developers in optimizing such applications on the JVM.*

Relevance: fork/join parallelism has an increasing popularity among developers targeting the JVM. It has been integrated to support parallel processing on the Java library, thread management in JVM languages and a variety of parallel applications based on Actors, MapReduce, etc.

Our proposal: coaching developers towards optimizing fork/join applications by diagnosing performance issues on such applications and further suggest concrete code refactoring to solve them.

Expected outcome: in contrast to the manual experimentation often required to tune fork/join applications on the JVM, we devise a tool able to automatically assist developers in optimizing a fork/join application.
What is a fork/join application?

```java
solve(Problem problem) {
    if (problem is small)
        directly solve problem sequentially
    else {
        recursively split problem into independent parts:
        fork new tasks to solve each part
        join all forked tasks
    }
}
```
The Java Fork/Join Framework

- The Java fork/join framework [1] is the implementation enabling fork/join applications on the JVM
  - It implements the work-stealing [2] scheduling strategy:

![Diagram of the Java Fork/Join Framework]

• The Java fork/join framework [1] is the implementation enabling fork/join applications on the JVM
  • It implements the work-stealing [2] scheduling strategy:

The Java Fork/Join Framework

- Supports parallel processing in the Java library:
  - `java.util.Arrays`
  - `java.util.streams (package)`
  - `java.util.concurrent.CompletableFuture<T>`

- Supports thread management for other JVM languages:
  - Scala
  - Apache Groovy
  - Clojure

- Supports diverse fork/join parallelism, including applications based on *Actors* and *MapReduce*
Many of the design forces encountered when implementing fork/join designs surround task granularity at four levels [3]:

- Maximizing locality
- Maximizing parallelism
- Minimizing contention
- Minimizing overheads

Example of a common performance issues 1/4

- Suboptimal forking
  - Excessive forking

Too fine-grained tasks

Parallelization overheads due to excessive:
  - Deque accesses
  - Object creation/reclaiming
Example of a common performance issues 2/4

- **Suboptimal forking**
  - **Sparse forking**

```
<table>
<thead>
<tr>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE</td>
</tr>
<tr>
<td>CORE</td>
</tr>
<tr>
<td>CORE</td>
</tr>
<tr>
<td>CORE</td>
</tr>
</tbody>
</table>
```

- Low CPU utilization
- Load imbalance

- Missed parallelization opportunities:
  - Few coarse-grained tasks
  - Idle cores
Despite the complexities associated with developing and tuning fork/join applications, there is little work focused on assisting developers towards optimizing such applications on the JVM.

The scope:

Fork/join applications running in a single JVM

A single shared-memory multicore
In contrast to manual experimentation used to tune a fork/join application, we propose an approach based on:
In contrast to manual experimentation often used to tune a fork/join application, we propose an approach based on:

Static and dynamic analysis to automatically diagnose performance issues.
In contrast to manual experimentation often used to tune a fork/join application, we propose an approach based on:

- **Static analysis**: to automatically inspect the source code to detect fork/join anti patterns.
- **Dynamic analysis**: to automatically diagnose performance issues noticeable at runtime (e.g., suboptimal forking, excessive garbage collection, low CPU usage, contention).
In contrast to manual experimentation often used to tune a fork/join application, we propose an approach based on:

**Optimization coaching**\[^4\]: processing the output generated by the compiler’s optimizer to suggest concrete code modifications that may enable the compiler to achieve missed optimizations.

In contrast to manual experimentation often used to tune a fork/join application, we propose an approach based on:

Inspired by Optimization Coaching, the goal is automatically suggesting concrete code modifications to solve the detected issues.
Future Work

- Methodology for the automatic diagnosing of performance issues:
  - Define a model to characterize fork/join tasks
  - Characterize all tasks spawned by a fork/join application
  - Determine the metrics and entities worth to consider to automatically diagnose performance issues

- Methodology for the automatic suggestion of optimizations:
  - Automatic recognition of fork/join anti patterns and matching to concrete suggestions to avoid them

- Validation of the results:
  - Discover fork/join workloads, suitable for validating both aforementioned methodologies
BACKUP SLIDES.
Related Work

- Analysis of parallel applications on the JVM
  - A number of parallelism profilers focus on the JVM [9][10]

<table>
<thead>
<tr>
<th>The goal</th>
<th>Characterizing processes or threads over time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitations</td>
<td>- None of the existing tools targets fork/join applications.</td>
</tr>
</tbody>
</table>


Related Work

- Assisted optimization of applications
  - “Optimization Coaching” was first coined to describe techniques to optimize Racket [4] and JavaScript [5] [6] applications

<table>
<thead>
<tr>
<th>The goal</th>
<th>Report to the developer precise changes in the code that may enable the compiler’s optimizer to achieve missed optimizations.</th>
</tr>
</thead>
</table>
| Limitations | - The techniques were not designed for optimizing parallel applications.  
- The prototyped techniques target only specific compilers. |

Related Work

- Analyses on the use of concurrency on the JVM
  - Documenting fork/join anti patterns on the JVM [7][8]

<table>
<thead>
<tr>
<th>The goal</th>
<th>Identification of common bad practices and bottlenecks on real fork/join applications.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limitations</td>
<td></td>
</tr>
</tbody>
</table>
  - Focus on detecting performance issues by using code inspection (manual code inspection and static analysis).
  - Do not consider the granularity of the tasks spawned by the fork/join application.
  - Do not mentor the developer towards optimizing a fork/join application. |


Challenges

- **Automatic detection of performance issues and suggestion of fixes**
  - Combination of program-analysis and machine-learning techniques to automatically identify performance problems, to pinpoint them in the source code, and to recommend concrete optimizations.

- **Accurately measure granularity of each task**
  - Recursion, fine-grained parallelism, task scheduling, exception handling, auxiliary task, etc.

- **Low perturbation in metric collection**
  - Use of efficient and reduced instrumentation code.
  - Avoid any heap allocation in the target application.
Ongoing Research

- **tgp**: a **Task-Granularity Profiler** for multi-threaded, task-parallel applications executing on the JVM [11]
  - Features as a vertical profiler [12] collecting metrics from the full system stack at runtime to characterize task granularity
  - Shows the impact of task granularity on application and system performance
  - Generates actionable profiles [13]
    - Developers can optimize code portions suggested by *tgp*

We analyzed task granularity in DaCapo [14] and ScalaBench [15].

We revealed fine- and coarse-grained tasks mainly in Java thread pools causing performance drawbacks [11].

We optimized suboptimal task granularity in pmd and lusearch [16].

- Speedups up to 1.53x (pmd) and 1.13x (lusearch)

Example of a common performance issues

- Heavy copying on fork

uggage copying on fork

✗ Parallelization overheads due to:
- Excessive object creation
- High memory load
- Frequent garbage collection
Automatic detection of performance issues

- **Static analysis**: analysis of source code for detecting **fork/join anti-patterns**, including:
  - **Heavy copy on fork** (e.g., detecting the use of methods such as `System.arraycopy`, `sublist`).
  - **Heavy merging on join** (e.g., detecting the use of methods such as `addAll`, `putAll`).
  - **Inappropriate synchronization** (e.g., detecting the use of improper synchronization during task execution, for example to wait for the result of another computation).
  - **The lack of a sequential threshold** (i.e., a threshold which determines whether a task will execute a sequential computation rather than forking parallel child tasks).
Automatic detection of performance issues

- **Dynamic analysis**: analysis of the fork/join application at runtime to deal with polymorphism and reflection along with detecting:
  - **Suboptimal forking** (i.e., the presence of too fine-grained tasks or few too coarse-grained tasks).
    - According to the Java fork/join framework documentation: “*a task should perform more than 100 and less than 10000 basic computational steps*”. [17]
  - **Strategy**: collection and automatic of metrics from the full system stack at runtime:
    - **Framework-level metrics** (e.g., the number of already queued tasks via `getSurplusQueuedTaskCount` method).
    - **JVM-level metrics** (e.g., garbage collections)
    - **OS-level metrics** (e.g., CPU usage, memory load)
    - **Hardware Performance Counters** (e.g., reference cycles, machine instructions)

Example of a common performance issues

- Inappropriate sharing

❌ Parallelization overheads due to significant:
  - Thread synchronization
  - Inter-thread communication
Automatic detection of performance issues

- **Dynamic analysis**: analysis of the fork/join application at runtime to detect:
  - **Inappropriate sharing of resources** (e.g., the use of shared objects, locks, files, data bases and other resources by several parallel tasks).
    - **Strategy**: collection and automatic analysis of performance metrics:
      - **VM-level metrics** (e.g., allocations in Java Heap)
      - **OS-level metrics** (e.g., context switches, cache misses, page faults)
Validation of the results

AutoBench [18], a toolchain combining:
- code repository crawling
- pluggable hybrid analyses, and
- workload characterization techniques

to discover candidate workloads satisfying the needs of domain-specific benchmarking.

### Achievements

#### Publications

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Title</th>
<th>Conference</th>
<th>Location</th>
<th>Year</th>
<th>ISBN</th>
<th>Pages</th>
</tr>
</thead>
</table>
• **Reference cycle**: reference cycle elapses at the nominal frequency of the CPU, even if the actual CPU frequency is scaled up or down.

• **Context switches**: occurs when the kernel switches the processor from one thread to another—for example, when a thread with a higher priority than the running thread becomes ready.

• **Page fault**: is a type of exception raised by computer hardware when a running program accesses a memory page that is not currently mapped by the memory management unit (MMU) into the virtual address space of a process.