Introducing Concurrency in Sequential Java via Laws

Rafael Duarte\(^a\), Alexandre Mota\(^{a, *}\), Augusto Sampaio\(^a\)

\(^a\)Centro de Informática – Universidade Federal de Pernambuco
Caixa Postal 7881 – Recife – PE – Brazil

Abstract
Nowadays multi-core processors can be found everywhere. It is well-known that one way of improving performance is by parallelization. In this paper we propose a parallelization strategy for Java using algebraic laws. We perform an experiment with two benchmarks and show that our strategy produces a gain similar to a specialized parallel version provided by the Java Grande Benchmark (JGB).

Keywords: formal methods, concurrency, parallel processing, performance evaluation, program correctness

1. Introduction
To explore multiple processing capabilities, software must be multi-threaded. A parallelization mechanism can be a solution. The idea is to hide the parallelism from the developer by providing either an automatic process or high-level abstractions. This involves generating a parallel version of a given sequential program, with minor or no human interference, which can be more cost-effective and safe than explicit parallelism, where human interference is intensive.

Java is an interesting language to explore parallelism. Recent surveys show that it is one of the most popular languages today that supports parallel programming with built-in threads, and has already proved its capabilities in different domains, such as web applications and even scientific computing [1].

Works reported in [2, 3, 4, 5, 6] propose ways to explore implicit parallelism in Java. They are carried out either at the bytecode or at the source code level. A common aspect shared among them is the presence of extra code introduced in the application either manually or by runtime tools. They provide good performance results, but they explore the parallelization in a very pragmatic way, without focusing on the correctness of the transformations. A promising approach is the use of algebraic laws, which provide insight on the algebraic
semantics of languages, and are also used as a basis for the definition of reliable refactorings [7].

In this paper we propose a parallelization strategy for Java using algebraic laws that transforms an original sequential program, introducing concurrency wherever possible. The new program can explore a parallel architecture more easily, aiming at improving its original execution performance.

To evaluate our strategy, we perform two real-world case studies extracted from the Java Grande Benchmark suite (JGB) [8]. In both case studies we obtained equivalent average speedups of the parallel version created by hand.

Although we do not provide formal proofs for our laws, each case study used the benchmark validation mechanism to give some confidence that the transformations preserve semantic behavior. Our main contributions reported here are: (i) A novel lightweight and safety focused strategy to Java parallelization; (ii) Experiments assessing its competitiveness.

2. Laws

We present only laws related to parallelization. The entire catalog that supports the strategy can be found in [9]. The laws are written in Java, adopting the convention that a Java system has the form \( \text{cds Main} \), where \( \text{cds} \) is a set of class declarations and \( \text{Main} \) is a class with the only main method present in the system\(^1\).

A condition required by a law is marked with \((\leftrightarrow)\) when it must hold in both transformation directions, or with \((\rightarrow)\) or \((\leftarrow)\), when it must hold only in the indicated direction.

2.1. Parallelization Laws

These laws are responsible to rewrite a sequential into a parallel Java program. They rely on the notion of independent commands, detectable by static (conservative) dependence analysis. This can diminish the cases where independent commands will be detected, but it will also reduce the analysis overhead.

As our focus is not on dependence analysis, we abstract details and simply use the predicate \( \text{indep}(\text{stmt}_1, \ldots, \text{stmt}_k) \) to indicate that these \( k \) statements are independent; that is, they cannot share variables, direct or indirectly (via aliasing), unless the variables are read-only.

Our first law changes the order of commands, grouping related commands to be run in different threads. It requires that the commands be independent. The notation \( \text{cds}, A \triangleright \text{cmd} \) means that \( \text{cmd} \) is a command in class \( A \) and is well-formed in the context of class declaration \( \text{cds}, A \).

**Law 1.** (change command order) If \( \text{cds}, A \triangleright \text{cmds}_1; \text{cmds}_2 \) then \( \text{cds}, A \triangleright \text{cmd}_1; \text{cmd}_2 = \text{cmd}_2; \text{cmd}_1 \)

**provided:** \((\leftrightarrow)\) \( \text{indep} (\text{cmd}_1, \text{cmd}_2) \)

Loops that have independent commands inside their body can be broken into two loops, increasing the chance of executing them in different threads. Loops that fulfill such conditions usually contain a fixed number of iterations.

\(^1\)Internal class declarations are assumed to be in the default package
Law 2. ⟨factorize loop⟩

\[
\begin{aligned}
\text{for} (\text{init}; \text{cond}; \text{incr}) \{ \\
\text{cmds}_1; \\
\text{cmds}_2
\}\end{aligned}
\]

\[
\begin{aligned}
\text{for} (\text{init}; \text{cond}; \text{incr}) \{ \\
\text{cmds}_1 \\
\}
\end{aligned}
\]

\[
\begin{aligned}
\text{for} (\text{init}; \text{cond}; \text{incr}) \{ \\
\text{cmds}_2
\}\end{aligned}
\]

\[
\begin{aligned}
\text{for} (\text{init}; \text{cond}; \text{incr}) \{ \\
\text{cmds}_1
\}
\end{aligned}
\]

\[
\begin{aligned}
\text{for} (\text{init}; \text{cond}; \text{incr}) \{ \\
\text{cmds}_2
\}\end{aligned}
\]

\[\text{provided: } (\rightarrow) \text{ indep(cmds}_1, \text{cmds}_2, \text{cond})\]

Another possibility is when the condition is a conjunction of two other conditions, each one being affected only by a partition of the commands. We can factorize the loop by data refinement, such that each new loop contains the related commands and conditions.

One of the most common cases of parallelization is when a loop presents independent iterations. Each iteration performs some computation that does not affect subsequent loop executions. To capture this condition in the law, we parameterized \text{cmds} with the index that controls the iteration being performed, and formulated their independence as a predicate involving \text{indep(cmds}_i, \text{cmds}_j)\), where \text{cmds}_i and \text{cmds}_j represent the executions of \text{cmds} in the \text{i}th and \text{j}th iterations respectively. In cases like this, the loop itself can be split, with each new loop executing a partition of the iterations from the original one.

Law 3. ⟨split loop iterations⟩

\[
\begin{aligned}
\text{for} (\text{int } i = K; \ i < F; \ i = i + \text{inc}) \\
\text{cmds}_1
\end{aligned}
\]

\[
\begin{aligned}
\text{for} (\text{int } i = K; \ i < F; \ i = i + \text{inc}) \\
\text{cmds}_1
\end{aligned}
\]

\[
\begin{aligned}
\text{for} (\text{int } i = J; \ i < F; \ i = i + \text{inc}) \\
\text{cmds}_1
\end{aligned}
\]

\[
\begin{aligned}
\text{for} (\text{int } i = J; \ i < F; \ i = i + \text{inc}) \\
\text{cmds}_1
\end{aligned}
\]

\[\text{provided: } (\rightarrow) (1) \forall i, j \ | \ K \leq i < F \land K \leq j < F \land i \neq j \bullet \text{indep(cmds}_i, \text{cmds}_j)\)\]

\[\text{Law 4. ⟨fork - join⟩ If } \text{cds, A } \triangleright \text{cmds}_1; \text{cmds}_2 \text{ then } \text{cds, A } \triangleright \]

\[
\begin{aligned}
\text{Thread } t; \\
t = \text{new Thread(} \\
\text{new Runnable() } \\
\{ \\
\text{public void run() } \\
\{ \\
\text{cmds}_1; \\
\}
\}
\}
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
As the method join throws the checked exception InterruptedException, we had to address this exception. We decided to catch and ignore it because this exception is thrown when another thread interrupts the one we are waiting for completion and in our case we control the execution of the thread. Thus we ensure that no other thread will try to interrupt it.

There is also another restriction, particular to Java, that forbids non-final variables to be accessed in methods within inner classes. Adding the final modifier to a local variable prevents it from being assigned more than once. In fact, this limitation affects mostly primitive types, because when working with reference types, we can change the values of their attributes, even if the referencing variable is declared as final.

Nevertheless, if we maintain the sequential execution, we cannot take advantage of multiple processors. Moreover, Law 4, when applied solely, worsens the program performance. To have a simultaneous execution, and improve performance, we can move the method join further, to after a command that is independent to the execution of the thread. In this manner, we can better explore the parallel execution of commands. In the following law, cmds refers to the commands executed by t.

**Law 5.** \(\text{⟨move join⟩} \) If \(cds, A △ t : \text{Thread}, \text{cmds}\) then \(cds, A △ t\)

\[
\begin{align*}
t &. start(); \\
& \text{try} \{t. join()\} \\
& \text{catch}(\text{InterruptedException } e)\{\}
\end{align*}
\]

\[
\begin{align*}
t &. start(); \\
& \text{cmds} \\
& \text{try} \{t. join()\} \\
& \text{catch}(\text{InterruptedException } e)\{\}
\end{align*}
\]

**provided**: \((\leftrightarrow) \text{indep}(\text{cmds}, \text{cmds}_t)\)

Whenever there are independent commands, we can continue moving the method join. If it becomes the last command within the method main, it can be eliminated, since there is no further commands to be executed, and therefore there is no command to wait for t to finish.

**Law 6.** \(\text{⟨eliminate join⟩} \)

\[
\begin{align*}
\text{public static void main}(\text{String[ ] args}) \{ \\
& \text{cmds} \\
& \text{try} \{t. join()\} \\
& \text{catch}(\text{InterruptedException } e)\{\}
\end{align*}
\]

\[
\begin{align*}
& =_{cds, \text{Main}} \\

\text{public static void main}(\text{String[ ] args}) \{ \\
& \text{cmds}
\end{align*}
\]

When the call to the method join is eliminated, it means that t is independent from all subsequent commands. In this situation, t can execute in parallel without the need of any external control.
2.2. Normal Form and Reduction Strategy

Prior to apply the previous laws, we need to transform the original source code into a normal form using a reduction strategy that conforms to the following conditions.

- Each class declaration in \(cds_{\text{internal}}\), except \(\text{Object}\), has no attributes;
- Methods are allowed only in the \(\text{Main}\) class, and they must be static;
- There is a \(\text{main}\) method in the \(\text{Main}\) class;
- The \(\text{Object}\) class may include only attribute declarations, and their types must be a primitive type, or \(\text{Object}\), or defined by an external class;
- All local declarations in the \(\text{main}\) method are declared with a primitive type, or \(\text{Object}\), or an external class;
- No type cast is allowed in the \(\text{main}\) method;
- There are no declarations of constructors.

This normal form can be obtained for an arbitrary Java program by using a reduction strategy detailed on [7, 9]. The major steps of this reduction strategy are.

1. Introduce a new class called \(\text{Object}\) and change the hierarchy of classes, to use \(\text{Object}\) as the new root superclass. After that, every attribute has its visibility changed to public and they are moved from subclass to superclass until they reach the root superclass \(\text{Object}\);
2. Eliminate all constructor declarations and calls, after having their calls inlined, including calls via \text{super} and \text{this};
3. Every method declaration is moved to the root superclass \(\text{Object}\) by successively applying laws to change the visibility of methods to public, and introduce trivial redefinitions. After that, we eliminate calls to super methods, and introduce trivial casts;
4. Finally, every method call is inlined, and the method declaration is eliminated, except for recursive methods.

2.3. Parallelization Strategy

Similarly to Section 2.2, and after the program is reduced to normal form, we need to apply a parallelization strategy to introduce concurrency, whenever possible\(^2\), using the laws of Section 2.1.

Our strategy can be divided into two major steps. The first one aims at reorganizing the source code, making it more amenable to the parallelization. This is performed by Laws 1, 2, and 3. Thereafter, in the second step, Laws 4, 5, and 6 are used to execute independent commands in parallel.

We give an overview of our parallelization strategy using a simple example. To save space we start with the normal form code of Figure 1(a). As stated before, the first step is to reorganize the code, grouping related commands and

\(^2\)Depending on how the input program is coupled, the conditions of the laws might not be satisfied and, therefore, our strategy might not apply.
splitting loops. In this case, there are no loops, but we can reorder the commands using Law 1. This law is applied repeatedly until all related commands are grouped together.

After reordering, we can now apply the laws to execute the commands in parallel. In the method `eval`, there are two blocks of independent commands, each one of them responsible for evaluating one side of the expression. Furthermore, they are followed by a command that uses their results. This scenario indicates Law 4. But for its conditions to hold, we have to add the `final` modifier to the local variables `lint`, `rint`, `le`, and `re` (laws in [9]). We then use Law 5 to move down the call to `join` until before the command that depends on the thread.

Now, the evaluation of each side of the expression is carried out by an independent thread, and the sequential execution continues until the result of their computation is needed. At this point, the method `join`, called on each thread, assures that the execution will resume only after they have finished. It is important to highlight that this example is intended to illustrate the strategy. In this situation, parallelizing this code would hardly deliver some speedup, due to the light computations performed in parallel. The final result is presented in Figure 1(b).

From Figure 1(b) we can see that our strategy does not change the structure of the source code. Instead, it changes how commands within method bodies are executed. Note that in Figure 1(a) we have a complete sequential code whereas in Figure 1(b) we have one with two threads.
3. Case Study

To perform our case studies, we obtained the original sequential code from the selected benchmarks. We isolated the code corresponding to the benchmark prior to applying our strategy. We have measured execution times for the three different input sizes provided for each benchmark (A, B, and C).

After finishing the code transformations, we compare each one of them with respect to their execution time and behavior preservation. The execution time of each benchmark is obtained by a timer provided by the JGB (Java grande benchmark). To calculate the execution time data, we execute each code ten times, calculate their average (arithmetic mean), and use this value in the comparison. The behavior preservation is also obtained by JGB using a validation mechanism based on testing.

We used an Intel Core i7 2.67 GHz Desktop, with 8GB of RAM, Ubuntu Linux (kernel 2.6.32-25), and Sun’s JDK version 1.6.0_22. We setup the same execution conditions on each test, to avoid external threats to validity.

Fourier Series. Our first benchmark computes the first 10,000 Fourier coefficients of the function $f(x) = (x + 1)^2$ on the interval 0-2. Prior to apply our strategy, we performed a few minor adaptations on the original classes: removing all package and import clauses, and eliminating one implements clause (without any effect to our code). We also considered two utility classes (JGFInstrumentor, JGFTimer) as external code (only compiled code).

The results displayed in Table 1 show a considerable speedup over the original sequential version, slightly better than the JGB’s parallel version. This was quite a surprise, since manually written parallel code tends to perform better. Moreover, the code in the normal form had a slightly better performance than the original one, providing some evidence that the normal form reduction strategy does not incur in execution overheads.

IDEA Encryption. This benchmark performs IDEA [10] (International Data Encryption Algorithm) encryption and decryption algorithm on an array of randomly generated three million bytes. The structure of its code is similar to the Fourier benchmark, but some implementation details hindered the use of our strategy. In addition to removing packages and implementation clauses as done in the previous case study, we had to perform a minor change in the algorithm, moving an external variable into a loop. Except for this, all transformations were covered by our normalization and parallelization strategies.

In this case, our approach provided equivalent speedups to JGB’s parallel version. We just noticed an unexpected behavior when executing the A size benchmark. In this case, the speedup was below the expected and the 4 threads version did not present a significant speedup over its 2 threaded counterpart. We believe it happened due to the very small input size, which was not enough to compensate thread creation overheads. This issue did not happen in JGB’s version because thread creation was manually fine tuned in their code. Concerning the normal form version, it performed similarly to the previous case study.
<table>
<thead>
<tr>
<th>Fourier Series</th>
<th>IDEA Encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>PV2</td>
<td>PV4</td>
</tr>
<tr>
<td>A 98.2%</td>
<td>A 33.9%</td>
</tr>
<tr>
<td>B 99%</td>
<td>B 97%</td>
</tr>
<tr>
<td>C 61.6%</td>
<td>C 98.8%</td>
</tr>
<tr>
<td>PV4</td>
<td>JGB2</td>
</tr>
<tr>
<td>A 277%</td>
<td>A 38.6%</td>
</tr>
<tr>
<td>B 262.7%</td>
<td>B 232.1%</td>
</tr>
<tr>
<td>C 206.5%</td>
<td>C 266.7%</td>
</tr>
<tr>
<td>JGB2</td>
<td>JGB4</td>
</tr>
<tr>
<td>A 97.5%</td>
<td>A 83.8%</td>
</tr>
<tr>
<td>B 99.2%</td>
<td>B 87.2%</td>
</tr>
<tr>
<td>C 61.2%</td>
<td>C 95.8%</td>
</tr>
<tr>
<td>JGB4</td>
<td>JGB4</td>
</tr>
<tr>
<td>A 269.1%</td>
<td>A 144.9%</td>
</tr>
<tr>
<td>B 293.1%</td>
<td>B 193.4%</td>
</tr>
<tr>
<td>C 162.8%</td>
<td>C 250.3%</td>
</tr>
</tbody>
</table>

Table 1: Case Study speedups for each input size (A, B, and C) and parallel version (PV: proposed version with 2 and 4 threads, JGB: java grande parallel version with 2 and 4 threads).

4. Related Work

There is some work on parallelizing Java programs, such as [11, 12, 5, 4, 3, 2]. The javar restructuring compiler [11] implements source to source transformations intended to parallelize java code. In order to detect which parts of the code will be parallelized, it relies on user annotations. In [12], the authors focus on automatic parallelization of loops, obtaining significant speedups over the original code. The parallelization is automatically performed in compilation time, by a specific compiler developed by the authors.

Java programs are parallelized in the bytecode level in [5]. This approach has very good results, as well as it can be applied even when there is no source code available. Its major drawback, however, is that it uses a processor specific optimization, restricting its practical use. The approaches presented in [4, 3] parallelize method calls, using annotations in the code and run-time engines. They present interesting results, but the granularity of parallelization is restricted to method calls, as well as they require the use of an extra framework and/or a runtime engine to work properly. A new technique, based on traces, is used in [2] to analyze a Java program, detect dependencies, and parallelize it. Their approach is automatic, and relies on a modified virtual machine to perform the analyses and transformations devised by them.

5. Concluding Remarks and Future Work

In this work we present how a subset (representative in practice) of sequential Java programs can be transformed into a parallel one by means of algebraic laws. The work presented in [13] served as inspiration to ours. Our work is shown (via a case study) to obtain performance gains by exploiting multi-core processors.

Our normal form reduction strategy is used as the starting point of a parallelization strategy. This coding structure eases the work with fine grained parallelization at command level. Also it seems that working with code in a controlled format facilitates automation and dependence analysis, although we have not experienced with it yet.

Our work stands as a first step towards a safe parallelization framework. In the current stage of our approach, laws still rely on human insights to be applied correctly and effectively. One of the goals of this formulation of laws is to facilitate their implementation on popular development environments that support program transformations, such as refactorings.
As future work we intend to create new laws for Java to deal with exceptions, interfaces, and generics as well as general open systems. Another trend concerns correctness by means of tool support and formal proofs; the major challenge is to find a suitable formal semantics for Java. As our parallelization laws are very simple (we must apply at least two laws in sequence), we could then define parallelization rules (compositions of laws).

Another point that requires more study is the heuristics used to define the level of parallelization. We have adopted heuristics that presented good results, but we believe they can be refined to deliver even better speedups.

References