Guava vs. Java4P: A Comparison of Two Approaches to Provide a Better Java Thread Synchronization Model

Roan Dawkins

Abstract

Unlike most other programming languages, Java has included support for multithreading in the Java Language Specification (JLS) since its inception in 1995 [4]. However, Java’s multithreading model is directly based on Hoare’s monitor concept [9] and uses low-level constructs such as synchronized code blocks, and library routines such as the wait(), notify(), and notifyAll() methods from class Thread. This approach is particularly error-prone as these constructs are easily abused or misunderstood by a programmer without sufficient multi-threading knowledge [4, 8].

Recent work in the field has been based on using one of the following techniques to overcome the shortcomings in Java’s multithreading model: 1) extensions to the monitor concept, 2) message passing, and 3) assertions and preconditions. Guava [1] and Java4P [3] (based on techniques 1, and 3 respectively) are two of several approaches proposed recently to provide a higher-level, less error-prone Java synchronization model. This paper evaluates these approaches using 1) applicability – is the proposed solution capable of solving common synchronization scenarios, 2) invasiveness – to what degree does the approach intrude on new and existing programs, and 3) flexibility – to what granularity is concurrency allowed, in an effort to determine if either approach is better from an application programmer’s perspective.

1 Introduction

Java’s current synchronization mechanism is adequate for its original purpose: windows programming and applets. However, Java is moving more and more into other application domains such as the safety- and mission-critical application domains as evidenced by the adoption of the Real-Time Specification for Java (RTSJ) [4]. Examples of applications in these domains include enterprise information systems, distributed programs, etc. [2]. In addition to being critical, these classes of applications are usually complex, and it is easy to make a programming error in a complex application when using low-level synchronization constructs such as those provided by Java [2].

As Java continues to evolve and as more demand is placed on the language from industry and from the scientific community to provide a better multithreaded mechanism, there appears to be a need for the language to support a higher-level abstraction for concurrency synchronization [2]. The aim of this research is to compare two current, state of the art techniques which have been proposed (Guava and Java4P), to provide a higher-level, less error-prone synchronization model for the Java language.

The approaches will be evaluated based on the following criteria: 1) applicability – can the approach be applied to solve common synchronization problems? 2) invasiveness – to what degree are new and existing programs affected by the approach? and 3) flexibility – to what granularity can the concurrency in programs be specified? The intent is to determine whether one technique or approach is significantly different from the other from an application programmer’s perspective. That is, does it allow the concurrent programmer to essentially concentrate on the design decisions that need to be made and the task at hand instead of on the particular synchronization implementation being used?

If and when the decision is made to extend Java’s multithreading model, the results of this research may help in deciding which general technique (extensions to the monitor concept or assertions) is most applicable to the Java programming language. This decision may in turn affect the direction of future work in the field.

The rest of the paper is organized as follows: Section 2 provides background on the approaches to be evaluated, Section 3 contains a description of the methodology used, an analysis of the results obtained is contained in Section 4, Section 5 presents the conclusions derived and Section 6 contains a discussion on future work.

2 Background

Below I present some background and a general overview of the two approaches involved in this research.

2.1 Guava

Guava [1] is a dialect of Java which aims to provide a better thread synchronization model for the language. Guava’s approach is to take Java’s existing monitor concept and extend it to guarantee that access to shared data can be achieved only through synchronized methods. Guava does this by differentiating between three categories of classes. These are 1) monitors, which may be shared but can only be accessed serially, 2) values, which are never shared, and 3) objects, which may be shared only within a single thread i.e. a single thread may have multiple references to it. So in essence, Guava disallows unsynchronized access to shared data completely [1].

Guava extends the single lock per object concept in Java by providing two locks (a read lock and a write lock) for each object, or rather monitor if we are speaking in Guava terms. Multiple threads may obtain the read lock concurrently but only one thread may hold the write lock at any given time. In addition, if a read lock is held, the write lock is not available and if the write lock is held, no thread may obtain the read lock. Every Guava method either requires a read lock or a write lock thus eliminating unsynchronized access [1].

2.2 Java4P

Java4P [3] is an extension of the Java language which aims to provide a simpler concurrency model. To achieve this, Java4P uses preconditions (or assertions) to specify the concurrent portions of programs. This allows the synchronization specification to be separate from the
functional specification of the program. In Java4P, concurrency is not intertwined with thread objects. Instead, threads are created and spawned on the fly asynchronously, as part of method invocation by using the send() command [3].

Every method in Java4P has one of two guards attached to it: an access guard, which is meant to control access to shared objects, or a boolean guard which is a condition that must evaluate to true in order for method execution to begin. Access guards can be divided further into exclusive, for exclusive access (similar to Guava’s write lock), observe, for shared access (similar to Guava’s read lock), and atomic access, which means that the object is available so long as it is not being accessed by the current thread. Note that if a method has more than one guard associated with it, then all resources must be available and/or all conditions must evaluate to true in order for method execution to proceed. Guards are evaluated as an atomic operation. Therefore, if a single guard does not evaluate successfully then all previously acquired resources (for this method execution) are released and the thread must wait until such time that it may attempt to acquire the resources again [3].

3 Methodology

This section describes the methodology which was used to conduct the research.

3.1 Measures

The focus of this study is on ease of programming from the perspective of the concurrent programmer and as such, the criteria used to evaluate both approaches are those of primary interest to the concurrent programmer. These are applicability, invasiveness, and flexibility.

When comparing approaches to a better Java synchronization model, recent work in the field [1, 2, 6] has focused, for the most part, on a performance-based comparison. There are at least two reasons for this: 1) performance-based comparison is strictly quantitative and primarily objective in nature, and 2) performance-based comparison forms the basis for a common ground (since all approaches must exhibit some performance characteristic) by which all approaches can be measured. Though the performance metrics of any approach are important to the concurrent programmer, the concurrent programmer should focus on the design decisions that need to be made i.e. what is the relationship between the multiple threads in the application [5] and is the approach applicable to the problem I am trying to solve? It is for this reason that I have chosen applicability as a criterion for evaluating both approaches.

The invasiveness of an approach is another important measure to the concurrent programmer. Invasiveness refers to the degree to which a class must be changed in order to support an approach. Therefore, with all else being equal, a low invasive approach is more preferred than a highly invasive one. The importance of a low invasive approach is evidenced by Katrib et.al.’s [7] attempt to ensure that their thread synchronization solution is as non-invasive as possible.

Flexibility refers to the degree of granularity with which concurrency can be specified in applications. There are three granularity levels to which multi-threading can be applied in object-oriented programs:

- Statement – Any individual statement or block of statements can be synchronized against concurrent execution.
- Method – Synchronization is permitted on a per-method basis. Therefore the entire method is synchronized against concurrent execution.
- Object – An entire object is synchronized against concurrent execution. Portions of code which access a synchronized object may not execute concurrently even if no interference exists among them.

The granularity levels to which multi-threading can be applied are not mutually exclusive with respect to one another. Indeed, an object-oriented programming language may provide multi-threading support within the language at more than one granularity level or it may not support multi-threading at all. The lower the granularity level the more flexible the approach and the higher the granularity level the less flexible the approach. The JLS has support for object granularity. However, through the use of library functions the concurrent programmer may achieve statement level granularity. The lack of fine-grained concurrency in Java serves as a limitation to the concurrent programmer [3]. In addition, the importance of the flexibility with respect to level of granularity can be seen by Bacon et.al. [1] and Nugroho et.al.’s [3] attempt to ensure that their solution to Java’s thread synchronization problem offers finer granularity concurrency. In the next section I shall demonstrate how each measure is achieved.

3.1.1 Applicability

Applicability refers to whether or not the proposed solution is capable of solving a particular synchronization problem. To obtain an applicability measure, the constructs of Guava and Java4P were applied to four classes that represent common synchronization scenarios. These are:

- Mutual exclusion – In this scenario only one thread may access a resource at any given time. There are three variations of this. With no wait queue, if a thread has locked a resource, any other thread requiring access to that resource will go into a spin loop, repeatedly attempting to acquire the resource until successful. With a (first-in first-out) FIFO wait queue, the thread will block itself and be placed in a FIFO queue awaiting control of the resource. Typically, the thread owning the resource will signal when the resource is no longer needed. This will cause the next thread on the queue to wake up and attempt to gain control of the resource. With an unordered wait queue, the next thread to acquire control of the resource may not necessarily be the next one in line. Typically, the next thread will be the one with the highest dispatching priority. For the purposes of this research, no distinction will be made between what variation of mutual exclusion is actually supported by the approach.
- Multiple readers/single writer – Instead of purely mutual exclusion access, the resource may be simultaneously accessed by multiple threads. The threads which may access the resource simultaneously are called readers because they are not expected to change the state of the resource being controlled. On the other hand writer threads exist that may change the state of the resource and so their access must be mutually exclusive. This scenario is typical in high-end systems where performance is critical and most threads accessing a resource are not expected to change it.
- Dining philosophers – In this scenario, multiple resources are required at the same time and in a certain...
order. This scenario is typical when access to more than one resource is required and the problem domain is simple enough that all required resources can be obtained at the same time. By adhering to a particular order, the threads have employed a deadlock avoidance mechanism i.e. there can never be deadlock. An important note about this class of problem is that either all resources are obtained or none are obtained i.e. if a thread cannot successfully acquire all the requested resources, any resource successfully obtained is released. The thread may then spin on the resources or place itself on a wait queue to be notified when to attempt acquiring the resources again.

- Multiple resources but not all obtained at the same time – This is probably one of the most complex classes of problems facing the concurrent programmer, especially those in complicated programming domains. At the root of this class of problem is that the order in which the resources will be obtained is not predetermined (and it cannot be) as in the previous class. This problem class is the probably at the root of more deadlocks than any other class. Since deadlock avoidance is impossible (as the order is unknown), the focus is usually on deadlock detection.

Each approach was given either an “Applicable” or a “Not Applicable” rating for each common synchronization scenario. For instance, if the constructs of Guava or Java4P allow me to code a class to a specification such as that outlined in class 1, then that approach received an applicability measure of “Applicable” for class 1. If not, then the rating would be “Not Applicable”. Note that this measure does not take into account the degree of difficulty involved in applying the approach to the class. This complexity is taken into account by another measure, invasiveness, which we shall discuss in the next section.

### 3.1.2 Invasiveness

Invasiveness is defined as the degree to which new and existing programs are affected by the approach. Each approach evaluated was given two invasiveness ratings for each class. The first rating is for a class without any prior Java synchronization code and the second is for a class which already contains Java synchronization code. In either case, the invasiveness rating is based on the following:

- If the number of lines added or changed for an approach in the process of applying it to a common class of synchronization problem is between zero and five, then the approach receives a rating of “Excellent” for that class.
- If the number of lines added or changed for an approach in the process of applying it to a common class of synchronization problem is between six and ten, then the approach receives a rating of “Average” for that class.
- If the number of lines added or changed for an approach in the process of applying it to a common class of synchronization problem is eleven or greater, then the approach receives a rating of “Poor” for that class.
- If the approach was not applicable to that class of problem then the approach receives a rating of “Not Applicable”.

The ranges are not scientific and are meant to give a general indication of the degree of difficulty involved.

### 3.1.3 Flexibility

Flexibility is defined as the granularity with which concurrency can be specified in programs. Recall from section 3.1 that granularity could be at the statement level, method level, or object level. Most programming languages either do not contain language support for concurrency when first developed or language concurrency is added to the language at a later date [4]. Java on the other hand has included support for concurrency since it was first introduced [4]. However, the granularity of such concurrency is at the object level. Other object-oriented programming languages, such as Ada, include support for method level concurrency [4] and recent work in the field [1, 3] has been focused on providing a more granular concurrency level for Java than that specified in the JLS.

In order to give Guava and Java4P a flexibility rating, the constructs of each approach were analyzed in order to determine the supported level of concurrency granularity. Depending on the level(s) supported, each approach was given one or more of the following ratings:

- "Statement" for statement-level concurrency,
- "Method" for method-level concurrency, or
- "Object" for object-level concurrency.

A lower granularity rating (statement) is often more desirable than a higher one (object) but this is not always the case.

### 3.2 Operational Procedure

This section describes, step-by-step, the procedure which was followed in taking the above measurements.

#### 3.2.1 Applicability

Four sample classes were created, each exhibiting one of the characteristics of the four common synchronization scenarios outlined in section 3.1. The constructs of both Guava and Java4P were then applied to each class in an effort to determine if the language constructs proposed by these approaches could be applied to common synchronization scenarios. Guava and Java4P were each given an applicability rating based on the criteria set forth in section 3.1.

#### 3.2.2 Invasiveness

The invasiveness rating for code without prior Java synchronization was taken by applying Guava and Java4P to each scenario and the count of the number of lines added per approach per class was taken. To obtain the invasiveness rating for code with prior Java synchronization, each scenario was implemented using synchronization constructs from the Java language. The Java language constructs were then replaced with the constructs of each approach and a count of the number of lines of code added, changed, or deleted was taken to obtain an invasiveness measure for existing code with prior Java synchronization.

#### 3.2.3 Flexibility

The language constructs of Guava and Java4P were analyzed and a measure of the flexibility was taken in accordance with section 3.1.3. Specifically, the granularity with which concurrency can be specified in the scenarios was measured.
4 Results
In this section I present the results obtained in addition to an analysis of those results.

4.1 Data

Table 1: Applicability measure

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Class 1</th>
<th>Class 2</th>
<th>Class 3</th>
<th>Class 4</th>
</tr>
</thead>
</table>

Table 1 shows the applicability measure for Guava and Java4P. The only difference between Guava and Java4P is in the class 3 scenarios (multiple resources obtained at once in order) for which Guava has no direct support.

Table 2a: Invasiveness measure for code without prior Java synchronization

<table>
<thead>
<tr>
<th>Invasiveness w/o</th>
<th>Class 1</th>
<th>Class 2</th>
<th>Class 3</th>
<th>Class 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guava</td>
<td>1: Excellent</td>
<td>1: Excellent</td>
<td>Not Applicable</td>
<td>1: Excellent</td>
</tr>
<tr>
<td>Java4P</td>
<td>2: Excellent</td>
<td>2: Excellent</td>
<td>1: Excellent</td>
<td>1: Excellent</td>
</tr>
</tbody>
</table>

Table 2a shows the invasiveness measure for Guava and Java4P for code without prior Java synchronization constructs. Recall from our applicability measure that Guava is not applicable to class 3 scenarios. The number preceding the colon ("=") represents the number of lines of code added, changed, or deleted and following the colon is the rating given based on the number of lines of code added, changed, or deleted. Except for class 3 scenarios for which Guava does not apply, both approaches can be considered equal.

Table 2b: Invasiveness measure for code with prior Java synchronization

<table>
<thead>
<tr>
<th>Invasiveness w/i</th>
<th>Class 1</th>
<th>Class 2</th>
<th>Class 3</th>
<th>Class 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guava</td>
<td>1: Excellent</td>
<td>14: Poor</td>
<td>Not Applicable</td>
<td>1: Excellent</td>
</tr>
<tr>
<td>Java4P</td>
<td>3: Excellent</td>
<td>15: Poor</td>
<td>1: Excellent</td>
<td>2: Excellent</td>
</tr>
</tbody>
</table>

Table 2b shows the invasiveness measure for code with prior Java synchronization constructs. Of interest here are the class 2 (multiple readers, single writer) ratings for both approaches. Of the fourteen lines of code which were added, deleted, or changed for Guava, thirteen were deletes and only one was an addition. This is due to the fact that using Java’s low-level synchronization constructs, it took thirteen lines of code to implement a simple multiple reader parallelism scenario. Only one line of Guava code is required to achieve the same functionality since multiple reader parallelism is the default in Guava. Using Java4P’s synchronization constructs, thirteen lines of Java synchronization code were replaced by two, hence the fifteen total lines of code added, changed, or deleted.

Table 3: Guava's flexibility measure

<table>
<thead>
<tr>
<th>Flexibility</th>
<th>Statement</th>
<th>Method</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guava</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Java4P</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3 illustrates the granularity measure for both Guava and Java4P. Note that neither approach provides statement-level concurrency granularity. In fact, some have argued that providing concurrency at that fine a granularity is at best dangerous [4] as it allows the concurrent portion of the program to be interleaved with the functional portions of the program [3]. Even though Java only supports object-level granularity, it allows programmers to embed synchronized blocks of code anywhere in their programs, even around a single statement, as long as they provide the object to be used. This has long been a drawback to Java’s synchronization model [4].

4.2 Analysis
In addition to the raw data illustrated above, we can summarize the ratings by class to provide a head-to-head illustration for each approach.

Table 4: Analysis of class 1 scenarios

<table>
<thead>
<tr>
<th>Class 1</th>
<th>Guava</th>
<th>Java4P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicability</td>
<td>Applicable</td>
<td>Applicable</td>
</tr>
<tr>
<td>Invasiveness w/o</td>
<td>1: Excellent</td>
<td>2: Excellent</td>
</tr>
<tr>
<td>Invasiveness w/i</td>
<td>1: Excellent</td>
<td>3: Excellent</td>
</tr>
</tbody>
</table>

It is evident from table 4 that both Guava and Java4P are similar with respect to class 1 scenarios (mutual exclusion). Though Java4P requires slightly more lines of code to implement the class, this is not large enough to cause a significant difference. It is not clear what variation of mutual exclusion is supported by each approach. Both [1] and [3] suggest that some form of wait queue is used to control threads requiring access to a resource which is not currently available, however, it is not clear whether the queue is a FIFO or unordered queue.

Table 5: Analysis of class 2 scenarios

<table>
<thead>
<tr>
<th>Class 2</th>
<th>Guava</th>
<th>Java4P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicability</td>
<td>Applicable</td>
<td>Applicable</td>
</tr>
<tr>
<td>Invasiveness w/o</td>
<td>1: Excellent</td>
<td>2: Excellent</td>
</tr>
<tr>
<td>Invasiveness w/i</td>
<td>14: Poor</td>
<td>15: Poor</td>
</tr>
</tbody>
</table>

Table 5 shows the ratings for class 2 scenarios (multiple readers/single writer). Again there are no significant differences between both approaches according to the measurement scheme for invasiveness.

Table 6: Analysis of class 3 scenarios

<table>
<thead>
<tr>
<th>Class 3</th>
<th>Guava</th>
<th>Java4P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicability</td>
<td>Not Applicable</td>
<td>Applicable</td>
</tr>
<tr>
<td>Invasiveness w/o</td>
<td>Not Applicable</td>
<td>1: Excellent</td>
</tr>
<tr>
<td>Invasiveness w/i</td>
<td>Not Applicable</td>
<td>1: Excellent</td>
</tr>
</tbody>
</table>

Table 6 shows the ratings for class 3 scenarios (multiple resources obtained at the same time in order).
This is the only scenario for which the results are significantly different. Whereas Guava does not provide support for this scenario, Java4P does and it can all be achieved in a single line of code.

Table 7: Analysis of class 4 scenarios

<table>
<thead>
<tr>
<th>Class</th>
<th>Guava</th>
<th>Java4P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicability</td>
<td>Applicable</td>
<td>Applicable</td>
</tr>
<tr>
<td>Invasiveness w/o</td>
<td>1: Excellent</td>
<td>1: Excellent</td>
</tr>
<tr>
<td>Invasiveness w/i</td>
<td>1: Excellent</td>
<td>2: Excellent</td>
</tr>
</tbody>
</table>

As with class 1 and 3 scenarios, there is no significant difference between both approaches for class 4 scenarios. This is evident from table 7.

5 Conclusion

Based on the results obtained, I conclude that Java4P provides a better model for Java synchronization than does Guava from a concurrent programmer’s perspective. However, the only criterion for which Java4P outranked Guava was in class 3 scenarios (multiple resources obtained in order at the same time) for which Guava provides no support. One may argue that class 3 scenarios are not very practical and possibly quite rare in a real-life programming environment. Nevertheless, there are situations in which class 3 scenarios are important. One such situation is deadlock prevention and one way to prevent deadlock between two or more threads is to specify an order in which multiple resources must be obtained [1] and to have all threads adhere to this order.

6 Future Work

One obvious area for future investigation is to investigate the class 1 scenarios in more detail by breaking up these scenarios into the three variations and measuring the applicability and invasiveness of Guava and Java4P. The actual variation of class 1 scenarios that is supported may be important to the concurrent programmer, especially those concerned with writing high-performance code. The work presented in this paper only performed a comparison of two approaches implementing two (assertions and extensions to the monitor concept) of the three main techniques for a better Java thread synchronization model. Therefore another area of study would be to expand this work to include an approach implementing the third main technique, message-passing, in the comparison. Of even more value would be a comparison of the general techniques themselves i.e. a comparison among assertions, extensions to the monitor concept, and message-passing. To gain more insight into the application programmer’s perspective, one could have actual programmers use each approach on either real or pseudo-real programming projects and rank or measure their experience based on some set of criteria.

References


20th Computer Science Seminar
SC3-T1-5