Fork/Join Parallelism in the Wild

Documenting Patterns and Anti-Patterns in Java Programs using the Fork/Join Framework

Mattias De Wael
Software Languages Lab
Vrije Universiteit Brussel, Belgium
madewael@vub.ac.be

Stefan Marr
RMoD, INRIA Lille – Nord Europe
France
stefan.marr@inria.fr

Tom Van Cutsem
Software Languages Lab
Vrije Universiteit Brussel, Belgium
tvcutsem@vub.ac.be

Abstract

Now that multicore processors are commonplace, developing parallel software has escaped the confines of high-performance computing and enters the mainstream. The Fork/Join framework, for instance, is part of the standard Java platform since version 7. Fork/Join is a high-level parallel programming model advocated to make parallelizing recursive divide-and-conquer algorithms particularly easy. While, in theory, Fork/Join is a simple and effective technique to expose parallelism in applications, it has not been investigated before whether and how the technique is applied in practice. We therefore performed an empirical study on a corpus of 120 open source Java projects that use the framework for roughly 362 different tasks.

On the one hand, we confirm the frequent use of four best-practice patterns (*Sequential Cutoff*, *Linked Subtasks*, *Leaf Tasks*, and avoiding unnecessary forking) in actual projects. On the other hand, we also discovered three recurring anti-patterns that potentially limit parallel performance: sub-optimal use of Java collections when splitting tasks into subtasks as well as when merging the results of subtasks, and finally the inappropriate sharing of resources between tasks. We document these anti-patterns and study their impact on performance.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Parallel programming; D.3.3 [*Programming Languages*]: Patterns; D.3.3 [*Programming Languages*]: Concurrent programming structures

General Terms Languages, Performance

Keywords Java, Fork/Join, empirical study, open source projects, patterns, anti-patterns

1. Introduction

Since roughly 2004, multicore processors have become commonplace on commodity machines such as desktops, laptops, and even mobile devices. To use such platforms to their full potential, programmers must turn to parallelism. This has caused parallel programming to outgrow its original niche of High Performance Computing (HPC). Today, parallel programming is no longer the territory of a handful of experts. As a testament to this, libraries and frameworks for parallel programming are now commonplace, even on so-called “managed runtimes”, such as the JVM and the CLR.

In general, writing parallel software remains notoriously difficult [20]. However, good abstractions exist for certain classes of applications. One such class is that of recursive divide-and-conquer algorithms, which can be parallelized using Fork/Join.

The use of Fork/Join as an efficient means of parallelizing divide-and-conquer algorithms was pioneered in MIT Cilk [3, 4, 8], which combined the Fork/Join programming model with an efficient so-called *work stealing* scheduler. Cilk has inspired modern parallel programming libraries for various mainstream languages, including Intel Cilk Plus and Intel Threading Building Blocks for C/C++, the Task Parallel Library for `.NET`, and the Java Fork/Join framework.

The efficiency of scheduling Fork/Join tasks with work stealing has been investigated from a theoretical [3] as well as from a practical [12] perspective. However, the actual use of the Fork/Join model has, to the best of our knowledge, never been studied before. In this paper, we study the use of the Java Fork/Join framework on a corpus of 120 open source Java projects.

Our goal is to obtain insight into how the Fork/Join model is applied in practice, primarily by identifying best-practice patterns and anti-patterns. These observations can then in turn 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.

Contributions The contributions of this paper are:

- A qualitative analysis of a corpus of 120 open source Java projects that use Fork/Join.
- The identification of three anti-patterns in these projects: (i) sub-optimal use of Java collections when dividing tasks into subtasks, (ii) sub-optimal use of Java collections when merging results of subtasks, and (iii) inappropriate sharing of resources between tasks.
- A study of the impact on performance of the three anti-patterns.
- Proposals on how future languages or frameworks could help avoid the identified anti-patterns.

Paper Outline In Section 2 we introduce the Java Fork/Join framework and some best-practices on how to use the framework effectively. We then turn to our empirical study of how the Fork/Join framework is used in practice. Section 3 describes our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPPJ '14, September 23 - 26 2014, Cracow, Poland.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2837-1/14/09...$15.00.
http://dx.doi.org/10.1145/2647508.2647511

methodology, that is, how we identified our corpus of 120 open source Java projects and how we identified the patterns. The peculiarities of benchmarking on the Java platform that we took into account are also discussed in Section 3. Section 4 sheds light on how the Java Fork/Join framework is used in practice. We confirm the use of four best-practice patterns (Section 4.1). Next, we identify and document three recurring anti-patterns (Section 4.2). Section 5 revisits the three anti-patterns and studies their impact on performance. In Section 6 we describe how the identified anti-patterns could be avoided. Finally, Section 7 highlights related empirical studies.

2. The Java Fork/Join Framework

We give an introduction on how to implement recursive divide-and-conquer algorithms with Java's Fork/Join framework. This section is primarily based on *Concurrent Programming in Java: Design Principles and Patterns* [11, Section 4.4], which documents best practices and gives advice on how to implement efficient parallel programs. All code examples in this text are Java snippets that resemble executable code as much as possible. For editorial reasons we sometimes omitted exception handling, concrete method implementations, and generic types.

Divide-and-conquer algorithms consist of a *base case*, in which the problem is small enough to be readily solved, and a *recursive case*, in which the problem is split into smaller subproblems and the solution to each subproblem is merged. Such algorithms are particularly easy to parallelize when the subproblems act on disjunct parts of the problem dataset. It then suffices to *split* the problem into subproblems, *fork* these computations in order to execute them potentially in parallel, and to *join* them, i. e., synchronize on a subproblem to *merge* the results.

2.1 Expressing an Algorithm as a Fork/Join Task.

In the Java Fork/Join framework, parallel computations are modelled as subclasses of `ForkJoinTask` or one of its subclasses `RecursiveTask` and `RecursiveAction`. In this paper, we refer to a subclass of `ForkJoinTask` as a *task type*, and to an instance of task type as a *task*.

As an initial example, fig. 1 shows a parallel implementation of the naive recursive divide-and-conquer algorithm to compute Fibonacci numbers. The `Fibonacci` class inherits from `RecursiveTask` to represent this computation. In general, task classes hold fields representing the *arguments* and *results* of the computation. Here, the input argument `n` represents the nth index of the Fibonacci sequence. Subclasses of `RecursiveTask` specify the actual computation in the `compute()` method which also returns the result.

In this example, `compute()` contains both the base case (lines 5-6) as well as the recursive case (lines 7-14). In order to reach the base case, each recursive step creates two new tasks initialized with the corresponding argument. A task `t` can subsequently be scheduled for parallel execution by calling `t.fork()`. Finally, synchronize on the completion of `t` and to obtain the result, call `t.join()`.

Figure 2 shows how to schedule a task on a `ForkJoinPool` to start the actual computation. Internally, a `ForkJoinPool` is implemented as a thread pool that uses work stealing to balance the load of executing tasks among parallel threads.

```
1  public class Fibonacci extends RecursiveTask<Integer> {
2    private final int n;
3    public Fibonacci(int n) { this.n = n; }
4    public Integer compute() {
5      if ( n<2 ) {
6        return n;
7      } else {
8        Fibonacci taskLeft  = new Fibonacci( n-1 );
9        Fibonacci taskRight = new Fibonacci( n-2 );
10       taskLeft.fork();
11       taskRight.fork();
12       int resultLeft  = taskLeft.join();
13       int resultRight = taskRight.join();
14       return resultLeft + resultRight;
15     }
16   }
17 }
```

Figure 1. A naive `RecursiveTask` computing the nth Fibonacci number.

```
1 ForkJoinPool poolOfWorkerThreads = new ForkJoinPool();
2 Fibonacci task = new Fibonacci( n );
3 int result = poolOfWorkerThreads.invoke( task );
```

Figure 2. Starting the Fibonacci computation on a `ForkJoinPool`

2.2 Efficiency Maxims for Fork/Join Computations.

To obtain maximum efficiency with Fork/Join, it is important to (i) *maximize parallelism*, (ii) *minimize overhead*, (iii) *minimize contention*, and (iv) *maximize locality*(Lea [11]). We refer to these as the *efficiency maxims* of the Fork/Join framework. The anti-patterns identified in Section 4.2 are essentially violations of these maxims. In the following paragraphs, we translate these efficiency maxims into documented best-practices.

Choose Task Granularity Carefully. In the case of recursive divide-and-conquer algorithms, the two maxims of maximizing parallelism and minimizing overhead need to be balanced against each other to achieve efficient execution. Spawning too many tasks may maximize parallelism at the expense of more overhead. Spawning too few tasks may minimize overhead at the expense of less parallelism.

The tradeoff between parallelism and overhead often made explicit in code in the form of a *sequential cutoff*, a threshold below which tasks are no longer split, but rather executed sequentially. A sequential cutoff acts like the base case for the recursive splitting of tasks. When chosen carefully, a sequential cutoff ensures that the overhead of creating, forking, scheduling and joining tasks does not outweigh the performance gained by parallel execution.

As a rule of thumb, the Fork/Join framework documentation[1] advises to have sequential tasks execute between 100 and 100, 000 instructions. Figure 3 applies this best-practice to the earlier Fibonacci example (cf. lines 2 and 19). As in our example, the sequential cutoff is often a constant, although it can also be dependent on runtime information, e. g., by taking the number of already queued tasks into account. The Fork/Join framework provides access to such information via the `getSurplusQueuedTaskCount()` method.

Keep Task Objects Small. Given its recursive divide-and-conquer nature, it is not uncommon for a Fork/Join computation to spawn a number of tasks that is exponential in the amount of divisions. That is, the computation spawns a tree of tasks, and the number of tasks at depth d is proportional to $O(n^d)$. When a computation is known to spawn tasks in abundance, it pays to minimize the *size* of a task object, as even a small reduction in the size of a task will pay for itself many times over. Therefore, it is a common pattern to reuse an instance field for both input and output values.

[1] *ForkJoinTask (Java Platform SE 7)*, Oracle, access date: 5 December 2013 http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html

```java
public class Fibonacci extends RecursiveTask<Integer> {
  private static final int SEQUENTIAL_CUTOFF = 13;
  private final int n;

  public Fibonacci(int n) {
    this.n = n;
  }

  private int computeSequentially(int n) {
    if ( n<2 ){
      return n;
    } else {
      int resultLeft  = computeSequentially(n-1);
      int resultRight = computeSequentially(n-2);
      return resultLeft + resultRight;
  }

  public Integer compute() {
    if ( n < SEQUENTIAL_CUTOFF ) {
      return computeSequentially( n );
    } else {
      Fibonacci taskLeft  = new Fibonacci( n-1 );
      Fibonacci taskRight = new Fibonacci( n-2 );
      resultLeft.fork();
      int resultRight = taskRight.compute();
      int resultLeft  = taskLeft.join();
      return resultLeft + resultRight;
    }
  }
}
```

Figure 3. An optimized Fibonacci implementation using Fork/Join.

Avoid Unnecessary Forking. To reduce the overhead of scheduling too many tasks and to avoid unnecessary task synchronization, if a thread has just split a task into two smaller subtasks, it can always execute one of the subtasks itself, without involving the scheduler. In general, this pattern avoids the scheduling and associated synchronization overhead for half of the created tasks.[2]

In fig. 3 (line 25), we apply this pattern by having the parent task directly call the `compute()` method on `taskRight`, rather than calling its `fork()` and `join()` methods. Note that the order of lines 24–25 is crucial: if these lines were to be reversed, no task would ever run in parallel.

Avoid Resource Contention. Fork/Join computations are most efficient when individual tasks do not share resources that require coordination. For instance, if multiple tasks acquire the same lock, chances are that the coordination overhead will outweigh the benefit of parallel execution. Sometimes resources can be shared efficiently with little overhead, e. g., one can share an array between tasks, where each task reads and/or writes to a non-overlapping subrange of the array. Note that, even if the ranges are logically non-overlapping, contention may still arise if parts of the array are stored on the same cache line, leading to cache effects such as *false sharing* [5].

2.3 Fork/Join Design Patterns.

In addition to the general efficiency maxims and best-practices described above, there also exist a number of design patterns that provide guidance on how to structure Fork/Join code for dealing with particular requirements. In section 4.1 we confirm that four best-practices described by Lea [11, Section 4.4] are actually used.

[2] *Beginner's Introduction to Java's ForkJoin Framework*, Dan Grossman, access date: 11 December 2013 http://homes.cs.washington.edu/~djg/teachingMaterials/spac/grossmanSPAC_forkJoinFramework.html

3. Methodology

Conducting a qualitative empirical study on a corpus of non-trivial open source projects gives insight into how Fork/Join is used in practice. However, we are unaware of any publicly available corpus containing a sufficiently large number of projects that use Fork/Join. Therefore, we built our own corpus which consists of 120 non-trivial open source Java projects. All projects are publicly available on the GitHub code hosting platform.

Before proceeding to the results of our study, we devote some time to discuss the scientific methodology that lead to these results. Figure 4 gives a high level overview of how the corpus of 120 GitHub projects was constructed, the details are discussed below. Then, we elaborate on the used methodology for measuring the impact on performance of the anti-patterns. We close this section by discussing threats to the validity of our results.

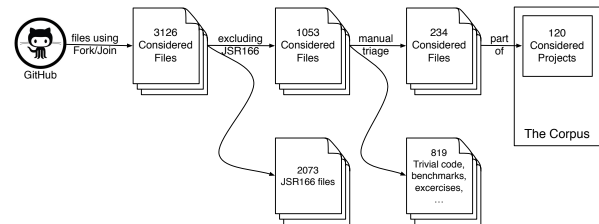


Figure 4. The process underlying the construction of our corpus.

Mining GitHub. GitHub is one of the most popular code repository sites for a wide range of open source projects.[3] To identify projects that use the Fork/Join framework, we searched for Java files containing classes that subclass one of the three Fork/Join task types provided by the framework: the class `ForkJoinTask` or one of its two direct known subclasses: `RecursiveAction` and `RecursiveTask`. At least one of these classes must be subclassed to implement a Fork/Join computation (cf. Section 2).

Before Fork/Join became part of the standard `java.util.concurrent` package, it was available as a Java Specification Request (JSR) 166.[4] Many projects include `jsr166` to use Fork/join on pre Java 7 platforms. In the corpus, we ignore Java files that belong to the `jsr166` package because the reference implementation is not relevant to our study. Table 1 shows the number of Java Fork/join files found on GitHub. Roughly 2 out of 3 Java Fork/Join files on Github are part of `jsr166` and are therefore excluded from the corpus.

`extends`	incl. `jsr166`	excl. `jsr166`
`ForkJoinTask`	1,074	56
`RecursiveTask`	1,148	491
`RecursiveAction`	904	506
Total	3126	1053

Table 1. Number of Java files found when querying GitHub for Fork/Join task types (October 2013)

Curating the Corpus We manually curated the 1053 non-`jsr166` Fork/Join files (cf.Table 1) and excluded those that were considered proof-of-concept experiments, textbook examples, or homework assignments: a) Fork/Join is a recent feature introduced in Java 7

[3] *Github Has Surpassed Sourceforge and Google Code in Popularity (June 2011)*, Klint Finley, access date: 10 December 2013 http://readwrite.com/2011/06/02/github-has-passed-sourceforge

[4] *Concurrency JSR-166 Interest Site*, Doug Lea, access date: 19 November 2013 http://g.oswego.edu/dl/concurrency-interest/

and many programmers want to learn the framework through experimentation. In particular calculating Fibonacci numbers, quicksort, mergesort, and numerically approximating integrals are frequently recurring example programs. b) Several textbooks on concurrent and parallel programming in Java introduce the Fork/Join model. We excluded all files that implement exercises from such textbooks. Files that are part of course preparations, blog posts, or presentations usually only show simple examples. We therefore excluded these as well. c) A final class of files that we did not consider for further review are those made as homework assignments or projects for Computer Science courses.

Applying these three filters to the 1053 Fork/Join files leaves 234 files for a detailed study. These 234 files belong to 120 distinct projects which we call *the Corpus*. An overview of the selected projects and instructions to download the corpus can be found on our website (http://soft.vub.ac.be/~madewael/w-JFJuse/).

Evaluating performance impact. As opposed to the best-practice patterns, the three anti-patterns have, to the best of our knowledge, not previously been identified as such in the context of Fork/join programming. Therefore, we document these anti-patterns both quantitatively (i. e., how often do they occur in the corpus, Section 4.2), as well as qualitatively (i. e., how do they affect performance, Section 5).

To assess the impact of the anti-patterns on program performance, one would ideally refactor the 120 projects such that the anti-pattern is remedied, and then measure performance before and after the change. Performing such refactorings on all the 120 projects, and subsequently benchmarking them is not practically feasible, as these applications have no readily available benchmarks (i. e., no harness, no input data and dimensioning, undocumented dependencies, various build frameworks). Additionally, none of the projects exhibit all three anti-patterns and electing one project per anti-pattern is highly susceptible for introducing biases. Moreover, benchmark results for a single project are not necessarily representative for the whole corpus.

Instead, we opted to use a set of synthetic benchmarks that clearly exhibit the anti-pattern. This approach allows us to investigate the effect of the anti-pattern on the efficiency of a Fork/Join program isolated from other aspects. The structure and computation of the benchmarks are discussed in Section 5, here we focus on the measuring methodology.

When benchmarking on the JVM it is important to take the various peculiarities of the managed runtime into account [2, 9]. It is important to be aware of JVM warm up time, the difference between interpreted mode and mixed mode, dead code elimination and other aggressive compiler optimisations, garbage collection effects, etc. We used a benchmarking framework[5] that takes these issues into account and repeats each benchmark until the gathered data is statistically significant.

The benchmarks are executed on a Ubuntu 13.10 server (Linux kernel version 3.11.0) with four AMD Opteron 6376 processors, forming an eight-node NUMA setup supporting 64 hardware threads. The machine has 64 GB of memory of which up to 16 GB could be allocated by the Java 1.7 Runtime Environment (OpenJDK JRE IcedTea 2.3.12).

Threats to Validity. As is the case with any empirical study, there are a number of threats to the validity of our results.

Our results are naturally biased toward our selected projects. None of the authors of this paper are involved in any of the projects in corpus. We thus cannot and do not make any claims about the overall *quality* of these projects.

The corpus is built only from projects found on GitHub. It is possible that different source code repositories attract different communities of programmers, which may use Fork/Join in different ways. We have not studied this dependency, and instead assume that Java code hosted on GitHub is representative for Java code in the large.

Our corpus was curated manually based on the criteria outlined above. We may have excluded projects that use the framework for non-trivial applications, and we may have retained projects that use the framework in trivial ways.

As we describe later, in the 120 selected projects, we identified a number of recurring patterns and anti-patterns. We do not claim to have identified *all* potentially relevant patterns. For the quantitative discussion on the presence of the patterns (and anti-patterns) we present the numbers as a *global percentage* (i.e., one number per pattern for the whole corpus). This percentage is global because we are unable to show a correlation between the presence of a Fork/Join patterns and other properties of the project. Figure 5 compares the distribution of LOC in projects without anti-patterns to the distribution of LOC in projects with anti-patterns. We conclude from the highly overlapping distribution that there is no correlation between project size and presence of anti-patterns in the Fork/Join corpus.

The projects were studied in a certain order and thus, the identification of patterns could be biased by the order in which they appear in the corpus. To avoid the bias of such a "learning effect", one would need to study the corpus with a larger group of people, where each person studies the projects in a random order. We have not attempted such a larger study. Nevertheless, we are confident that the identified patterns already give relevant insight in how the Fork/Join framework is used in practice.

As we did not know beforehand what patterns we were looking for, tool support to structurally query code was of little use to us. We identified the patterns and anti-patterns using manual search. As a result, the reported numbers may not accurately report on all the occurrences of the patterns or anti-patterns. The precise details of how we counted the occurrences of a particular pattern is explained in the following section, where we introduce the actual patterns.

4. Fork/Join in the Wild: Patterns and Anti-Patterns

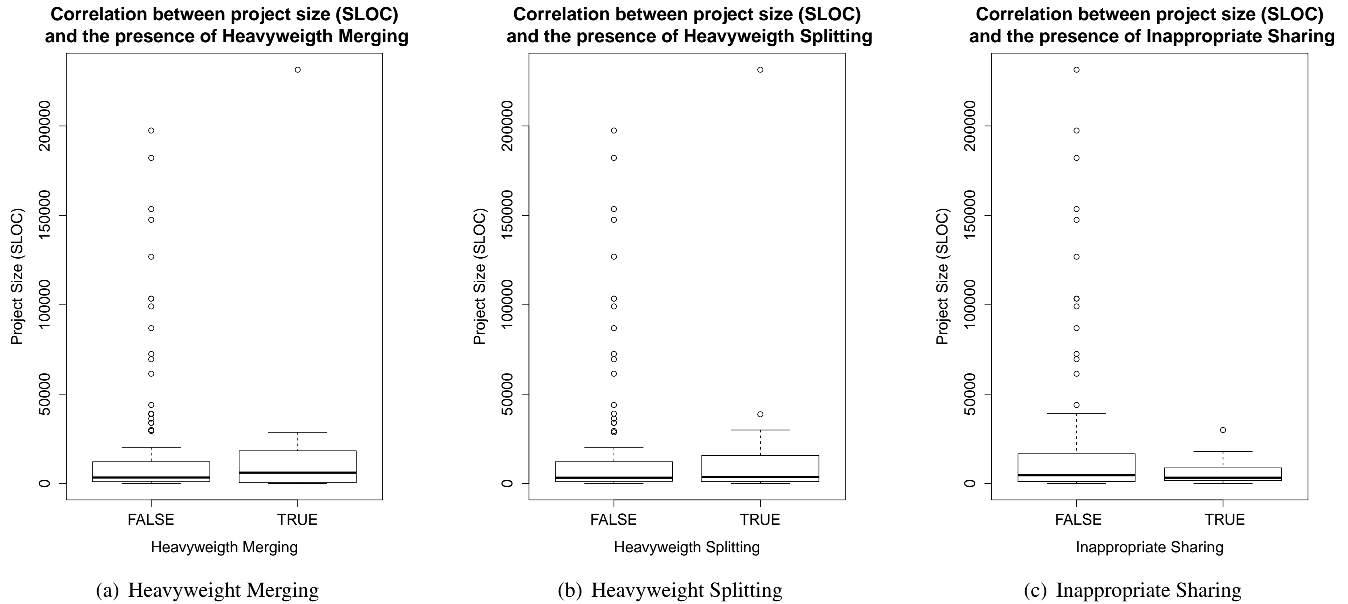
We report on recurring code patterns in Fork/Join tasks drawn from our corpus. We categorize these patterns as follows:

- Patterns that are considered "best-practice", this can be for performance as well as from a software engineering perspective. We describe these in section 4.1. Retrospectively, the patterns found in the corpus can be tied back to best-practices documented in textbooks or tutorials.
- Some patterns identify "bad smells" that go against the efficiency maxims mentioned in section 2, i.e. *maximizing parallelism*, *minimizing overhead*, *minimizing contention*, and *maximizing locality*. We refer to these patterns as *anti-patterns* and describe them in section 4.2.

4.1 Design Patterns

Lea [11, Section 4.4] introduces multiple design patterns worth considering when developing a Java Fork/Join program. Our evaluation of the corpus of Fork/Join programs reveals that two of the presented patterns occur frequently in actual open source projects. We will highlight differences found between concrete code and the textbook description of the pattern.

[5] *Robust Java benchmarking*, Brent Boyer, access date: 3 March 2014 http://www.ibm.com/developerworks/library/j-benchmark1/



(a) Heavyweight Merging (b) Heavyweight Splitting (c) Inappropriate Sharing

Figure 5. The distribution of the project sizes of the projects with (left box plots) and without (right box plots) the anti-patterns are largely overlapping. The overlapping distributions indicates that there is no obvious correlation between project size and the presence of Fork/Join Anti-Patterns.

4.1.1 Linked Subtasks

In the Fibonacci example from section 2, a task is always split into just *two* subtasks. Other problems may require a task to be split into a larger number of subtasks. The precise number of subtasks may even be dynamically determined. A textbook example could be the recursive traversal of a directory tree, where the number of subtasks required to process a directory depends on the number of subdirectories.

When a task spawns a dynamic number of subtasks, this raises the question of how to accumulate the results of the sibling subtasks with minimal overhead. We observe that most often, the sibling subtasks are gathered into a collection object. This collection is then iterated over when all subtasks have finished. Concretely, we identify two variants of this pattern in our corpus.

```
1  public class DirectoryTask extends RecursiveTask {
2    protected Long compute() {
3      List<RecursiveTask> tasks = new ArrayList<>();
4      for (File f : dir.listFiles() ) {
5        if ( f.isDirectory() ){
6          tasks.add( new DirectoryTask( f ) );
7        } else {
8          tasks.add( new      FileTask( f ) );
9        }
10     }
11     long sum = 0;
12     for (RecursiveTask task : invokeAll( tasks ) ) {
13             // exception handling omitted
14             sum += task.get();
15     }
16     return sum;
17   }
18 }
```

Figure 6. Recursively traversing a directory hierarchy where the tasks in the list `subTasks` are invoked all at once.

Figure 6 shows the first variant where all tasks are created and added to a collection first (lines 4–10) and then simultaneously invoked (line 12).[6] Finally, the results are collected and combined by iterating over the completed tasks and getting the results (lines 12–15).

```
1  public class DirectoryTask extends RecursiveTask {
2    protected Long compute() {
3      List<RecursiveTask> tasks = new ArrayList<>();
4      for (File f : dir.listFiles() ) {
5        if ( f.isDirectory() ){
6          tasks.add( new DirectoryTask( f ).fork() );
7        } else {
8          tasks.add( new      FileTask( f ).fork() );
9        }
10     }
11     long sum = 0;
12     for (RecursiveTask<Long> task : tasks ) {
13       sum += task.join();
14     }
15     return sum;
16   }
17 }
```

Figure 7. Recursively traversing a directory hierarchy with directly forked tasks.

The observed alternative is shown in fig. 7, where already forked tasks are collected (lines 4–10) and then the tasks are joined one by one while collecting and combining their results (lines 12–14). Both alternatives are presented here, as they both occur frequently.[7] Lea [11] proposes a third alternative, where instead of an explicit instance of a `Collection` class (i. e., a `List`), a linked list of subtasks is encoded within the tasks themselves (i. e., each subtask

[6] The `invokeAll()` method takes a collection of tasks and behaves as if all tasks are first forking and then joining all tasks. It returns a collection of completed tasks.

[7] We did not count the alternatives separately, instead the reported number of occurrences is the aggregate of both.

holds a `next` field to the next sibling). This avoids the overhead of using a generic collection object. We have not observed any projects in our corpus that apply this optimization.

We determined the number of occurrences of the *Linked Subtasks* pattern by counting the number of task types that combine the instantiation of a `List` with *either* a call to `invokeAll()` and a `for`-loop; *or* with a `for`-loop containing a call to `fork()`. In our corpus of 120 projects, we encountered this pattern in 39 projects (33% of the corpus). This indicates that it is quite common for a problem to be split into a dynamic number of subproblems.

4.1.2 Leaf Tasks

Fork/Join computations can naturally express tree traversals. Often, processing the leaves of a tree is computationally different from processing intermediary nodes. As part of the *Computational Tree* pattern, Lea [11] indicates that it is good practice to create two separate task types: one task type for the intermediary nodes and one task type for the leaf nodes.

This is exemplified in fig. 7, which depicts a parallel traversal of a directory tree. The code creates a `DirectoryTask` to process directories (intermediate nodes in the tree) and a `FileTask` to process files (leaves in the tree).

We determined the number of occurrences of this pattern by counting the number of task types that fork a task of a different type `T` in their `compute()`-method, and the forked type `T` does not itself spawn any other tasks in its own `compute()`-method.

We encountered the "Leaf Task" pattern in 33 projects (28% of the corpus). Thus, it seems fairly common for developers to separate base cases into separate task types.

4.1.3 Avoid Unnecessary Forking.

In section 2 we mentioned that when a task creates two subtasks, it is not necessary to `fork()` both of them. Forking one subtask and computing the other directly reduces the overhead of scheduling and task synchronization by a factor of two, while the amount of parallelism remains the same.

We manually counted the occurrence of this pattern (a call to `fork()` followed by a call to `compute()`) and found that 23 projects (19% of the corpus) avoid unnecessary forking in this manner.

Note that of the remaining 81% of the projects that do not use this pattern, some use the `invokeAll()` method to schedule two or more tasks, which also schedules tasks with lower overhead than naively calling `fork()` and `join()`. We do not count those, as it is not possible to determine whether the call to `invokeAll()` is made to avoid unnecessary forking or for an other reason.

4.1.4 Sequential Cutoff

In section 2 we discussed the importance of choosing task granularity carefully, and the use of a sequential cutoff to achieve the desired granularity. A well chosen sequential cutoff ensures that the overhead of creating, forking, scheduling and joining tasks does not outweigh the performance gained by parallel execution.

The documentation of the class `ForkJoinTask` states that developers should aim to have tasks that execute between 100 and $10,000$ "basic computational steps". It is a best-practice to determine the ideal Sequential Cutoff experimentally w.r.t. performance. Whereas, choosing an *Inappropriate Sequential Cutoff* could be considered to be an anti-pattern. Determining for each of the projects of the corpus whether the sequential cutoff is appropriate or not is infeasible (see section 3). Therefore, we only report on the quantitative occurrence of this pattern.

In 54 projects (45% of the corpus) we did not find the use of an explicit sequential cutoff. In 11 projects (9% of the corpus) the sequential cutoff corresponds to the base case of the sequential algorithm (for instance, tests such as "`size == 1`" where `size` denotes the problem size, or "`from >= to`" where `from` denotes the low bound and `to` the high bound of a range). In the remaining 55 projects (46% of the corpus) we observe an explicit—non trivial—sequential cutoff.

4.2 Anti-Patterns

In this section we document the observed anti-patterns that go against the efficiency maxims of Fork/Join and report on how often they occur in the corpus. In section 5 we zoom in on their impact on performance. In section 6 we offer proposals on how these anti-patterns can be avoided or resolved.

4.2.1 Heavyweight Splitting

Many Fork/Join tasks operate on an indexable data structure (e. g., a `List` or an array), with subtasks operating on contiguous partitions of this data structure. Typical examples include parallel sort and search algorithms.

As part of the recursive step, it is often necessary to split the input data structure into two smaller structures on which the subtasks can operate. When the indexable data structure `i` implements the `List` interface, a view on a partition can be obtained with a call to `i.subList()`, which returns a new object that also implements the `List` interface. An example using `subList()` to split a data structure is shown in fig. 8. Alternatively, when the indexable data structure is an array a similar result can be obtained with a call to `System.arraycopy(src, srcPos, dest, destPos, length)` where `length` is halve of the length of the original array `src`.

```
1  public class ListTask<S> extends RecursiveTask<T> {
2    private List<S> source;
3
4    private T computeSequentially(List<S> source) ...
5    private T combine(T resultLeft, T resultRight) ...
6
7    public T compute() {
8      int size = source.size();
9      if ( size < SEQUENTIAL_CUTOFF ) {
10       return computeSequentially( in );
11     } else {
12       int mid = size/2;
13       ListTask taskLeft  =
14         new ListTask( source.subList(0, mid) );
15       ListTask taskRight =
16         new ListTask( source.subList(mid, size) );
17       taskLeft.fork();
18       T resultRight = taskRight.compute();
19       T resultLeft  = taskLeft.join()
20       return combine( resultLeft, resultRight );
21     }
22   }
23 }
```

Figure 8. Using `subList()` to recursively split a data structure.

The problem with this pattern is that, at each stage of the recursive split, a new object is allocated and initialized. In the case of `subList()`, performance depends on the concrete list implementation. When using an `ArrayList`, the use of sublists to `get()` or `set()` elements via an index is relatively lightweight. For other kinds of lists however, recursive creation of sublists may lead to a stack of views on the original data structure, which must be traversed on each individual element access.

When using arrays and `System.arraycopy()`, the situation is even worse as the entire array is copied on each subdivision.

A more lightweight approach to split an indexable data structure is simply for the tasks themselves to keep track of the subrange that they must operate on, e. g., by passing around `lo` and `hi` indices, and by indexing directly into the original data structure.

The use of `subList()` or `System.arraycopy()` violates the maxim of minimizing overhead. We determined the number of occurrences of this anti-pattern by counting the number of task types that call `subList()` or `System.arraycopy()` on the input data in their `compute()` method. 24 projects (20% of the corpus) exhibit the pattern. This indicates that users of the Fork/Join framework do not always minimize the cost associated with splitting a problem into smaller parts when using standard Java collections.

4.2.2 Heavyweight Merging

After having joined on its subtasks, a task must usually combine the results of the subtasks into a result for the larger problem. In simple cases such as the Fibonacci example in fig. 3, the result is a simple scalar and combining results is as simple as adding them. However, it is not uncommon for tasks to return more complex data structures that need to be *merged*. There are Fork/Join computations where the size and shape of the result data structure is known a priori. A prototypical example is a parallel map over a collection. In this case, the output data structure typically has the same size and shape as the input data structure, and can be pre-allocated.

There also exist Fork/Join computations where the size is not known a-priori but rather depends on the problem input. For example, when performing an exhaustive parallel search, the number of matching results is not known. Typically, the result of a task will be the merger of the results of the subtasks.

When working with a Java `Collection c`, a common way of merging results is to use `c.addAll(c2)`, where `c2` is another `Collection`. This will add all elements of `c2` to `c`. Figure 9 shows how this operation can be used to combine two sublists into a single larger list.

```
1  public class MergeTask<S>
2      extends RecursiveTask<List<T>> {
3    private S source;
4    private List<T> computeSequentially(S source) ...
5    private S[] split(S source) ...
6    public List<T> compute() {
7      if ( size( source ) < SEQUENTIAL_CUTOFF ) {
8        return computeSequentially( source );
9      } else {
10       S[] parts = split( source );
11       MergeTask taskLeft  = new MergeTask( parts[0] );
12       MergeTask taskRight = new MergeTask( parts[1] );
13       taskLeft.fork();
14       List<T> result      = new ArrayList<T>();
15       List<T> resultRight = taskRight.compute();
16       List<T> resultLeft  = taskLeft.join();
17       result.addAll( resultLeft );
18       result.addAll( resultRight );
19       return result;
20     }
21   }
22 }
```

Figure 9. Using `addAll()` to recursively merge task results.

The performance of `addAll()` is highly dependent on the used data structure. For instance for `ArrayList`, `addAll()` is an expensive operation because data may be excessively copied. Its implementation is shown in fig. 10.[8] If the argument `c` to `addAll()` is itself an `ArrayList`, the call to `toArray()` on line 2 will call `System.arraycopy()`. The call to `System.arraycopy()` on line 5 then copies the same data again. If the receiver array does not have enough capacity to hold the result, the call on line 4 leads to another call to `System.arraycopy()` to resize the array.

[8] *ArrayList.java*, Josh Bloch, Neal Gafter, access date: 11 December, 2013 http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/classes/java/util/ArrayList.java

```
1 public boolean addAll(Collection<? extends E> c) {
2     Object[] a = c.toArray();
3     int numNew = a.length;
4     ensureCapacityInternal(size + numNew);
5     System.arraycopy(a, 0, elementData, size, numNew);
6     size += numNew;
7     return numNew != 0;
8 }
```

Figure 10. Implementation of `addAll()` in `ArrayList` from openJDK7.

It is generally not a good idea to recursively merge collections by copying all the individual elements. Pointer-based data structures such as trees can generally be combined more efficiently into larger data structures [19].

We determined the number of occurrences of this anti-pattern by counting the number of task types that, inside their `compute()` method, call `addAll()` on an `ArrayList` or `putAll()` on a `HashMap`. We encountered this anti-pattern in 15 projects (13% of the corpus). As in the case of heavyweight splitting, it seems that developers do not always consider the overhead associated with merging solutions of subproblems when using standard Java collections.

4.2.3 Inappropriate Sharing of Resources among Tasks

We now turn to the efficiency maxim of minimizing contention. Fork/Join is a programming model for parallel programming, and operates at its best when the forked tasks each operate on an isolated set of resources. The term "resources" can be interpreted broadly to mean a set of objects, arrays, files, etc. Quoting Lea [11]:

> Tasks should minimize (in most cases, eliminate) use of shared resources, global (static) variables, locks, and other dependencies. Ideally, each task would contain simple straight-line code that runs to completion and then terminates.

When tasks do need to share resources, the Fork/Join framework itself offers no direct support to handle the coordination of concurrent accesses on these resources. To manage synchronization, programmers must turn to other utilities in the `java.util.concurrent` package, or resort to Java's built-in locks.

Unless synchronization mechanisms are applied with the greatest care, they can significantly reduce parallel performance. Sharing resources among tasks is not in itself an anti-pattern, unless sharing could have been easily avoided, or synchronization on the shared tasks is done in a naive or incorrect way.

By "inappropriate sharing" we mean one of three things:

- Tasks unnecessarily share resources, i. e., sharing could have been avoided in the first place. For example, we observed projects where all tasks increment a single globally shared counter. More appropriate would be for each task to increment a local variable, and then to sum up these local variables when joining the tasks.
- Tasks necessarily share resources, but synchronization is naive. For example, tasks may share a collection which they query and update, such as a `Map`. A naive synchronization strategy is to either guard the entire `Map` using a single lock or to wrap the map using `Collections.synchronizedMap()`, where the better alternative would be to use a genuine concurrent collection such as a `ConcurrentHashMap`. For example, one of the projects in the corpus wraps an `ArrayList` with `Collections.synchronizedList()` to share it among its tasks.

  Even when developers make use of concurrent collections, they do not always use them correctly. For instance, a `CopyOnWrite-`

Pattern	Occurrence	Percentage of the Corpus
Explicit Sequential Cutoff	55	46%
Linked Subtasks	39	33%
Leaf Tasks	33	28%
Avoid Unnecessary Forking	23	19%
Heavyweight Split	24	20%
Heavyweight Merge	15	13%
Inappropriate Sharing	18	15%

Table 2. Overview of the patterns and anti-patterns and their occurrence in the corpus.

ArrayList creates a fresh copy of an ArrayList on each update to the list, to avoid contention. This strategy only pays when the number of query operations on the list vastly outnumber updates. We observed one project that used a shared CopyOnWriteArrayList, but where all the leaf tasks write to the list, causing repeated copying of the list.

- Tasks necessarily share resources, but synchronization is wrong. For instance, tasks may be operating on overlapping subranges of a shared array, causing a race condition when they perform a simultaneous read and write on an overlapping index without acquiring a lock. Another example, observed in our corpus, is when tasks add elements to a shared, non-thread safe collection.

We observed a total of 18 projects (15% of the corpus) that exhibit a form of inappropriate sharing of resources among tasks.

4.2.4 Summary

Table 2 summarizes the occurrence of the patterns and anti-patterns that were discussed in this section. We discuss the impact on performance of the anti-patterns in the following section. In section 6 we interpret the results of our study of the corpus.

5. Performance Impact of the Anti-Patterns

Section 4.2 discusses the three anti-patterns discovered in the corpus. This section studies the performance impact of these anti-patterns on Fork/Join programs. To this end we compare for each of the anti-patterns the execution times of two equivalent programs, one with the anti-pattern and one without the anti-pattern. Our benchmarks show that a program with anti-pattern can be up to two orders of magnitude slower than the equivalent program without the anti-patterns.

The methodology of measuring benchmark programs is discussed in section 3. Here, we present the results of our measurements as graphs showing program execution time in function of problem input size. Note the logarithmic scale on both axes. The confidence intervals for all our benchmarks are insignificant and therefore omitted from the graphs.

As mentioned above, for each anti-pattern we compare the execution times of equivalent programs. For the *Heavyweight Splitting* and *Heavyweight Merging* anti-patterns, we also take the data structure that is being split or merged into account. We limit our experiments to the two data structures that are most commonly used in the corpus, i. e., plain Java arrays and ArrayLists.

All benchmark programs discussed in this section are based on the code skeleton shown in fig. 11. These benchmark programs vary in constructor and member fields, in how the problem is divided into subproblems (cf. lines 2 and 3), in how the leaf tasks behave (cf. line 7–11), or in how subresults are combined (cf. line 21). The concrete variations are discussed in the remainder of this section

```
1  public class Task extends RecursiveTask<T> {
2    public Task createLeftTask() ...
3    public Task createRightTask() ...
4
5    public T compute() {
6      if ( (to-from) < SEQUENTIAL_CUTOFF ) {
7        T res = null;
8        for (int i=from ; i<to ; i++) {
9          ...
10       }
11       return res;
12     } else {
13       int mid = (from+to)/2;
14       Task taskLeft  = createLeftTask();
15       Task taskRight = createRightTask();
16
17       taskLeft.fork();
18       taskRight.compute();
19       taskLeft.join();
20
21       return ...;
22     }
23   }
24 }
```

Figure 11. Recursively split problem in 2 until some threshold is reached.

per anti-pattern. The SEQUENTIAL_CUTOFF on the other hand is kept constant among all benchmark programs and was set to 32.

All benchmarks are run on input sizes between 10^1 and 10^8 elements (in steps of factor 10). We consider the small end of this range (i. e., $< 10^4$) to be an inappropriate input size for a parallel program. To benefit from parallel execution, the input size must be larger. Nevertheless, we show the execution times for these small input sizes to highlight the influence of input size on performance degradation.

5.1 Heavyweight Splitting

The *Heavyweight Splitting* benchmark programs compute the sum of the elements of an array or ArrayList. The *Heavyweight Splitting* anti-pattern occurs in the recursive case, where this data structure is partitioned and distributed among subtasks. The *heavyweight program* (i. e., the program with the anti-pattern) and the *lightweight program* (i. e., the program without the anti-pattern) differ in the way the data structure is partitioned and distributed.

The heavyweight program using arrays splits the data structure by means of a call to System.arraycopy (fig. 13). The heavyweight program using ArrayLists splits the data structure by means of a call to List.subList. The lightweight programs instead pass a reference to the complete data structure to their subtasks together with a *range* to process, represented simply as two integers: a lower and upper bound (fig. 12).

```
1 private int[] source;
2 private int from;
3 private int to;
4
5 public Task createLeftTask() {
6   new Task(source, from, (from+to)/2);
7 }
```

Figure 12. Member fields and left subtasks factory method for lightweight splitting of an array.

Figure 15 shows that the impact of *Heavyweight Splitting* of ArrayLists introduces some slowdown. However, even for larger input sizes the effect is negligible. In the worst case (input size 10^8) we observe a slowdown of 20% compared to the lightweight program. For *Heavyweight Splitting* of plain arrays, on the other

```
1 private int[] source;
2
3 public Task createLeftTask() {
4   int mid = source.length/2;
5   int[] newSource = new int[mid];
6   System.arraycopy(source, 0, newSource, 0, mid);
7   return new SplitArrayHeavy( newSource );
8 }
```

Figure 13. Member fields and left subtasks factory method for heavyweight splitting of an array.

hand, we observed a slowdown of 28× on an input size of 10^8 elements. Thus, depending on the concrete implementation of the data structure that is split, the *Heavyweight Splitting* anti-pattern can have a dramatic impact on a Fork/Join program's performance.

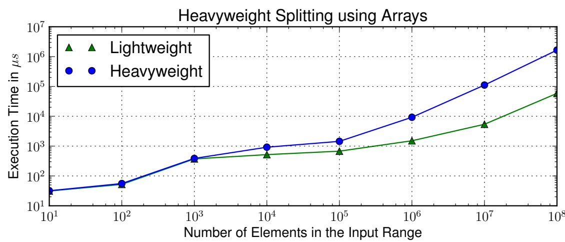


Figure 14. Comparing the executions times of splitting an array.

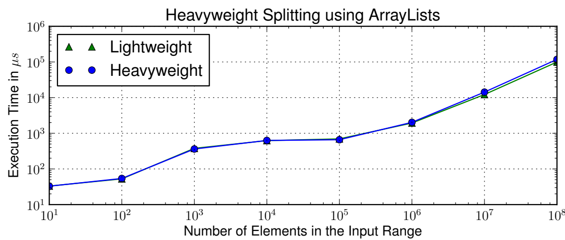


Figure 15. Comparing the executions times of splitting an `ArrayList`.

5.2 Heavyweight Merging

The *Heavyweight Merging* benchmark programs create a deep copy of an array or `ArrayList`. The *Heavyweight Merging* anti-pattern occurs in the recursive case, where the results of the subtasks are combined. The heavyweight programs copy their data into a newly allocated data structure by means of a bulk copy operation, i. e., `System.arraycopy()` for the program using arrays and a call to `addAll()` in the program using the `ArrayList` (fig. 17). The lightweight programs, where a *target* data structure is pre-allocated before invoking the initial Fork/Join task, and is passed along to the subtasks. In the recursive case no explicit combination of the subresults is needed. In the base case, the elements are copied into the pre-allocated data structure.

Figure 19 shows that *Heavyweight Merging* can cause a 17× slowdown in the heavyweight programs with arrays of length 10^8. In the heavyweight programs with `ArrayLists` the slowdown can be as high as 98× when copying 10^8 elements (fig. 18). Thus, for both types of data structure, factoring out the *Heavyweight Merging* anti-pattern can significantly improve the performance of a Fork/Join program.

```
1 private List source;
2 private List target;
3 private int from;
4 private int to;
5
6 public void compute() {
7   if ( (to-from) < SEQUENTIAL_CUTOFF ){
8     for (int i=from; i<to ; i++) {
9       target.set(i, source.get(i));
10    }
11  } else {
12    Task leftTask = createLeftTask().fork();
13    createRightTask().compute();
14    leftTask.join();
15  }
16 }
```

Figure 16. Member fields and `compute`-method for lightweight merging of a list.

```
1 private List source;
2 private int from;
3 private int to;
4
5 public List compute() {
6   if ( (to-from) < SEQUENTIAL_CUTOFF ) {
7     List target = new ArrayList<Integer>();
8     for (int i=from; i<to ; i++) {
9       target.add( source.get( i ) );
10    }
11    return target;
12  } else {
13    Task leftTask = createLeftTask().fork();
14    List result = new ArrayList<Integer>();
15
16    List rightResult = createRightTask().compute();
17    result.addAll( leftTask.join() );
18    result.addAll( rightResult );
19
20    return result;
21  }
22 }
```

Figure 17. Member fields and `compute`-method for heavyweight merging of a list.

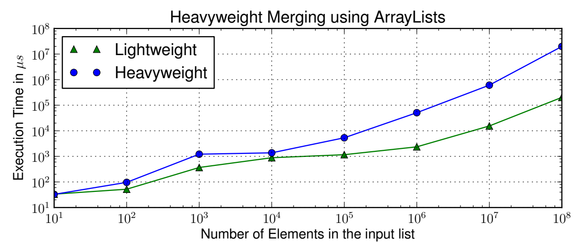


Figure 18. Comparing the executions times of merging an `ArrayList`.

5.3 Inappropriate Sharing

The *Inappropriate Sharing* benchmark programs compute the sum of all integers in the interval $[0, n]$. The programs without *Inappropriate Sharing* simply sum the results of the subtasks in the recursive case (fig. 21). The programs exhibiting *Inappropriate Sharing* increment a shared `AtomicInteger` in the base case, as is shown in fig. 20.

Figure 22 shows the performance impact of using a shared atomic counter instead of recursively summing subresults. Com-

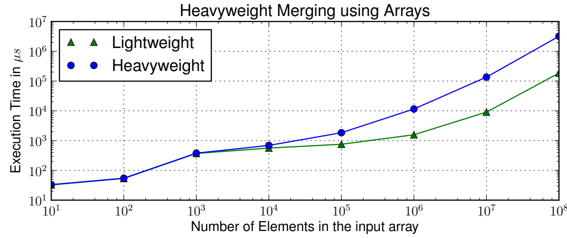


Figure 19. Comparing the executions times of merging an array.

```
1  private AtomicInteger accumulator;
2  private int from;
3  private int to;
4
5  public void compute() {
6    if ( (to-from) < SEQUENTIAL_CUTOFF ) {
7      int local = 0;
8      for (int i=from; i<to ; i++) local += i;
9      this.accumulator.addAndGet( local );
10   } else {
11     Task leftTask = this.createLeftTask().fork();
12     createRightTask().compute();
13     leftTask.join();
14   }
15 }
```

Figure 20. Member fields and `compute`-method for a task that accumulates integers into a shared `AtomicInteger`.

```
1  private int from;
2  private int to;
3
4  public int compute() {
5    if ( (to-from) < SEQUENTIAL_CUTOFF ) {
6      int local = 0;
7      for (int i=from ; i<to ; i++) local += i;
8      return local;
9    } else {
10     Task leftTask = createLeftTask().fork();
11     int resultRight = createRightTask().compute();
12     return leftTask.join() + resultRight;
13   }
14 }
```

Figure 21. Member fields and `compute`-method for a task that recursively accumulates the integers from its subtasks.

puting the sum of 10^8 integers with a shared `AtomicInteger` is $95\times$ slower than the proper Fork/Join program shown in fig. 21.

We did not investigate the relation between execution time and accuracy in programs where a non-atomic integer without synchronisation is used. Neither did we perform experiments where a non-atomic integer is protected using a shared lock. A thorough discussion on counting in parallel and the trade-off between accuracy and performance is given by McKenney [15, Chapter 5].

5.4 Conclusion

We see that for small input sizes the effect of having an anti-pattern in a Fork/Join program is negligible. However, to exploit the full potential of Fork/Join, programs have to be run on large inputs. We showed that when the input size increases, the impact of the anti-pattern also increases. Our benchmarks showed that factoring out the *Heavyweight Splitting* in the context of `ArrayLists` may not be necessary. For all other anti-patterns the effort of refactoring can result in significant performance improvements.

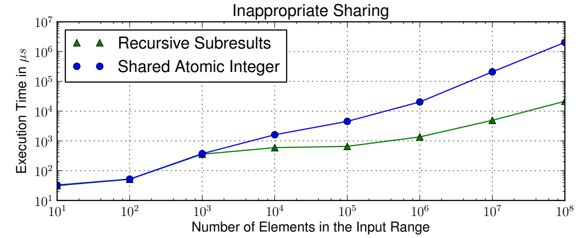


Figure 22. Comparing the executions times of incrementing a shared atomic counter and recursive propagation of local results.

6. Discussion

In this section, we interpret the results of our study. We focus in particular on the observed anti-patterns and formulate proposals on how these could be mitigated.

6.1 Inappropriate Sequential Cutoff

In Section 4.1.4 we stated that 54 projects (45% of the corpus) show no sign of using an explicit sequential cutoff to regulate task granularity. We did not benchmark these projects, so we do not know whether these projects suffer indeed from the high overhead that usually occurs when no sequential cutoff is used. Nevertheless, we were surprised by the high number of projects that make no attempt to coarsen task granularity.

This may indicate that the Fork/Join framework could benefit from an abstraction that automates the coarsening, for instance, by considering the current load on the scheduler to determine whether to fork a task or to compute it sequentially instead.

The documentation of the Fork/Join framework states that a task ideally executes between 100 and $10,000$ “basic computational steps”. It is not clear what is meant by “basic computational steps”, since Java expressions do rarely map one-to-one onto operations executed by the underling JVM. Others[3] advise that a task should perform anywhere between $1,000$ to $100,000$ arithmetic operations.

When a sequential cutoff is not used, the problem is often that the overhead of the creation of many small tasks outweighs the benefit of parallelism. One approach to reduce the overhead of task creation is to make the runtime smarter. Kumar et al. [10] describe a scheduler that defers the overhead of task creation to the point where tasks get stolen. Overhead is minimized when a task is executed by the same thread. Given such runtime support, the use of a sequential cutoff would become less important.

In the absence of such advanced runtime support, choosing an appropriate sequential cutoff remains somewhat of an art and requires application-specific measuring and tuning. Unfortunately, from what we observed, there are still many developers that do not use a sequential cutoff. Even when they do, it is rare to find a project that documents why a particular threshold value was chosen.

6.2 Heavyweight Splitting and Merging

As is reported in Sections 4.2.1 and 4.2.2 respectively, 24 projects (20% of the corpus) exhibit the heavyweight task splitting pattern and 15 projects (13% of the corpus) exhibit the heavyweight merging pattern.

We believe these numbers are due to the fact that many developers use the standard Java collections that they are familiar with (the use of `ArrayLists` is particularly common), even though these collections were not designed with Fork/Join programming in mind. Thus, it may be worthwhile for the framework to include its own variations on existing Java collections, augmented with efficient means to split and merge them.

The observation that Fork/Join can benefit from tailored data structures that can be efficiently split and merged is not new. Our study merely illustrates that the need for such data structures is real and performance is hampered without them. Guy Steele, in his ICFP 2009 keynote made the observation that effective divide-and-conquer parallelism requires tree-like data structures [19]. He proposes to represent lists not as linked lists of cons-cells, but rather as trees of "conc"-cells ("conc lists"), which can be split and merged in constant time.

Fortunately developers need not always use dedicated data structures to avoid heavyweight splits and merges. When using indexable data structures such as arrays or lists, a common practice is to split and merge the *indices* rather than to split and merge the data structure itself. For instance, if a task only reads the elements of a shared array `a` between `a[from]` and `a[to-1]`, it can easily split the range into `[from-mid[` and `[mid-to[` subranges by calculating `mid = (from+to)/2`.

Given that the splitting and merging of ranges is a highly recurring pattern, it may make sense for the framework to provide a "range" abstraction that automates the pattern for a variety of indexable data structures.

Data Parallel Programming using Fork/Join Fork/Join programming is primarily designed to express task parallelism. However, it can also be used to implement data parallelism. For example, it is possible to implement a parallel `map`-operation using Fork/Join by recursively partitioning a data structure and applying the mapped function in parallel in the base case. The use of Fork/Join to implement data parallelism is very common in the programs in our corpus. This partly explains the frequent occurrence of the *Heavyweight Merging* and *Heavyweight Splitting* anti-patterns.

Java 8 introduced the `Stream` API to more directly support data parallelism. In combination with Java 8 lambdas (anonymous functions), many of the Fork/Join programs of the corpus could be rewritten using this API, which results in shorter code that is not susceptible to the Heavyweight Splitting or Heavyweight Merging anti-pattern.

To validate the latter claim, we rewrote the Fork/Join tasks found in the projects that exhibit both the Heavyweight Splitting and the Heavyweight Merging anti-pattern. We refer to our website where these tasks are reimplemented using streams and lambdas from Java 8 (http://soft.vub.ac.be/~madewael/w-JFJuse/).

6.3 Inappropriate Sharing of Resources Among Tasks

Finally, in Section 4.2.3 we reported a total of 18 projects (15% of the corpus) that exhibit a form of inappropriate sharing of resources among tasks. Of all the observed anti-patterns, we believe this one is most likely the hardest to avoid in practice, due to the very implicit sharing of resources made possible by shared-memory multithreading platforms, such as the JVM. Nevertheless, for specific problems, the framework may provide abstractions that help tasks to operate on local resources. The `SplittableRandom` class, as of Java 8, is a good example of a solution to a specific problem[9].

`SplittableRandom` aims to resolve the recurring problem of how to efficiently share a random number generator among tasks. The standard `java.util.Random` class is thread safe, so that it can be safely accessed concurrently by multiple tasks, but this often leads to contention on the shared random number generator. Using a per-task or per-thread random number generator removes this contention, but this setup runs the risk of changing the semantics of the program: sampling multiple independent random number generators does not necessarily lead to the same statistical properties as querying a single random number generator.

The class `java.util.SplittableRandom` provides a solution to this problem. Unlike `Random`, it is not thread safe. However, its interface provides a `split()` method to create a new instance of `SplittableRandom` that, when used in parallel with the original, maintains the statistical properties of the random number generator. The underlying algorithm is based on work by Salmon et al. [18] and Leiserson et al. [13].

Looking beyond platforms such as the JVM, we note that abstractions such as partitioned global address spaces (PGAS), as used in languages such as X10 [7], Chapel [6], and Fortress [1] may help developers reason more explicitly about shared versus local resources. In addition, explicit address spaces may help reasoning about locality of data, which is another efficiency maxim of parallel programming.

7. Related Work

Empirical studies such as ours, which aim to study the use of language features or APIs in the wild, require a sufficiently large corpus of non-trivial projects. Tempero et al. [22] curate a collection of Java projects with the explicit goal to facilitate such studies (e. g., [16, 23]). While their effort is commendable, their corpus unfortunately does not contain a sufficient amount of projects that actively make use of Fork/Join.

Similar to our methodology, Tasharofi et al. [21] build a corpus of 16 open source Scala projects to study issues related to concurrency and parallelism. More specifically, they are interested in whether and how Scala developers mix the Actor Model with other concurrency models. They similarly start from a set of 750 Scala projects that use actors, which they found on GitHub, and then curate that set to end up with a corpus of 16 selected projects.

Okur and Dig [17] study the use of Microsoft's parallel libraries in 655 open source projects in C#, which they collected from GitHub and Microsoft CodePlex. Their methodology to build a corpus is similar to ours in that they too consider only publicly available open source projects, and filter out "toy projects" based on a set of documented criteria. Whereas our study focuses on a specific library, i. e., Fork/Join, their study is more general and considers many different kinds of libraries such as the TPL, PLINQ, concurrent collections, etc. Moreover, whereas we study recurring patterns and anti-patterns, they are primarily interested in the frequency of use of particular libraries, and for each library, what parts of the API are used most often.

Lin and Dig [14] automatically mine a corpus of 28 widely used open source Java projects for "Check-Then-Act" misuse of concurrent collections. Like our work, their work documents a set of anti-patterns (which they call idioms) which they hope will inform library designers to build more resilient APIs.

8. Conclusion

We studied a corpus of 120 open source Java projects that make use of the Fork/Join framework. We observe that many developers who use the framework do apply documented best-practices, such as making use of a sequential cutoff (55% of the corpus) and avoiding unnecessary forking (19% of the corpus). Design patterns such as *Linked Subtasks* and *Leaf Tasks* are also encountered in practice.

On the other hand, we identified three anti-patterns that occur sufficiently frequently to make us conclude that writing efficient Fork/Join programs is not that easy in practice, despite the attractive simplicity of the Fork/Join model itself. These anti-patterns include the inefficient splitting and merging of Java collections, and inappropriate sharing of resources among tasks. Moreover, we studied the impact on performance of these anti-patterns by com-

[9] *SplittableRandom*, Oracle, access date: 20 March, 2014 http://download.java.net/jdk8/docs/api/java/util/SplittableRandom.html

paring equivalent programs, with and without an anti-pattern. Our benchmarks show that a slowdown of two orders of magnitude is possible.

We posit that these anti-patterns can provide insight to language designers and framework developers that aim to expose Fork/Join parallelism. The anti-patterns essentially reveal aspects of parallel programming where developers receive little or no support from the Fork/Join API directly. In particular, we observe that the habit of developers to use standard Java collections not tailored to Fork/Join programming leads to an unnecessarily high overhead when splitting input data and when merging result data. Given the observed usage patterns, a better integration between the Fork/Join framework and the collections library is one avenue of future work.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt. The fortress language specification. Technical report, Sun Microsystems, Inc., 2008. Version 1.0.

[2] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, Aug. 2008.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[5] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Proc. of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, 1993.

[6] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *Int. Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th ACM SIGPLAN Int. Conference on Object Oriented Programming Systems Languages and Applications*, pages 519–538, 2005.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *ACM SIGPLAN Notices*, pages 212–223, 1998.

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 57–76. ACM, 2007.

[10] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu. Work-stealing without the baggage. In *Proc. of the ACM SIGPLAN Int. Conference on Object Oriented Programming Systems Languages and Applications*, pages 297–314, 2012.

[11] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999. ISBN 0201310090.

[12] D. Lea. A java fork/join framework. In *Proc. of the ACM 2000 Conference on Java Grande*, pages 36–43, 2000.

[13] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–204, 2012.

[14] Y. Lin and D. Dig. Check-then-act misuse of java concurrent collections. In *Proc. of the 6th IEEE Int. Conference on Software Testing, Verification and Validation*, pages 164–173, 2013.

[15] P. E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* 2014. URL http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e1.pdf.

[16] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. In *Proc. of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 563–582, 2008.

[17] S. Okur and D. Dig. How do developers use parallel libraries? In *Proc. of the 20th ACM SIGSOFT Int. Symposium on the Foundations of Software Engineering*, pages 54:1–54:11, 2012.

[18] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proc. of the Int. Conference for High Performance Computing, Networking, Storage and Analysis*, pages 16:1–16:12, 2011.

[19] G. L. Steele, Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *Proc. of the 14th ACM SIGPLAN Int. Conference on Functional Programming*, pages 1–2, 2009.

[20] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, pages 54–62, 2005.

[21] S. Tasharofi, P. Dinges, and R. E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proc. of the 27th European Conference on Object-Oriented Programming*, pages 302–326, 2013.

[22] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, 2010.

[23] E. Tempero, H. Y. Yang, and J. Noble. What programmers do with inheritance in java. In *Proc. of the 27th European Conference on Object-Oriented Programming*, pages 577–601, 2013.