Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions

Semih Okur[1], Cansu Erdogan[1], and Danny Dig[2]

[1] University of Illinois at Urbana-Champaign, USA
{okur2,cerdoga2}@illinois.edu
[2] Oregon State University, USA
digd@eecs.oregonstate.edu

Abstract. Parallel libraries continuously evolve from low-level to higher-level abstractions. However, developers are not up-to-date with these higher-level abstractions, thus their parallel code might be hard to read, slow, and unscalable. Using a corpus of 880 open-source C# applications, we found that developers still use the old `Thread` and `ThreadPool` abstractions in 62% of the cases when they use parallel abstractions. Converting code to higher-level abstractions is (i) tedious and (ii) error-prone. e.g., it can harm performance and silence the uncaught exceptions.

We present two automated migration tools, TASKIFIER and SIMPLIFIER that work for C# code. The first tool transforms old style `Thread` and `ThreadPool` abstractions to `Task` abstractions. The second tool transforms code with `Task` abstractions into higher-level design patterns. Using our code corpus, we have applied these tools 3026 and 405 times, respectively. Our empirical evaluation shows that the tools (i) are highly applicable, (ii) reduce the code bloat, (iii) are much safer than manual transformations. We submitted 66 patches generated by our tools, and the open-source developers accepted 53.

1 Introduction

In the quest to support programmers with faster, more scalable, and readable code, parallel libraries continuously evolve from low-level to higher-level abstractions. For example, Java 6 (2006) improved the performance and scalability of its concurrent collections (e.g., `ConcurrentHashMap`), Java 7 (2011) added higher-level abstractions such as lightweight tasks, Java 8 (2014) added lambda expressions that dramatically improve the readability of parallel code. Similarly, in the C# ecosystem, .NET 1.0 (2002) supported a Threading library, .NET 4.0 (2010) added lightweight tasks, declarative parallel queries, and concurrent collections, .NET 4.5 (2012) added reactive asynchronous operations.

Low-level abstractions, such as `Thread`, make parallel code more complex, less scalable, and slower. Because `Thread` represents an actual OS-level thread, developers need to take into account the hardware (e.g., the number of cores) while coding. Threads are *heavyweight*: each OS thread consumes a non-trivial amount of memory, and starting and cleaning up after a retired thread takes hundreds

R. Jones (Ed.): ECOOP 2014, LNCS 8586, pp. 515–540, 2014.

of thousands of CPU cycles. Even though a .NET developer can use `ThreadPool` to amortize the cost of creating and recycling threads, she cannot control the behavior of the computation on `ThreadPool`. Moreover, new platforms such as Microsoft Surface Tablet no longer support `Thread`. .NET also does not allow using the new features (e.g., `async/await` abstractions) with `Thread` and `ThreadPool`. Furthermore, when developers mix old and new parallel abstractions in their code, it makes it hard to reason about the code because all these abstractions have different scheduling rules.

Higher-level abstractions such as .NET `Task`, a unit of parallel work, make the code less complex. `Task` gives advanced control to the developer (e.g., chaining, cancellation, futures, callbacks), and is more scalable than `Thread`. Unlike threads, tasks are *lightweight*: they have a much smaller performance overhead and the runtime system automatically balances the workload. Microsoft now encourages developers to use `Task` in order to write scalable, hardware independent, fast, and readable parallel code [26].

However, most developers are oblivious to the benefits brought by the higher-level parallel abstractions. In recent empirical studies for C# [18] and Java [25], researchers found that `Thread` is still the primary choice for most developers. In this paper we find similar evidence. Our corpus of the most popular and active 880 C# applications on Github [12] that we prepared for this paper, shows that when developers use parallel abstractions they still use the old `Thread` and `ThreadPool` 62% of the time, despite the availability of better options. Therefore, a lot of code needs to be migrated from low-level parallel abstractions to their higher-level equivalents.

The migration has several challenges. First, developers need to be aware of the different nature of the computation. While blocking operations (e.g., I/O operations, `Thread.Sleep`) do not cause a problem in Thread-based code, they can cause a serious performance issue (called thread-starvation) in Task-based code. Because the developers need to search for such operations deep in the call graph of the concurrent abstraction, it is easy to overlook them. For example, in our corpus of 880 C# applications, we found that 32% of tasks have at least one I/O blocking operation and 9% use `Thread.Sleep` that blocks the thread longer than 1 sec. Second, developers need to be aware of differences in handling exceptions, otherwise exceptions become ineffective or can get lost.

In this paper, we present an automated migration tool, Taskifier, that transforms old style `Thread` and `ThreadPool` abstractions to higher-level `Task` abstractions in C# code. During the migration, Taskifier automatically addresses the non-trivial challenges such as transforming blocking to non-blocking operations, and preserving the exception-handling behavior.

The recent versions of parallel libraries provide even higher-level abstractions on top of Tasks. For example, the `Parallel` abstraction in C# supports parallel programming design patterns: data parallelism in the form of parallel loops, and fork-join task parallelism in the form of parallel tasks co-invoked in parallel. These dramatically improve the readability of the parallel code. Consider the example in Code listing 1.1, taken from ravendb [1] application. Code