How do Developers Use Parallel Libraries?

Semih Okur, Danny Dig
University of Illinois at Urbana-Champaign, IL, USA
okur2@illinois.edu, dig@illinois.edu

ABSTRACT

Parallel programming is hard. The industry leaders hope to convert the hard problem of *using parallelism* into the easier problem of *using a parallel library*. Yet, we know little about how programmers adopt these libraries in practice. Without such knowledge, other programmers cannot educate themselves about the state of the practice, library designers are unaware of API misusage, researchers make wrong assumptions, and tool vendors do not support common usage of library constructs.

We present the first study that analyzes the usage of parallel libraries in a large scale experiment. We analyzed 655 open-source applications that adopted Microsoft's new parallel libraries – Task Parallel Library (TPL) and Parallel Language Integrated Query (PLINQ) – comprising 17.6M lines of code written in C#. These applications are developed by 1609 programmers. Using this data, we answer 8 research questions and we uncover some interesting facts. For example, (i) for two of the fundamental parallel constructs, in at least 10% of the cases developers misuse them so that the code runs sequentially instead of concurrently, (ii) developers make their parallel code unnecessarily complex, (iii) applications of different size have different adoption trends. The library designers confirmed that our findings are useful and will influence the future development of the libraries.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Object-oriented constructs*

General Terms

Measurement, Experimentation

Keywords

Multi-core; empirical study; parallel libraries; C#.

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

1. INTRODUCTION

The computing hardware industry has resorted to multicore CPUs in order to keep up with the previous prediction of Moore's law. While the number of transistors will keep doubling, the multicore revolution puts pressure on software developers to use parallelism if they want to benefit from future hardware improvements. At the time, this seemed like a huge gamble: will software developers embrace parallelism in their applications? A few years after the irreversible conversion to multicore, we can finally answer such questions.

Parallel programming is hard. In the desktop computing, the dominant paradigm is thread-based parallelism on shared-memory systems. Under this paradigm, parallel programming is regarded as the art to balance conflicting forces: making code thread-safe requires protecting accesses to shared variables through synchronization, but this in turn reduces the scalability of parallel applications. Parallelism can also obfuscate the intent of the original sequential code [6]. Despite books on parallel programming and API documentation of parallel constructs [2, 10, 14–17], parallel programming education is lagging behind. Developers miss examples [32] of successful applications that use parallelism.

The industry leaders hope to convert the hard problem of using parallelism into the easier problem of using a parallel library. Microsoft provides Task Parallel Library (TPL) [15], Parallel Language Integrated Query (PLINQ) [23], Collections.Concurrent (CC) [3] and Threading [33] for .NET languages (e.g., C#). Java developers uses `java.util.concurrent` package. Intel provides Threading Building Blocks (TBB) [34] for C++. Despite syntactic differences, these libraries provide similar features such as scalable concurrent collections (e.g., `ConcurrentDictionary`), high-level parallel constructs (e.g., `Parallel.For`), and lightweight tasks. Their runtime systems also provides automatic load balancing [37]. Despite the recent surge in the number of these libraries, we know little about how practitioners adopt these libraries in practice.

We present the first empirical study that answers questions about parallel library usage in-depth and on a large scale. We analyzed 655 open-source applications that adopted Microsoft's new TPL and PLINQ libraries. In this corpus, we studied the usage of all four .NET parallel libraries (both old and new). These applications are hosted on Github [8] and Microsoft's CodePlex [4], and they comprise 17.6M non-blank, non-comment lines of code written in C# by 1609 programmers. We implemented a semantic analysis that uses type information to collect *precise* statistics about parallel constructs.

Using this data, we are able to answer several questions.

Q1: Are developers embracing multi-threading? Our data shows that 37% of all open-source C# applications in the most active code repositories use multi-threading. Out of these applications, 74% use multi-threading for concurrency and 39% use it for parallelism.

Q2: How quickly do developers start using the new TPL & PLINQ libraries? TPL and PLINQ have been released nearly 2 years ago (in April 2010). However, we found significant differences between the times when developers start using these libraries. We found that applications of different size have a different adoption tipping point. We also found that more applications are becoming parallel, and existing parallel applications are becoming more parallel.

Q3: Which parallel constructs do developers use most often? 10% of the API methods account for 90% of the library usage, thus newcomers can focus on learning a smaller subset of the parallel libraries.

Q4: How do developers protect accesses to shared variables? Locks are still the most used synchronization construct, but developers use a wide variety of alternatives.

Q5: Which parallel patterns do developers embrace? Out of the six widely-used parallel patterns that we analyzed, loop parallelism is the most common.

Q6: Which advanced features do developers use? We found that developers rarely use optional parameters such as customized task schedulers, aggregate exception handling, controlling the level of parallelism, etc.

Q7: Do developers make their parallel code unnecessarily complex? We found that developers sometimes use more powerful task constructs instead of the equivalent but simpler task constructs, even though they never use the extra power. Thus they make their code less readable and more verbose than it needs to be.

Q8: Are there constructs that developers commonly misuse? We found that for two of the fundamental parallel constructs, in at least 10% of the cases developers misuse them: the code runs sequentially instead of concurrently.

Our study has several practical implications. First, it is a tremendous resource for educating developers. The most common way to learn a new library is to study relevant examples of the API. Newcomers can start learning the APIs that are most widely used (see Q1 and Q3), and we can point them to the kinds of applications that are most likely to use the libraries (Q2). Newcomers should avoid common misuses (Q8) and constructs that unnecessarily increase the code complexity and the likelihood of errors (Q7). Our study also educates developers by showing real-world examples of parallel patterns (Q5).

Second, designers of these libraries can learn how to make the APIs easier to use (Q6). They can learn from observing which constructs do programmers embrace (Q3), and which ones are tedious to use or error-prone (Q8).

Third, researchers and tool vendors can focus their efforts on the constructs that are commonly used (Q3) or tedious or error-prone to use (Q8). For example, the refactoring community can decide which refactorings to automate. The testing and verification community can study the synchronization idioms that programmers use (Q4).

This paper makes the following contributions:

- To the best of our knowledge, this is the first empirical study to answer questions about parallel library usage on a large-scale, using semantic analysis.
- We present implications of our findings from the perspective of three different audiences: developers, library designers, and researchers.
- The tools and data are publicly available, as a tremendous education resource: `http://LearnParallelism.NET`

2. BACKGROUND

2.1 Parallel programming in .NET

We first give a brief introduction to parallel programming in .NET framework. The earlier versions provide the Threading library which contains many low-level constructs for building concurrent applications. `Thread` is the primary construct for encapsulating concurrent computation, and `ThreadPool` allows one to reuse threads. Synchronization constructs include three types: locks, signals, and non-blocking.

.NET 4.0 was enhanced with higher-level constructs. The new TPL library enables programmers to introduce task parallelism in their applications. `Parallel`, `Task`, and `TaskFactory` classes are the most important constructs in TPL.

`Task` is a lightweight thread-like entity that encapsulates an asynchronous operation. Using tasks instead of threads has many benefits [15] - not only are tasks more efficient, they also abstract away from the underlying hardware and the OS specific thread scheduler. `Task<>` is a generic class where the associated action returns a result; it essentially encapsulates the concept of a "Future" computation. `TaskFactory` creates and schedules tasks. Here is a fork/join task example from the *passwordgenerator* [28] application:

```
for (uint i = 0; i < tasks.Length; i++)
  tasks[i] = tf.StartNew(() => GeneratePassword(
       length, forceNumbers, ...), _cancellation.Token
       );
try{ Task.WaitAll(tasks, _cancellation.Token); } ...
```

The code creates and spawns several tasks stored in an array of tasks (the fork step), and then waits for all tasks to complete (the join step).

`Parallel` class supports parallel loops with `For` and `ForEach` methods, and structured fork-join tasks with `Invoke` method. The most basic parallel loop requires invoking `Parallel.For` with three arguments. Here is a usage example from the *ravendb* [30] application:

```
Parallel.For(0, 10, counter => {... ProcessTask(
     counter, database, table)} )
```

The first two arguments specify the iteration domain, and the third argument is a C# lambda function called for each iteration. TPL also provides more advanced variations of `Parallel.For`, useful in map/reduce computations.

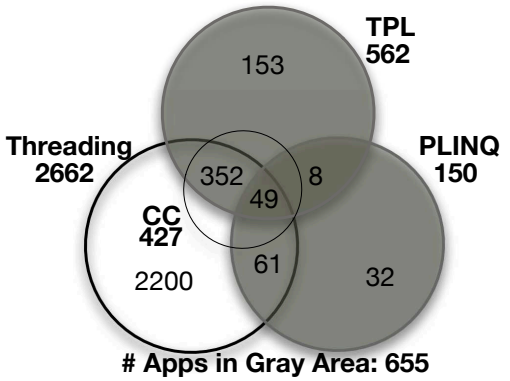
.NET also provides the CC library, which supports several thread-safe, scalable collections such as `ConcurrentDictionary`.

.NET 4.0 provides a fourth parallel library, the Parallel Language-Integrated Query (PLINQ) library, which supports a declarative programming style. PLINQ queries operate on `IEnumarable` objects by calling `AsParallel()`. Here is an example from the *AppVisum* [24] application:

```
assembly.GetTypes().AsParallel()
  .Where(t => t.IsSubclassOf(typeof(ControllerBase)))
  .Select(t => new ...)
  .ForAll(t => controllersCache.Add(t.Name, t.Type));
```

After the `AsParallel`, the data is partitioned to worker threads, and each worker thread executes in parallel the following `Where`, `Select`, and `ForAll`.

Figure 1: Number of applications that use Threading, TPL, PLINQ or CC libraries.



2.2 Roslyn

The Microsoft Visual Studio team has recently released Roslyn [31], as a community technology preview, with the goal to expose the compiler-as-a-service through APIs to other tools like code generation, analysis, and refactoring. Roslyn has components such as Syntax, Symbol Table, and Binding and Flow Analysis APIs.

The Syntax API allows one to parse the structure of a program. While a C# file can be syntactically analyzed in isolation, we cannot ask questions such as "what is the type of this variable". The type may be dependent on assembly references, namespace imports, or other code files. To further improve the analysis, we use the Symbol and Binding APIs to get semantic information such as type information, compiler options (e.g., targeting .NET 4.0). We used Syntax, Symbol and Binding APIs to parse our corpus data and statically analyze the usage of concurrent constructs.

3. METHODOLOGY

In this section we briefly describe the set of applications, the experimental setup, and the analysis infrastructure.

3.1 Corpus of Data

We analyze all open-source C# applications from two repositories, CodePlex [4] and Github [8]. We chose these two repositories because according to a recent study [19], most C# applications reside in these two repositories. Codeplex is Microsoft's code repository, and Github is now the most popular open source software repository, surpassing Google Code and SourceForge.

From these repositories, we want to filter those applications that use TPL, PLINQ, CC, and Threading libraries. For this, we implemented a tool, Collector. Next we explain how Collector works.

Collector downloaded all C# applications that contain at least one commit after April 2010, the release date of TPL and PLINQ. In the Git community, developers often fork an application and start making changes in their own copies. Sometimes, the main application might merge changes from the forked applications, but many times the forked applications start evolving independently. Collector ignores all forked applications. It also ignores the "toy applications", i.e., the ones that have less than 1000 non-comment, non-blank lines of code (SLOC). We discard such applications because many are just experimentally written by developers who learn a new construct, and they do not represent realistic usage of production code.

After eliminating applications that do not compile due to the missing libraries, incorrect configurations, etc, we had 7778 applications targetting .NET 4.0. From these, we want to select the applications that truly use the parallel libraries. For example, 648 applications imported the TPL library, but only 562 actually invoke functions from the TPL libraries. Thus, Collector removed the applications that import but never invoke any parallel library construct. Table 1 shows 2855 applications that truly use the parallel libraries.

Figure 1 shows that some applications use only one library, while other applications use these four libraries together. The TPL or PLINQ applications that also use Threading does not imply that these applications use threads. Threading library also provides synchronization constructs, and they are used in conjunct with TPL and PLINQ. The 2200 applications that only use the Threading library use multi-threading with explicit threads and thread pools. We excluded applications that use the Threading library to only insert delays and timers.

In the rest of the paper, we will focus on the applications that adopted the new parallel libraries, TPL and PLINQ. In this corpus, we also study the usage of Threading and CC. After all the filters, Collector retained 655 applications (shown within the gray area inside Fig. 1), comprising 17.6M SLOC, produced by 1609 developers. The only exception is our research question Q1 (the adoption of multi-threading), where we take into account all applications in Fig. 1.

We analyze all these 655 applications, without sampling, and these applications are from the most widely used C# repositories. This makes our findings representative.

3.2 Analysis Infrastructure

We implemented another tool, Analyzer, that performs the static analysis and gathers statistical usage data. We run Analyzer over each application from our corpus data. For each of these applications, Analyzer inspects the version from the main development trunk as of Jan 31st, 2012. The only exception is Q2 (the trends in adoption), where we analyze monthly code snapshots.

We implemented a specific analysis for each question using Roslyn's API. Since two projects in an application can share the same source file, Analyzer ensures that each source file is counted only once. Also, a .NET project can import system libraries in source format, so Analyzer ignores any classes that reside in the `System` namespace. This ensures that we are not studying the usage patterns in Microsoft's library code, but we study the usage only in the applications' code. When we discuss each empirical question, we present the static analysis that we used in order to collect the results.

4. RESULTS

Q1: Are developers embracing multi-threading?

As seen in Table 1, 37% of the 7778 applications use at least one of the four parallel libraries, which means they use some form of multi-threading. When we take into account only the category of large projects, 87% use multi-threading.

Why do programmers use multi-threading? Sometimes, multi-threaded code is a *functional* requirement. For example, an operating system with a graphical user interface must support concurrency in order to display more than one window at a time. Sometimes it is more convenient to write multi-threaded code even when it runs on a uniprocessor machine. For example, online transaction processing, reactive,

Table 1: Corpus Data

Type	Small (1K-10K)	Medium (10K-100K)	Large (>100K)	Total
# Applications compilable and targetting .NET 4.0	6020	1553	205	7778
# Multi-threaded Applications	1761	916	178	2855
# Applications adopted new libraries (TPL, PLINQ)	412	203	40	655

event-driven code is easier to express with threads. In such scenarios developers use multi-threading for *concurrency*.

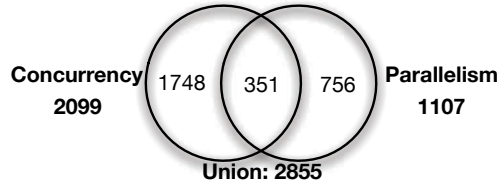
However, other times developers use multi-threading to improve a *non-functional* requirement such as performance. For this, they use multiple threads that run on multicore machine, thus they use multi-threading for *parallelism*.

Out of the applications that use multi-threading, 74% use it for concurrency and 39% use it for parallelism. Figure 2 shows the distribution. Some applications using multi-threading for both concurrency and parallelism

Next we manually analyzed the top 50 applications that highly use parallelism. We aim to find the killer applications for parallelism. We list their domain and how many applications we found from each domain: developer tools (7), data mining (7), multimedia (6), graphics (6), games (5), cloud computing (5), finance (3), database (3), networking (3), social media (2), office productivity (2), web server (1).

Program Analysis: To find whether an application uses multi-threading for concurrency or for parallelism, Analyzer first tabulates the usage of the multi-threading constructs (e.g., `Thread`, `Task`, `Parallel.For`, etc.) from each library in each application. Some constructs are clearly intended for concurrency (e.g., `FromAsync`, `TaskCompletionSource`, UI event dispatching thread) or for parallelism (e.g., `Parallel.For`, all PLINQ constructs). Other constructs (e.g., `Thread`, `Task`) can be used for either concurrency or parallelism. A typical usage scenario is to spawn threads in the iterations of a `for` loop. If the main thread waits for the child threads to finish, it means that the intent of the programmer is to have the threads execute *at the same time*, thus it is an example of parallelism. If the main thread does not wait for the child threads, it means that the intent is to have the threads *be in progress*, which is an example of concurrency. Thus, Analyzer checks whether the spawned constructs are waited or joined in the calling context.

Figure 2: Concurrency vs. Parallelism



Many applications have embraced multi-threading, however many of them use it for concurrency rather than parallelism.

Q2: How quickly do developers start using the new TPL & PLINQ libraries?

In the rest of the paper we move away from the applications that only use the Threading library and will focus on the 655 applications that adopted the new libraries (in the gray area in Fig. 1). Microsoft released the new libraries along with .NET 4.0 in April 2010. We want to find out how long it takes for developers to start using such libraries.

To analyze such adoption trends, from the set of 655 applications that eventually use TPL/PLINQ we select the subset of applications that exist in the repository as of April 2010. This subset comprises of 54 applications. If we had analyzed all TPL/PLINQ applications, regardless of their starting date, then as time goes by, we would see an increased number of constructs due to adding more applications.

For each of these 54 applications, we analyze monthly snapshots. In total, we analyze 31.9MLOC, comprising 694 different versions.

Figure 3 shows the number of applications that use at least one construct in each month. We split the 54 applications according to the size of their source code (small, medium, large). This prevents the trends in the small applications to obscure the trends in the larger applications (notice the different vertical scale in Fig 3). The results show that more applications are using the libraries as time goes by.

Figure 4 shows the average number of constructs per application. Here is an example of how we compute this number for the month of June 2010 for small applications. There are 24 constructs and 9 applications that use TPL at this time, so the average usage per application is $24/9 = 2.6$. In April 2010 the average usage for small and medium applications is not zero because these applications were using the "developer preview release" of the libraries.

Looking at both Fig. 3 and 4, we can notice a very different adoption rate among the three sizes of applications. If we look for the "tipping point" [9], i.e., the point in time when there is a major increase in the adoption rate (noticeable by a steep gradient of the slope), we can notice very different trends. The small applications are the early adopters of new libraries (2-3 months after the release), medium applications adopt around 4-5 months, and large applications are late adopters (8-9 months after the release).

Figures 3 and 4 show complementary data: the former shows that more applications are becoming parallel, whereas the latter shows that each application is becoming more parallel, i.e., it uses more parallel constructs.

Figure 5 shows the average number of Threading constructs per application does not decrease over time. This makes sense because most of the synchronization constructs are in the Threading library. Also, one can notice that compared with the TPL/PLINQ average density, Threading density is higher; this makes sense because the latter library has lower-level constructs.

Program Analysis: To find whether an application exists in April 2010, Collector looks at the creation date of each application, as listed in Github or Codeplex. After determining the set of 54 applications, our script checks out the source code snapshot for each month from April 2010 to February 2012. Then, for each snapshot, Analyzer collects usage details of TPL/PLINQ libraries. In the next question (Q3) we provide more information on how Analyzer collects usage details for one single snapshot.

Applications of different size adopt the new parallel libraries differently.

Figure 3: Number of (a) small-, (b) medium-, (c) large-size applications that use TPL/PLINQ

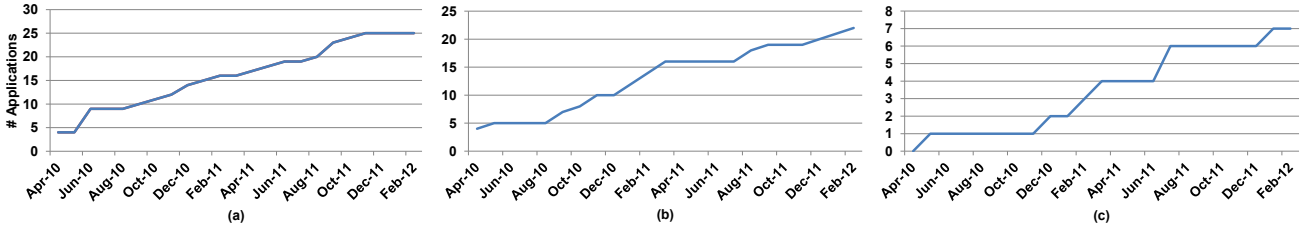
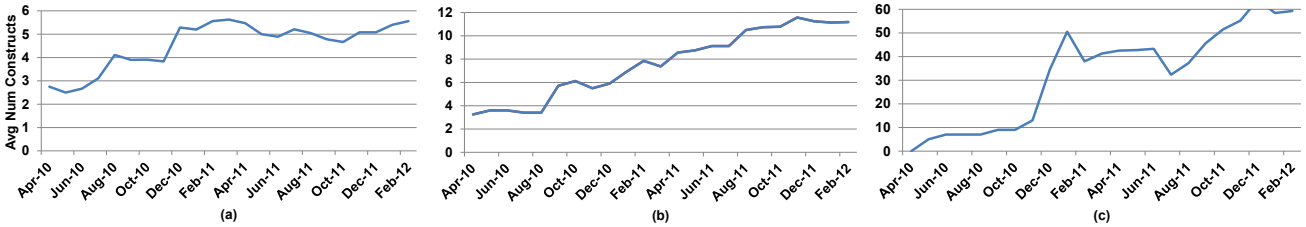


Figure 4: Average number of TPL/PLINQ constructs per application for (a) small-, (b) medium-, (c) large-size applications.



Q3: Which parallel constructs do developers use most often?

Table 2 tabulates the constructs that developers used most often from the TPL, Threading, PLINQ, and CC libraries. For example, lets drill down inside the TPL library and see the usage of class `Task`. Its methods account for 23% of all method call sites for the TPL library. One particular method, `Start`, has 243 call sites in 92 different applications. These call sites account for 18% of all call sites for methods from `Task` class.

Among these 4 libraries, they define 138 classes containing 1651 methods (counting constructors and overloaded methods). In table 2 we show combined usage for overloaded methods (e.g., we combine all 17 overloaded `StartNew` methods into one single method). Analyzer collects usage details for each of these methods. Due to the space limitations, we only tabulate the most used classes and methods for each library. The companion website [36] presents a complete fine-grained view.

The data shows that among the 1651 methods, some methods are used much more frequently than othes. For example, 10% of the methods are used 90% of the times. 1114 methods are never used. While similar trends are expected for any rich library APIs, it is important that we find the widely used APIs so that developers can focus on these.

We now discuss some of the findings for each library.

TPL: As shown in Table 2, `Parallel`, `Task`, and `TaskFactory` are the TPL classes most commonly used. When it comes to creating tasks, developers prefer to use the factory method `TaskFactory.StartNew` rather than invoking the task constructor. `Task<>` (i.e., the "Future" construct) is used nearly half as many times as `Task`.

Threading: `WaitHandle` is an abstract class for synchronization primitives, e.g., semaphore, mutex, so it is the second most popular class after `Thread`, the main class of the library.

Concurrent Collection: `ConcurrentDictionary`, a thread-safe implementation of `HashMap` is the most widely used.

Program Analysis: To accurately detect usage of a particular method, Analyzer needs type and binding information. Analyzer needs to know not only the name of the method, but also the type of the receiver object and the type of the arguments, and where does a method bind. This lets the analysis differentiate between `t.start()` when `t` is an instance of `Thread`, and the cases when `t` is an instance of a business class defined by the application. Because Analyzer uses the Symbol and Binding services of Roslyn, our reported usage numbers are 100% precise. Other empirical studies of library usage [1, 11, 35] have only used syntactical analysis, which can limit the accuracy of the results.

Parallel library usage follows a power-law distribution: 10% of the API methods account for 90% of the total usage.

Q4: How do developers protect accesses to shared variables?

Table 3 shows the type of synchronization, the name of the library constructs, how many times each construct was employed, and what is the usage frequency in comparison with other constructs within the same type of synchronization. Table 3 list *all* five kinds of synchronization constructs. `lock` and volatile accesses are language features, `Task.Wait` is a method of TPL, implicit synchronization constructs are from CC, and the rest of all is from Threading. To compute the number of implicit synchronization constructs, we sum the number of call sites for each API method that has implicit synchronization in its implementation. Notice that `lock` is by far the most dominant construct followed by `Volatile` accesses.

Program Analysis: To count one usage of a lock, Analyzer tries to match a pair of lock acquire and release operations. When one of the acquire or release operations is used more often than the other, we take the minimum number of these operations. Similarly, a pair of signal and wait operations count as one occurrence.

Finding accesses to volatile variables takes most of the analysis running time. Using the binding information, Analyzer looks up the definition of each accessed variable and field and checks whether it is `volatile` variable or field.

While locks are still very popular, developers use a wide variety of other synchronization constructs.

Figure 5: Average number of Threading constructs per application for (a) small-, (b) medium-, (c) large-size applications.

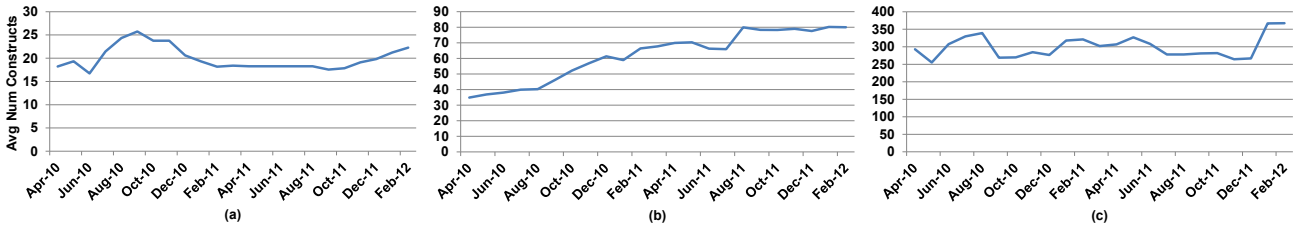
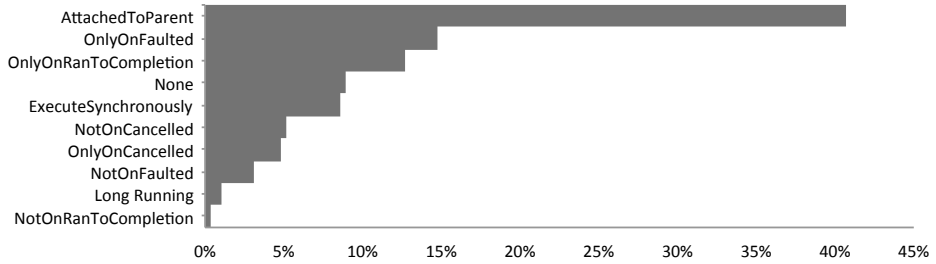


Figure 6: Distribution of Task Continuation Options



Q5: Which parallel patterns do developers embrace? Using the classification from the .NET Parallel Programming book [2], we analyzed the usage of six parallel patterns. Table 4 tabulates the usage of these patterns. The second column reports the popularity of task vs. data parallelism. The third column provides the names of patterns within each category, and the fourth column gives a brief explanation of the pattern. Last two columns show the number of individual instances of patterns, and the popularity percentage within its category.

Program Analysis: To automatically detect these patterns, we developed heuristics. We also randomly sampled from the inferred patterns to ensure that the reported patterns are inferred correctly. Because these patterns have several syntactical variations, it is very hard to detect all instances of patterns. Thus, the numbers that we report may be under-estimated, but not over-estimated.

For instance, to detect fork/join tasks pattern, Analyzer tries to match pairs of statements that create tasks and statements that wait for tasks completion. Our heuristic is to match such pairs intra-procedurally, not inter-procedurally. Although this heuristic correctly labels many cases, it fails to label a pattern that creates tasks in one method and waits for completion in another method.

Second, to detect data parallelism, Analyzer collects `Parallel.For`, `Parallel.ForEach` and `AsParallel` method calls. Since these method calls are perfect examples of data parallelism, we do not need to use heuristics. Loops that iterate over collections and launch a task to process each element are also counted by Analyzer as data parallelism.

Next we describe how Analyzer finds aggregation patterns. In a parallel aggregation pattern, the parallel loop uses unshared, local variables, that are combined at the end to compute the final result. Analyzer searches for `Parallel.ForEach` and `Parallel.For` method calls that use a `ThreadLocal` object as a parameter. This is the parameter that encapsulates the unshared variable. As for PLINQ's code, Analyzer checks whether the `AsParallel` method calls are followed by `Sum`, `Aggregate`, `Min`, etc. methods.

Finally, we illustrate how Analyzer detects tasks that dynamically spawn other tasks, e.g., in a recursive divide-and-conquer algorithm. Starting from a task's body, it analyzes the method invocations inside. If one of these invocations calls recursively the method which encapsulates the starting task, Analyzer labels it a dynamic task pattern.

> *Regular data parallelism is the most used parallel pattern in practice.*

Q6: Which advanced features do developers use? Now we focus on the most important parallel classes, `Parallel` and `Task`. Their methods take optional arguments related to performance and exception handling. Since these optional arguments distinguish TPL from other parallel libraries (e.g., TBB or Java's ForkJoinTask), we wonder if developers use them.

Parallel Class: `Parallel` class has `Invoke`, `For`, and `ForEach` methods. These methods can take an optional argument, `ParallelOptions`. With `ParallelOptions`, one can insert a cancellation token, limit the maximum concurrency, and specify a custom task scheduler. Of 852 method calls of `Parallel` class, only 3% use `ParallelOptions`.

Similarly, `For` and `ForEach` methods calls can take an optional `ParallelLoopState` which enables iterations to signal events (e.g., interrupt) to other iterations. Of 852 calls, only 3% use `ParallelLoopState`.

Task Class: When creating tasks, a developer can specify the execution order or the granularity of the task with an optional argument `TaskCreationOptions`. However, only 12% of task creation method calls use `TaskCreationOptions`.

Another advanced feature, `TaskContinuationOptions`, specifies the behavior for a task that is created as a continuation of another task. 28% of the continuation tasks use `TaskContinuationOptions`. Figure 6 tabulates the distribution of various continuation options.

Program Analysis: Because `TaskCreationOptions` and `TaskContinuosOperations` are enums, Analyzer also visits field accesses.

> *The advanced features and optional arguments are rarely used in practice.*

Table 2: Usage of TPL, Threading, PLINQ, and CC classes and their methods. The third column shows the percentage of usages of a class in comparison with usages of all classes from the library. The fourth column lists the main parallel methods in the parallel class. The fifth column shows the number of call sites for each method. The sixth column shows the percentage of usage of a method from one parallel class. The last column shows how many applications use this method.

Library	Class Name	% in Library	Method Name	# Call Sites	% in Class	# Apps
TPL	TaskFactory	30	StartNew	1256	72	286
			FromAsync	121	7	32
	Task	23	ContinueWith	372	28	122
			Wait	273	20	110
			Start	243	18	92
			Constructor	225	17	82
			WaitAll	172	13	91
	Parallel	14	For	450	53	102
			ForEach	365	43	133
			Invoke	37	4	23
	Task<TResult>	11	ContinueWith	536	86	113
			Constructor	85	14	40
Threading	Thread	17	Start	985	32	212
			Constructor	937	30	206
			Join	382	12	101
			Abort	294	10	82
	WaitHandle	11	WaitOne	1585	81	206
			Close	176	9	46
	Interlocked	10	CompareExchange	580	34	95
			CompareExchange	518	31	126
	ThreadPool	5	QueueUserWorkItem	814	90	125
PLINQ	ParallelEnumerable	100	AsParallel	221	24	150
			Select	136	15	46
			Where	62	7	30
			ForAll	61	7	29
CC	ConcurrentDictionary	72	Constructor	883	32	140
			TryGetValue	458	17	83
	ConcurrentQueue	13	Enqueue	194	38	63
			Constructor	178	35	70
	BlockingCollection	7	Add	85	30	25
			Constructor	78	28	25

Q7: Do developers make their parallel code unnecessarily complex? TPL provides some high-level constructs that allow developers to implement parallel code more concisely. These constructs decrease the number of lines of code and makes the parallel code easier to read, thus improving code quality.

Consider the example below, taken from *backgrounded* [25] application. It illustrates fork-join task parallelism.

The code on the bottom is the equivalent of the code on the top. It is much simpler to read because it uses `Parallel.Invoke`, a higher-level construct.

```
var runDaemons = new Task(RunDaemonJobs, ..token);
.....
var runScheduledJobs = new Task(RunScheduledJobs, ..
    token);
var tasks = new[] {runDaemons, ..., runScheduledJobs
    };
Array.ForEach(tasks, x => x.Start());
Task.WaitAll(tasks);
```

⇌

```
Parallel.Invoke(new ParallelOptions(CancellationToken
    =..token),
      RunDaemonJobs, ..., RunScheduledJobs);
```

Analyzer found that in 63 out of 268 regular fork/join task parallelism, the programmers could have used `Parallel.Invoke`, which would have reduced the complexity of the parallel code.

```
for (int i = 1; i <= threadCount; i++)
{
  var copy = i;
  var taskHandle = Task.Factory.StartNew(() =>
       DoInefficientInsert(server.Database.
       Configuration.ServerUrl, copy));
  tasks.Add(taskHandle);
}
Task.WaitAll(tasks);
```

⇌

```
Parallel.For(1,threadCount, (i)=> DoInefficientInsert
    (server.Database.Configuration.ServerUrl, i));
```

Analyzer found 189 `for/foreach` loops that launch tasks inside. Launching tasks inside a `for` loop is not only increasing the number of lines of code, but is also error-prone. In the code example above from *ravendb* [30], the programmer needs to make sure the iteration variable `i` is local to each task, otherwise the reading/writing accesses would exhibit data-races. 55 out of 189 cases could have used `Parallel.For` or `Parallel.ForEach`.

Table 3: Usage of Synchronization Constructs

Type	% in Types	Name	#	% in Type	# Apps
Locking	39	lock (language feature)	6643	89	361
		ReaderWriterLockSlim	258	3	68
		Monitor - Enter/Exit	245	3	66
		Mutex	94	1	46
		Semaphore	75	1	23
		ReaderWriterLock	65	1	24
		SpinLock	31	0.4	11
		SemaphoreSlim	20	0.3	10
Non-Blocking	26	Volatile Accesses	3212	65	152
		Interlocked Methods	1696	34	126
		Thread.MemoryBarrier	50	1	15
Implicit	21	CC Operations	4021	100	283
Signaling	9	ManualResetEvent	671	38	150
		AutoResetEvent	647	37	102
		Monitor - Wait/Pulse	168	10	31
		ManualResetEventSlim	167	10	37
		CountdownEvent	58	3	9
		Barrier	33	2	6
Blocking	5	Thread.Join	382	38	101
		Thread.Sleep	350	35	132
		Task.Wait	273	27	110

Table 4: Usage of Parallelism Patterns.

Main Pattern	%	Pattern Name	Brief explaination	#	%
Data Parallelism	68	Regular	parallel loops with `For, ForEach,` and PLINQ	954	92
		Aggregation	parallel dependent loops (map reduce algorithms)	82	8
Task Parallelism	32	Regular	regular fork&join tasks	268	56
		Futures	task dependency on results	155	32
		Pipeline	assembly line parallelism with `BlockingCollection`	41	8
		Dynamic	dynamically created tasks	18	4

There might be many other patterns of accidental complexity. We focused on two of them based on our own observations and discussions with the library designers.

Program Analysis: To detect tasks that could have used the `Parallel.Invoke`, Analyzer filters those tasks that are created and are also waited upon immediately.c More precisely, Analyzer checks that the main thread does not execute other statements between the statements that create and wait for tasks. It also checks that there are no dependencies among the created tasks, e.g., tasks are not linked with continuations like `ContinueWith`. In addition, Analyzer also discards the fork-join tasks that use `TaskCreationOptions` since `Parallel.Invoke` does not provide such a feature.

Despite the fact that parallel programs are already complex, developers make them even more complex than they need to be.

Q8: Are there constructs that developers commonly misuse?

`Parallel.Invoke(params action)` is a construct that executes in parallel the actions passed as arguments. It is a fork-join with blocking semantics: the main thread will wait until all actions specified as arguments have finished. Our analysis found that 11% of all usages of `Parallel.Invoke` take one action parameter in different applications. Consider the example from the *gpxviewer* [27] application:

```
Parallel.Invoke(() => i.ImportGPX(null, GPXFile));
```

Notice that in this case there is only one single action to be performed, and the main thread will block until this action has finished. In this case, the parallelism has no effect: the code executes sequentially, `ImportGPX` followed by the main thread. Developers might erroneously believe that `ImportGPX` will execute in parallel with the main thread, when in fact it doesn't.

When we look at PLINQ code, the `AsParallel` method converts an `Enumerable` into an `ParallelEnumerable` collection. Any method called on such a parallel enumeration will execute in parallel. We found 27 cases in 19 applications (representing 12% of all `AsParallel` usages) where developers misuse a parallel enumeration as the iteration source of a sequential `for` or `foreach` loop. Consider the example from the *profit* [29] application:

```
foreach (var module in Modules.AsParallel())
    module.Refresh();
```

Notice that despite `AsParallel` being placed at the end of the `Modules` collection, there is no operation performed on the "parallel" `Modules`. The `foreach` proceeds sequentially. Developers might erroneously believe that the code runs in parallel, when in fact it runs sequentially.

Program Analysis: To answer misusage questions, ANALYZER encodes the erroneous usage patterns. For example, it searches for calls to `Parallel.Invoke` with one single argument, where the argument is an `Action` object (e.g., a method name or a lambda expression). For the PLINQ misusage, ANALYZER searches for expressions where `AsParallel` is the last subexpression. We then manually analyze whether it is present in `for` or `foreach` loop whose iteration does not create any threads.

Misuse of parallel constructs can lead to code with parallel syntax but sequential execution.

5. IMPLICATIONS

There are several implications of our study. We organize them based on the community for which they are relevant.

5.1 Developers

Q1 (adoption): Becoming proficient with a new programming model requires a long-term commitment. Developers without parallel programming experience might ask themselves: should we learn how to use parallel libraries, or should we avoid them because they are a passing fad. Our data shows that 37% of all applications use the multi-threaded paradigm, so many developers will not be able to completely avoid multi-threaded programming. Sooner or later, most programmers will have to become familiar with this model.

Q2 (trends in adoption): Learning how to use effectively a library requires studying examples of the library API in real code. Where can developers find such examples? Our data shows that smaller applications are the early adopters of the parallel libraries. In addition, these applications have a much higher density of parallel constructs per thousand of SLOC. Looking in Fig 4, we can divide the average number of parallel constructs by 1K, 10K, 100K for small, medium, and large applications respectively. The average density is 5‰, 1.2‰, and .6‰ respectively. When taking into account the effort to understand unknown code, developers are better off looking for examples in small applications.

Q3 (usage): We notice a power-law distribution: 10% of the API methods are responsible for 90% of all usages. If we look at the classes, 15% of classes are responsible for 85% of all usages. This is good news for developers who are just learning parallel libraries: they can focus on learning a relatively small subset of the library APIs and still be able to master a large number of parallelism scenarios.

5.2 Library Designers

Q3 (usage): Surprisingly lower usage numbers like the ones for PLINQ can highlight the APIs that need better documentation and more advertisement on mailing lists, developer forums, etc.

Q4 (synchronization): Designers of concurrent data structures and synchronization constructs are always asking themselves on what to focus. Table 3 shows that developers are more likely to use the faster synchronization constructs. For example, `ReaderWriterLockSlim` is used four times more often than the slower `ReaderWriterLock`.

Q6 (advanced features): Library designers pay special attention to making the APIs easier to use. This involves making the syntax for the common case more concise. We observed in Figure 6 that programmers prefer to create new tasks attached to the parent task (40% are `AttachedToParent`). So, library designers could make this the default behavior for nested tasks. Similarly, 80% of times when developers used `ParallelOptions` they only specify one single option, `MaxDegreeOfParallelism`. Library designers may make this an argument to `Parallel` class methods instead of encapsulating it in `ParallelOptions`.

Additionally, 60% of the times developers overwrite `MaxDegreeOfParallelism`; they make it equal with the number of processors found at runtime. This means that developers are not happy about the degree of parallelism chosen by .NET. TPL architects should consider making the number of processors the default value for the max degree of parallelism. Stephen Toub, who is one of the main architects of TPL, confirmed our suggestion.

Q8 (misusage): Library designers can also remove the constructs that are error-prone. We found that developers are not aware that `Parallel.Invoke` is a blocking operation, so they invoke it with one single action parameter (which results in executing the code sequentially). Library designers may consider removing `Parallel.Invoke` version that takes only one action parameter.

5.3 Researchers

Q1 (adoption): Since we list the domains and the applications that use parallelism most heavily, the researchers can use them to create benchmarks for parallel programming.

Q4 (synchronization): Researchers that work on ensuring correctness (e.g., data-race detection) should notice from Table 3 that developers use a wide variety of synchronization constructs. Thus, data-race detectors should also model these other synchronization constructs.

.NET parallel libraries provide more than 20 synchronization constructs divided into 5 different categories. It is difficult for developers to select the most appropriate one. Each construct has tradeoffs, depending on the context where it is used. This is an opportunity for developing intelligent tools that suggest which constructs developers should use in a particular context.

Q7 (complexity): Researchers in the refactoring community can get a wealth of information from the usage patterns. For example, developers should use higher-level constructs to manage the complexity of the parallel code: 24% of fork-join tasks can be converted to `Parallel.Invoke`, which reduces many lines of code. Refactorings that allow programmers to improve the readability of their parallel code have never been automated before, but are invaluable.

6. THREATS TO VALIDITY

Construct: Are we asking the right questions? We are interested to asses the state of the practice w.r.t. usage of parallel libraries, so we think our questions provide a unique insight and value for different stakeholders: potential users of the library, designers of the library, researchers.

Internal: Is there something inherent to how we collect and analyze the usage that could skew the accuracy of our results? Microsoft's Roslyn, on which we built our program analysis, is now in the Community Technology Preview and has known issues (we also discovered and reported new bugs). For some AST nodes, we did not get semantic information. We printed these nodes, and they are not parallel constructs, thus they do not affect the accuracy.

Second, the study is only focusing on static usage of parallel constructs, but one use of a construct (i.e., a call site) could correspond to a large percentage of execution time, making it a very parallel program. Likewise, the opposite could be true. However, we are interested in the developer's view of writing, understanding, maintaining, evolving the code, not on the performance tools' view of the code (i.e., how much of the total running time is spent in multithreaded code). For our purpose, a static usage is much more appropriate.

Third, do the large applications shadow the usage of constructs in the smaller applications? Tables 2 and 3 provide the total tally of constructs across all applications and there is a possibility that most usages come from a few large applications. To eliminate this concern, the last column in the two tables list the number of applications that use each kind of construct. Due to lack of space, we do not present the mean, max, min, standard deviation in the paper, but they are available on the companion website [36].

Fourth, static analysis offers limited insight in the performance of parallel applications. While the real purpose of using parallel libraries is to improve performance, we can not estimate this based solely on static analysis.

External: Are the results generalizable to other programming languages, libraries, and applications? First, despite the fact that our corpus contains only open-source applications, the 655 applications span a wide range from tools, IDEs, games, databases, image processing, video encoding/decoding, search engines, web systems, etc., to third party libraries. They are developed by different teams with 1609 contributors from a large and varied community. Still, we cannot be sure whether this usage is representative for proprietary applications.

While we answer the questions for the C# ecosystem, we expect they can cross the boundary from C# to Java and C++. For example, we expect such empirical studies that reveal pain-points and common errors in using parallel library APIs to be useful to the TBB/C++ and `j.u.c.`/Java designers since these libraries provide very similar abstractions. Furthermore, C# with .NET is used on wide range of platforms – desktop, server, mobile, and web applications.

Reliability: Can others replicate our study? A detailed description of our results with fine-grained reports and analysis tools are available online [36].

7. RELATED WORK

There are several empirical studies [1, 11, 13, 35] on the usage of libraries or programing language features. These studies rely only on syntactic analysis. To best of our knowledge, ours is the first large-scale study that uses both syntactic and semantic analysis, thus increasing the accuracy of the usage statistics.

Robillard and DeLine [32] study what makes large APIs hard to learn and conclude that one of the important factors is the lack of usage examples. Our current study provides lots of usage examples from real code which can hopefully educate newcomers to the parallel library.

Monperrus et al. [18] study the API documentation of several libraries and propose a set of 23 guidelines for writing effective API documentation.

Dig et al. [7] and Pankratius et al. [21] analyzed concurrency-related transformations in a few Java applications. Our current study does not look at the evolution of concurrent applications, but at how developers use parallel libraries.

Pankratius [20] proposes to evaluate the usability of parallel language constructs by extending the Eclipse IDE to record usage patterns and then infer correlations using data mining techniques.

Other empirical studies on the practice of multicore programming [5] focused on identifying the contented resources (e.g., shared cache) that adversely impact the parallel performance. Our fourth research question identifies a wide variety of synchronization constructs that impact performance.

In the same spirit like our paper, Parnin et al. [22] study the adoption patterns of Java generics in open-source applications. While some of our research questions specifically address adoption patterns (Q1 and Q2), the remaining questions provide an extensive exploration into the practice of using parallel libraries.

Others [12] have studied the correlation between usage of the MPI parallel library and productivity of the developers.

The closest work to ours is done by Weslley et al. [35] on the usage of concurrent programming constructs in Java. They study around 2,000 applications and give some coarse-grain usage results like the number of synchronized blocks and the number of classes extending `Thread`. In contrast, our study looks at every parallel construct in the parallel libraries, and we also look at how these constructs form patterns and structures. Although they analyze the usage of very few constructs, their results are not accurate due to missing type information because they only perform lexical analysis. Also, their count of the constructs' usage can be misleading. For example, they measure the usage of `java.util.concurrent` by counting statements that import the library. In our study, there are many applications that import TPL but never invoke any construct. For example, there is an application, *DotNetWebToolkit* [26], that imports TPL 111 times but invokes TPL just once.

8. CONCLUSION

Parallelism is not a passing fad; it is here for the foreseeable future. To encourage more programmers to embrace parallelism, we must understand how parallel libraries are currently used. Our empirical study on the usage of modern parallel libraries reveals that programmers are already embracing the new programming models. Our study provides tremendous education value for developers who can educate themselves on how to correctly use the new parallel constructs. It also provides insights into the state of the practice in using these constructs, i.e., which constructs developers find tedious and error-prone. Armed with this information, library designers and researchers can develop effective tools and techniques to better match the current practice and transform it.

More studies are needed if we want to fully understand the state of the practice, and we hope that our study inspires follow-up studies.

Acknowledgements: This research was partially funded by Microsoft through an SEIF award, and by Intel through a gift grant. The authors would like to thank Stephen Toub, Kevin Pilch-Bisson, Shan Lu, Mark Grechanik, Michael Hind, Stas Negara, Cosmin Radoi, Yu Lin, Phil Miller, Milos Gligoric, Samira Tasharofi, Mohsen Vakilian, Yun Young Lee, Darko Marinov, and anonymous reviewers for providing helpful feedback on earlier drafts of this paper.

9. REFERENCES

[1] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: the case of smalltalk. In *MSR '11: Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 23–32, 2011.

[2] C. Campbell, R. Johnson, A. Miller, and S. Toub. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 2010.

[3] Collections.Concurrent (CC). July'12, `http://msdn.microsoft.com/en-us/library/dd997305.aspx/`.

[4] CodePlex. July'12, `http://codeplex.com`.

[5] T. Dey, Wei Wang, J.W. Davidson, and M.L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 76–86, 2011.

[6] D. Dig. A refactoring approach to parallelism. *Software, IEEE*, 28(1):17–22, 2011.

[7] D. Dig, J. Marrero, and M. D. Ernst. How do programs become more concurrent? a story of program transformations. In *IWMSE '11: Proceedings of the 4th International Workshop on Multicore Software Engineering*, pages 43–50, 2011.

[8] Github. July'12, `https://github.com`.

[9] M. Gladwell. *The Tipping Point: How Little Things Can Make a Big Difference*. Back Bay Books, 2002.

[10] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[11] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale java open source code repository. In *ESEM '10: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2010.

[12] L. Hochstein, F. Shull, and L. B. Reid. The role of mpi in development time: a case study. In *SC Conference*, pages 1–10, 2008.

[13] S. Karus and H. Gall. A study of language usage evolution in open source software. In *MSR '11: Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 13–22, 2011.

[14] D. Lea. *Concurrent Programming in Java‡: Design Principles and Pattern*. Prentice Hall, 1999.

[15] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *ACM SIGPLAN Not.*, 44(10):227–242, 2009.

[16] B. P. Lester. *The Art of Parallel Programming*. 1st World Publishing, Inc., 2006.

[17] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2005.

[18] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, Online Edition, 2011.

[19] Survival of the Forgest. July'12, `http://redmonk.com/sograd y/2011/06/02/blackduck-webinar/`.

[20] V. Pankratius. Automated usability evaluation of parallel programming constructs. In *ICSE '11 (NIER track): Proceedings of the 33rd International Conference on Software Engineering*, pages 936–939, 2011.

[21] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 53–60, 2008.

[22] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *MSR '11: Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 3–12, 2011.

[23] Parallel Language Integrated Query (PLINQ). July'12, `http://msdn.microsoft.com/en-us/library/dd460688.aspx/`.

[24] AppVisum Project. July'12, `https://github.com/Alxandr/AppVisum.Sys`.

[25] Backgrounded Project. July'12, `http://www.github.com/swedishkid/backgrounded`.

[26] DotNetWebToolkit Project. July'12 `https://github.com/chrisdunelm/DotNetWebToolkit`.

[27] Gpxviewer Project. July'12, `https://github.com/andrewgee/gpxviewer`.

[28] PasswordGenerator Project. July'12, `https://github.com/PanosSakkos/PasswordGenerator`.

[29] Profit Project. July'12, `http://profit.codeplex.com/`.

[30] Ravendb Project. July'12, `https://github.com/ravendb/ravendb`.

[31] The Roslyn Project. July'12, `http://msdn.microsoft.com/en-us/hh500769`.

[32] Martin P. Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[33] System.Threading. July'12, `http://msdn.microsoft.com/en-us/library/system.threading`.

[34] Threading Building Block (TBB). July'12, `http://threadingbuildingblocks.org/`.

[35] W. Torres, G. Pinto, B. Fernandes, J. P. Oliveira, F. A. Ximenes, and F. Castor. Are java programmers transitioning to multicore?: a large scale study of java floss. In *SPLASH '11 Workshops*, pages 123–128, 2011.

[36] Companion TPL usage data. July'12, `http://learnparallelism.net`.

[37] ForkJoinTask Doug Lea's Workstation. July'12, `http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/ForkJoinTask.html`.