An Empirical Study on Parallelism in Modern Open-Source Projects

Marc Kiefer
Karlsruhe Institute of Technology
kiefer@kit.edu

Daniel Warzel
Karlsruhe Institute of Technology
daniel.warzel@student.kit.edu

Walter Tichy
Karlsruhe Institute of Technology
tichy@kit.edu

Abstract

Writing parallel programs is hard, especially for inexperienced programmers. Parallel language features are still being added on a regular basis to most modern object-oriented languages and this trend is likely to continue. Being able to support developers with tools for writing and optimizing parallel programs requires a deep understanding of how programmers approach and implement parallelism.

We present an empirical study of 135 parallel open-source projects in Java, C# and C++ ranging from small (<1000 lines of code) to very large (>2M lines of code) codebases. We examine the projects to find out how language features, synchronization mechanisms, parallel data structures and libraries are used by developers to express parallelism. We also determine which common parallel patterns are used and how the implemented solutions compare to typical textbook advice.

The results show that similar parallel constructs are used equally often across languages, but usage also heavily depends on how easy to use a certain language feature is. Patterns that do not map well to a language are much rarer compared to other languages. Bad practices are prevalent in hobby projects but also occur in larger projects.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

Keywords study, parallelism, object orientation, languages, open source

1. Introduction

Modern software needs to make use of parallelism to leverage the full computing power of current processors. Parallel programming requires considerably more skills than sequential programming since it introduces an additional layer of complexity and even new types of errors. [1, 17] The potential for deadlocks and data races makes parallel programming more error prone. Although the demand for parallelism has been rising in the last few years, most programmers still lack expertise in parallel software development.[16]

Textbooks offer advice and best practices on how to implement parallelism[3, 6, 12], but popular object-oriented languages like *Java*, *C#* and *C++* sometimes do not offer advanced parallel constructs or synchronization primitives required to follow the advice. Low-level parallelization and synchronization primitives have been in object-oriented languages for some time now, but high level constructs are still being added bit by bit to mainstream languages (for example, Java has no simple parallel for loop). High-level constructs simplify code and make programming less error prone, while also increasing readability and maintainability. C++ was lacking native parallel support for a long time and was heavily reliant on libraries to provide the desired functionality.

With different sets of parallel constructs in different languages and textbook advice for typical parallel problems, which is often far from reality, we conducted a study on how programmers implement parallelism and which language features and libraries they use to tackle parallel problems.

2. Motivation and Research Questions

Lots of legacy code is still strictly sequential. With the inevitable advancement of multicore processors some of this code and much of what gets written from scratch today needs to be parallel. More and more code already contains some kind of parallelism, but does not speed up the application to its full potential. In many cases, these refactoring efforts just mean simple loop parallelism or a bunch of threads handling independent tasks.

The research community is trying to support developers in their struggle to implement (possibly complex) parallelism[2]. Tools for automated or semi-automated parallelization exist [4, 8, 11, 14, 18], but in order to improve

SEPS '15, Oct 27, 2015, Pittsburgh, PA, United States.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

these partially parallelized programs, we want to understand better how programmers actually implement parallelism.

To correctly detect, recognize and handle parallelism, we need to have an idea of how developers express parallelism in their code. Introducing additional parallelism as well as improving existing parallel regions requires a deep understanding of what the programmer intended, i.e. what constructs and patterns were implemented. Depending on the general tendency of programmers to use more low-level synchronisation constructs or high-level parallel patterns, new tools for parallelization support can be focused on the actual demand of either high- or low-level constructs. To gain insight into programmers habits of introducing parallelism, we want to answer the following research questions:

RQ1: Which language features, synchronization mechanisms, parallel data structures and libraries are used by developers to express parallelism? All three languages (Java, C#, and C++) have different means of expressing parallelism. Almost every new version of each language brought new parallel features and this trend is likely to continue. While Java and C# had more regular updates and therefore more parallel constructs added in shorter periods of time, C++ had parallel support through various external libraries (PThreads[13], OpenMP[5], Boost[7], Intel TBB[15], ...).

The features, synchronization mechanisms and parallel data structures available in each language vary wildly. While simple synchronization mechanisms are based on the same basic concepts such as locks and condition variables, they appear in different variants between languages. Because their interfaces are different, some of them are easier to understand and use than others.

The same applies to parallel data structures. Thread-safe containers and atomic basic types are available in different varieties ranging from generic containers to very specific ones and even wrappers to synchronize accesses to other non thread-safe containers.

Libraries also play an important role, especially for C++, because they can provide functionality that is not (yet) available in the standard library of a language. C++ probably benefits most from external libraries, as important parallel features were not part of the language until recently, but there are also libraries for C# and Java that provide missing features.

Because there is such a complex integration of parallel features and primitives between languages, we are interested in which of these features, primitives and possibilities are actually used by programmers to implement parallelism.

RQ2: Which high- or low-level parallel patterns are used? Parallel patterns provide solutions to recurring problems in a parallel programming context. They have been crafted by experts to make life easier for less experienced parallel programmers. These patterns come in great variety from generic ones to very specific ones. Some of them can be applied at a high level of abstraction (master-worker, pipeline,...) while others are better suited at the low level (fork/join, parallel loop, ...).

Java	C#	C++
Cassandra	OpenSimulator	Thunderbird
Neo4j	StockSharp	OpenH264
OpenML	Azure Power Shell	LLVM
Consulo IDE	Smuxi	Libre Office
JetBrains MPS	OpenRA	MySQL

Table 1. Application examples for each language

We would like to know which parallel patterns are used in practice and how they are distributed among the languages. Some patterns might be easier to implement in a certain language and might therefore be used more often.

RQ3: How do programmers' solutions to common parallel problems compare to textbook solutions? When parallelizing existing software or writing parallel programs from scratch, programmers face the same problems over and over again. There are limits to the number of sensible ways of parallelizing a piece of code. Textbooks cover most of them and offer advice on transferring sequential code into a parallel version. In reality, sequential problems do not look quite like the idealized problems in textbooks.

Data dependencies and structural differences sometimes prevent easy solutions or require non-trivial refactoring. For non-expert parallel programmers, it often is not obvious which of these approaches can or cannot be applied to their code. Less experienced programmers may also not be aware of these proven approaches and try to come up with their own solutions. This is why we want to find out how often these guidelines are used in reality, which ones are used heavily (if any), and how close real solutions are to their textbook counterparts.

3. Methodology

Our goal is to find out how programmers implement parallelism in *Java*, *C#* and *C++*.

We studied 135 open source projects, consisting of 46 *Java* projects, 45 *C#* projects and 44 *C++* projects. The source code files were obtained from the major open source code repositories SourceForge and GitHub as well as self-hosted project pages (e.g.*apache.org*). We tried to cover a broad spectrum of projects from small hobby projects (<1000 lines of code) to large software projects (>2M lines of code) with several professional full-time developers. The applications come from different domains including research, graphics, computer vision, databases, emulators, compilers, office and games. Table 1 lists some of the more popular projects for each language.

The criteria a project had to meet to be included in our corpus were the following: 1) The project had to contain some kind of parallelism. Since the project repositories give no indication if a project contains parallel regions, we had to search the source code for evidence and filter out projects which did not make use of parallelism. 2) We only wanted projects that were actively developed, so we ignored projects without a commit in the last six months. To be able to look at the current state of open source projects, projects that were not actively developed would not be helpful and the chance of them using the latest parallel language features was minimal.

To gather our data, we first programmatically searched the code with a set of keywords specific to each language to find the code locations where parallel instructions might be found. These keywords included all the language features which are used to implement parallelism, as well as parallel containers and popular parallel libraries. We also searched code comments to find hints for manual implementations of parallel patterns or synchronization primitives. Some projects use wrappers around parallel language features, so we also included them in project specific keyword lists. Once we had our compiled list of code locations, we manually inspected the code to confirm the findings and look for possible high- or low-level patterns.

Our corpus consists of 52221 Java, 49087 C# and 20187 C++ source code files. Our search found parallel code locations in 5483 Java, 4900 C# and 2117 C++ files. So, parallel regions are included in roughly 10% of all source code files, which is also true for each of the three languages individually.

4. Results

We present the results based around our research questions and focus on the interesting parts of the data we gathered. We also found some interesting facts not directly related to the research questions which will be presented at appropriate locations.

The first thing we noticed while we collected projects for the corpus was that parallelism was a lot less common in C++ project than in C# or Java. We had to filter out many more C++ projects in comparison to the other languages because they did not match our criterion of containing parallelism. This is probably due to the fact that parallel features have been introduced much earlier into the standard libraries of Java and C# than C++. This can also be seen by the fact that Java and C# projects barely use external libraries to implement parallelism, whereas in C++ libraries are the most common way of expressing parallelism.

From the raw numbers of classes and features supporting parallelism for each language, C# and Java are pretty much on par, while C++ places a distant third. The recent additions to the standard and the available libraries helped in that regard, but the more low-level nature of the language is clearly visible by the absence of convenient high-level parallel constructs. Concerning thread-safe containers, Java is the clear winner. It hast three times the number of containers as C#, while C++ only supports thread-safe containers through third-party libraries.

4.1 RQ1: Which language features, synchronization mechanisms, parallel data structures and libraries are used by developers to express parallelism?

This research question investigates to which extent developers make use of the available parallel features in each language.

4.1.1 High-level Features

Features and language constructs in this work are considered high-level when they provide a layer of abstraction and convenience above the low-level work of manually handling threads and synchronization. A typical example would be the `Parallel.For` loop in C#, which completely hides the complexity of thread management and synchronization. Since `Delayed` in Java or `Task` in C# are used by many of the high-level constructs, they are also regarded relevant here.

Tables 2 and 3 contain the high-level constructs for Java and C# with the number of projects in which they appear as well as the percentage of total projects (for each language) this corresponds to. The arguably closest counterpart to these high-level patterns in C++ is OpenMP, which is supported by most modern compilers, although technically it is a library. The results can be found in Table 4.

In Java the most frequently used feature is `Executors`. Two thirds of all the projects we examined use them to avoid managing `Threads` and `ThreadPools` manually. The next three in line are `ExecutorService`, `Callable` and `Future`, which go hand in hand with the `Executor`. Further down in the ranking are some more classes related to `Executors`, which leads to the conclusion that this is the prevalent method of implementing task parallelism in Java. Another heavily used concept is that of `BlockingQueue` and its variants, which often gets used to implement producer-consumer patterns. It is a buffer which blocks take requests when the queue is empty as well as put requests when the queue is full to regulate the produce/consume ratio. Features that are not used are `ForkJoinTask`, `ForkJoinWorkerThread` and `RunnableScheduledFuture`. Those are specialized classes and interfaces; it is not surprising that they do not get used. `ForkJoinTask` essentially is a lightweight `Future`, which has specialized subclasses `RecursiveAction` and `RecursiveTask`, which do get used in some projects.

When it comes to C#, there are fewer classes and predefined specializations available compared to Java. But they cover the same aspects of task parallelism. Additionally, C# offers a simple way of implementing data parallelism, which is lacking in Java. A `Task` is a lightweight version of a thread, but it is considered a high-level construct since it also supports the functionality of a Future. It is the most frequently used parallel class in our C#

Feature	# projects	% of total
Executor	31	67.39%
ExecutorService	28	60.87%
Callable	24	52.17%
Future	23	50.00%
BlockingQueue	20	43.48%
ScheduledExecutorService	16	34.78%
ThreadPoolExecutor	15	32.61%
ArrayBlockingQueue	13	28.26%
LinkedBlockingQueue	13	28.26%
ScheduledFuture	11	23.91%
FutureTask	8	17.39%
ScheduledThreadPoolExecutor	8	17.39%
PriorityBlockingQueue	3	6.52%
CompletionService	2	4.35%
AbstractExecutorService	2	4.35%
Delayed	2	4.35%
DelayQueue	2	4.35%
ExecutorCompletionService	2	4.35%
ForkJoinPool	2	4.35%
RecursiveAction	2	4.35%
RunnableFuture	2	4.35%
SynchronousQueue	2	4.35%
BlockingDeque	1	2.17%
LinkedBlockingDeque	1	2.17%
LinkedTransferQueue	1	2.17%
RecursiveTask	1	2.17%
TransferQueue	1	2.17%
ForkJoinTask	0	0.00%
ForkJoinWorkerThread	0	0.00%
RunnableScheduledFuture	0	0.00%

Table 2. Usage of high-level constructs in Java

corpus. ThreadPool takes second place, which is probably attributable to its variety of convenience features. A TaskScheduler allows fine control over the scheduling policy in a ThreadPool. Their usage is roughly consistent with the usage numbers of their correspondig Java counterparts. The BlockingCollection however, which is similar to BlockingQueue in Java, gets much less use with only 6 out of 45 projects. With Parallel.For and Parallel.ForEach C# also offers data parallelism constructs, which are used in 13.33% of all projects. Parallel.Invoke offers a fork/join mechanism, which is only used in one project. An interesting concept in C# is the Partitioner class, which simplifies partitioning of arrays, lists and enumerables into chunks for parallel processing. It can also be used together with Parallel.ForEach to create application-specific data partitioning schemes. Surprisingly, it is never used in our corpus.

C++ only offers a single one of the features discussed with Java and C#, namely the future. As a fairly new feature introduced in 2011, it is used in only 3 out of 44 projects. The related shared_future as well as packaged_task, which is comparable to Callable in Java (top 3 in the rankings), are nowhere to be found in our corpus. Since OpenMP is a common substitute for the missing data parallel constructs and most compilers support it natively, we take a look at the usage of these pragmas. The most common feature is #pragma omp parallel for, which corresponds nicely to the Parallel.For numbers seen in the C# projects. #pragma omp parallel is also used in the G'MIC and OpenCV projects to unconditionally execute code on all available OpenMP threads. The reduction clause did not appear in any of the projects.

Feature	# projects	% of total
Task	33	73.33%
ThreadPool	19	42.22%
TaskScheduler	10	22.22%
Parallel.For	8	17.78%
BlockingCollection	6	13.33%
Parallel.ForEach	6	13.33%
TaskFactory	5	11.11%
Parallel.Invoke	1	2.22%
Partitioner	0	0.00%

Table 3. Usage of high-level constructs in C#

Feature	# projects	% of total
#pragma omp parallel for	6	13.64%
future/promise	3	6.82%
#pragma omp parallel	2	4.55%
packaged_task	0	0.00%
shared_future	0	0.00%

Table 4. Usage of high-level constructs in C++

The use of high-level parallel constructs in each language obviously depend on the availability of language features. While Java focuses more on task parallelism and C++ more on data parallelism through OpenMP, C# offers support for both. Yet it is interesting to see that the use percentages are in the same range for the thread pool constructs and blocking collections between C# and Java as well as the data parallel constructs between C# and C++.

4.1.2 Synchronization Mechanisms

Regarding the basic building blocks for parallel applications, we looked at the available synchronization mechanisms for each language. Again Java and C# are pretty much on par covering basic primitives as well as some more specialized and convenient constructs. C++ is only covering the basics.

Starting with Java, synchronized is by far the most used synchronization mechanism (87%), as can be seen in Table 5. Although it is a coarse grained mechanism (especially when used for whole methods, which is the major use

Feature	# projects	% of total
synchronized	40	86,96%
Lock	19	41.30%
ReadWriteLock	15	32.61%
ReentrantReadWriteLock	14	30.43%
CountDownLatch	12	26.09%
ReentrantLock	11	23.91%
Semaphore	9	19.57%
Condition	5	10.87%
CyclicBarrier	4	8.70%
AbstractQueuedSynchronizer	2	4.35%
LockSupport	2	4.35%
AbstractOwnableSynchronizer	0	0.00%
AbstractQueuedLongSynchronizer	0	0.00%
Exchanger	0	0.00%
Phaser	0	0.00%

Table 5. Usage of synchronization primitives in Java

case in our corpus), it is easy to use and hardly susceptible to errors. With 41%, the next mutual exclusion feature is `Lock`, followed closely by the specialized versions for single writer, multiple reader problems. `CountDownLatch` and `CyclicBarrier` are moderately used, while their more complex sibling `Phaser` is not used at all. `Exchanger` is also not in use. `LockSupport` is a building block for synchronization primitives similar to a semaphore and is in fact used in Consulo to build a custom version of a concurrent hashmap. Neo4j als makes use of this class in several places e.g. to implement a simpler and faster latch or a prefetching mechanism which synchronizes regularly with the main thread. `AbstractQueuedSynchronizer` is a similar case; it is used in Consulo to implement a semaphore.

Similar to Java, C# has a lock built-in to every class. The mutual exclusion feature `lock()` is also the top item on our list in Table 6. While it is common in Java to lock methods with the `synchronized` keyword, the corresponding C# method attribute `MethodImplOptions.Synchronized` is used in fewer than 5% of the projects. C# offers slim versions of some of their synchronization features, which trade functionality for performance . In most cases the slim version is preferred, which makes us wonder why developers chose `ManualResetEvent` three times more often than `ManualResetEventSlim`. `Mutex` provides the same functionality as `lock()`, but extends beyond process borders. Since this is often not required, `Mutex` gets used far less. Busy waiting is generally considered bad practice, which is why `SpinLock` gets not used at all by developers. But sometimes it is the right tool, when wait times are short and performance is critical, which is exactly what happens in the four projects where `SpinWait` gets used. C# also provides an explicit memory barrier `Interlocked.MemoryBarrier`, which prevents instruction reordering across the barrier.

Feature	# projects	% of total
lock()	42	93.33%
ManualResetEvent	22	48.89%
Monitor	17	37.78%
AutoResetEvent	16	35.56%
ReaderWriterLockSlim	15	33.33%
WaitHandle	13	28.89%
EventWaitHandle	10	22.22%
Mutex	10	22.22%
ManualResetEventSlim	8	17.78%
Barrier	7	15.56%
Semaphore	6	13.33%
SpinWait	4	8.89%
CountdownEvent	3	6.67%
ReaderWriterLock	3	6.67%
SemaphoreSlim	3	6.67%
MethodImplOptions Synchronized	2	4.44%
Interlocked.MemoryBarrier	1	2,22%
SpinLock	0	0.00%

Table 6. Usage of synchronization primitives in C#

Feature	# projects	% of total
mutex	39	88.64%
condition_variable	28	63.63%
Semaphore	18	40.91%
CriticalSection	17	38.64%
unique_lock	16	36.36%
lock_guard	12	27.27%
barrier	5	11.36%
#pragma omp critical	3	6.82%

Table 7. Usage of synchronization primitives in C++

Again C++ offers only the most basic synchronization primitives. Native C++ did not offer any of these constructs until C++11 and then only added mutexes and condition variables. Until then these were mostly provided by the boost and pthread libraries. The numbers in Table 7 contain the library versions as well. `barrier` also refers to both the boost and pthread versions. `Semaphore` and `CriticalSection` are only available on Windows. The OpenMP `critical` clause can only be used together with other OpenMP constructs, which is why it is only used in 3 projects (half of the projects using OpenMP). The usage of the mutual exclusion feature is in line with the numbers of its C# and Java counterpart. Although `mutex` and `condition_variable` have only been available since 2011 it is good to see that they are already gaining some momentum. Developers use them in 11 out of 39 projects for the mutex and 6 out of 28 for the condition variable.

All three languages have different tools for synchronizing parallel executions. C++ covers the basics and relies on the developer to implement more sophisticated synchronization schemes tailored to the use case. C# has a much wider variety of options to handle typical synchronization problems such as single writer, multiple reader scenarios. Java additionally offers some basic blocks purpose-built for constructing advanced synchronization mechanisms.

4.1.3 Data Structures

Parallel programs typically operate on data stored in thread-safe containers. Locking state variables is expensive, which is why atomic variables are preferred. We take a look at the offerings of our three languages regarding concurrent data structures and atomic variables.

Java provides developers with various thread-safe map, list, set and queue implementations. Additionally it offers wrapper classes to make arbitrary container implementations thread-safe as long as they implement one of the generic container interfaces. Table 8 shows that the developers use both variants, but often do not benefit from the performance gains of the specialized container when using `synchronizedMap` together with `HashMap`. Of these containers `ConcurrentHashMap` is by far the most common, with over 50% of all projects using it. Java also has atomic versions for various numeric types, which implement increment and decrement operations for atomic counters. They also provide `compareAndSet` operations to implement non-blocking algorithms. There are wrappers for volatile variables, which provide atomic behavior, when this is only occasionally necessary or putting large numbers of variables in atomic objects is too memory-intensive. Again, programmers do not use them often in our corpus.

C# developers only have four thread-safe containers to choose from, but similar to Java they prefer the key-value map implementation named `ConcurrentDictionary`. Queue and bag/set containers also show similar usage numbers compared to Java. C# does not provide atomic types. Instead it offers methods implementing atomic operations on standard numeric types. The `Interlocked` class, which is used in 22 of the 45 projects, comes with increment, decrement, add as well as compare-and-swap operations. The `Interlocked.read()` method is only required on 32-bit systems when doing a 64-bit read.

With C++11 came atomic data types while concurrent containers are entirely lacking from the standard library. Developers are required to explicitly synchronize accesses to the standard non thread-safe containers. Three of the projects in our corpus use the concurrent collections provided in the Intel TBB library. 20 projects use atomic variables, while both C++ style (`std::atomic<int>`) and C style (`atomic_int`) variants are equally present. The C-style functions for modifying atomic variables are also listed in Table 10. There are a lot more atomic types in C++, for example `int_least64_t` which typically defaults to the small-

Feature	# projects	% of total
ConcurrentHashMap	26	56.52%
synchronizedMap	15	32.61%
CopyOnWriteArrayList	14	30.43%
ConcurrentLinkedQueue	11	23.91%
synchronizedList	10	21.74%
SynchronizedSet	9	19.57%
CopyOnWriteArraySet	6	13.04%
ConcurrentSkipListSet	3	6.52%
synchronizedCollection	3	6.52%
ConcurrentLinkedDeque	1	2.17%
ConcurrentSkipListMap	1	2.17%
synchronizedSortedMap	1	2.17%
synchronizedSortedSet	1	2.17%
ConcurrentNavigableMap	0	0.00%
AtomicInteger	28	60.87%
AtomicBoolean	17	36.96%
AtomicLong	16	34.78%
AtomicReference	13	28.26%
AtomicReferenceArray	7	15.22%
AtomicIntegerArray	5	10.87%
AtomicLongArray	4	8.70%
AtomicIntegerFieldUpdater	2	4.35%
AtomicReferenceFieldUpdater	2	4.35%
AtomicLongFieldUpdater	1	2.17%
AtomicMarkableReference	1	2.17%
AtomicStampedReference	0	0.00%

Table 8. Usage of concurrent data types and collections in Java

Feature	# projects	% of total
ConcurrentDictionary	22	48.89%
ConcurrentQueue	11	24.44%
ConcurrentBag	5	11.11%
ConcurrentStack	0	0.00%
Interlocked.Increment	19	42.22%
Interlocked.Exchange	12	26.67%
Interlocked.CompareExchange	10	22.22%
Interlocked.Decrement	10	22.22%
Interlocked.Add	3	6.67%
Interlocked.Read	3	6.67%

Table 9. Usage of concurrent data types and collections in C#

Feature	# projects	% of total
std::atomic<bool>	7	15,91%
atomic_int	6	13.64%
atomic_fetch	6	13.64%
atomic_exchange	6	13.64%
std::atomic<int>	5	11,36%
atomic_compare	4	9.09%
atomic_load	4	9.09%
atomic_store	4	9.09%
atomic_bool	3	6.82%
std::atomic<size_t>	3	6,82%
atomic_uint	2	4.55%
std::atomic<uint32_t>	2	4,55%
std::atomic<uint64_t>	2	4,55%
atomic_long	1	2.27%
atomic_int_fast	1	2.27%
atomic_intptr	1	2.27%
atomic_size_t	1	2.27%

Table 10. Usage of C++11 atomic data types

est type with at least 64 bits or `int_fast16_t` which may default to a 32 bit type because it provides faster arithmetic or can be optimized better.

There are two approaches for handling atomic variables in our three languages. Classes providing atomic methods for their encapsulated data type and special functions for atomically modifying volatile variables. C++ and Java support both, while C# only provides the latter. When it comes to concurrent data structures, C++ provides none, while C# and Java supply roughly the same types of structures. While Java has a wider selection of variants, developers mostly use thread-safe key-value maps in both languages.

4.1.4 Libraries

Why reinvent the wheel when there is already a library available that does exactly what you want? While this is true for many aspects of programming, we focus on the libraries dedicated to provide parallel features. From the topics discussed above, we already know that C++ is a perfect candidate for parallel libraries, since many desired features are lacking from the standard library, which is even more true for code written before C++11. Java and C# have much less demand for third-party libraries because they natively cover a much broader spectrum of useful features. In fact, we did not find any external parallel library used in multiple projects across our corpus. In C# there are a lot of cases, where developers implemented features already available in the standard library. This is due to the fact that the .NET runtime library comes in variants for desktop PCs, tablets, mobile phones, embedded devices and gaming consoles which do not always support all features related to parallelism.

Regarding C++, this was a totally different story. The most used libraries are listed in Table 11. PThreads and Windows-threads are used in half of all the projects. Sometimes both are used to enable cross-platform applications. Boost threads are also popular, followed by OpenMP, Intel TBB and MPI. Microsoft PPL and Apple GCD are only used in one project. We found that developers tend to rely on a single library and do not try to combine parallel features from different libraries. In some cases, developers provide their own implementations of features which would be readily available in one of these common libraries.

Library	# projects	% of total
phtreads	22	50%
Windows-threads	20	45%
Boost-thread	7	16%
OpenMP	6	14%
Intel TBB	3	1%
MPI	2	<1%
Microsoft PPL	1	<1%
Apple GCD	1	<1%

Table 11. Usage of parallel libraries in C++

A notable exception regarding library use is the OpenCV project. It supports many platforms and is heavily optimized for each one of them. Besides supporting an amazing number of graphics libraries, it can be configured to use many different parallel libraries. Listing 1 shows a code example which demonstrates the use of many different parallel libraries. Similar patterns are found all over the OpenCV source code.

```
#ifndef HAVE_TBB
    #if defined HAVE_CSTRIPES
        #include "C=.h"
        #undef shared
    #elif defined HAVE_OPENMP
        #include <omp.h>
    #elif defined HAVE_GCD
        #include <dispatch/dispatch.h>
        #include <pthread.h>
    #elif defined WINRT
        #include <ppltasks.h>
    #elif defined HAVE_CONCURRENCY
        #include <ppl.h>
    #endif
#endif
```

Listing 1. OpenCV parallel.cpp code example

4.2 RQ2: Which high- or low-level parallel patterns are used?

Parallel patterns provide solutions to recurring parallel problems, which can be found in various textbooks [9, 10]. We searched our corpus for popular patterns to find out how often they are implemented by developers. Table 12 con-

Pattern	Java	C#	C++
Master-worker	29	23	23
Producer-consumer	10	5	5
Pipeline	8	9	8
Parallel Loop	1	8	5
Fork-join	3	1	1

Table 12. Number of projects containing parallel patterns

tains our findings. We consider master-worker, producer-consumer and pipeline high-level patterns, parallel loop and fork-join are low-level patterns. Although a parallel loop can be easily implemented using the master-worker pattern, the master-worker pattern is much more versatile. We consider a parallel loop low-level because of its simple data parallel intent.

The master-worker pattern is by far the most frequent parallel pattern in our corpus. This is probably due to the fact that it is the most obvious and simple way of implementing parallelism. In its basic form this pattern creates a number of threads that perform work and report back when they are finished. It is equally popular across languages with a slight advantage for Java, maybe due to the ease-of-use of the `ExecutorService` which appears in almost all master-worker occurrences in Java.

Next up is the producer-consumer pattern which gets 10 uses for Java and 5 each for C# and C++. Java and C# provide convenient blocking collections to handle synchronization between producers and consumers, while in C++ developers need to implement this mechanism.

The pipeline pattern gets equal use in all three languages. In C# and Java developers are mostly using the same blocking collections as buffers between pipeline stages as in the producer-consumer pattern. Although Java offers the `Exchanger` class which is designed to be a buffer between pipeline stages, this feature is never used.

The Parallel loop pattern is a good candidate for showing that the use of a pattern depends on ease of use. C++ and C# provide easy ways to implement a data parallel loop through `#pragma omp parallel for` and `Parallel.For/Each` respectively. In Java there is no such construct and only one project implements it by means of the aforementioned master-worker pattern.

Fork-join is the simplest task parallel pattern, but not used much because of the availability of more convenient higher-level options. Fork-join is a building block for the divide-and-conquer pattern, which we do not consider here, but it can be observed in the Java code that makes use of `RecursiveAction` and `ForkJoinPool`. Fork-join can be easily realized in C# by calling `Parallel.Invoke`, while in C++ developers need to handle the threads explicitly. The `fork()` and `join()` functions by themselves are not considered to match our patterns.

4.3 RQ3: How do programmers' solutions to common parallel problems compare to textbook solutions?

There are many textbooks about parallel patterns and best practices for parallel programming, but not all developers seem to be aware of them or at least do not adhere to them. In our corpus, hobby projects with arguably less experienced programmers show more violations of best practices compared to larger projects with more developers.

Parallel patterns are inherently problematic because real world problems often do not exactly match a pattern, so developers start being creative and come up with their own solutions instead of trying to reuse existing functionality. This leads to more error prone code. Smaller projects tend to reimplement slight variations of existing functionality, while projects with a larger codebase often reuse existing functionality and extend these classes via subclassing.

An exception is the master-worker pattern. It seems that it is such an intuitive pattern, that some developers seem to implement it by accident. Unexperienced developers just create threads and distribute their work to them. In larger projects comments and class naming suggests that they are aware of the master-worker pattern and implement it on purpose, which is not the case in most small projects.

Prime examples of bad pratice are the `synchronized()` and `lock()` features in Java and C#. Both language references strongly advise against locking on publicly accessible types or classes. Nevertheless this is what happens in most instances in both Java and C# regardless of the size of the project.

5. Threats to Validity

This paper provides a window on the current state of parallelism in open source projects. There might be some cases where we missed parallel code locations. One such case are libraries. We included the most popular parallel libraries into our search, but there are a lot more libraries that offer parallel functionality. If such a library was used and its methods were named differently from what the typical names for parallel constructs are, then we probably missed it. The other case would be manual implementations of parallel constructs, which are also not conventionally named and do not contain hints in the comments. Such an example would be a french programmer implementing a barrier and naming it *barrière*. While it would be easy to account for language differences, this was not our goal and such cases can be handled by the techniques described in the following paragraphs. In other cases we did catch manual implementations of parallel primitives. Here are some examples: The OpenSim C# project implemented its own `EventWaitHandle` class, because it is not natively available on Windows CE devices. Another project called Enyim Memcached Client implemented its own `CountDownEvent` class, which is very close in functionality to the construct contained in the standard library. In C++ Kythe ships with a custom `Mutex` implementation, which internally uses the Windows `CriticalSection` API instead of using the pthread, boost, or C++11 counterparts. There are a few other examples which we also accounted for in our statistics.

Detecting the use of parallel patterns could benefit from a dynamic analysis approach. Seeing the real data dependence and control dependence graphs from an instrumented execution would allow for a better understanding of the parallel behavior of these programs.

Static code analysis based on ASTs would also lead to a better understanding of the code than our regular expression based approach. Certain ambiguities would be easier to resolve and comments could be ignored without manual filtering. This would lead to a higher accuracy with less room for human error.

6. Conclusion

In order to be able to support programmers with tools for multicore development, we need to understand how developers approach and implement parallelism. We need to know which language features and parallel patterns they use and which errors they make. As a first step we examined object-oriented software projects to gain some insight.

In this paper we have presented a study of 135 parallel open-source projects, consisting of 45 C#, 44 C++ and 46 Java programs. Parallel constructs were found in 10% of the source files, which was consistent across languages. Developers focus on hotspots and bottlenecks in their applications and parallelize those first.

We further looked at the use of language features related to parallelism. While C# and Java provide a great variety of options, C++ is lacking high-level constructs and even third-party libraries do not close this gap. Concurrent containers are dominated by key-value maps, which are easy to use but hardly the optimal choice in every scenario.

When it comes to synchronization primitives all three languages cover the basic building blocks, while C# and Java also offer more sophisticated features. Looking closer at the use of these primitives reveals that the ease of use of `synchronized` blocks far outweighs the performance benefits of finer locking schemes. Best practices are also mostly ignored especially with the `synchronized` feature.

Developers tend to intuitively prefer the master-worker pattern, because it is simple to understand and easy to implement. The other parallel patterns we examined were much less common. There are many additional parallel patterns we did not consider here, but might explore in the future to get a deeper understanding of how programmers use parallelism.

From the data we gathered, we now have some hints on where to support developers with tools for implementing parallelism. With respect to the most frequently used parallel patterns, we can focus our attention on them instead of wasting resources on rarely used patterns. Making those that are heavily used easier to implement and less error-prone would benefit inexperienced parallel programmers.

Our study gives us a good first insight into how developers make use of the available language features to create parallel programs. More sophisticated methods such as static analysis could gather additional data. This information will help create better tools for supporting programmers in writing parallel programs from scratch and improving existing parallel software.

Acknowledgments

We thank Siemens Corporate Technology for their financial support. We also appreciate the support of the Initiative for Excellence at the Karlsruhe Institute of Technology.

References

[1] S. Amarasinghe. The Looming Software Crisis due to the Multicore Menace, 2006. URL `http://groups.csail.mit.edu/commit/papers/06/MulticoreMenace.pdf`.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, and others. A view of the parallel computing landscape. *Communications of the ACM*, 52(10): 56–67, 2009.

[3] J. Bloch. *Effective java*. Pearson Education India, 2008. ISBN 81-317-2659-2.

[4] C. Brown, K. Hammond, M. Danelutto, and P. Kilpatrick. A Language-independent Parallel Refactoring Framework. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 54–58, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1500-5. .

[5] L. Dagum and R. Enon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. ISSN 1070-9924.

[6] D. Gove. *Multicore Application Programming: For Windows, Linux, and Oracle Solaris*. Addison-Wesley Professional, 2010. ISBN 0-321-71137-8.

[7] B. Karlsson. *Beyond the C++ standard library: an introduction to boost*. Pearson Education, 2005. ISBN 0-672-33415-1.

[8] P. Larsen, R. Ladelsky, J. Lidman, S. A. McKee, S. Karlsson, and A. Zaks. Automatic Loop Parallelization via Compiler Guided Refactoring. Technical report, Technical University of Denmark, 2011.

[9] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, Boston, Mass., 05001 edition, Sept. 2004. ISBN 978-0-321-22811-6.

[10] M. McCool, J. Reinders, and A. D. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, June 2012. ISBN 978-0-12-415993-8.

[11] K. Molitorisz, T. Müller, and W. F. Tichy. Patty: A pattern-based parallelization tool for the multicore age. In *The 6th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), to be held in conjunction with the 20th ACM SIGPLAN Symposium on*

Principles and Practice of Parallel Programming (PPoPP), 02 2015.

[12] T. Nash. Threading in C#. In *Accelerated C# 2010*, pages 361–427. Apress, 2010. ISBN 978-1-4302-2537-9. .

[13] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996. ISBN 1-56592-115-1.

[14] S. Okur, C. Erdogan, and D. Dig. Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, number 8586 in Lecture Notes in Computer Science, pages 515–540. Springer Berlin Heidelberg, July 2014. ISBN 978-3-662-44201-2 978-3-662-44202-9.

[15] C. Pheatt. Intel threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008. ISSN 1937-4771.

[16] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? *Communications of the ACM*, 58(5): 77–86, Apr. 2015. ISSN 00010782. .

[17] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[18] H. Vandierendonck and T. Mens. Techniques and Tools for Parallelizing Software. *IEEE Software*, 29(2):22–25, Mar. 2012. ISSN 0740-7459. .