An empirical study on C++ concurrency constructs

Di Wu Lin Chen Yuming Zhou Baowen Xu*

State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing, China

nju.wudi@gmail.com lchen@nju.edu.cn zhouyuming@nju.edu.cn bwxu@nju.edu.cn

Abstract—**Nowadays concurrent programming is in large demand. The inherent support for concurrency is therefore increasingly important in programming languages. As for C++, an abundance of standard concurrency constructs have been supported since C++11. However, to date there is little work investigating how these constructs are actually used in developing real software. In this paper, we perform an empirical study to investigate the adoption of C++ concurrency constructs in open-source applications, with the goal to provide insightful information for practitioners to use concurrency constructs efficiently. To this end, we analyze 127 open-source applications that adopt C++ concurrency constructs, comprising 34 million lines of C++ code, to conduct the experiment. The experimental results show that: (1) to implement concurrency code, thread-based constructs are significantly more often used than atomics-based constructs and task-based constructs; (2) to manage synchronization, lock-based constructs are significantly more often used than lock-free constructs and blocking constructs; (3) among the key thread-based constructs and task-based constructs (i.e. mutex, promise, and future), there is not a construct significantly more commonly misused than others; (4) small-size applications introduce concurrency constructs more intensively and more quickly than medium-size applications and large-size applications; and (5) an increasing use of standard concurrency constructs does not result in a substantially decreasing use of unstandardized concurrency constructs. Based on these findings, we make actionable suggestions for language designers, developers, and novices to assist them in designing and using C++ concurrency constructs.**

Keywords- Programming Language, Empirical Study, C++, Concurrency

I. INTRODUCTION

Concurrency is widely used to improve the performance of software through enhancing its throughput and responsiveness on multiprocessor platforms. Also, concurrency is a good way to separate distinct areas of functionality, making it easy to implement complex systems using separation of concerns [3]. For these reasons, many modern languages have provided well-designed concurrency constructs. As for C++, it has started to support concurrency since C++11 [1], which provides concurrency constructs at different abstraction levels (including atomics-based, thread-based, and task-based constructs) and corresponding synchronization mechanisms to manage interference between concurrent operations. With these constructs, programmers can efficiently implement portable and quality concurrent programs.

Ever since these standard concurrency constructs were introduced, many studies have been performed to guarantee the correctness of concurrent programs through theoretical verification [4, 5, 6]. However, few studies investigate the actual adoption of C++ concurrency constructs in an empirical way. This may lead to several problems. For language designers, they cannot know which concurrency constructs are practically needed and which new concurrency features need to be introduced in the future language standard. For developers, they are unaware of whether they have properly used concurrency constructs and whether their concurrency code needs to be revised. For novices, they do not understand which concurrency constructs are the most useful and need to be learnt.

In this paper, we perform an empirical study to investigate the actual adoption of C++ concurrency constructs in open-source applications, with the goal to provide insightful information for practitioners to use concurrency constructs efficiently. To be specific, we propose the following research questions:

- RQ1: Which concurrency constructs are the most often[1] used?
- RQ2: Which synchronization constructs are the most often used?
- RQ3: Are there concurrency constructs that developers commonly misuse?
- RQ4: Do applications with different size adopt concurrency constructs differently?
- RQ5: Does an increasing use of standard concurrency constructs result in a substantially decreasing use of unstandardized concurrency constructs?

These research questions are critically important, as their results can provide empirical evidence for improving the design and use of C++ concurrency constructs. More specifically, the purpose of RQ1 investigates whether there exist a few concurrency constructs that are more often used than others. Based on its result, we can find out whether developers act in accordance with the expert's suggestion that programmers should "think in terms of tasks that can be executed concurrently, rather than directly in terms of threads" [2]. The purpose of RQ2 investigates which synchronization constructs play a key role in managing interference between concurrent operations. With this body of knowledge, we can advise C++ novices to focus on learning the most commonly-used synchronization constructs. The purpose of RQ3 investigates the misuse of concurrency constructs. In [2],

* Corresponding author: Baowen Xu; Email: bwxu@nju.edu.cn

[1] Throughout this paper, we use the terms "often" and "common" to refer to the percentage of use for a given construct. Moreover, the term "intensive" refers to the number of uses of concurrency constructs within a project. In addition, the terms "use", "adopt", and "utilize" are used interchangeably.

Stroustrup highlighted the misuses of three important concurrency constructs (i.e. mutex, promise, and future). However, there is no data reporting how commonly these constructs are misused in real software. We attempt to fill this gap by answering RQ3. In particular, if some kind of common misuses are detected, we can give advice to developers to pay attention to avoiding these misuses. The purpose of RQ4 investigates how application size influences the adoption of concurrency constructs. More specifically, we break RQ4 into two sub-RQs: (1) RQ4.1: Do large-size applications adopt concurrency constructs more intensively than small-size applications and medium-size applications? and (2) RQ4.2: Do small-size applications adopt concurrency constructs more quickly than medium-size applications and large-size applications? By answering these two questions, we can find out a group of applications that intensively and quickly adopt concurrency constructs and suggest C++ learners taking these applications as real-world examples to learn the usages of C++ concurrency constructs. The purpose of RQ5 investigates the correlation between the adoption of standard concurrency constructs and the adoption of unstandardized concurrency constructs. Since many standard concurrency constructs originate from third-party libraries and they have been put into use for only four years, developers might be still using the concurrency constructs provided by the third-party libraries, such as TBB [27] and PPL [28]. Due to the fact that most third-party libraries are domain specific, using those unstandardized concurrency constructs may degrade program portability and code quality. Transforming the adoption of unstandardized constructs to the adoption of standard alternatives is a wise choice. Therefore, we propose RQ5 to investigate whether this transformation practically makes a difference.

In order to answer these research questions, we analyze 127 open-source applications that adopt C++ concurrency constructs, comprising 34 million lines of C++ code, to conduct the experiment. The experimental results show that: (1) to implement concurrency code, thread-based constructs are significantly more often used than atomics-based constructs and task-based constructs; (2) to manage synchronization, lock-based constructs are significantly more often used than lock-free constructs and blocking constructs; (3) among the key thread-based constructs and task-based constructs (i.e. mutex, promise, and future), there is not a construct significantly more commonly misused than others; (4) small-size applications introduce concurrency constructs more intensively and more quickly than medium-size applications and large-size applications; and (5) an increasing use of standard concurrency constructs does not result in a substantially decreasing use of unstandardized concurrency constructs. Based on these findings, we make actionable suggestions for language designers, developers, and novices to assist them in designing and using C++ concurrency constructs.

The rest of the paper is organized as follows. Section II introduces C++ concurrency constructs in a nutshell. Section III describes the studied subjects, data collection procedure, and data analysis methods. Section IV reports the experimental results. Section V sheds light on the implications of the empirical results. Section VI examines the threats to validity. Section VII discusses related work. Section VIII concludes the paper and outlines the direction for future work.

II. An Overview of C++ Concurrency Constructs

According to the C++11 specification [1], all standard concurrency constructs can be categorized as atomics-based, thread-based, or task-based constructs, in accordance to their abstraction levels for concurrency. To be specific, the **atomics-based constructs**, which denote the primitive classes and functions that do not suffer data races, are the lowest-level concurrency constructs. They provide a set of techniques for writing concurrent programs without using explicit locks. For this reason, the synchronization operations used by atomics-based constructs are called **lock-free operations**, which include atomic flag functions and atomic fence functions. As for **thread-based constructs**, they stand for the classes and functions which are used for thread creation and management. To implement multithreading synchronization (for example, thread exclusion and message passing), C++ provides a set of **lock-based constructs**, including mutexes, mutex ownership wrappers, and condition variables. Also, threads are allowed to be **blocked** by using the waiting operations provided in the "this_thread" namespace. In terms of **task-based constructs**, they provide the abstraction for concurrency at the highest level, meaning that users do not need to care about the complex interference among concurrent operations. To run a task, users only need to call the "async" function or create an instance of the "packaged_task" class. Message passing between tasks can be implemented by using "**promise**" and "**future**", which work for storing and accessing the result of a task, respectively. To explicitly control the execution of tasks, users can employ the **blocking operations** such as "future::wait" and "shared_future::wait". Generally speaking, the execution efficiency of atomics-based, thread-based, and task-based constructs is from high to low. However, using atomics-based and thread-based constructs can bring about a high risk of error. Therefore, Stroustrup suggested developers leaving lock-free operations to experts and thinking in tasks instead of threads [2].

In [2], Stroustrup also highlighted three kinds of **misuses of concurrency constructs**, namely, the mismatched *lock/unlock* operations on mutexes, the multiple *set_value/set_exception* operations on promises, and the multiple *get* operations on futures. Since mutexes, promises, and futures are the most important thread-based and task-based constructs, their misuses need developers' attention. Therefore, we attempt to investigate the actual occurrence of these misuses by analyzing real concurrent programs.

Apart from the C++ standard libraries, many third-party libraries also provide various concurrency constructs. The most well-known concurrency libraries are the TBB library which contains generic algorithms and data structures for thread building, the PPL library which includes the containers and objects for task parallelism, and the Thrust library which provides a rich collection of data parallel primitives for implementing complex parallel algorithms. In addition, there exist many other famous concurrency libraries, including OpenMP, HPX, boost::interprocess, boost::lockfree, boost::thread, and boost::atomics [26].

III. Research Method

In this section, we first introduce the subjects used for investigating our research questions. Then, we report the data collection procedure. Finally, we describe the data analysis methods.

A. Studied subjects

To investigate the proposed research questions, we analyze the open-source C++ applications listed on GitHub [23], the most popular open-source software repository. More specifically, the subject applications are selected by the following steps. In the first step, we rank all C++ applications on GitHub according to the number of followers. Then, we select the top 1000 applications as the target subjects. We do this in order to guarantee the most popular applications to be included in our data corpus. In the second step, we download the source code of the 1000 applications by executing the *git clone* command. In the third step, we cleanse the data corpus by filtering out the applications whose source code repositories are invalid (no longer maintained, or duplicated with others). After doing this, 926 applications remain. In the fourth step, we preliminarily analyze the source code of selected applications with the purpose to find out the subjects that adopt C++ concurrency constructs. This is done by examining the included C++ header files. To be specific, if the header files of standard concurrency libraries such as "atomics" and "thread" are detected, we will recognize the application as the one that uses standard concurrency constructs. If the header files of third-party concurrency libraries such as "Boost.Interprocess" and "TBB" are found, we will regard the application as the one that adopts unstandardized concurrency constructs. By doing this, we divide all applications into different partitions. Figure 1 shows the result of categorization. Among the 926 applications, 127 (14%) of them adopt C++ concurrency constructs, with 113 using standard concurrency constructs, 8 employing unstandardized concurrency constructs, and 6 utilizing both standard and unstandardized concurrency constructs. Since this paper focuses on the adoption of concurrency constructs, we select the 127 applications (the gray areas in Figure 1) as our target subjects to investigate the proposed research questions. In the final step, we categorize these 127 applications based on their code size. By doing this, we find the number of small-size (C++ SLOC $\leq$ 10K), medium-size (10K $<$ C++ SLOC $\leq$ 100K), and large-size (C++ SLOC $>$ 100K) applications are 20, 49, and 58, respectively. Table I shows the basic information of these subjects. From this table, we find that both applications' size and the number of developers vary in a large range.

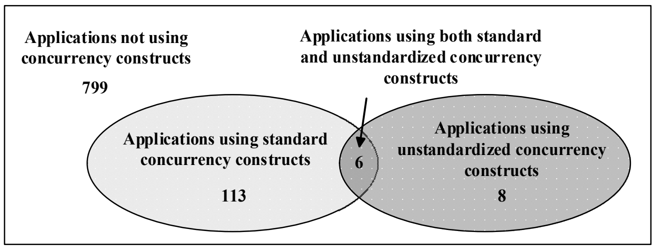
Applications not using concurrency constructs
799
Applications using both standard and unstandardized concurrency constructs
Applications using standard concurrency constructs
113
6
Applications using unstandardized concurrency constructs
8

Figure 1. Categorization of all applications

B. Data collection procedure

We collect the experimental data by using "Understand" [24], a program analysis tool. To be specific, the experimental data is collected by the following steps. In the first step, we obtain C++ files by using the "C++ Strict" option provided by "Understand" and build an Understand database for each studied application. In the second step, we process Understand databases to identify the use sites of concurrency constructs by calling Understand APIs. In the third step, we pick out the use sites of synchronization constructs by filtering the use sites of all concurrency constructs. In the fourth step, we detect the misuses of concurrency constructs by analyzing the use sites of mutexes, promises, and futures. To be specific, for each mutex, we match its *lock* operations with its *unlock* operations. If a *lock* operation is not matched with corresponding *unlock* operations, it will be reported as a misuse of mutex. For each promise, we examine the number of its *set_value* and *set_exception* operations. If multiple *set_value/set_exception* operations are found in the same basic block[2], they will be reported as misuses of a promise. For each future, we examine the number of its *get* operations. If multiple *get* operations are detected in the same basic block, they will be reported as misuses of a future. In particular, the above-mentioned detection of misuses is automatically processed along with the data flow analysis. In the final step, we process Understand databases to find out the use sites of unstandardized concurrency constructs. Since all unstandardized concurrency constructs are prefixed with the third-party namespaces, they can be easily detected by matching their names with the namespaces such as "boost::interprocess::" and "tbb::".

C. Data analysis methods

To answer the proposed research questions, we employ three standard statistical inference techniques, including the Wilcoxon signed-rank analysis, the Mann-Whitney U-test, and the Spearman's correlation analysis. The detailed data analysis methods are discussed as follows:

(1) *Wilcoxon signed-rank analysis for RQ1 and RQ2*

RQ1 aims to investigate which kind of concurrency constructs are the most often used: atomics-based, thread-based, or task-based? In order to answer RQ1, we first calculate the percentage of use for these three kinds of concurrency constructs in each project. Then, we employ the Wilcoxon signed-rank test to compare their percentages of use pairwise in order to examine whether there is a significant difference at the significance level of 0.05. If one kind of concurrency constructs has a percentage of use significantly higher than the other two kinds, it will be identified as the most commonly used kind of concurrency constructs.

RQ2 aims to investigate which type of synchronization constructs are the most often used: lock-free, lock-based, or blocking? To answer RQ2, we first calculate the percentage of use for these types of synchronization constructs in each application. Then, we compare their percentages of use pairwise to test whether there exists a significant difference (p-value < 0.05). If one type of synchronization constructs exceed

[2] A basic block is a straight-line sequence of code with only one entry point and only one exit. For example, "if…else…" correspond to two basic blocks.

Table I. Basic information of 127 studied subjects
(Note: "SLOC" means "source lines of code excluding comments"; "KSLOC" means "thousand source lines of code excluding comments")

		Min.	25%	Median	75%	Max.	Mean	Std. dev.
Small-size applications (C++ SLOC ≤ 10K)	**C++ KSLOC**	0.101	0.978	2.359	3.637	9.928	3.464	3.354
	# developers	1	3	3.5	5	24	5.450	5.316
Medium-size applications (10K < C++ SLOC ≤ 100K)	**C++ KSLOC**	10.206	17.176	25.718	38.636	98.581	33.901	24.372
	# developers	1	12	24	50	219	35.898	37.652
Large-size applications (C++ SLOC > 100K)	**C++ KSLOC**	100.377	153.320	293.127	687.181	2520.027	564.452	601.122
	# developers	1	6.25	53.5	140.75	1335	125.345	220.432

other two types in the percentage of use significantly, we will recognize it as the most commonly used type of synchronization constructs.

After performing each Wilcoxon signed-rank analysis, we further apply the Cliff's δ, which is used for median comparison, to examine whether the magnitude of difference is important [20]. By convention, the magnitude of the difference is considered either trivial ($|\delta| < 0.147$), small (0.147-0.33), medium (0.33-0.474), or large (> 0.474) [21].

(2) *Mann-Whitney U-test for RQ3, RQ4.1, and RQ4.2*

RQ3 aims to investigate which concurrency construct is the most commonly misused: mutex, promise, or future? To answer RQ3, we first calculate the occurrence rate for each kind of misuse. To be specific, for mutex misuse, its occurrence rate is computed as the number of mutexes with unmatched *lock/unlock* operations divided by the total number of mutexes that apply these operations. For promise misuse, its occurrence rate is calculated as the number of promises with multiple *set_value/set_exception* operations divided by the total number of promises that apply these operations. In terms of future misuse, its occurrence rate is calculated as the number of futures with multiple *get* operations divided by the total number of futures applying the *get* operations. Then, we employ the Mann-Whitney U-test to compare the three groups of occurrence rates pairwise to examine whether they show a significant difference (p-value < 0.05). If the occurrence rate of one kind of misuse is significantly higher than others, we will confirm that this kind of misuse is the most likely to occur. Otherwise, we will conclude that there is not a concurrency construct that is more commonly misused than others.

RQ4.1 aims to investigate which kind of applications adopt concurrency constructs most intensively: small-size, medium-size, or large-size? In order to answer RQ4.1, we first calculate the density of use for concurrency constructs in each application. The density is calculated as the number of use for concurrency constructs in an application divided by its KSLOC value. Here, we employ the density instead of the raw number of uses for concurrency constructs in order to remove the impact brought by different application size. Then, we use the Mann-Whitney U-test to compare the density values of small-size, medium-size and large-size applications in order to figure out which kind of applications applies concurrency constructs most intensively at the significance level of 0.05.

RQ4.2 aims to investigate which kind of applications adopt concurrency constructs most quickly: small-size, medium-size, or large-size? We answer this research question by investigating the interval days between the application's establishing time and it first use of concurrency constructs. To be specific, we first calculate for each application its interval days. Then, we use the Mann-Whitney U-test to compare the interval days among different application groups. If one group of applications has significantly smaller interval days than other groups (p-value < 0.05), we will conclude that this group of applications adopt concurrency constructs most quickly.

After performing each Mann-Whitney U-test, we also apply the Cliff's δ to examine whether the magnitude of difference is important.

(3) *Spearman's rank correlation analysis for RQ5*

RQ5 aims to investigate whether the use of standard concurrency constructs and the use of unstandardized constructs are negatively correlated. In order to answer RQ5, we first pick out the 6 applications that utilize both standard and unstandardized concurrency constructs (applications shown in the overlapped area in Figure 1). Then, we calculate the density of use for standard/unstandardized concurrency constructs in each historical revision of the 6 applications. Finally, we use the *R* language [25] to automatically calculate the Spearman's coefficient (rho) of the correlation. In particular, we examine whether a correlation is significant at the significance level of 0.05. The answer for RQ5 will be "Yes" if the result shows a significantly negative correlation.

IV. Experimental Results

In this section, we report in detail the experimental results for the proposed research questions.

A. *RQ1: Which concurrency constructs are the most often used?*

We employ the result from the Wilcoxon signed-rank analysis for the use of concurrency constructs to answer RQ1. In particular, Figure 2 describes the percentage of use for concurrency constructs of different abstraction levels. In this figure, each boxplot shows the median (the horizontal line within the box), the 25th and 75th percentiles (the lower and upper sides of the box), and the mean value (the small red rectangle inside the box). From Figure 2, we can see that the percentage of use for thread-based constructs is larger than the percentages of other groups, indicating that thread-based constructs are the most commonly used by developers. The data listed in Table II confirms our observation. In Table II, we show the result by reporting the significance (p-value) and the magnitude (Cliff's δ) of the difference, respectively. To be specific, for thread-based constructs, they significantly exceed atomics-based constructs and task-based constructs with respect to the percentage of use (both p-values < 0.001). Moreover, the effect sizes are large in terms of Cliff's δ ($0.522 \leq |\delta| \leq 0.767$). As for atomics-based constructs, their percentage of use is also significantly larger than task-based constructs (p-value < 0.001). To summarize, the conclusion for RQ1 based on the core observation from Table II is that:

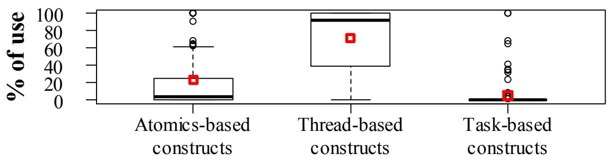

Figure 2. Boxplot showing the percentage of use for concurrency constructs of different abstraction levels

Table II. Result of Wilcoxon signed-rank analysis for RQ1

Groups for comparison	p	δ
Atomics-based constructs vs. Thread-based constructs	<0.001	-0.522
Atomics-based constructs vs. Task-based constructs	<0.001	0.445
Thread-based constructs vs. Task-based constructs	<0.001	0.767

* All p-values are BH-adjusted

Table III. Usage of the most commonly-used concurrency constructs

Category	# use	% in categories	Class name	# use	% in category	Member function name	# use	% in class
Thread-based constructs	8228	62.59%	mutex	2611	31.73%	unlock	288	11.03%
						lock	241	9.23%
			thread	1590	19.32%	join	465	29.25%
						detach	104	6.54%
			unique_lock	1262	15.34%	unlock	142	11.25%
						lock	45	3.57%
			lock_guard	1079	13.11%	-	-	-
			condition_variable	636	7.73%	wait	157	24.69%
						notifiy_one	127	19.97%
			recursive_mutex	306	3.72%	unlock	33	10.78%
						lock	28	9.15%
Atomics-based constructs	3726	28.34%	atomic	1814	48.68%	load	105	5.79%
						compare_exchange_strong	66	3.64%
						store	42	2.32%
			atomic_init	706	18.95%	-	-	-
			atomic_flag	244	6.55%	test_and_set	99	40.57%
						clear	78	31.97%
Task-based constructs	1192	9.07%	promise	415	34.82%	get_future	143	34.46%
						set_value	56	13.49%
						set_exception	11	2.65%
			future	370	31.04%	get	89	24.05%
						operator=	36	9.73%
			packaged_task	157	13.17%	get_future	31	19.75%
						valid	31	19.75%
			shared_future	108	9.06%	operator=	16	14.81%
						get	7	6.48%

Table V. Usage of all synchronization constructs

Type	# use	% in types	Category	# use	% in type	Construct name	# use	% in category
Lock-based constructs	6243	85.97%	Mutexes	3081	49.35%	mutex	2611	84.75%
						recursive_mutex	306	9.93%
						timed_mutex	113	3.67%
						recursive_timed_mutex	51	1.66%
			Mutex ownership wrappers	2341	37.50%	unique_lock	1262	53.91%
						lock_guard	1079	46.09%
			Condition variables	745	11.93%	condition_variable	636	85.37%
						condition_variable_any	109	14.63%
			Generic algorithms	68	1.09%	try_lock	37	54.41%
						lock	31	45.59%
			Tag types for locking strategy	8	0.13%	adapt_lock_t	3	37.50%
						try_to_lock_t	3	37.50%
						defer_lock_t	2	25.00%
Blocking constructs	721	9.93%	Thread blocking operations	721	100.00%	thread::join	465	64.49%
						this_thread::sleep_for	225	31.21%
						this_thread::yield	25	3.47%
						this_thread::sleep_until	6	0.83%
			Task blocking operations	0	0.00%	-	-	-
Lock-free constructs	298	4.10%	Atomic flags	266	89.26%	atomic_flag	244	91.73%
						atomic_flag_test_and_set_explicit	12	4.51%
						atomic_flag_clear_explicit	6	2.26%
						atomic_flag_clear	2	0.75%
						atomic_flag_test_and_set	2	0.75%
			Atomic fences	32	10.74%	atomic_thread_fence	21	65.63%
						atomic_signal_fence	9	28.13%
						kill_dependency	2	6.25%

To implement concurrency code, thread-based constructs are significantly more often used than atomics-based constructs and task-based constructs.

Based on this conclusion, we pick out the most commonly-used concurrency constructs from our studied subjects. Table III reports the information about the usage of these constructs, which are sorted with respect to their percentages of use. From Table III, we find that thread-based constructs amount to 62.59% of total uses for concurrency constructs. The mutex classes (mutex and recursive_mutex) and lock classes (unique_lock and lock_guard) are the main contributors, because they jointly account for 66% of uses for thread-based constructs. This implies that developers spend a lot of effort to manage multi-threading interference by using mutexes and locks. By comparison, the task-based constructs are seldomly used in practice (9.07%). This empirical evidence indicates that in the practical adoption of concurrency constructs, developers do not follow Stroustrup's advice that "thinking in terms of tasks that can be executed concurrently, rather than directly in terms of threads" [2].

B. *RQ2: Which synchronization constructs are the most often used?*

We employ the result from the Wilcoxon signed-rank analysis for the use of synchronization constructs to answer RQ2. Figure 3 is applied to show the percentage of use for the synchronization constructs of different types. According to this figure, lock-based constructs are more often used than lock-free constructs and blocking constructs. To verify this observation, we tabulate the inferential result with Table IV, which reports the significance (p-value) and the magnitude (Cliff's δ) of the difference. Based on Table IV, we find that all comparisons show significant results (p-values < 0.001). Furthermore, the effect sizes of difference are large in terms of Cliff's δ ($0.587 \le |\delta| \le 0.795$). This indicates that lock-based constructs are significantly more often used than both lock-free constructs and blocking constructs, and that blocking constructs are significantly more intensively used than lock-free constructs. From this reasoning, we draw the following conclusion for RQ2:

To manage synchronization, lock-based constructs are significantly more often used than lock-free constructs and blocking constructs.

In order to understand why lock-based constructs are the most commonly used, we analyze the usage of each synchronization construct, whose information is shown in Table V. According to this table, we find that lock-based constructs contribute to more than 85% of total uses for all synchronization constructs. Moreover, the mutexes (3081 uses) and mutex ownership wrappers (2341 uses) are adopted in a large amount. By comparison, the lock-free constructs (298 uses) and blocking constructs (721 uses) are relatively seldomly adopted. In particular, the task blocking operations, which are applied to manage task exclusion, are not used at all. In general, the statistics shown in Table V is basically consistent with the findings from Table III. Hence, we deduce that the high use ratio of lock-based constructs is mainly due to the large amount of threads created in concurrent programs. In other words, developers have to use more mutexes and locking operations for thread exclusion when they need to secure the correctness of multi-threading executions.

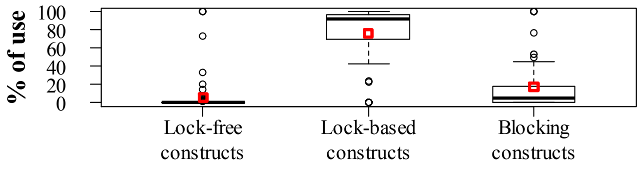

Figure 3. Boxplot showing the percentage of use for synchronization constructs of different types

Table IV. Result of Wilcoxon signed-rank analysis for RQ2

Groups for comparison	**p**	δ
Lock-free constructs vs. Lock-based constructs	<0.001	-0.795
Lock-free constructs vs. Blocking constructs	<0.001	-0.587
Lock-based constructs vs. Blocking constructs	<0.001	0.654

* All p-values are BH-adjusted

C. *RQ3: Are there concurrency constructs that developers commonly misuse?*

We employ the result from the Mann-Whitney U-test for the misuse of concurrency constructs to answer RQ3. Among the studied subjects, we find 45 applications containing the *lock/unlock* operations on mutexes, the *set_value/set_exception* operations on promises, and the *get* operations on futures. By analyzing these subjects, we find 50 misuses in total, comprising 39 mutex misuses, 6 promise misuses, and 5 future misuses. This data clearly shows that mutexes are misused at the highest amount. However, it cannot reveal the occurrence rate of misuse. In order to answer RQ3 with reliable empirical data, we report the statistical result from the Mann-Whitney U-test in Table VI, which shows the significance (p-value) and the magnitude (Cliff's δ) of the difference among the three kinds of misuses according to their occurrence rates. By observing Table VI, we find that the occurrence rate of mutex misuses is significantly lower than that of promise misuses (p = 0.014). Moreover, the effect size is large in terms of Cliff's δ ($|\delta| = 0.636$). However, other two comparisons do not show significant differences (both p-values > 0.05). Since there is not a kind of misuse whose occurrence rate is significantly larger than others, we draw the conclusion for RQ3 as follows:

Among the key thread-based constructs and task-based constructs (i.e. mutex, promise, and future), there is not a construct significantly more commonly misused than others.

Table VI. Result of Mann-Whitney U-test for RQ3

Groups for comparison	**p**	δ
Occurrence rate of mutex misuses vs. Occurrence rate of promise misuses	0.014	-0.636
Occurrence rate of mutex misuses vs. Occurrence rate of future misuses	0.521	-0.154
Occurrence rate of promise misuses vs. Occurrence rate of future misuses	0.268	0.429

* All p-values are BH-adjusted; p-values > 0.05 are shown in gray background.

D. RQ4: Do applications with different size adopt concurrency constructs differently?

(1)RQ4.1: Do large-size applications adopt concurrency constructs more intensively than small-size applications and medium-size applications?

We employ the result from the Mann-Whitney U-test for the density of use for concurrency constructs to answer RQ4.1. Figure 4 shows the distribution of the data for three different groups. According to this figure, we can see that small-size applications tend to use concurrency constructs at a higher density than both medium-size and large-size applications. To verify this observation, we show the inferential statistics with Table VII, which reports the significance (p-value) and the magnitude (Cliff's δ) of the difference, respectively. From Table VII, we find that all three pairs of comparisons show significant results (p-values < 0.001). Moreover, the effect sizes of difference are large regarding Cliff's δ ($0.563 \leq |\delta| \leq 0.893$). To summarize, the core observation from Table VII is that the density of using concurrency constructs in small-size applications is significantly larger than the values in medium-size applications and large-size applications. From this reasoning, the conclusion of RQ4.1 can be drawn as:

Small-size applications adopt concurrency constructs significantly more intensively than medium-size applications and large-size applications.

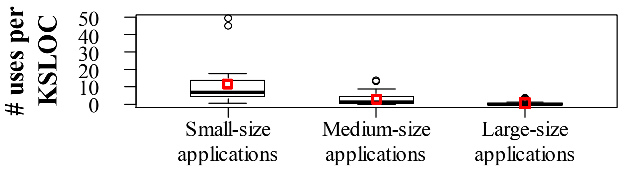

Figure 4. Boxplot showing the density of use for concurrency constructs

Table VII. Result of Mann-Whitney U-test for RQ4.1

Groups for comparison	**p**	δ
Small-size applications vs. Medium-size applications	<0.001	0.594
Small-size applications vs. Large-size applications	<0.001	0.893
Medium-size applications vs. Large-size applications	<0.001	0.563

* All p-values are BH-adjusted

This conclusion is completely contrary to our initial expectation that large-size applications should use concurrency constructs most intensively. One possible explanation for this is that the adoption of concurrency constructs in large applications is championed by a relatively small proportion of developers. This is confirmed by our experimental data. To be specific, we find that the average numbers of developers using concurrency constructs in small-size, medium-size, and large-size applications are 1.45, 2.54, and 3.29, respectively. However, the proportions of these developers are 34.55%, 16.29%, and 16.00%, respectively. Obviously, small-size applications exceed medium-size and large-size applications regarding the ratio of developers who utilize concurrency constructs. For this reason, small-size applications adopt concurrency constructs most intensively.

(2) RQ4.2: Do small-size applications adopt concurrency constructs more quickly than medium-size applications and large-size applications?

We employ the result from the Mann-Whitney U-test for the interval days between application's establishing time and its first use of concurrency constructs to answer RQ4.2. In particular, we apply Figure 5 to show the distribution of the data for the studied applications. On average, the interval days for the three groups are 190, 571, and 1225, respectively. In other words, small-size applications spend fewer days to start using concurrency constructs than medium-size and large-size applications. The result of statistical analysis in Table VIII confirms our observation from Figure 5. To be specific, Table VIII lists the significance (p-value) and magnitude (Cliff's δ) of the difference, respectively. According to this table, small-size applications significantly spend fewer days to introduce concurrency constructs compared to medium-size applications (p-value = 0.016) and large-size applications (p-value = 0.002). Moreover, the effect sizes are either medium or large in terms of Cliff's δ ($0.375 \leq |\delta| \leq 0.514$). In addition, the comparison between medium-size applications and large-size applications also shows a significant result (p-value = 0.016). The magnitude of difference, however, is small regarding Cliff's δ ($|\delta| = 0.297$). In general, the core observation from Table VIII is that small-size applications significantly spend the fewest days to start using concurrency constructs. Therefore, we have the following conclusion for RQ4.2:

Small-size applications adopt concurrency constructs significantly more quickly than medium-size applications and large-size applications.

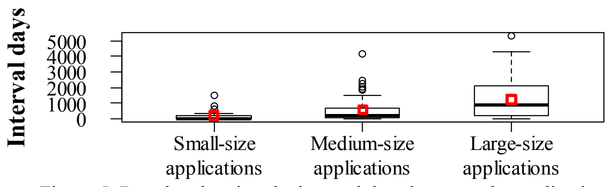

Figure 5. Boxplot showing the interval days between the application establishing time and the first use of concurrency constructs

Table VIII. Result of Mann-Whitney U-test for RQ4.2

Groups for comparison	**p**	δ
Small-size applications vs. Medium-size applications	0.016	-0.375
Small-size applications vs. Large-size applications	0.002	-0.514
Medium-size applications vs. Large-size applications	0.016	-0.297

* All p-values are BH-adjusted

We deduce the possible explanation for this conclusion lies in the different ways to manage the development of open-source applications. More specifically, in comparison with large-size applications (128 developers on average) and medium-size applications (36 developers on average), small-size applications (5 developers on average) generally have a smaller number of developers. With fewer populations in the project group, small-size applications are easier to manage. For this reason, the developers of small-size applications may have more freedom to develop the software in their own ways. As a result, some practitioners can have opportunities to try the advanced language features, such the concurrency constructs introduced by C++11.

Combining the results for RQ4.1 and RQ4.2, we come to the following conclusion for RQ4:

Applications with different size adopt concurrency constructs differently. In particular, small-size applications introduce concurrency constructs more intensively and more quickly than medium-size applications and large-size applications.

E. RQ5: Does an increasing use of standard concurrency constructs result in a substantially decreasing use of unstandardized concurrency constructs?

We employ the result from the Spearman's rank correlation analysis to answer RQ5. Table IX reports the detailed result of the correlation between the use of standard concurrency constructs and the use of unstandardized concurrency constructs. The columns "rho" and "p-value" state for each application the Spearman's correlation coefficient and the statistical significance of the coefficient, respectively. In particular, those insignificant results (p-values > 0.05) are marked with gray background. From this table, we find that only one subject, Stan, shows that the use of standard concurrency constructs and the use of unstandardized concurrency constructs are significantly negatively correlated (p-value = 0.025, rho = -0.513). However, the other five subjects do not show significant results. In particular, four subjects indicate complete irrelevance (p-values = 1). From this reasoning, we cannot confirm the negative correlation between the use of standard concurrency constructs and the use of unstandardized concurrency constructs. Therefore, we draw the following conclusion for RQ5:

An increasing use of standard concurrency constructs does not result in a substantially decreasing use of unstandardized concurrency constructs.

Table IX. Result of Spearman's rank correlation analysis for RQ5

Application name	**rho**	**p-value**
ConcurrentQueue	-0.041	0.416
Hexagen	1.000	1.000
metashell	0.957	1.000
mongo	0.702	1.000
Stan	-0.513	0.025
SuperCollider	0.824	1.000

* p-values > 0.05 are shown in gray background

In order to understand why there are few unstandardized concurrency constructs converted to corresponding standard concurrency constructs, we investigate the existing uses of unstandardized concurrency constructs in our studied subjects. Table X shows the information about the usage of typical unstandardized concurrency constructs. From this table, we see that the constructs provided by the boost::interprocess library is the most widely adopted, amounting to over 50% of total uses. Additionally, the other third-party libraries including TBB, Thrust, OpenMP are also adopted in large amounts. On the contrary, the libraries such as PPL, and HPX are completely not used in practice. By analyzing the inclusive constructs in these libraries, we divide the third-party libraries into two categories. One category consists of the third-party concurrency constructs that cannot be replaced by standard concurrency constructs. The total proportion of this category of constructs is about 70%, mainly contributed by the libraries including boost::interprocess, Thrust, OpenMP, boost::lockfree, and boost::context. The other category is composed of the third-party concurrency constructs which are replaceable by their standard alternatives. These constructs include tbb::task, tbb::atomic, and boost::thread::id. However, developers do not transform the uses of these constructs into the uses of their standardized alternatives. Based on this analysis, we deduce that the explanation for the conclusion of RQ5 lies in that most existing third-party concurrency constructs are not convertible to standard concurrency constructs. Hence, there is not a negative correlation between the use of standard concurrency constructs and the use of unstandardized concurrency constructs.

Table X. Usage of typical unstandardized concurrency constructs

Library name	**% in libraries**	**Construct name**	**% in library**
boost::interprocess	50.70%	interprocess_exception	3.78%
		shared_memory_object	3.19%
		Permissions	3.05%
		scoped_lock	3.03%
		mapped_region	2.58%
TBB	27.74%	Task	12.59%
		Atomic	3.93%
		task_group_context	3.72%
Thrust	9.76%	Complex	14.54%
		detail::functional::actor	9.11%
		null_type	5.47%
OpenMP	8.31%	communicator	29.10%
		Status	10.56%
		Request	8.42%
		packed_iarchive	5.78%
		packed_oarchive	5.42%
boost::lockfree	2.48%	detail::tagged_index	17.58%
		stack::node	15.75%
		queue::node	11.47%
boost::thread	0.75%	Id	100.00%
boost::context	0.26%	guarded_stack_allocator	98.53%
		simple_stack_allocator	1.47%

V. IMPLICATIONS

In this section, we discuss the implications of the experimental results corresponding to the five research questions for developers, novices, and language designers.

For developers, we suggest them paying attention to avoiding the misuses of mutexes, promises, and futures. The result for RQ3 suggests that among the most commonly-used thread-constructs and task-based constructs (mutex, promise, and future), there is not a construct significantly more commonly misused than others. Thus, developers should pay equal attention to avoiding the misuses of these constructs.

For novices, we give the following advice to help them learn C++ concurrency constructs efficiently:

- Suggestion 1: Being familiar with the most commonly-used concurrency constructs. Novices without concurrent programming experience might be confused about which library constructs need to be learnt. The empirical results of RQ1 and RQ2 provide a solution to this problem. To be specific, Table III and Table V show the most commonly-used concurrency constructs and synchronization constructs, respectively. To learn C++ concurrent programming efficiently, novices can pay their attention to understanding the usage of these constructs and practising themselves in writing concurrent programs by using these constructs.

- Suggestion 2: Understanding the actual use of concurrency constructs by learning the real examples from small applications. According to [22], users can efficiently understand an API if they are provided with examples that demonstrate "best practices" for using the API. Since all concurrency constructs are provided as APIs in the C++ libraries, novices can learn their usage by reading real examples from open-source code. Moreover, the result for RQ4 indicates that small-size applications adopt concurrency constructs more intensively and more quickly than medium-size and large-size applications. On ground of this, we suggest novices reading the concurrency code snippets of small applications, which can save their learning time without understanding the complex functionalities of large applications. Also, this can help them quickly come into contact with new C++ features.

For language designers, we advise them to improve the design of standard concurrency constructs by drawing lessons from the practical use of third-party concurrency libraries. By analyzing the result for RQ5, we find that over 2/3 of third-party concurrency constructs do not have corresponding standard alternatives. This indicates that the C++ standard libraries still need to be expanded. For this reason, we recommend C++ designers to introduce more widely-used and domain-independent concurrency constructs (such as the process-based constructs in the boost::interprocess library and parallel constructs in the TBB and PPL libraries) into the future C++ standard library.

VI. Threats to Validity

The possible threat to the **construct validity** is the utilization of the tool "Understand". As introduced in Section III.B, we detect the use sites of concurrency constructs by calling Understand APIs. Thus, the correctness of our collected data depends on the tool's ability in program analysis. Since many historical studies have produced reliable empirical results by using "Understand" [18, 19], the data in our study can also be considered as acceptable.

The possible threat to the **internal validity** is the way to calculate the density of use for concurrency constructs. To answer RQ4.1 and RQ5, we compute the density as the average number of construct uses per thousand source lines of C++ code excluding comments (C++ KSLOC). To validate the findings for RQ4.1 and RQ5, we use another way to calculate the density, namely, the average number of construct uses per file. From the inferential statistics for the file-level density, we have a consistent result. Due to page limitation, we do not report the data in this paper. But this result confirms the conclusions of RQ4.1 and RQ5.

The possible threat to the **external validity** is that we only investigate open-source applications in this study. For this reason, the empirical results may be not applicable to industrial applications, as different ways of software development probably make a difference in the writing concurrent programs. In the future, we will replicate this study on more industrial applications.

VII. Related Work

To our limited knowledge, this paper is the first study to investigate the actual adoption of C++ concurrency constructs. There are three existing studies closest to ours, which focus on C# parallelism and Java concurrency, respectively.

The first related work to our study is an empirical study on the adoption of C# parallel libraries [7]. In [7], Okur and Dig proposed eight research questions and obtained corresponding empirical results. Here, we list some remarkable findings in their paper: (1) in at least 10% of the cases, developers misuse parallel constructs; (2) developers usually make their parallel code unnecessarily complex; and (3) applications of different size have different adoption trends. Due to the fact that C++ does not provide parallel constructs and paralleling patterns as C# does, we cannot completely follow the same way in investigating C# parallelism. However, our study focuses on some different research questions which are specific to C++ concurrency, such as the comparison between the adoption of high-level and low-level constructs, and the correlation between the use of standard concurrency constructs and the use of unstandardized concurrency constructs.

The second related work to our study is an empirical investigation on the adoption of Java concurrency constructs [8, 9]. In [8], the authors put forward two research questions regarding the commonly-used Java concurrency constructs and the transition from concurrency to parallelism, respectively. In a recent paper [9], the authors extended their previous work by investigating research questions with regard to library-based concurrency, shared variables' protection, thread creation and management, thread-safe data structures, condition-based synchronization, and concurrency's exception handling. Due to the fact that C++'s concurrency constructs are not as abundant as Java's concurrency constructs (for example, C++ does not provide thread-safe constructs like ConcurrentLinkedQueue and ConcurrentHashMap in Java), many research questions proposed in [9] are not applicable to be investigated in our study. For this reason, both research questions and research results of the two studies are different in terms of their goals and their nature.

The third related work to our study is an investigation on the adoption of Java concurrency libraries in the Qualitas Corpus [10]. The authors focused on the most-commonly used concurrency classes and methods, and the usage trend of Java concurrency library. The empirical results in [10] showed that: (1) lock and concurrency maps are the most intensively used concurrency constructs; (2) lock/unlock methods are the most often used operations; and (3) concurrency library constructs are increasingly used in the evolvement of the Qualitas Corpus. Compared with this work [10], our study not only employs more precise data analysis methods, but also outlines the implications of the empirical results, thus making our contributions more actionable.

Besides the empirical studies on concurrency constructs, some researchers particularly focus on thread synchronization. Recently, Gu and colleagues [11] have empirically investigated C/C++ software projects to understand how critical sections are changed, how concurrency bugs are introduced, and how developers handle over-synchronization problems.

In addition, other two empirical studies on concurrency investigated the differences between concurrent programming languages [12, 13]. More specifically, in [12], Nanz and colleagues employed a controlled experiment to compare two concurrent programming languages, Java and SCOOP, and they found that SCOOP generally outperforms Java in terms of program comprehending and debugging. Furthermore, in [13], the authors conducted another controlled experiment to compare the usability and performance of multicore languages. To be specific, they compared the languages (Chapel, Cilk, Go, and TBB) with respect to their source code size, coding time, execution time, and speedup, respectively.

Apart from the studies on concurrency, there are a wealth of empirical studies on other language features, such as Java generics [14], C++ templates [15], C *goto* statements [16], and C/C++ assertions [17].

VIII. Conclusion and Future Work

This paper empirically studies C++ concurrency constructs. The whole study is performed by investigating five research questions regarding the use of concurrency constructs, the use of synchronization constructs, the misuse of concurrency constructs, the utilization of concurrency constructs in applications with different size, and the relationship between the use of standard concurrency constructs and the use of unstandardized concurrency constructs. By employing standard statistical inference techniques, we get reasonable results for the proposed research questions. Based on the empirical results, we give salutary suggestions for practitioners to help them design and use concurrency constructs. In future work, we will investigate more research questions and perform an empirical study on more applications to understand the adoption of C++ concurrency constructs in depth. Also, we will show more real examples of the adoption of concurrency constructs in an extended paper.

Acknowledgment

This work is partially supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61170071, 61432001, 91418202, 61472175, 61472178), the National Natural Science Foundation of Jiangsu Province (BK20130014), and the program B for Outstanding PhD candidate of Nanjing University.

References

[1] ISO/IEC. Information Technology—Programming Languages—C++, Third Edition. ISO/IEC 14882-2011. 2011.

[2] B. Stroustrup. The C++ programming language – Fourth Edition. Addison-Wesley, 2013.

[3] A. Williams. C++ concurrency in action. Manning Publications Co., 2012.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, T. Weber. Mathematizing C++ concurrency. In: Proceedings of the ACM SIGPLAN 38th International Symposium on Principles of Programming Languages, 2011: 55-66.

[5] J. C. Blanchette, T. Weber, M. Batty, S. Owens, S. Sarkar. Nitpicking C++ concurrency. In: Proceedings of the 13th International Symposium on Principles and Practices of Declarative Programming, 2011: 113-124.

[6] B. Norris, B. Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, 2013: 131-150.

[7] S. Okur, D. Dig. How do developers use parallel libraries? In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering, 2012: Article No. 54.

[8] W. Torres, G. Pinto, B. Fernandes, J. P. Oliveira, F. Ximenes, F. Castor. Are Java programmers transitioning to multicore: A large scale study of Java FLOSS. In: SPLASH'11 Workshops, 2011: 123-128.

[9] G. Pinto, W. Torres, B. Fernandes, F. Castor, R. S. M. Barros. A large-scale study on the usage of Java's concurrent programming constructs. Journal of Systems and Software, 106, 2015: 59-81.

[10] S. Blom, J. Kiniry, M. Huisman. How do developers use APIs? A case study in concurrency. In: Proceedings of 18th International Conference on Engineering of Complex Computer Systems, 2013: 212-221.

[11] R. Gu, G. Jin, L. Song, L. Zhu, S. Lu. What change history tells us about thread synchronization. In: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2015.

[12] S. Nanz, F. Torshizi, M. Pedroni, B. Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. In: Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, 2011: 325-334.

[13] S. Nanz, S. West, K. S. da Silveira, B. Meyer. Benchmarking usability and performance of multicore languages. In: Proceedings of the 7th International Symposium on Empirical Software Engineering and Measurement, 2013: 183-192.

[14] C. Parnin, C. Bird, E. Murphy-Hill. Adoption and use of Java generics. Empirical Software Engineering, 18(6), 2013:1047-1089.

[15] D. Wu, L. Chen, Y. Zhou, B. Xu. An empirical study on the adoption of C++ templates: library templates versus user defined templates. In: Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering, 2014: 144-149.

[16] M. Nagappan, R. Robbes, Y. Kamei, É. Tanter, S. McIntosh, A. Mockus, A. E. Hassan. An empirical study of goto in C code from GitHub repositories. In: Proceedings of 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2015.

[17] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, B. Ray. Assert use in GitHub projects. In: Proceedings of the 37th International Conference on Software Engineering, 2015.

[18] A. Koru, J. Tian. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. IEEE Transactions on Software Engineering, 31(8), 2005: 625-642.

[19] Y. Zhou, H. Leung, B. Xu. Examining the potentially confounding effect of class size on the associations between object-oriented Metrics and change-proneness. IEEE Transactions on Software Engineering, 35(5), 2009: 607-623.

[20] E. Arisholm, L. Briand, B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software, 83(1), 2010: 2-17.

[21] J. Romano, J. Kromrey, J. Coraggio, J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In: Annual Meeting of the Florida Association of Institutional Research, 2006: 1-3.

[22] M. P. Robillard, R. DeLine. A field study of API learning obstacles. Empirical Software Engineering, 16(6), 2011: 703-732.

[23] GitHub. https://github.com.

[24] SciTool Understand. https://scitools.com/.

[25] The R language. http://www.r-project.org/.

[26] C++ open-source libraries. http://en.cppreference.com/w/cpp/links/libs#Concurrency.

[27] The TBB library. https://www.threadingbuildingblocks.org/.

[28] The PPL library. https://msdn.microsoft.com/en-us/library/dd492418.aspx.