Journal of Information Technology Education: Innovations in Practice

An Official Publication of the Informing Science Institute InformingScience.org

JITEiip.org

Volume 19, 2020

NOVICE PROGRAMMERS' COPING WITH MULTI-THREADED SOFTWARE DESIGN

Rami Rashkovits*	Peres Academic Center, Rehovot, Israel	ramir@pac.ac.il
Ilana Lavy	Yezreel Valley College, Afula, Israel	ilanal@yvc.ac.il

* Corresponding author

ABSTRACT

Aim/Purpose	Multi-threaded software design is considered to be difficult, especially to novice programmers. In this study, we explored how students cope with a task that its solution requires a multi-threaded architecture to achieve optimal runtime.
Background	An efficient exploit of multicore processors architecture requires computer programs that use parallel programming techniques. However, parallel programming is difficult to understand and apply by novice programmers.
Methodology	The students had to address a two-stage problem: (1) design an optimal runtime solution to a given problem with no additional instructions; and (2) provide an optimal runtime multi-threaded design to the same problem. Interviews were conducted with a representative group of students to understand the underlying causes of their provided designs. We used qualitative research methods to gain refined insights regarding the students' decision making during the design process. To analyze the gained data, we used content analysis tools.
Contribution	The case study presented in this paper will help the teacher to stress the merits and limitations of various parallel architectures and confront students with the consequences of their solutions via performances' benchmark.
Findings	Analysis of the student's solutions to the first stage revealed that the majority of them did not provide a multi-threaded solution ignoring the optimal runtime requirement. At the second stage, seven various architectures were provided differing in the number of involved threads, the data structures used, and the synchronization mechanism employed. The majority of the solutions were sub-optimal and only a few students provided an optimal one.
Recommendations for Practitioners	We recommend conducting class discussions that will follow a task similar to the one used in this study.

Accepting Editor Justin Filippou │ Received: May 10, 2020 │ Revised: June 22, July 2, July 15, 2020 │ Accepted: July 20, 2020.

Cite as: Rashkovits, R., & Lavy, I. (2020). Novice programmers' coping with multi-threaded software design. *Journal of Information Technology Education: Innovations in Practice, 19,* 75-89. https://doi.org/10.28945/4609

Recommendations for Researchers	To be able to generalize the received results this research should be repeated with larger study participant groups from various academic institutions.
Impact on Society	Understanding the difficulties of novice programmers may lead to quality software systems.
Future Research	To be able to generalize the received results this research should be repeated with larger study participant groups from various academic institutions.

Keywords	multi-threading, parallel programing, novice programmers, thread-synchronization

INTRODUCTION

Multicore processors entered the computer industry rapidly and became the standard architecture of computing hardware (Creeger, 2005). An efficient exploit of multicore processors architecture requires computer programs that use parallel programming techniques (Lee, 2006). The curriculum of computer science programs has always included a capstone course on operating systems, in which issues related to parallel computing was addressed (Sahami et al., 2013). With the development of multi-processors and multicore technologies the importance of parallel programming increased. Any software built nowadays, such as application servers, e-commerce infrastructures, or ERP systems utilizes the advantages of parallel architecture to yield better performances.

The concepts involved in parallel programming are introduced in mandatory and elective courses (Sahami, 2013). The main issues discussed in these courses refer to mechanisms in which a program is divided into subprograms (i.e. threads) running in parallel each handling an independent task. Efficient parallel programming addresses two main issues: (1) the number of concurrent subprograms; (2) the coordination among subprograms to avoid collisions. Each subprogram requires computing resources such as allocated stack, operating system scheduling, dynamic memory, and so forth. The number of concurrent subprograms may affect the performances of the program. On the one hand, too little may not optimally utilize the available computational power but, on the other hand, too many may lead to an overload of computational resources and hence harm performances (Khot, 2018). Subprograms running concurrently sometimes need to coordinate activities. Moreover, they often use common data structures and need to coordinate access to them to avoid collisions. To achieve this coordination, a variety of synchronization mechanisms was developed (e.g., semaphore, mutex, a monitor) (Khot, 2018). Proper use of these mechanisms is essential for accomplishing consistent results. However, using coordination mechanisms in a way that harms concurrency unnecessarily will result in performance degradation.

The issue of parallel programming is difficult to understand and apply by novice programmers (Sutter & Larus, 2005), especially to students (Benaya & Zur, 2007). Benava and Zur (2007) found that students encounter difficulties in understanding the synchronization mechanism embedded in threads implementation. For instance, when designing a multi-threaded solution, one has to identify possible conflicting code segments and design proper synchronization between them to avoid collisions that may lead to faulty execution. However, one must consider the impact of the chosen solution on concurrency and avoid redundant synchronization that will cause unnecessary performance reduction. Designing a proper solution that will be best organized, consistent, and efficient necessitates high order thinking abilities. To cope with such a design task, which amalgamates knowledge of data structures, algorithms, programming, hardware (i.e., memory, multicore, stack), operating systems (i.e. process scheduling, memory management) and software engineering (i. e. modularity, classes, and method design), one has to be able to analyze, synthesize, and evaluate all of the above into one coherent solution. Such capabilities were ranked as high order thinking abilities in Bloom's taxonomy (Krathwohl, 2002). Awareness of the complexity involved in designing proper and efficient solutions as described above yielded the development of assisting tools to help students visualize and internalize multi-threaded programs (Adams et al., 2018; Wakatani & Maeda, 2018). Yet, the use of such tools during academic studies is not widely spread.

The current study aims to explore novice programmers' understanding and applying multi-threading. For that matter we: (1) explore the level of knowledge and understanding of novice programmers regarding parallel programming; (2) characterize novice programmers' difficulties in designing solutions to problems that require parallel programming; and (3) recommend suggestions for the instruction of parallel programming subjected to findings. We conducted a study in which students were provided with a design task where a parallel design was required. We classified the students' solution strategies and difficulties regarding the threading-architecture they chose and the synchronization mechanisms they implemented.

BACKGROUND

A thread of execution is a sequence of programmed instructions that can be managed independently by the operating system (Butenhof, 1997). Threads were developed to enable parallel executions of sub-processes within a single process. For instance, when a web server accepts concurrent requests, it uses threads to address each request in parallel. The threads share computing resources such as CPU and memory. If more than one processor is available, the threads are executed concurrently. Otherwise, they switch the CPU according to their priorities. If the priorities of several threads are the same, the operating system will flip the CPU among them, providing each time slot for execution. A thread of execution may need several time slots. Since threads may also share common memory, they might harm each other's consistency. Hence, conflicting code sections (i.e., critical sections) using shared memory must synchronize access to these data structures to avoid collisions. Many programming mechanisms are available for this purpose, among them monitors and semaphores (Andrews, 1991; González, 2017). A monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait in a blocking state for a certain condition to become false. Monitors also have a mechanism for signaling other threads that their condition has been met. They use a lock object in a way that to execute a critical section of code a thread must lock the object. This can be achieved only if the lock object is not locked by a conflicting thread. If the lock object is not available, a thread gets into a waiting list, and cannot continue. When a thread finishes its critical section, the lock object is freed and one of the waiting threads is signaled and dequeued, enabling it to catch the lock object and continue to execute its critical section. Semaphores use similar constructs; however, they allow multiple threads to use simultaneously shared resources (González, 2017). However, it is the responsibility of the designer and programmer to detect the conflicting sections and synchronize their access.

When using multi-threading, one should decide whether to use a multi-threaded or thread-pool architecture. While the first refers to the creation of dedicated thread to each subtask, as many as they are, the second refers to a pool of N threads handling all the subtasks that are in a queue, one at the time. An overflow of threads might occur when using the first architecture, which may lead to an overload of computing resources and hence harm execution. The use of the second architecture necessitates the setting of N, which is a fixed number of threads that are responsible to execute the subtasks. Setting an appropriate value for N might maximize concurrency (performances) without overloading the system capabilities. However, inaccurate value for N may lead to either over or underuse of computational resources. An appropriate value may be set according to trial and error tests or via a sophisticated algorithm based on the environment where the solution is deployed.

In some solutions, additional problems related to multi-threading may arise. A thread may starve if it is endlessly waiting to execute due to low priority or biased assignment of computing resources. A cycle of threads may be formed if each thread is waiting for computing resources (e.g. shared memory) occupied by other threads causing a deadlock in which all threads are in hold. These two phenomena require sophisticated algorithmic treatments such as dynamic priorities and cycle detection algorithms.

Teaching multi-threaded related issues to novice programmers is considered to be a difficult task. The students have to get used to parallel thinking, which is far more complex than single-threaded thinking, and develop parallel computing skills. The teachers have to find effective ways to teach complex issues such as concurrent execution and synchronization mechanism among threads for coordination. Shene and Carr (1998) discuss the problems and difficulties teachers encountered when teaching multi-threading programming and present a set of comprehensive and flexible course materials. They also suggest pedagogical tools for visualization and experiment with various multi-threaded concepts. Many approaches for teaching multi-threaded programming were suggested in the literature, including the use of analogies and visual demonstrations such as house building and demonstration of the building process using single vs multi-threading techniques (Giacaman, 2012). Yet another example of such an analogy is the modified version of the virtual world game Minecraft that was used to teach multi-threading (Förster et al., 2016). The students implemented multi-threaded agent software.

Since concurrent activities are difficult to follow, many researchers suggested visual tools to facilitate students' understanding of threads' behaviors. Bi and Beidler (2007) provide a visual tool that displays the execution of threads and the synchronization among them in a graphical manner. ThreadMentor is another pedagogical tool used to facilitate students' understanding of C++ multi-threading concepts (Carr et al., 2003) and visual debugging and testing tools for multi-threading environments were suggested in Larson and Pating (2013).

Students can use these tools to track threads' states and follow synchronization operations. JThreadSpy is another supportive tool that collects execution traces of threads and displays them using UML sequence diagram (Malnati et al., 2008), enabling students to follow complex states such as deadlock situations, and understand their causes.

In this paper, we do not focus on teaching approaches. Instead, we test the perception of multi-threading concepts by novice programmers. We provide them with a problem in which parallelism would certainly contribute to the solution efficiency. However, novice architects must understand the consequences of the parallelism mechanism they chose and select proper synchronization mechanisms to support parallelism. If these mechanisms would not be chosen very carefully, efficiency may be harmed dramatically.

THE STUDY

In this section, we provide information about the research tools, the study participants, the data research, and analysis tools.

THE PROBLEM

In this section, we describe the design problem that was handed to the study participants. The problem was designed in a way that an efficient solution would make use of parallel programming. To avoid redundant complexities, we chose a rather simple problem that can be easily perceived by the problem study participants, enabling them to focus their efforts on the solution design rather than the understanding of the problem. Based on the students' prior knowledge in Java programming in general, and multi-threading Java architectures in particular, we conducted a pilot study with ten students and handed them the first version of the problem (part A) in which no specific requirements regarding threading architecture of any kind were specified. After reviewing the students' solutions, realizing that many of them did not use multi-threading architecture, we added the second version of the problem (part B) in which specific instruction concerning the use of multi-threaded architecture was provided. The two-stage problem was then handed to the thirty study participants, while part B was handed upon completion of part A.

Part A

Design a Java program that calculates the number of occurrences of all alphabetic letters ‘a-z’ (case-insensitive) in a given folder of text files. Assume that the number of files in the folder and their length is unknown. Also, assume that there already exists a mechanism that uploads a chunk of text into a queue. Also, there is no prior knowledge concerning the computer on which the program will run on, namely the hardware and operating system. Provide an algorithm and data structures that solve the above problem. The proposed algorithm will focus on minimizing runtime.

Part B

Design a parallel multi-threaded solution to the former problem, aiming to minimize runtime. In your detailed design refer to the following: (1) threads and their pseudo-code; (2) data structures used by the threads; (3) synchronization between threads - if relevant.

SOLUTION DESIGN

The algorithm suits counting letters are the counting sort procedure, which is built on an array of buckets, each counts a certain letter’s occurrences. Figure 1 presents a schematic description of the data structure’s constituents. It also includes synchronization mechanisms (keys representing monitors) required for solutions that use multi-thread architecture in which several threads need to access simultaneously to the same buckets inside the array.

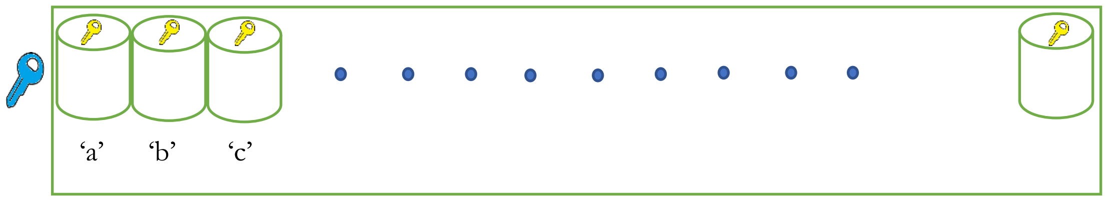

Figure 1: Array of 26 buckets (one for each letter)

In the process of the problem design and its solution, we came up with three available architectures based on the number of used threads: (1) a single thread architecture in which all text processing is done by one thread, each text chunk at a time; (2) a multi-threaded architecture in which an unlimited number of threads are processing text chunks each by a dedicated thread; (3) a thread-pool architecture in which a predefined number of threads, N, process all text chunks that are waiting in a queue.

Table 1. Solutions architectures

No.	No. of threads	Data structures	Synchronization
1	1	An array of 26 “buckets”	
2	Unlimited – one per block	Global Array of 26 “buckets”	Access to array
3			Access to an array index
4		Local arrays of 26 “buckets”	No synchronization. Buckets are collected after all threads are done
5	A pool of N threads	Global Array of 26 “buckets”	Access to array
6			Access to an array index
7		Local arrays of 26 “buckets”	None

In addition to the above classification, one should decide whether threads will share the array of buckets or each will occupy its array. If a common array of buckets is chosen, one should decide how the threads will access the buckets simultaneously without damaging consistency.

Based on the observations described above, we came up with seven architecture variations shown in Table 1. In what follows, we provide a brief description of the architectures shown in Table 1 and discuss their advantages and shortcomings.

Single-Threaded Architecture

Architecture 1. This architecture is the simplest. It requires only one array of "buckets", one for each alphabetic letter. Then, the single thread scans the queue until the queue is empty, each text chunk is removed from the queue and scanned. The buckets are updated accordingly without any need to synchronize access with other threads. The advantage of this architecture is its simplicity. However, this architecture lacks the utilization of multi-processors (if exist) for concurrent analysis of text chunks, which lead to a long runtime.

Multi-Thread Architecture

In this architecture, the text chunks that are uploaded to a queue are each processed by a dedicated thread. That is, each text chunk is removed from the queue and a new thread is created to process it, in parallel to others. The following architectures are variations of the above architecture, each uses different data-structures and synchronization mechanism.

Architecture 2. All the threads access a global array of buckets (shared memory) simultaneously, hence they should synchronize with other threads upon accessing the global array (see Figure 1 – blue key). The advantage of this architecture is the utilization of multiprocessors (if exist) for concurrent analysis of text chunks. This concurrency can improve performance. However, the creation of a vast number of threads, each takes time to construct, each consumes computing resources may lead to a decrease in performance. Also, synchronizing the threads over a single array dramatically damages concurrency. To illustrate the problematics of this solution, consider a thread that needs to update the 'a' bucket that must wait until another thread finishes to update 'b' bucket. The execution of the threads becomes sequential due to the synchronization mechanism used.

Architecture 3. The only difference between this architecture and the previous one is that the threads in this architecture synchronize the access to the buckets rather than to the array which holds the buckets, so that a thread that wants to update a certain bucket collides only with threads that access the same bucket (see Figure 1 – yellow keys) and not with all the other threads as described above. The advantage of this architecture over the previous one is that the level of concurrency increases dramatically since much fewer collides occur. However, the same amount of threads are constructed and activated, consuming many computing resources and thus harming efficiency. Though fewer collides occur between the threads, many conflicts will still happen between threads that simultaneously need to update the same bucket, yet concurrency is still affected, and performances decrease.

Architecture 4. In this architecture, similarly to the previous one, each text chunk is processed by a new thread. However, in this case, each thread manages its array of buckets, and no global array is used, hence there is no need to synchronize with other threads. After all threads end, all the local buckets are collected. The advantage of this architecture is that there is no need to synchronize the threads, and hence concurrency is not damaged. The shortcomings are identical to the ones detailed in the previous architecture. Also, an overload of memory dedicated to the creation of many local arrays (one per thread) might occur, causing a waste of memory. In the case of very large input, the program can be aborted due to insufficient memory.

Thread-Pool Architecture

In this architecture, instead of creating new threads to handle each text chunk, the program constructs a pool of N threads. Each of these threads removes one text chunk from the queue and processes it. Upon completion of one chunk, the thread removes another text chunk from the queue and so on. The number of threads in the pool is determined according to the number of available processors in the target host to maximize concurrent processing. Similar to the previous architectures, we distinguished several variations of this pool-based architecture based on data structure and synchronization used.

Architecture 5. Resembling architecture 2, the N threads are using a global array of buckets and synchronize the access to the array. The shortcomings of this architecture remain the same as in Architecture 2. The only improvement over architecture 2 is that it does not overload the system with too many threads.

Architecture 6. Resembling architecture 3, the N threads are using a global array of buckets but synchronize over the array's buckets rather than the array itself. The only improvement over architecture 3 is that it does not overload the system with too many threads.

Architecture 7. Resembling architecture 4, each one of the N threads creates its local array of buckets, updating the buckets without the need to synchronize with other threads. Upon completion, all N arrays are collected to yield the result. The only improvement over architecture 4 is that it does not overload the system with too many threads.

The Study Participants

The data were collected during the academic years 2019. The study participants were third-year Information Systems students in a rural academic college. Thirty students participated in the research and all were graduated from the following programming courses: "Object-oriented Programming" and "Data Structures and Algorithms". In these courses, the students were exposed to multi-thread programming. They were familiar with the concepts involved in multi-threaded programs including thread synchronization methods. They also practiced these concepts via relevant programming examples and assignments.

Data Collection and Analysis Tools

To address the research aims, a task that includes a problem comprised of two parts was built (see Problem section). The first part aimed to explore whether the solution chosen by the students includes multi-threaded architecture. In the second part, we aimed to examine how students implement the solution using threads. That is, the tasks assigned to each thread, the data structures used, the identification of conflicting critical sections, and the synchronization between the threads. The first part of the problem addresses the question of whether the students use threads, and the second question refers to how they design a multi-thread solution. The goal of minimum runtime was set to examine whether students are using synchronization mechanisms efficiently without harming concurrency.

We used qualitative research methods to gain refined insights regarding the students' decision making during the design process of a solution to a problem that includes the use of multi-threading. The research data were the students' solutions to the given problems. After scanning the students' solutions and classifying them based on the architectures described in Table 1, open interviews were conducted with a representative group of 12 students to figure out the underlying reasons of their provided solutions. The students were asked to elaborate on their decision making as regards to their provided solution and to specify the underlying reasons for each step in it. These interviews were transcribed and were analyzed using content analysis tools (Krippendorff, 2004; Neuendorf, 2016) to identify typical patterns focusing on multi-threading programming and design. Guba and Lincoln (1981) stated that inferences can be derived from content itself since, in their opinion, the process of discovering the characteristics of ideas embedded in the researched content is constructed and based on what is in the material itself rather than from theoretical assumptions or studies that are supposedly imposed on the text.

Firstly, we calculated the frequency of one-threaded vs multi-threaded Part-A solutions used. These frequencies may point to the students' perceptions as regards to multi-threading mechanism. Secondly, we summarized the underlying reasons for Part-A solutions according to the use of one-threaded vs multi-threaded solutions to learn about the consequences of the above perceptions. Thirdly, after careful studying of the students' solutions as regards to Part-B of the problem, we tried to find the best match between each solution and one of the solution architectures presented in Table 1. Fourthly, we calculated the frequencies of each solution architecture to learn on the students' tendencies. Finally, we looked at the interviews' transcriptions to find justifications for each architecture solution to learn about the students' difficulties and to come with implications for instruction.

RESULTS

In this section, we present results separated according to the parts of the given problem.

PART-A

As shown in Figure 2, out of the thirty participants, only 43% (13 students) suggested a solution based on multi-threaded architecture. All the other 57% (17 students) provided a single-threaded solution.

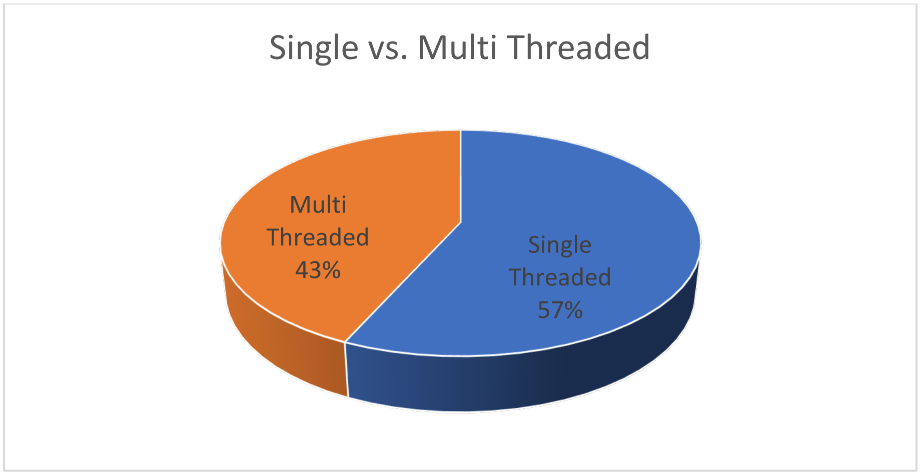

Figure 2. Single vs Multi-threaded solutions

During the interviews conducted with a selective group of the study participants, we asked the students who provided a single-thread architecture to justify their solutions. Here are some of their answers:

"I did not see a need for concurrent processing. In my view there is a need to scan the input serially, hence concurrent processing was not an option."

"Single-thread solution is the simplest to implement, and I saw no reason to complicate it."

"The problem seems to me rather simple, and it did not cross my mind to use multi-thread architecture."

"I got the impression from the courses I took that multi-thread should be used for very complex programs such as database or web servers."

From the above quotes, we can imply that these students ignored the “minimize runtime” constraint and chose the easiest way to cope with the problem. Some attributed their choice to their perceived impression that a multi-threaded solution requires “overkill efforts” to such a simple problem.

Part-B

Figure 3 presents the frequencies of the students’ solutions according to the architectures in Table 1.

As shown in Figure 3, twelve participants provided a solution like Architecture-2, in which each text chunk is handled by a newly created thread, which synchronizes the access to a single array of buckets to update the bucket.

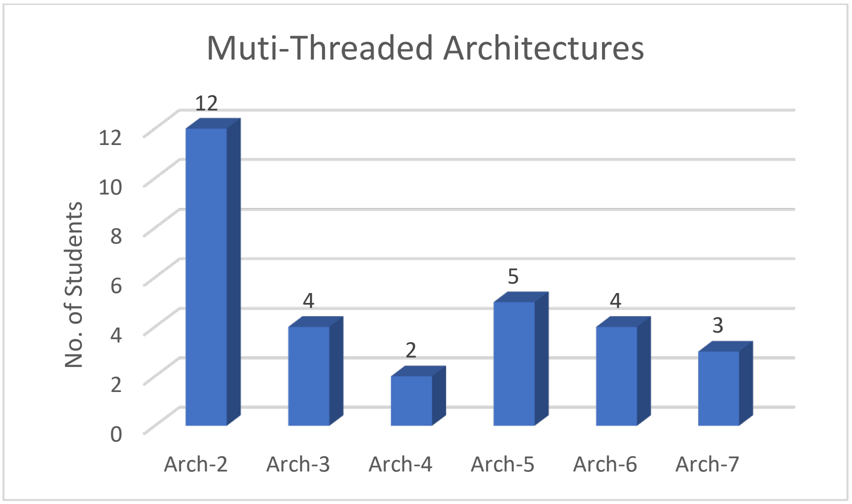

Figure 3. Multi-threaded architectures

Here are some of the students’ reflections from the interviews:

> *“Since we were asked to provide a multi-threaded solution, I found this solution to be appropriate. I assumed that a single folder of text file does not contain too many input files, thus the threads’ number would not exceed the maximum allowed. As to the synchronization, I did not notice that accessing the array would cause such a performance derogation.”*

> *“I can see the flaws in my solution now. If I had coded the solution rather than just sketch a solution design, I would probably come up with a better solution.”*

From the above quotes, it is obvious that the students did not reflect on their solution upon its completion and did not validate their solution against the problem constraints. To justify this behavior, they made up assumptions regarding the input size and the complexity level of the problem in a way that enables them to provide simple solution ignoring the runtime constraint. The second quote illuminates additional problems referring to the design process, which necessitates high order thinking capabilities. These capabilities are necessary for simultaneously gain an overview of the problem constituents and be able to figure out the practical consequences of each (Malnati et al., 2008). Even though novice programmers learned when and how to use threads to gain concurrency and improve performances, yet being inexperienced in such tasks, they need the concrete phase (i.e., coding) to find flaws in their design.

Four participants provided a solution resembling architecture 3, in which each chunk is handled by a newly created thread, but the synchronization between the threads is executed over the buckets inside the array rather than the array itself, which improves the concurrency level in comparison to the previous architecture.

Here are some of the students' reflections from the interviews:

"I thought a lot how to address the problem's constraints and I noticed that synchronizing access to the array is a bad idea since all threads compete with all the others. Hence I suggested a better synchronization mechanism to increase concurrency."

"After I thought a lot about a proper solution it came to my mind to use Thread-Pool architecture, but I found it difficult to design. I believe such a solution fits many complex problems."

From the above quotes, we can learn that these students invested efforts in designing proper synchronization mechanisms, and came up with a "halfway" solution, in which concurrency is improved but is not yet optimal. The concept of Thread-Pool was considered by some of them but eventually was not chosen because they found it difficult to conceptualize.

Only two participants provided a solution resembling architecture 4, in which each chunk is handled by a newly created thread, but unlike architecture 4, the synchronization between the threads is avoided, since each thread holds its array of buckets. Hence concurrency is increased.

Here are the students' reflections from the interviews:

"I tried to think of ways to avoid synchronization since I know it harms concurrency. I must admit it took me a while to get to this idea."

"Unfortunately, I did not notice that with an unlimited number of threads I waste a lot of 'space' to hold their buckets until they are collected at the end."

To come up with this solution, one must 'run' different solution scenarios in mind to examine the virtues and shortcomings of each one. However, when focusing on one of the solution's aspects, one might neglect other important issues, as occurred in this solution. If not all the design aspects are considered, it may result in a suboptimal solution.

Five participants provided a solution resembling architecture 5, in which a pool of fixed number of threads handles the text chunks; however, they are all synchronized over a single array. Hence concurrency is decreased.

"I used thread-pool architecture since I understood there are many text chunks to process. But I did not pay sufficient attention to the synchronization mechanism and failed to provide an efficient one."

Resembling the students' utterances concerning architecture 4, it turns out that when focusing on one aspect sometimes not enough attention is paid to other ones. As in the previous case, the solution is suboptimal.

Four participants provided a solution resembling architecture 6, in which a pool of fixed number of threads handles the text chunks, with synchronization over the buckets rather than the array as in the previous architecture. Hence concurrency is increased in comparison with architecture 5.

"I think my solution was a pretty good one. If I had thought that synchronization can be completely avoided, I would have surely definitely used such a solution."

Though this solution is very good, yet creative thinking was needed to come up with a better one. Although the synchronization mechanism used in this architecture is the best possible, there is a solution that avoids synchronization completely. However, the latter is far less intuitive than the former, hence fewer students were able to think of it. Creative thinking is usually developed with exposure to a variety of scenarios gained via professional experience (Cennamo et al., 2011; Treffinger et al., 2002). Since our participants are students considered as novices, they did not yet develop such skills.

Three participants provided a solution resembling architecture 7, in which a pool of fixed number of threads handles the text chunks, each holds its array of buckets. After all processing threads end, the buckets are collected to calculate the frequencies of the letters. Since no synchronization between the threads is required, the concurrency remains intact and the performance is increased to the maximum.

Here is one of the students' reflections from the interviews:

> *"I love challenges. I understood that to provide the best possible solution I must limit the number of threads involved. I also realized that a key factor in raising performances lies in high concurrency, which is minimum synchronization among the threads."*

The students that provided the optimal architecture succeeded to identify the key factors that lead to the achievement of the optimal solution. The first factor refers to saving of computer's resources by limiting the number of threads, and the second factor refers to the increase of concurrency to the maximum. The optimal solution was achieved via creative 'out of the box' design in which all aspects of the solution are considered, and an optimal design decision was taken to address each.

DISCUSSION

When we first introduced the problem to the study participants, we did not include any hint as regards to the desired solution architecture. Surprisingly, more than half of them (see Fig. 2) provided a single-threaded architecture despite the requirement for runtime minimization. The students justified this selection by uttering that they perceived the problem as one that a sequential solution better fits than parallel one, and that multi-threading should be used in more complex problems. We may conclude from these findings and reflections that multi-threading has not yet become an integral part of the students' professional toolbox although they are acquainted with these issues. This might be attributed to the lack of the participants' experience in solving problems out of a course context and the absence of high cognitive abilities among many of them. This absence is expressed in their inability to break down a problem into smaller ones, inability to understand that parallel processing of the files reduces runtime, and their inability to integrate a solution that addresses the runtime minimization requirement. In Lee (2006), we may find an additional explanation of the obtained results. He suggested that despite humans' ability to reason about concurrent physical dynamics, one cannot extend this ability to concurrent programming because programming environment does not resemble the physical world's concurrency. Moreover, the sequential structure of programming languages does not contribute to the development of parallel design (Banerjee et al., 1993). This is in line with our findings that many novice programmers do not possess sufficient multi-threading skills.

We mapped the solutions-based architectures according to main dimensions: thread-architecture and synchronization configuration. As to thread-architecture, the students had to figure out which of the following meets the problem requirements: single-thread, multi-thread, or thread-pool. As to the Synchronization setup they had to figure out which mechanism will give the best performances to address the problem requirements. Figure 4 presents our mapping. Each participant provided one of the architectures stated above. Her choice points to certain levels of understanding of each of the above dimensions. Table 2 summarizes the theoretical and practical knowledge required to end with such decisions.

Architecture			
Thread-Pool	Arch-5	Arch-6	Arch-7
Multi-Thread	Arch-2	Arch-3	Arch-4
Single-Thread			Arch-1
	Array	Index	None
		Synchronization	

Figure 4. Architectures according to number of threads and synchronization resolution

Table 2 demonstrates that to end up with the optimal solution (Architecture 7), one should master all theoretical and practical knowledge on both thread architectures and synchronization mechanisms. That is, one understands that limiting the number of involved threads is necessary to control the computing resources. One also understands that synchronization harms concurrency hence avoiding conflicting sections is even better than solving them efficiently. As a result, one ends up with a creative solution that uses a pool of threads who process the input files, each holding its buckets avoiding the need to synchronize with other threads. This solution maximizes parallelism leading to the best performance. The level of understanding required to provide such a solution is very high, which can explain the small number of such solutions (see Figure 3).

Table 2. Actions based on theoretical and practical knowledge

Theoretical knowledge	**Practical knowledge**	**Action**
Concurrent Programming	Multi-Threading	Divide a task into independent sub-tasks
Computing Resource Utilization (processors, memory, etc.)	Thread-Pool Vs. Multi-Threading	Use Thread-pool architecture to limit the number of threads
Conflict of Critical Sections	Synchronization Mechanism	Use monitors to protect access to the common array
Synchronization mechanisms harm performances	Reduce to minimum conflicting sections	Use a monitor to protect specific buckets, not on the entire array
Synchronization mechanisms harm performances	Avoid concurrency damage by bypassing synchronization	Use a separate array for each thread (avoid synchronization completely)

Students who end-up with architecture 6, also demonstrated a high level of understanding, but they could not think of a mechanism that completely avoids the synchronization. They chose thread-pool architecture, and their solution reduces the conflicting sections to a minimum, by allowing threads to process concurrently unless they conflict on the same letter (e.g., need to update the same bucket in parallel). As in the previous case, only a few students demonstrated the knowledge required for this solution which is the second-best. Students who provided architecture 5 as a solution, demonstrated mastery over the knowledge of the thread-pool architecture; however, as to the synchronization dimension, they demonstrated only basic understanding. That is, they understood the need for synchronization between conflicting sections, but they failed to understand the impact of their solution on performance. They prevented parallel processing of different letters unnecessarily by synchronizing the threads over the array rather than the buckets.

As to the students who provided solutions based on architectures 2, 3, 4 they all demonstrated partial understanding regarding thread architecture; namely, they allowed an unlimited number of threads to operate in parallel. However, these students were differed by their levels of understanding as regards to the synchronization dimension. Students who provided solutions based on Architecture 4 demonstrated a high level of understanding regarding the synchronization mechanism by providing a solution that avoids synchronization, while students who provided solutions based on architecture 2 and 3 demonstrated a low to medium level of understanding regarding synchronization by failing to provide a bypass solution.

As to the students who provided solutions based on architectures 2, 3, 4, they all demonstrated partial understanding regarding thread architecture. Namely, they allowed an unlimited number of threads to operate in parallel. However, these students were differed by their levels of understanding as regards to the synchronization dimension. Students who provided solutions based on Architecture

4 demonstrated a high level of understanding regarding the synchronization mechanism by providing a solution that avoids synchronization, while students who provided solutions based on architecture 2 and 3 demonstrated a low to medium level of understanding regarding synchronization by failing to provide a bypass solution.

Parallelism calls for thinking that is not intuitive for the human brain (Marowka, 2008). Concurrent programming refers to many concepts that are not part of humans' sequential thinking (e.g., deadlocks, starvations, shared memory, synchronization) and hence makes it most difficult for programmers to comprehend (Malnati et al., 2008). Moreover, an increase in program concurrency may cause a reduction in performance rather than an improvement, if not properly designed (McKenney, 2017). To be able to properly utilize multi-threading, one has to be proficient in the hardware and software aspects relating to concurrent programming and be aware of its merits and limitations. Also, parallel computer programs are difficult to understand and debug (Adams et al., 2018) hence designing such programs is even more complex. All the above necessitates abstraction abilities and high order thinking skills.

CONCLUDING REMARKS AND IMPLICATIONS FOR INSTRUCTION

In this study, we found that many students, who are considered novice programmers, find it difficult to come up with an optimal design of a solution that includes multi-threading. We found that many of them avoid multi-threading, although runtime efficiency is required. When they are instructed to use multi-threading, many of them choose an uncontrolled number of threads and perform unnecessary synchronization operations that harm concurrency for no reason.

Hence, we suggest the following recommendations for the instruction of multi-threading:

1. In addition to the instruction of the multi-threading constituents (i.e., work partition, critical sections, synchronization mechanisms) we recommend conducting class discussions that will follow a task resembling the one used in this study. The different solutions will be analyzed and discussed thoroughly stressing the merits and limitations stemming from each solution.
2. As part of the discussion, the students will be asked to implement their solutions. A benchmark will be conducted on various inputs to compare the performances of each provided solution. To emphasize the differences between solutions, the teacher will draw a performance graph to enable students to visualize the consequences of their design and internalize it. This activity is crucial to the development of reflection abilities that are essential to design tasks.
3. Following the discussion, the students would get another assignment in which they will be asked to solve a similar problem. This assignment aims to validate that the students internalized the concepts involved in multi-threading.

When difficult and complex concepts are concerned that require analytical thinking skills, we believe that the above suggestions might facilitate the understanding and internalization of such concepts. Class discussions and exposing to a variety of solutions and implementations will enable students to develop reflection abilities that will eventually result in effective learning.

STUDY LIMITATIONS AND FUTURE WORK

This study was conducted in a case study format (Mills et al., 2017) to gain in-depth insights into why novice programmers choose suboptimal solutions to a problem that necessitates multi-threaded architecture.

To be able to generalize the received results, this research should be repeated with larger study participant groups from various academic institutions. Also, we intend to elaborate on additional issues relating to multi-threading such as threads' priorities, threads' starvation, and threads' deadlocks. We aim to explore novice programmers' understanding and implantation abilities of these issues.

REFERENCES

Adams, J. C., Crain, P. A., Dilley, C. P., Hazlett, C. D., Koning, E. R., Nelesen, S. M., & Stel, M. B. V. (2018). TSGL: A tool for visualizing multi-threaded behavior, *Journal of Parallel and Distributed Computing*, *118*, 233-246. https://doi.org/10.1016/j.jpdc.2018.02.025

Andrews, G. R. (1991). *Concurrent programming: principles and practice*. Benjamin/Cummings Publishing Company.

Banerjee, U., Eigenmann, R., Nicolau, A., & Padua, D. A. (1993). Automatic program parallelization. *Proceedings of the IEEE*, *81*(2), 211-243. https://doi.org/10.1109/5.214548

Benaya, T., & Zur, E. (2007). Understanding threads in an advanced Java course. *ACM SIGCSE Bulletin*, *39*(3), 323-323. https://doi.org/10.1145/1269900.1268890

Bi, Y., & Beidler, J. (2007). A visual tool for teaching multi-threading in Java. *Journal of Computing Sciences in Colleges*, *22*(6), 156-163.

Butenhof, D. R. (1997). *Programming with POSIX threads*. Addison-Wesley Professional.

Carr, S., Mayo, J., & Shene, C. K. (2003). ThreadMentor: A pedagogical tool for multi-threaded programming. *Journal on Educational Resources in Computing* (JERIC), *3*(1), 1-es. https://doi.org/10.1145/958795.958796

Cennamo, K., Douglas, S. A., Vernon, M., Brandt, C., Scott, B., Reimer, Y., & McGrath, M. (2011, March). Promoting creativity in the computer science design studio. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 649-654. https://doi.org/10.1145/1953163.1953344

Creeger, M. (2005). Multicore CPUs for the masses. *ACM Queue*, *3*(7), 63-64. https://doi.org/10.1145/1095408.1095423

Förster, K. T., König, M., & Wattenhofer, R. (2016, September). A concept for an introduction to parallelization in Java: Multi-threading with programmable robots in Minecraft. *Proceedings of the 17th Annual Conference on Information Technology Education, Boston, MA,* 169. https://doi.org/10.1145/2978192.2978243

Giacaman, N. (2012) Teaching by example: Using analogies and live coding demonstrations to teach parallel computing concepts to undergraduate students. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshop & PhD Forum, Shanghai,* 1295-1298. https://doi.org/10.1109/IPDPSW.2012.158

González, J. F. (2017). *Java 9 concurrency cookbook*. Packt Publishing Ltd.

Guba, E. G., & Lincoln, Y. S. (1981). *Effective evaluation: Improving the usefulness of evaluation results through responsive and naturalistic approaches*. Jossey-Bass.

Khot, A. S. (2018). *Concurrent Patterns and Best Practices: Build scalable apps with patterns in multi-threading, synchronization, and functional programming*. Packt Publishing Ltd.

Krathwohl, D. R. (2002). A revision of Bloom's taxonomy: An overview. *Theory into Practice*, *41*(4), 212-218. https://doi.org/10.1207/s15430421tip4104_2

Krippendorff, K. (2004). *Content analysis: An introduction to its methodology*. Sage Publications.

Larson, E., & Palting, R. (2013, March). Mdat: A multi-threading debugging and testing tool. *Proceeding of the 44th ACM technical symposium on Computer Science Education, Denver, CO,* 403-408. https://doi.org/10.1145/2445196.2445318

Lee, E. A. (2006). The problem with threads. *Computer*, *39*(5), 33-42. https://doi.org/10.1109/MC.2006.180

Malnati, G., Cuva, C. M., & Barberis, C. (2008, December). JThreadSpy: A tool for improving the effectiveness of concurrent system teaching and learning. *International Conference on Computer Science and Software Engineering, Hubei,* 549-552. https://doi.org/10.1109/CSSE.2008.11

Marowka, A. (2008). Think parallel: Teaching parallel programming today. *IEEE Distributed Systems Online*, *9*(8), 1. https://doi.org/10.1109/MDSO.2008.24

McKenney, P. E. (2017). *Is parallel programming hard, and, if so, what can you do about it?* Linux Technology Center, IBM Beaverton.

Mills, J., Harrison, H., Franklin, R., & Birks, M. (2017). Case study research: Foundations and methodological orientations. *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, *18*(1).

Neuendorf, K. A. (2016). *The content analysis guidebook*. Sage. https://doi.org/10.4135/9781071802878

Sutter, H. & Larus, J. (2005). Software and the concurrency revolution, *ACM Queue*, *3*(7), 54-62. https://doi.org/10.1145/1095408.1095421

Sahami, M., Roach, S., Cuadros-Vargas, E., & LeBlanc, R. (2013, March). ACM/IEEE-CS computer science curriculum 2013: Reviewing the Ironman Report, *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, *Denver, CO*. 13-14. https://doi.org/10.1145/2445196.2445206

Shene, C. K., & Carr, S. (1998). The design of a multi-threaded programming course and its accompanying software tools. *The Journal of Computing in Small Colleges*, *14*(1), 12-24.

Treffinger, D. J., Young, G. C., Selby, E. C., & Shepardson, C. (2002). *Assessing creativity: A guide for educators*. National Research Center on the Gifted and Talented.

Wakatani, A., & Maeda, T. (2018, March). Web applications for the education of parallel programming. *Society for Information Technology & Teacher Education International Conference*, *Washington, DC*, 262-267.

BIOGRAPHIES

Dr Rami Rashkovits is a senior lecturer at Peres Academic Center, head of the Department of Management Information Systems. His PhD dissertation (in the Technion – Israel Institute of Technology) focused on content management in wide-area networks using profiles concerning users' expectations. His research interests are in the fields of distributed content management as well as computer sciences education. He has published over thirty papers and research reports.

Professor Ilana Lavy is an associate professor with tenure at the Academic College of Yezreel Valley. Her PhD dissertation (in the Technion – Israel Institute of Technology) focused on the understanding of basic concepts in elementary number theory. After finishing a doctorate, she was a post-doctoral research fellow at the Education faculty of Haifa University. Her research interests are in the field of pre-service and mathematics teachers' professional development as well as the acquisition and understanding of mathematical and computer science concepts. She has published over a hundred papers and research reports (part of them is in Hebrew).