On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVM

D. Beronić, P. Pufek, B. Mihaljević and A. Radovan
Rochester Institute of Technology Croatia, Zagreb, Croatia
dora.beronic@mail.rit.edu, paula.pufek@mail.rit.edu,
branko.mihaljevic@croatia.rit.edu, aleksander.radovan@croatia.rit.edu

2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO) | 978-953-233-101-1/21/$31.00 ©2021 10.23919/MIPRO52101.2021.9596855

***Abstract* - Various blocking operations, such as input/output (IO) operations, are an essential functionality of concurrent applications. However, these operations often cause efficiency-related challenges in multithreaded applications because of the single-thread access policy. IO operations' speed and efficiency heavily rely on operating system (OS) threads and scheduling processes, which is time consuming because of the insufficient distribution of OS resources. Because of their overgeneralization, required to support various usage scenarios, robust OS kernel threads are commonly heavyweight and oversized, created in a limited number, and heavily dependent on OS schedulers.**

Various asynchronous programming techniques tackled issues in implementing modern concurrent applications with blocking operations with the complete reconstruction of threads and their schedulers. Based on our previous preliminary research, we further explored a model of structured concurrency in Java Virtual Machine (JVM), through the updated implementation of virtual threads and accompanying schedulers, as a novel form of lightweight user-mode threads in the JVM, decoupled from OS threads and managed within the JVM.

In this paper, we present our findings about the efficiency of those schedulers utilized for virtual threads on blocking operations, compared with the implementation of OS threads' schedulers, and review possibilities of reducing memory footprints and context switching costs.

Keywords – Concurrent Programming, Virtual Threads, Java Virtual Machine, Thread Scheduling, Structured Concurrency, Java

I. INTRODUCTION

In recent times, the increase of resources that are being processed and commonly stored on the computer has urged software developers to search for a solution to optimize the contemporary software performance that accomplishes tasks by performing a significant number of commonly heavyweight asynchronous processes. To improve efficiency during such operations, several programming concepts have been proposed to ease the process of improving the program's software architecture quality attributes – most notably, scalability, reusability, and maintainability.

Some programming languages implement the support for concurrent data exchange, the software feature that supports different tasks running concurrently where several underlying operations are individually running in parallel and working on specific suboperations. Examples of those include internal implementations and APIs that are platform-specific and depend on the external hardware resources. The software industry benefits from concurrent features in common multicore environments. For instance, recent proposals include libraries and constructs implemented into existing platforms; Microsoft developed the Task Parallel Library (TPL) and Collections.Concurrent (CC)[1], Intel provides the Threading Building Blocks (TBB) [2], the Java community uses java.util.concurrent (j.u.c) [3] library, and Android framework provides AsyncTask, IntentService, and AsyncTaskLoader[4].

In contrast to synchronous programming, using concurrency might significantly improve the throughput and reduce pause times. Given that asynchrony performance occupies the CPU execution due to the extensive OS resource usage, such underlying processes commonly require the support of both hardware and internal system resources [1]. Additionally, this might also contribute to software developers' decreased efficiency in terms of writing non-blocking asynchronous code that should handle heavy data workload. The overall goal (best-case scenario) would be to entirely exclude the need for developers' activity to manage such constructs' creation and safety, with providing such resource management by the runtime. In a modern computing environment, software developers should be able to write scalable applications that make use of any amount of hardware, with processor cores running in parallel while preserving software reliability.

[1] Task Parallel Library (TLP), https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl
[2] Intel Thread Building Blocks, https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html
[3] Java SE 11 & JDK 11, https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html
[4] Async and AsyncTaskLoader, https://developer.android.com/codelabs/android-training-asynctask-asynctaskloader#0

The focus of this paper would mostly be concerned with the implementation of concurrency in the Java platform. To investigate the performance of the newly introduced lightweight constructs that operate on the JVM-level, in comparison to the traditional approach to multithreading in Java, we have analyzed their executions performed under different tasks and workloads.

II. Background

The concept of threads in computer science and software development represents sequential flow controls and could be described as processor abstractions, and events refer to the abstractions of hardware operations. Threads are typically attributed with weight – how much contextual information must be reserved by a processor to schedule them to accomplish tasks successfully. For example, the context of a Unix process uses the hardware register, kernel stack, user-level stack, and process id, and the time required for switching between Unix processes is considered to be relatively long; therefore, such threads are also called heavyweight threads. In contrast, some other operating systems' kernels, such as macOS and Mach, allow multiple threads working on the same process, decreasing the amount of information context that must be saved for each one, and such threads are considered mediumweight threads. Finally, if context and thread operations could be exposed at the user level, running applications would need a minimal amount of information, and context switching could be further reduced with lightweight threads.

Whenever a new thread is instantiated, its creation is delegated to the operating system, which contains a scheduler that is responsible for managing CPU resources and controls its lifespan. Thread scheduling is responsible for ensuring that every thread will eventually run successfully. Each thread resides in the "pool," and there are many scheduling policies such as Priorities, First Come First Served (FCFS), Shortest Process Next (SPN), Shortest Remaining Time (SRT), and Round Robin [2], upon which this process can be implemented. Round Robin scheduler is commonly used when modeling interactive systems and guarantees that processor resources are equally distributed among threads that must be executed. At such conditions, a low priority task operates on the shared resource that is held in higher-order operation, which consequently might contribute to the delays in the execution of the high priority tasks.

With the rapid development of multi-processor machines, the demand for scaling systems in the modern world of mobile and other connected digital devices is growing. Achieving optimal efficiency of large-scale applications nowadays is possible with the use of the concept of concurrency. Concurrency provides a solution to make effective use of multiple processor cores. The use of concurrent programming has increased, but its basic constructs have still not reached their optimal performance in some real-life scenarios. In recent Java versions, *runtime-managed concurrency* [3] presented the idea of provisioning implicit capabilities by a runtime with the purpose of enabling efficient performance, thus taking care of low-level details. Some of these proposals are addressed in the following sections.

In the asynchronous programming model, code is structured and separated into tasks that execute concurrently. Software developers could make use of concurrency as a tool to respond to a large number of incoming requests, database transactions, I/O operations, or only to perform some operations that would otherwise make the applications pause due to the blocking operations. Although concurrency constructs have evolved in recent times, many developers are building upon low-level control flows without sufficient knowledge of their internal workings, whereas they are able to have control over it if thoroughly understood. For example, two types of concurrent constructs, Java *concurrent collections* and Android *async constructs*, have been studied and analyzed with the aim to fix their misuse and help programmers understand how to use them correctly. Moreover, it addresses the design of existing concurrent constructs and proposes refactoring techniques that introduce new use of such constructs [4].

III. Concurrency in Java

The threading system implemented in Java allows multiple threads to execute concurrently along with the main application thread. Every time a thread is started, its allocation is delegated to the OS and a new native thread is created within an OS and assigned a JVM bytecode interpreter. Once it is run, the interpreter starts its execution. The allocated part of the CPU is controlled by the OS scheduler, responsible for managing processor time and ensuring that an application thread does not exceed its allocated time.

Java's threading model is built on three main concepts:

- *Shared, visible-by-default mutable state* – objects shared between threads in processes, can be mutated by any thread holding a reference to them
- *Thread scheduling* – OS thread scheduler that allocates space on/off a core at any time
- *Locks* – could be hard to use, making the state more vulnerable

Recently, there have been some improvements introduced in the Java platform through concurrency APIs and other contemporary technologies. Some of the prominent models for concurrency in Java and JVM involve [5]:

- Synchronization
- Future and ExecutorService object
- Software transactional memory (STM)
- Actor-based model (Akka)
- Reactive Streams (RxJava, Reactor)

Whenever a resource shared among multiple threads is modified, it can be controlled, and it could be "protected" by using *synchronize* keyword, which is a low-level construct, and, according to recent reports [6], it should be avoided. The Future interface is primarily used in asynchronous computation and it exposes several methods, including checking whether the asynchronous computation

is complete, waiting for the completion, blocking the call (with the ability to provide optional timeout duration), and retrieval of computation's result.

The CompletableFuture class represents a synchronization mechanism introduced in the Java 8 concurrency API and updated in Java 9, extending the Future interface and improving its methods. As with a Future object, the CompletableFuture class represents a result of an asynchronous computation, but the result of CompletableFuture can be created by any thread. However, the exposed interface has several drawbacks, including the callbacks that must be implemented to prevent the blocking of operations.

Other languages for the JVM, for example, Scala and Clojure, have been built with the intention to support concurrency from the start. Also, the Scala-based actor framework Akka can be used within Java code. In comparison to Java, a similar approach might require assigning each task to a heavyweight OS thread that would make a system less scalable and consequently increase the expense of memory usage. Thus, to provide the ability to develop non-blocking concurrent applications, Reactive systems have also been implemented in Java. They define a standard for asynchronous stream processing with non-blocking backpressure – the ability to skip or queue data that is coming too fast to be processed. At their core, they provide high responsiveness of applications and handle more requests in the same unit of time [7].

Even though there are many existing implementations for asynchronous processing, such as Java's Future, CompletableFuture, and parallel streams, not all of them provide standard support for asynchronous handling of heavy data workload. The main concern of such implementation is resource consumption due to the potential overload that might happen when published information becomes too excessive to the subscribers – data receivers.

IV. Resources

Since the dawn of Java, the approach to concurrency has been achieved with the thread mechanism, and the whole platform is based on the concept of thread-like components.

A. Threads

The thread represents an independent scheduled execution unit of concurrency. It is mapped to a software unit that executes a program's operations sequentially at the operating system level, and could be considered as thin wrappers around the OS threads.

OS threads must support various languages, and it is not known in advance how much of the stack will be needed to run efficiently; therefore, a larger contiguous stack might be allocated and they tend to have a higher footprint. Also, task switching happens through the kernel, which might result in additional overhead. The architecture of such a system consists of abstraction and implementation with lightweight processes that act as mediators and allow the creation of faster threads. Although the multithreading implementation provided a standardized interface, simpler model, and more efficient performance for every process in a multithreaded operating environment, these "lightweight" processes still operate as virtual CPUs, structured and scheduled by the OS.

The JVM "virtualizes" resources, either from a processor or I/O devices, and schedules threads' use. In such a process, each computer operation might be employed by different hardware units at different times synchronously, in a blocking manner, with commonly waiting for the occurrence of some event that triggers the continuation of a started process. This wrapper spawned over operating system threads has shown to be insufficient for extensive load in some environments because it requires a lot of resources, and is considered heavyweight [8]. These "carrier threads" are bound to the CPU and mapped to threads with One-To-One unidirectional relationships. Such threads utilize the ForkJoinPool scheduler to create more threads when blocking operations are invoked. If an application has to operate asynchronously at such conditions, the ForkJoinPool scheduler would have to create more threads to keep the parallelism high.

Applying a synchronous and asynchronous approach to perform complex computational tasks is expensive, one in terms of hardware resources and another one wasteful in time. The level of concurrency of an application might be lower than the level of concurrency that the server can support in general, meaning that more resources are needed, or the code should be refactored and implemented in a more performance-wise manner. In many cases, both approaches cannot provide optimal solutions.

B. Virtual Threads

Virtual threads do not remove the existing functionalities of threads but differ in other features such as the footprint and scheduling. The concept of virtual threads in terms of JVM introduces a different approach to software development where certain adjustments are required. With the implementation of these constructs, blocking code should not be avoided since it is not costly. In comparison to the thread approach, the concurrent access to some service could be limited by executing a particular task or subtask in its own thread, from start to end, and implement some "notification" in the form of a limiter for the corresponding service to indicate the task's completion.

According to Java Language Specification [9], the way of setting up threads' resources is not strictly defined. Mapping threads to operating system level threads is flexible; thus, it is possible to extend to the existing application programming interface such as the Thread API. An example of such implementation is Project Loom, which introduces a different execution model where an object representing the execution context does not have to be scheduled by the operating system. Since threads commonly represent a scaling bottleneck for modern JVM applications, the goal of this restructuring was to reduce the cost of these constructs and exclude the need for additional external resources when running complex, large-scale JVM applications.

The main goal of Java's higher-level constructs and virtual thread's implementation is to decouple JVM threads from OS threads. Similar to tasks executed with reactive

programming (streams), the main idea is to release the running thread during blocking operations as soon as it finishes, mostly by providing the virtual thread information context at the user mode level. With this approach, heavy-load thread utilization could be distributed to internal resources and would result in the overall application's higher throughput. Virtual threads are not run on "carried" for long by the OS threads; once they complete the assigned tasks' executions, they are destroyed and released from the memory and cannot be reused. Regular threads are managed by the operating system and blocked inside the JVM, while virtual threads are managed by the JVM and blocked in the Application layer, as presented in Fig. 1.

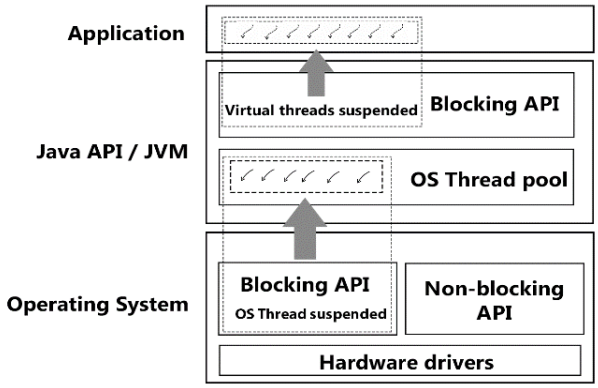

Figure 1. Model of threads and virtual threads creation

Within the Project Loom scope, instantiating and running of virtual threads, offered an alternative approach to the traditional threads. Operations that must be performed in an asynchronous manner are frequently limited by threads supported by native resources. With this approach, we might be able to use millions of virtual threads on a single JVM since they are implemented in the user space by the Java runtime [10]. Furthermore, it could ease the process of debugging the virtual threads. In this case, the entire stack trace of a particular concurrent task is separated into its own, whereas with the OS thread, a thread that contains asynchronous code also contains its stack trace, which might impact the process of analysis of software performance.

V. Testing Environment

A. Testing Environment Setup

Our academic testing environment relies on the OpenJDK Runtime Environment, OpenJDK 64-Bit Server VM installed on ubuntu-20.04 with 13 GB of RAM, 8-core processor, and 80 GB of Hard Disk. For the compilation, we used the early access JDK loom-17ea.2 version of the Project Loom, which supports the updated last version of the creation of virtual threads. The Ubuntu operating system was hosted by Windows 10 64-bit OS with an x64-based processor and 16 GB of RAM.

B. Test Cases

Based on our previous research [11], where the previous version of virtual threads implementation with "fibers" outperformed standard thread implementation in a multithreaded web server, we expected at least similar results and higher concurrency level in specific cases. To measure and compare the strengths and weaknesses for both standard and virtual threads, we selected a specific benchmark that tests threads' creation speed, concurrency, and efficiency level and how they handle blocking operations and synchronization, with different test scenarios. For our analysis, we used benchmarks with common Merge sort, Sequential, and Parallel Run algorithms and compared results in regards of scalability and how they perform under heavier load.

VI. Preliminary Results

After two pre-runs for warm-up, each test was run 10 consecutive times, from which we calculated the average runtime. We analyzed the results based on each scenario's scalability and made a comparison between threads and virtual threads**.** To explain and clarify the results of virtual threads, we examined the processes of thread creation in the scope of several benchmark tests, and while explaining our results, we offered some possible predictions of their usage.

A. Sequential Run

Since the creation of virtual threads relies entirely on the JVM and does not depend on the availability and the size of OS kernel threads, our hypothesis was that in common cases it should be faster than the standard thread creation. To test this, we created a simpler test that created a larger number of both regular threads and virtual threads, and measured how long the creation would last and how many resources it would occupy, with the focus on the usage of OS kernel threads and how fast they can be mapped.

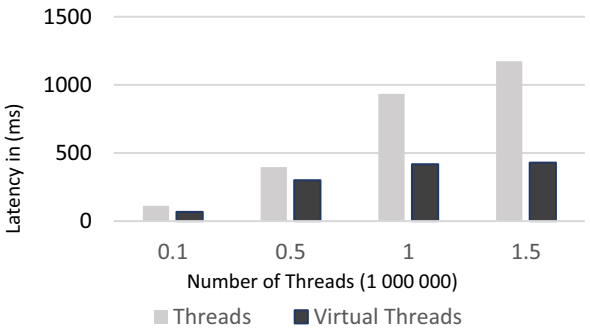

Figure 2. Variation of latencies based on the number of created threads and virtual threads in a Sequential Run

As presented in Fig. 2. we started with creating 100 000 threads, which were on average 1.67 times slower than creating the same number of virtual threads. As we increased the number of threads, the difference became more significant. The average creation of 1 000 000 threads was 2.23 times slower than the average creation of the same number of virtual threads, and for the creation of 1 500 000 threads it was 2.74 times slower. According to these preliminary results, both threads and virtual threads measure a linear increase in their creation; however, in the general case, virtual threads remain more resilient regarding scalability.

B. Multi-threaded Merge Sort

Since examined virtual threads are Java-specific, their schedulers and actions are managed by the JVM, which enables them to be more efficient in blocking operations resulting in better synchronization. To test this, we created a Multi-threaded merge sort and measured how efficiently both threads and virtual threads are able to sort an array. The test was performed on multiple array sizes to test the scalability.

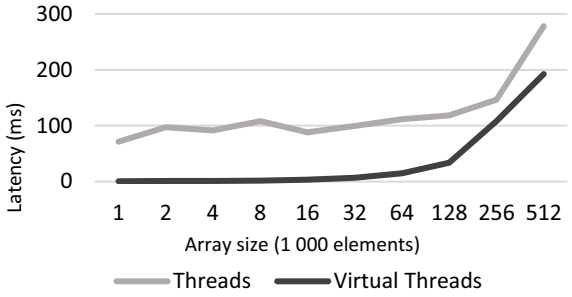

Figure 3. Variation of latencies based on the number of created threads and virtual threads used to perform the Multi-threaded merge sort algorithm

As shown in Fig. 3. we started with the array size of 1 000 and scaled it up to 512 000 elements, measuring how long it would take 16 threads or virtual threads to sort it. For the smaller arrays up to 64 000 elements, virtual thread implementation significantly outperformed threads by more than 80 times. However, the difference in the performance became less significant as the array size increased. The smallest difference was measured with the largest array size of 512 000 elements. However, virtual threads still outperformed threads on average by 30.82 %. Based on these results, we concluded that in most common cases, virtual threads could achieve faster synchronization because of their specific Java scheduler.

C. Parallel Run

The main goal of concurrency is to execute multiple independent tasks simultaneously and join them up again before the main execution completes. The number of requests an application can handle simultaneously determines the concurrency level of that application. The concurrency level is based on the number of requests and the time it takes to execute them. To examine both threads and virtual threads' scalability, we tested their performance based on the latency in performing a specific task. Then again, we increased the number of concurrent threads and measured the increase in latency for the same task.

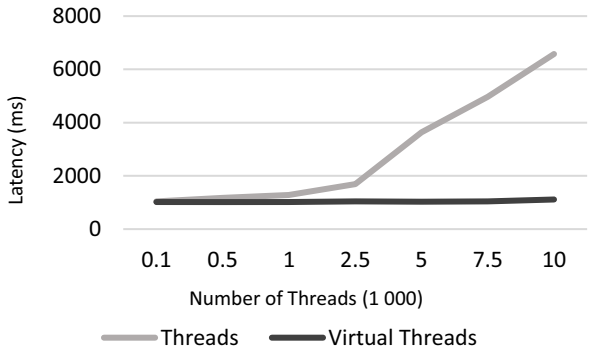

Figure 4. Variation of latencies based on the number of created threads and virtual threads in a Parallel Run

As presented in our test results in Fig. 4, the execution time with low request rates, starting with 100 to around 2500 requests, does not measure a significant difference between the two types of threads. However, as the latency level increases with the increase of a request rate, the difference becomes more noticeable. At the rate of 2500 requests, virtual threads outperformed threads by 38.24 %. As the request rate further increased, the efficiency of the threads falls, while virtual threads obtain a relatively constant response rate. At the rate of 10000 requests, virtual threads outperformed threads by 591%, from which we can conclude that, because of their scheduling processes, they should be able to handle significantly heavier loads with the same amount of CPU. Therefore, in the general case, we believe that virtual threads could achieve much higher concurrency levels than regular threads.

VII. Discussion

The primary purpose of concurrent programming is to increase performance without acquiring more resources. To achieve this, the applications should perform on a higher concurrency level based on the load an application can handle. This approach is common within multithreaded web servers, using the thread per request model [11]. When using virtual threads, higher levels of concurrency enable the applications to have better performance and withstand a heavier load. However, we believe that this is just one of the applications for this model of concurrency.

For comparison, to simplify the implementation of the code that should be executed asynchronously, a coroutine system has been added to Kotlin, and their concept is similar to those established in other languages. Some key characteristics of coroutines include lightweight concurrency – many of them could be run on a single thread without blocking others, and thus reduce the possibility of potential memory leaks by using structured concurrency to perform operations grouped within scopes, and integrated cancellation support (with automatic propagation through coroutine hierarchy).

In [12], the image decoding and encoding processes were analyzed, which require a lot of CPU resources and are commonly time-consuming. A model for decoding jpeg2000 images on lightweight devices was proposed and uses coroutines, lightweight processes that need fewer resources than AsyncTask set of Threads operating on the Android mobile OS. The product of this decoding

implementation results in a faster, economically efficient compressed image. Compared to the traditional thread-based model, this implementation gained better performance results with regard to time, CPU, and memory usage.

Difficulties in determining the order of access to shared resources and thread synchronization make the process of debugging concurrent applications hard. Since concurrent programs commonly distribute computation with forking and joining multiple threads, developers are forced to investigate the relations between the threads they consume. The need for optimizations of such procedures resulted in the proposal of a unified thread debugger model that enables debugging child threads inside the context of the parent thread. With such implementation, we could retain the history of multiple inherited threads by making a copy of the parent's stack. When an exception arises in the child thread, a single virtual thread holds a call stack that merges the child thread with its parent threads. Also, threads with multiple child threads only contain the information about the initial thread where the error occurred. The proposed model was tested on four different use cases, and each has shown to be able to provide the possibility for the reconstruction of the relation between parent and child thread to provide the unified view of all of the activated threads in the debugger. A further analysis reported the significant performance overhead when utilizing hundreds of threads. The proposed solution to reach the optimized performance includes leveraging underlying VM support in terms of resources for copying operations, as described in [13].

VIII. Conclusion

The novel approach to structured concurrency within the JVM introduced with virtual threads has a lot of potential for significant improvement in performance for concurrent large-scale Java applications. Based on our preliminary test results, virtual threads might be the solution for better scalability in Java, and provide needed resource usage optimization to Java-based web applications. Since virtual threads are entirely managed by and within the JVM, the associated Garbage Collection processes and algorithms, as well as Heap and Stack usage, would also be worth exploring during such processes. Through the development of Project Loom, we have witnessed several proposals of significant improvements to the existing JVM platform, and we are looking forward to its future development as well as the implementation of the debugging support.

References

[1] Y. Zhang, L. Li, and D. Zhang, "A survey of concurrency-oriented refactoring," School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, China, Dec 1, 2020, vol. 28, issue 4, pp. 319-330, doi: 10.1177/1063293X20958932

[2] M. Toro, "How to Implement Lightweight Threads," Universidad EAFIT, Jan 5, 2019. doi: 10.31219/osf.io/84tvj

[3] B. Evans, and D. Flanagan, "Java in a Nutshell: A Desktop Quick Reference," 7th ed. Sebastopol: O'Reilly Media, Inc., 2018.

[4] Y. Lin, "Automated Refactoring for Java Concurrency," Ph.D. diss., University of Illinois at Urbana-Champaign, US, 2015.

[5] R. Malhotra, "Rapid Java Persistance and Microservice: Persistence Made Easy Using Java, EE8, JPA and Spring," Apress, Berkeley, 2019. doi: 10.1007/978-1-4842-4476-0

[6] A. L. Davis, "Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams," Apress, Berkeley, CA, doi: 10.1007/978-1-4842-4176-9

[7] J. F. González, "Mastering Concurrency Programming with Java 9," 2nd ed. Packt Publishing Ltd., July 2017.

[8] R. Pressler, "State of Loom," May 2020. [Online]. Available: https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part1.html Accessed on: Jan 15, 2020.

[9] J. Gosling, et al. "The Java ® Language Specification, Java SE 15 Edition," Oracle America, 2020.

[10] B. Evans, "Going inside Java's Project Loom and virtual threads" Java magazine, Jan 15, 2021.

[11] P. Pufek, D. Beronić, B Mihaljević, and A. Radovan, "Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine," Proc. of the 43rd International Convention on Information Communication, and Electronic Technology MIPRO 2020, Opatija, Croatia, 2020.

[12] R. Saoungoumi-Sourpele, J. M. Nlong, J. K. Kamdjoug, and G. Yufui "Improve Image Decoding in Lightweight Environment Using a Coroutines Based Approach," Journal of Computer and Communications, vol. 8, no. 10, October, 2020, pp. 60-74. doi: 10.4236/jcc.2020.810007

[13] M. Leske, A. Chiş, and O. Nierstrasz, "Improving live debugging of concurrent threads through thread histories," Science of Computer Programming, vol. 161, 2018, pp. 122-148. doi: 10.1016/j.scico.2017.10.005