Let's Put the Memory Model Front and Center When Teaching Parallel Programming in C++

Jiri Dokulil
Faculty of Computer Science
University of Vienna
Vienna, Austria
jiri.dokulil@univie.ac.at

***Abstract*—When teaching parallel programming in C++, the memory model is often treated as an afterthought. Even if it is included in the lectures, it may only be covered as an isolated topic near the end of the whole course. We have reorganized the Parallel Computing course at the University of Vienna to make the memory model an integral part of the course, starting from the very first lecture. Being aware of the memory model and understanding its basic principles helps the students better comprehend parallel programming in C++, even when the memory model is only discussed informally. In the paper, we describe how we integrated the memory model into the course. Based on test results, assignments, and feedback from the students, we consider this approach to be successful.**

I. Introduction

Parallel programming is hard [1], but so is teaching parallel programming [2]. Efforts like the NSF/TCPP curriculum initiative [3] try to provide guidelines to educators, but there are still many challenges ahead.

Despite decades of research and innovation, there are no universally accepted, high-level programming models that would be able to allow programmers to easily target the ever-increasing range of parallel architectures. There were even efforts to design such tools specifically for education [4]. Just like MPI (a very low-level tool) remains the tool of choice for parallel computing on supercomputers, low-level languages like C++ or (plain) Java are often the go-to tools for parallel computing, even though they were not necessarily intended as such from the start. This is obvious from the absence of memory model from early C++ standards and the fact that the original Java memory model was flawed [5]. These issues were later fixed [6], [7] and since C++11 [8], the memory model has been an integral part of the standard, even though some problems still remain [9], [10].

A common approach in parallel programming courses is to cover the memory model near the end of the course, as an advanced topic. The aforementioned NSF/TCPP curriculum puts memory models into "advanced", but they do suggest covering the topic in the parallel programming course, as we do. But they only see the memory model in the context of sequential-vs-relaxed models without going into sufficient detail to cover the C++11 memory model. Sequential consistency and relaxed consistency are listed as separate topics, without indicating where and how they should be covered.

In our Parallel Computing undergraduate course at the University of Vienna, we have made the memory model an integral part of the course, building up to it from the very first lecture. The course is taught in C++, so we also teach the standardized C++ memory model. It is not possible to fully explain the model, but our experience shows that a basic understanding of the model can be taught even without devoting much time to the topic and some of the students could actually apply their understanding of the memory model in practical assignments. The Parallel Computing course teaches other models besides parallelism in standard C++, but they don't deal with the memory model and are therefore outside the scope of this paper. For simplicity, we will refer to the C++ part as "the course", even though there are in fact two more equally important components (OpenMP and MPI), where the memory model is only a marginal topic. There are five lectures (90 minutes each) in the C++ part, including the memory model.

In this paper, we:

- describe the organization of our Parallel Computing course;
- explain how we integrated the memory model;
- evaluate the course based on results of students' tests, assignments, and a feedback survey.

Based on the students' results and feedback, we consider the restructuring of the course around the memory model to be a success and a viable alternative to teaching the memory model as a separate, stand-alone topic within the course. We believe it provides the students with a deeper understanding of parallelism in C++.

II. Memory models in practice

On the theoretical level, a memory model describes the visibility of changes made by commands in the program. If command A changes the value of shared variable X, will command B in a different thread see the changed value, the original value, or is the value undefined? When transformed to actual software and hardware, the main role of the memory model is to restrict operation reordering. The reason is that most of today's hardware architectures provide automatic cache coherency and strong guarantees for the ordering of memory operations. Preventing the compiler from reordering operations and forcing it to read/write from memory (instead of registers) is therefore the most impactful consequence of the memory model.

Based on our experience, this is not obvious to many students. They expect the generated assembly code to closely reflect the original C++ code. It is surprising for them that a loop control variable is likely to be stored in a register and not written to the main memory at all, despite passing a programming language course earlier. This is a significant obstacle, which prevents them from properly understanding what happens when such a program is executed in parallel.

The second code example in our course looks like this:

```
void threadA()
{
  ++counter;
  cout<<counter;
}
void threadB()
{
  ++counter;
  cout<<counter;
}
```

It is easy to explain why there is a race condition if the counter is not incremented atomically. The situation starts getting surprisingly complicated when we also consider the command that prints out the counter. Even if all the operations happen atomically, it's possible that the compiler will store the incremented value of the counter in a register and reuse it when the value is printed. Once again, this seems to be surprising to some of the students.

When writing sequential codes, the students can build a memory model that tells them that what they write in their source code is what the processor will do. This view is rarely challenged if no parallelism is involved. They may encounter this in a system programming course, where they see that handling signals can be tricky and they need to be more careful. But in general, it is easy to adopt the WYSIWYG approach to programming. We make a significant effort in our course to break this view as soon as possible.

As discussed earlier, we put more emphasis on what the compiler does to optimize the code. The impact that a CPU has on execution is only covered in a more advanced course on parallel architectures intended for Masters students. We have considered including the topic already in the introductory parallel programming course but then decided against it. The compiler optimizations are sufficient to motivate the need to have a memory model and if a program is written correctly using C++11, it should run correctly irrespective of the CPU architecture, so the students don't need to be aware of the CPU architecture to use standard C++11 to write correct parallel programs.

III. Course Outline

In this section, we will present the architecture of the C++ parallel programming course, which has the C++ memory model at its core. The memory model plays a different role in different parts of the course. The following outline shows which parts have very little to do with the memory model, where the model is provided as *context*, and where it plays an **essential role**.

- motivation:
  - race conditions
  - **visibility of changes**
- **POSIX threads in C++**
- threads
- *mutexes*
- **atomic operations**
- **memory_order**
- other C++11 parallel primitives:
  - **call_once**
  - promises and futures
  - tasks
  - condition variables

As you can see, the C++ memory model is not equally important everywhere. Most parts are either focused on the memory model or the memory model is only briefly mentioned for context. The only exception are mutexes, since they have an important role in both assuring mutual exclusion and ensuring that changes are visible.

A. Motivation

The second code example in the first lecture increments a counter from two threads in parallel, as shown earlier. It is also the first example of a race condition, since the very first example is embarrassingly parallel. This example serves to motivate the need for synchronization (in this case mutual exclusion), but then we use a rewritten version of the program to show what might happen if the variables are "cached" in CPU registers.

The next example then demonstrates why something like this might be a problem in a more realistic scenario of two threads that need to exchange a value:

```
done = false;
void threadA()
{
  data=10;
  done=true;
}
void threadB()
{
  while(!done) /* NOP */;
  cout<<data;
}
```

It is natural to expect the value of data to be 10 in thread B, but if we allow the compiler to reorder the assignments in thread A, it doesn't have to be. Such reordering can easily be justified if we use something more complex to compute the value to be stored in the data. In fact, the compiler could go even further and read the value of `done` once at the beginning of the thread B code and loop indefinitely if the value was false.

B. Pre-C++11 model

To show a simple memory model that clearly works, we show the students how POSIX threads worked together with the pre-C++11 standards to ensure correct parallel execution. Consider this simple POSIX threads example:

```
done = false;
void threadA() {
  pthread_mutex_lock(mutex);
  data=10;
  done=true;
  pthread_mutex_unlock(mutex);
}
void threadB() {
  int my_data;
  for(;;) {
    pthread_mutex_lock(mutex);
    if (done) {
      my_data = data;
      pthread_mutex_unlock(mutex);
      break;
    }
    pthread_mutex_unlock(mutex);
  }
}
```

Since both `pthread_mutex_lock` and `pthread_mutex_unlock` are library calls, C++ guarantees that no operations are reordered around these calls, all changes are written to memory before the calls, and all data is read from memory after the call. Combined with the guarantees provided by POSIX, we get a guarantee that data is 10. This way, the students see that obtaining the proper visibility of changes does not have to be difficult. Here, we get it for free as a side-effect of proper synchronization. The same is basically true for C++11 mutexes, but we will still re-visit mutexes later to properly explain them.

C. Introduce atomic operations

Since the main strength of C++11 are the atomic operations, we introduce them next. Naturally, the first example goes back to the parallel increment of a counter, which the atomic operations solve beautifully. But this beauty comes at a price. The potential extra cost of atomicity is obvious, but the students might not immediately see the implications that the atomic operations have on the memory model. This is easily shown on the example from Section III-A. Even if the atomicity of the two changes guaranteed, it does not solve the problem if the operations can be reordered.

Up until now, the materials presented to the students serve mainly as a motivation to give the students some idea what the main issues of parallel programming in C++ are. We put significant emphasis on the fact that atomicity and mutual exclusion are not the only problems. Since nearly all of the students are already somewhat familiar with those, we actually put more emphasis on the change visibility problem.

D. C++11 threads

To allow the students to write some parallel C++ programs, we need to start with basic threads. But even this topic is relevant to the memory model. If thread A creates thread B, the memory model guarantees that all changes made by A before B was created are visible in B. Similarly, if thread B is joined into thread C, thread C sees all changes made by thread B. Nearly all students implicitly assume that this is the case, but we have seen a few cases (in a test) where someone expected that the changes made by a thread are not visible after it is joined.

This is a very important thing to keep in mind while using our memory-model-centered course structure. It is easy to put the students into a state of mind where they assume that nothing is guaranteed unless they explicitly use atomic operations with proper memory order to enforce it. Even if the actual behavior of thread join is explained in the lecture, it can be very easy for the students to miss. Most of them will go along with their (correct) implicit assumption, but it is good to keep this in mind to avoid unintentionally creating trick test questions. We have not seen this in any of the practical assignments, but some students tend to overthink test questions and then this is an easy trap to fall into.

E. Mutexes

We cover C++11 mutexes, not focusing on the memory model. Their behavior with regard to the memory model is fairly intuitive, similar to POSIX mutexes. However, there are some differences, which we try to explain on this example:

```
m.lock()
++counter;
m.unlock()
```

A POSIX mutex always provides a two-sided fence. It guarantees that all changes done before using the mutex are written to memory and that all read operations done after using the mutex are read from memory. But the C++ mutex lock and unlock operations are actually only one-sided. Lock does *acquire* and unlock does *release*. By chaining two operations on the counter and only considering the two relevant unlock/lock operations, we get:

```
++counter;
m.unlock()
m.lock()
++counter;
```

This is a basic chain of write-release-acquire-read operations, which demonstrates how the memory model actually works. We first introduce the *release* and *acquire* terms in this example. It is as simple as possible and we think that using mutexes instead of atomic variables to introduce the concept is beneficial, since the primary purpose of mutexes is synchronization and they don't carry the extra responsibility of modifying the data, as atomic operations do.

We spend considerable time explaining how to use mutexes and provide more sophisticated examples. But as none of these examples is too interesting from the point of view of the memory model, we do not discuss it further in this part.

However, there is one interesting example and some of the test answers seem to indicate it would be beneficial to discuss memory model implications there as well. When the vector container is used in the code, there are two kinds of operations with very different behavior as far as the memory model is concerned: whole-vector operations and individual

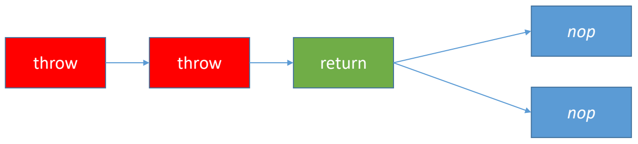


Fig. 1. An example of synchronization among related `call_once` calls. The helper function is invoked five times by different threads in order to run function *f* exactly once. Initially, *f* fails twice (red boxes), then succeeds (green box). The final two calls (blue boxes) do not attempt to call *f*, but they still provide synchronization and visibility of changes. The arrows show both synchronization (origin of the arrow must run before the destination) and data visibility (all changes made by the origin are visible at the destination).

item operations. If two threads add items to a vector (a whole-vector operation), they need to be properly synchronized and they need to ensure that all changes are visible to everyone who might need to see them. A mutex used as a critical section for the operation would serve this purpose well. But if two threads access (or even modify) two different items in the vector, they can be completely independent and not create a race condition. We have not explained this in the lecture and test answers clearly indicate that some of the students cannot see the difference on their own.

This is a difficult topic, since it requires a deeper understanding of the vector container. It is necessary to understand when and why references and iterators are invalidated and what implications does this have for parallel access to the container and the elements it contains. Furthermore, the whole-vector operations need to be protected by a mutex or something similar, but the operation on individual items might be performed using atomic operations or not synchronized at all. Such combinations can be very tricky in C++. Therefore, we plan not to deal with the topic at this stage of the course, but instead include a stand-alone section near the end to specifically deal with this issue.

F. Call once

The `call_once` helper function, which is also part of the C++11 standard library, is interesting for the memory model, since the synchronization performed by the function is very straightforward, but special care has been taken when dealing with the memory model. The purpose of `call_once` is to run a function *f* exactly once, even when invoked multiple times in parallel. But *f* is allowed to throw an exception, in which case another call to *f* is permitted. All such re-tries can see the result of previous invocations, but after a successful call, the subsequent calls (that don't actually invoke *f*) only synchronize with the successful one, not among themselves, as shown in Figure 1.

It's possible to use this example to show the students that even if this sounds unnecessarily complex, it is actually the most appropriate, natural behavior. The function *f* is probably doing some initialization. To retry it, it's obviously good to know why it failed before (therefore the linear chain of red boxes) but once it succeeds, everyone only needs to see the successfully initialized values (the independent blue boxes connected to the green one).

G. Atomic operations

We first explain the atomic operations without involving the memory model. There are more complex operations like compare-exchange, so we first focus on the atomicity and behavior of the operations, ignoring the memory model. But this part is relatively short. We revisit an earlier example of a counter that needs to be incremented atomically and show what it would actually look like in C++11. We also revisit the example with the vector, but also avoid mixing the whole-vector and per-item operations, only doing the independent per-item operation in parallel.

H. Memory order

At this point, we introduce the `memory_order` enumeration, which lists the possible memory orders of an atomic operation. The students have already seen release and acquire in the mutex example (Section III-E). The other memory models are only briefly explained and we quickly move to examples that use only release and acquire in different situations.

We show how the example from Section III-B can be handled with atomic variables, but also include more complex ones. We demonstrate how a value can be passed among more than two threads using one or more atomic variables. In all these examples, the main focus is on the memory model and visibility of changes, since the synchronization part is trivial.

One important point that we believe should be mentioned multiple times is that to establish a proper release-acquire chain, the atomic operation needs to be the same (this is also true for mutexes). We have an example with a release on one atomic variable and acquire on a different one, but maybe the topic should be discussed even more, as there were examples of students not doing this correctly in their assignments or tests.

I. Happens-before

The *happens-before* relation is at the core of the C++ memory model. It defines which operations are guaranteed to happen in a certain order. Inside a thread, the relation is based on the sequential execution. But the relation is also defined among operations in different threads, which can be synchronized using the primitives provided by C++. In the lecture, we only explain the relation informally. In fact, there is only one slide in the presentation directly dedicated to happens-before. On the other hand, we spend considerable time explaining this slide and discussing it with the students.

We believe that this is enough to provide the students with a working understanding of the topic, allowing them to write correct C++ programs that use release and acquire semantics of atomic operations. The work that the students do in the assignments suggests that at least for a significant fraction of the students, this is true. See Section IV for more details.

We first demonstrate the happens-before and synchronization using atomic operations by showing the students how to

memory order use	number of students
correct use of acquire-release	13
sequential consistency (too strong)	1
relaxed consistency (correct use)	2
relaxed consistency (possibly incorrect)	2
total number of submissions	35

TABLE I
THE USE OF MEMORY ORDER IN AN ASSIGNMENT.

correctly implement a spin mutex using atomic operations. One might naturally expect that the operation that locks the mutex should have acquire semantics and the operation that unlocks the mutex should have release semantics. This is actually the case, but it is good to spend the time and explain to the students why it is this way. It also provides some connection between mutexes and atomic operations, demonstrating that the fundamentals (w.r.t. the memory model) are actually the same.

J. Other topics

There are other parts of C++11 that facilitate parallel programming and we cover them in the rest of the lecture. These include examples of using compare-exchange for lock-free data structures, futures (and related C++11 constructs), and condition variables. Even though they also have memory model implications, we only briefly mention them without going into greater details. The fact that the behavior of atomic operations is very simple (especially when compared to something like condition variables) makes them the ideal topic to teach the memory model. Once the students are aware of the problem and have a basic understanding of happens-before and visibility of changes, it should be much easier for them to apply this to other areas as well.

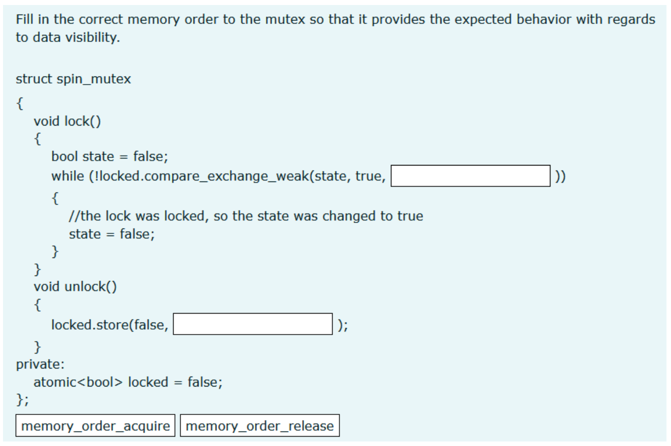
K. Example problem

At the end of the course, we show the students different ways of implementing one task in multiple ways. The task is to have one thread perform 1000 iterations of some computation and let another thread know what is the index of the last completed iteration. The second thread uses this information to print out progress at fixed intervals. This can be done with mutexes or atomic variables.

Using atomic variables is interesting in this case, since it is possible to use relaxed memory order. This is actually the first case where we show this to the students. Then, we put this into contrast with a more complex case, where we need to not only send the iteration index, but also some other data (in our case, it is a single string). This requires the memory order to be changed from relaxed to release-acquire, to facilitate the transfer of the other data. We also show the default (sequentially consistent) order to cover all the options, with the exception of consumer memory order, which is being revised and its use is discouraged [11], [12].

IV. Evaluation

We have not performed a formal study to investigate the effects of making the memory model a central topic of the



Fill in the correct memory order to the mutex so that it provides the expected behavior with regards to data visibility.

```
struct spin_mutex
{
  void lock()
  {
    bool state = false;
    while (!locked.compare_exchange_weak(state, true, [          ]))
    {
      //the lock was locked, so the state was changed to true
      state = false;
    }
  }
  void unlock()
  {
    locked.store(false, [          ]);
  }
private:
  atomic<bool> locked = false;
};
```

Fig. 2. The memory model test question. The students need to drag the two boxes at the bottom to the correct position. This option was chosen to make the question depend on the students' understanding of the memory order, instead of forcing them to remember the exact C++11 interface.

course. There were other changes to the course, making direct comparison impossible. However, we can still make some conclusions with the data that we have. As their home assignment, the students were asked to modify existing source code to correctly use atomic operations for synchronization. The key part was a *slot allocator* – an object with a fixed number of slots that can be acquired and released by worker threads. This is similar to an array of mutexes, but the object automatically finds a free mutex if one is available.

The default memory order for C++ atomic operations is sequential consistency, which implies acquire (for load operations) and release (for store operations). This is sufficient for the assignment. The students were not asked to explicitly provide memory order. Therefore, all instances that specify it are based on their own active decision. Table I provides the number of student assignments based on their use of explicit memory order.

As you can see, in 18 of the 35 submissions the students have decided to provide memory order. This shows that at least half of the students were thinking about the memory model when working on the assignment. One student used sequential consistency, which is too strong. Four students used relaxed consistency. In general, this is not the right decision for the kind of synchronization object that they were asked to design.

However, in two of the four cases, it is obvious (from the comments in the code) that the students were actually well aware of the implications of the relaxed memory order and based their decision on the fact that the code that uses the object does not rely on the acquire-release semantics and will provide correct results even with relaxed memory order. In two cases, no such comments were provided. It is still possible that these students still made the decision based on correct assumptions, but it is also possible that they did not.

Overall, we consider the results to be very good, with most students selecting the right memory order. However, one needs to consider the fact that the code is similar to the spin mutex that the students have seen in the lecture. It is therefore possible that they based their decision on that similarity. Unfortunately, our data do not allow us to distinguish between these two cases.

In the final test, one question required students to fill in the correct memory order in an example derived from the spin mutex. The question is shown in Figure 2. It was answered correctly by 32 of 39 students. One multiple choice question required a basic understanding of the happens-before relation. It was answered correctly by 31 students. Interestingly, providing a correct answer in the two questions seems to be mostly independent and only two students answered both incorrectly.

The students were also asked to fill a standard survey to provide feedback for the course. Only 9 of the students participated, but the results are encouraging, with most students being satisfied with the course overall, but also with the speed at which the topics were presented. Note that the answers in the survey pertain to the whole semester and C++ parallelism is in fact only one third of the whole course. But individual (informal) feedback from the students indicated that they were indeed satisfied with the C++ lectures.

V. Conclusion

We consider the course restructuring to be a success. The students were able to demonstrate a basic level of understanding of the C++ memory model in a test. More importantly, many of the students have shown to be aware of the memory model when working on their assignments, providing correct memory order even though they were not explicitly asked to do that.

In the future, we plan to further refine the course. The biggest consideration is dealing with the whole-vector and per-item operation and the interaction of atomic operations, non-atomic operations, and mutexes in general. The integration of atomic and non-atomic operations is challenging even for the C++ memory model itself [9], [10], so we need to find a good compromise between accuracy and practical usability.

We are also looking into ways of making the assignments more involved with regards to the memory model. One way would be to have a larger number of smaller assignments, each covering a different kind of memory order. However, the current format based on Moodle is not well suited for this and it would be necessary to make significant changes, possibly even towards interactive educational software and gamification [13], [14], [15].

It would also be beneficial to see how the knowledge of the C++ memory model translates to other models and languages, like OpenMP or Java.

Acknowledgment

The work was supported in part by the Austrian Science Fund (FWF) project P 29783 Dynamic Runtime System for Future Parallel Architectures.

References

[1] P. E. McKenney, "Is parallel programming hard, and, if so, what can you do about it? (v2017.01.02a)," *CoRR*, vol. abs/1701.00854, 2017. [Online]. Available: http://arxiv.org/abs/1701.00854

[2] L. R. Scott, T. Clark, and B. Bagheri, "Education and research challenges in parallel computing," in *Computational Science – ICCS 2005*, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 44–51.

[3] S. K. Prasad, T. Estrada, S. Ghafoor, A. Gupta, K. Kant, C. Stunkel, A. Sussman, R. Vaidyanathan, C. Weems, K. Agrawal, M. Barnas, D. W. Brown, R. Bryant, D. P. Bunde, C. Busch, D. Deb, E. Freudenthal, J. Jaja, M. Parashar, C. Phillips, B. Robey, A. Rosenberg, E. Saule, and C. Shen, "Curriculum initiative on parallel and distributed computing - core topics for undergraduates, version II-beta," Tech. Rep., 2020. [Online]. Available: http://tcpp.cs.gsu.edu/curriculum/

[4] I. Finlayson, J. Mueller, S. Rajapakse, and D. Easterling, "Introducing tetra: An educational parallel programming system," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 746–751.

[5] H.-J. Boehm and S. V. Adve, "You don't know jack about shared variables or memory models," *Commun. ACM*, vol. 55, no. 2, pp. 48–54, Feb. 2012.

[6] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," *SIGPLAN Not.*, vol. 40, no. 1, pp. 378–391, Jan. 2005.

[7] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," *SIGPLAN Not.*, vol. 43, no. 6, pp. 68–78, Jun. 2008.

[8] "ISO/IEC 14882:2011 information technology – programming languages – C++," Tech. Rep., 2011. [Online]. Available: https://www.iso.org/standard/50372.html

[9] H.-J. Boehm and B. Demsky, "Outlawing ghosts: Avoiding out-of-thin-air results," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:6.

[10] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in c/c++11," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 618–632.

[11] P. E. McKenney, T. Riegel, J. Preshing, H. Boehm, C. Nelson, O. Giroux, and L. Crow, "P0098r1: Towards implementation and use of memory_order_consume," Tech. Rep., Jan 2016. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0098r1.pdf

[12] J. Bastien and P. E. McKenney, "p0750r1 Consume," Tech. Rep., Feb 2018. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0750r1.html

[13] M. Abernethy, O. Sinnen, J. Adams, G. De Ruvo, and N. Giacaman, "ParallelAR: An augmented reality app and instructional approach for learning parallel programming scheduling concepts," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 324–331.

[14] R. F. Maia and F. R. Graeml, "Playing and learning with gamification: An in-class concurrent and distributed programming activity," in *2015 IEEE Frontiers in Education Conference (FIE)*, 2015, pp. 1–6.

[15] E. Buzek and M. Kruliš, "An entertaining approach to parallel programming education," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 340–346.