Formal Automated Verification of a Work-Stealing Deque

C.M. van Kampen
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
c.m.vankampen@student.utwente.nl

ABSTRACT

Multi-core systems brought the possibility for concurrent programs. In task-based parallelism, work-stealing has been an important development. Many designs of a work-stealing framework make use of deques, but they do not properly prove the correctness of the deque. Therefore, the deque from the Lace framework has been implemented and verified for functional correctness using the VerCors verification tool. Under our assumptions, the verification in VerCors passes and shows that all tasks are executed and return the correct result. The verification has been partially validated, but more testing and validation is required before a statement can be made about the validity of the proof.

Keywords

Concurrency, task-based parallelism, deque, work-stealing, deductive verification, VerCors

1. INTRODUCTION

The development of multi-core systems has been of great impact in computer science. It means that programs can be executed in parallel and show the results much faster. An important paradigm in parallel programming is the task-based paradigm. A program is seen as a tree of tasks that need to be executed one after another. However, some tasks are independent and can be executed in parallel. By having specific workers that each run on a separate core, they can execute the tasks in parallel. Some workers might be idle, because they are done with their tasks earlier than others. To solve this problem of inefficiency, a specific form of task-based programming involves work-stealing (see 2.1). This means that workers can steal tasks from other workers if they are idle. Each worker has a task pool which is often implemented as a double-ended queue or deque. Although a lot of designs for such a deque have been proposed, they have not been formally proven to be correct and are prone to data races or other concurrency issues. To prove that the deque functions correctly, it needs to be verified that each task in the deque is executed and contains the correct result.

Several work-stealing frameworks exist and have shown good performance. Some examples include Cilk [4] and Wool [7]. Recently, a new work-stealing framework, called Lace, has been developed that shows comparable, and in some cases improved, performance. The Lace framework implements a non-blocking split task deque [16]. Van Dijk has provided a formal proof on paper but also states that an automated proof using a verification tool is necessary. It has been shown that deductive verification is a useful verification technique. A specific deductive verification tool is VerCors [3]. Based on behavioural constructs, VerCors can verify whether a program does not violate this behaviour.

This research has implemented the deque from the Lace framework and attempts to verify functional correctness using VerCors. Due to time limitations, we assumed a strongly consistent memory model. Accordingly, the central research question that this paper attempts to answer is: *To what extent can Lace's work-stealing deque be verified for functional correctness using the VerCors verification tool and assuming a strongly consistent memory model?*

During the research, some challenges were encountered with regard to VerCors. These challenges were partly overcome, resulting in a passing verification. However, the specification should be validated before anything can be said about the validity of the proof. When a proof is justified, the next step is to verify correctness in the TSO (Total Store Order) weak consistency model.

1.1 Contributions

The contributed work of this paper is a formalized specification of Lace's deque in PVL to verify its functional correctness in a strongly consistent memory model. Furthermore, this paper reports on the challenges, specifically with regard to VerCors, that have been tackled. In the end, we have decent confidence in the verification of the deque. However, due to limitations, a formal proof cannot be justified. Therefore, we propose directions for future work.

2. BACKGROUND

This section provides more background information for this research. First, we explain work-stealing in more detail. Secondly, we discuss the deque along with the distinction between strong and weak consistency models. Next, we provide a short explanation of the memory fence. Finally, we consider the Lace framework and deductive verification with VerCors.

2.1 Work-stealing

In task-based parallelism, a program can be seen as a tree of tasks or problems to solve. The fork-join model defines that each task can create subtasks that need to finish first as seen in Figure 1. To efficiently execute all tasks, they can be run in parallel. However, parallelism adds extra overhead on the performance time for initializing separate threads and synchronizing between them. Therefore, to reduce the overall execution time, it is important to effectively make use of parallelism.

In a multi-core system, each core is assigned a specific worker that has its own task pool which it runs through. To minimize each workers' idle time, the work should be split in equally partitioned tasks. Though, it is often unknown how long each task will take.

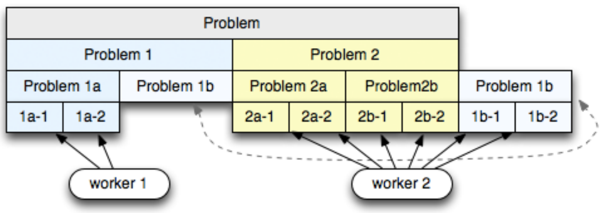

Figure 1. An example of two workers solving a task-based problem together. [1]

Work-stealing resolves this problem of *load balancing* by giving workers the ability to steal tasks from each other if they are experiencing idle time. A worker is idle when it does not have any more tasks or when all its tasks are stolen. The main worker starts running the root task which creates subtasks that the other workers steal. A stealing worker is called a *thief* and the worker it steals from is a *victim.* Several implementations of work-stealing frameworks exist such as Cilk [4] and Wool [7]. Wool has been found to be slightly more efficient than Cilk [14]. Another recent development which has shown similar performance to Wool is the Lace framework by Van Dijk [17].

A problem that occurs in work-stealing is when a thief steals a task from a victim. Therefore, the victim might run out of tasks and has to steal from other workers in turn. A consequence would be that the size of the task pool could grow beyond the size needed for a sequential program [7], since the stolen task is added on top of the current task that is blocked until a join. By stealing back from the thief instead, the worker steals subtasks created from the stolen task and thus helps itself finish its initially owned task. This is called *leapfrogging* [18].

2.2 Deque

The task pool that workers use is often implemented as a *deque.* A deque or double-ended queue is a type of queue where elements can be added and removed, from the front and back, using *push* and *pop* instructions respectively as seen in Figure 2. Often also a *peek* method is defined that returns the value of the front element. The worker holding the deque is the *owner* of this deque and of all the tasks it contains.

A deque can be implemented in many ways. An example is by using a circular array and two descriptors called front and back [6]. In work-stealing, the front can only be used by thieves and its value can only be incremented. The back is used by the owner to add and remove tasks.

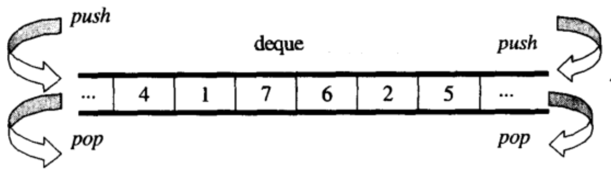

Figure 2. An illustration of a deque where elements can be pushed to and popped from both ends. [2]

An optimal deque is non-blocking. This means that an error in a running thread should not lead to any errors in other threads. To minimize the concurrency overhead of synchronizing between workers, it should only require atomic instructions like compare-and-swap (i.e. check if the known value of a variable still holds and change it if this is the case) and memory fences.

2.3 Consistency models

2.3.1 *Strong consistency model*

In a *strong consistency model*, access to a shared variable always holds the same value for all observing (parallel) processes. This means that storing a new value from a process can directly be seen by all other processes. Therefore, when a thief tries to steal a non-stolen task, it can be assured that it is indeed not stolen and the thief will be the only one completing it. In practice, such optimal conditions cannot be met.

2.3.2 *Weak consistency model*

Most computer architectures make use of a *weakly consistent memory model*, i.e. loads can be reordered before stores. In a work-stealing environment, this means that a thief can steal a task from another worker that started executing said task, but has not yet told the rest of the workers. Therefore, a memory fence or barrier is required in the *pop* command to enforce execution order. However, these are expensive to execute and can add a lot of overhead to the performance time [12, 15].

Many types of weak consistency models exist. TSO (Total Store Order) is the best known and most used.

It is important that we state the difference between the strong and weak consistency models. Overall, an algorithm that functions correctly in a sequential context, cannot be trivially assumed to also work in concurrent programs. In concurrent programs, data races could lead to incorrect execution. Additionally, the behaviour of concurrent programs cannot be expected to be the same in a strong and a weak consistency model.

2.4 Memory fence

A *memory fence* or *barrier* is used to enforce the ordering of memory instructions (loads and stores). This means that instructions before the memory fence will always happen before the ones that are written after the fence. Some instructions might occur out of order, because of compiler optimizations. Usually, this will not result in unexpected behaviour. However, in a concurrent program, some shared variables might be accessed by different threads which the compiler does not take into account when optimizing the execution order. As a result, a variable might be loaded before another thread is able to store a new value.

As an example, assume a program has a boolean variable called *wait* and an integer variable called x that start out as *true* and 0 respectively. Two threads are started that work on both variables. The first thread waits for the boolean to become false and then prints x:

```
while (wait) {}
// Require memory fence
print x;
```

The second thread sets x to 5 and *wait* to *false*:

```
x = 5;
// Require memory fence
wait = false;
```

If some operations are executed out-of-order, it might happen that the second thread updates *wait* before *x*. As a result, the first thread could jump out of the *while* loop and print x as 0. Furthermore, it could also happen that x is printed before the *while* loop. Consequently, memory fences are necessary at the specified positions.

2.5 The Lace framework

Several work-stealing frameworks have been designed. Examples include Cilk [4], Cilk-5 [9] and Wool [7]. These frameworks often use a deque to implement the task pool. The *Lace* framework is a recently developed work-stealing framework [16], taking in mind the designs of some aforementioned frameworks. The task pool is implemented as a non-blocking deque with directly stored tasks (instead of pointers to a shared deque). It is split in a shared and a private part and is described by the tail, split point and head variables. Thieves can only steal tasks after the split point. Stolen tasks remain in the deque; the result of a stolen task is written back to the stored descriptor. The work-stealing algorithm also makes use of leapfrogging.

The Lace framework clearly shows a lot of advantages over other frameworks. The deque is non-blocking, and only requires one memory fence and one compare-and-swap operation. However, an important problem is that there is only a proof on paper [16] and no formal automated proof using a verification tool that the deque functions correctly, i.e. that every task inserted in the deque is eventually executed exactly once. The goal of this research is to implement the deque from the Lace framework and prove its functional correctness.

2.6 Deductive verification

As a result of weak consistency models, concurrency issues can occur in a multi-core system. For instance, the reordering of loads and stores introduces the possibility for data races. Therefore, an implementation should always be tested. However, this is limited to specific test cases and a lot of situations can be overlooked because of unexpected behaviour as a result of data races. Therefore, it is necessary to prove the correctness of concurrent programs. A possible approach is to specify a formal mathematical proof using induction. However, this can become tedious in a concurrent program. Therefore, an automated verification tool is often used. One technique is *deductive verification* which verifies for a program that it complies with the modelled behaviour in every possible scenario [13].

Hoare logic, proposed by Tony Hoare and Robert Floyd, provides logical rules for reasoning about sequential programs [10] and is a key concept in deductive verification. The most important feature is called *Hoare triple* and characterizes a triple $\{P\}s\{Q\}$ where s is a program statement, and P and Q are the pre- and postconditions respectively. It specifies that a program being in a state where P is satisfied, the state after execution of s satisfies Q [8, 13].

For concurrent programs, Hoare logic has been extended as CSL (*concurrent separation logic*) [13]. It provides logic for concurrent programs using a shared memory heap. It is also able to examine basic locking patterns. CSL extends Hoare logic with ownership and disjointness of ownerships. Predicates can be defined as $l \overset{\pi}{\hookrightarrow} v$, stating that value v is defined at location l on the heap. Additionally, $\pi \in (0, 1]$ describes the amount of ownership that is available for location l where a value of 1 expresses write permission and otherwise only read permission. The concept of disjointness of ownership is specified by the separating conjunction $P * Q$, declaring that P and Q cannot assert write access to the same memory location. Finally, one of the most important proof rules is the parallel composition rule [5, 13] which states that if programs C_1 through C_n, operating on disjoint parts of the heap, can be proven for the Hoare triples $\{P_1\}C_1\{Q_1\}$ through $\{P_n\}C_n\{Q_n\}$, then it can be concluded that they can also operate correctly in parallel.

In this research, we use VerCors, a deductive verifier, as the verification tool. Its logic is build on IDF (Implicit Dynamic Frames), a variation of CSL.

VerCors is a tool for verification specifically useful for concurrent programs [3]. It checks Java, C, OpenCL and OpenMP programs. Furthermore, it defines its own specification language called PVL (Prototypal Verification Language). A specification language consists of verification constructs, such as pre- and postconditions and invariants, integrated in a programming language [8].

In PVL, the pre- and postconditions are defined on method declarations by *requires* P and *ensures* P where P is a predicate. To avoid duplication, *context* P can be used to specify that the predicate must hold as both a pre- and postcondition.

Furthermore, VerCors provides *resource invariants* which bundle a set of permissions and boolean expressions. This makes it easier to reuse a collection of permissions. The pre- and postconditions can take in resource invariants requiring that all bundled expressions must hold. A method requiring a resource invariant can $unfold$ the resource to be able to use the permissions and $fold$ it back to ensure the resource again.

Additionally, VerCors enables us to lock on a resource invariant, so other threads cannot use the permissions bundled in the resource. The resource invariant needs to be called *lock_invariant* and needs to be defined in the target class. The lock can then be acquired using: *lock o* where *o* is the target object with the *lock_invariant*.

Another important construct in VerCors is the loop invariant which makes assertions about a loop which hold before, in between iterations and after termination of the loop.

VerCors translates an input program to the Viper language. Subsequently, the Viper verifier uses the Z3 SMT (Satisfiability Modulo Theories) solver for verification [13].

VerCors makes use of a layered structure. In the first layer, it verifies data race freedom. The second layer is to prove memory safety and the final layer is about the functional correctness of programs. The logic in VerCors builds on IDF (Implicit Dynamic Frames) which is comparable to CSL, but ownership is specified in a slightly different manner [13]. In the second layer, ownership of shared memory is specified by permissions in the form: $Perm(o.f, \pi)$ where π is the amount of ownership available for the field f of object o.

By specifying all the aforementioned behavioural constructs inside your code, the VerCors tool will generate verification conditions which are verified by Z3 to determine whether the program is correct.

3. RELATED WORK

A lot of research has already been done in the field of task-based parallelism and work-stealing. Researchers have discovered new techniques to implement a work-stealing deque and make improvements upon existing ones. The Lace framework took several designs in mind to create an improved work-stealing framework with similar performance to Wool [16, 17]. In his dissertation, Van Dijk gives a formal mathematical proof on paper, but also states that a full correctness proof should be performed using a verification tool [16].

Filliâtre specifically discusses the challenges in proving program correctness [8]. It is already difficult to prove safety, i.e. not running into any fatal errors, or termination of a program. However, a deductive verification tool definitely speeds up the verification progress in comparison with a manually written proof.

Some more challenges in deductive verification are discussed by Hähnle and Huisman [11]. For example, they state that in large programs, it is difficult to make sure all required specifications such as pre- and postconditions are defined. These challenges have to be kept in mind in this research when using VerCors.

Oortwijn stresses that the strength of a deductive verifier is that it analyzes all possible execution scenarios [13] some of which might be overseen when manually trying to prove correctness. For that purpose, Oortwijn uses VerCors to verify parallel graph algorithms.

Overall, the literature shows that deductive verification is a useful technique to prove the correctness of concurrent programs. Therefore, VerCors will be used in this research to verify the correctness of the deque from the Lace framework. However, some challenges might have to be overcome or avoided beforehand.

VerCors has already been used to verify the *ConcurrentLinkedQueue* class from the Java concurrency library [3]. This is a non-blocking deque and mostly involved the verification of atomic operations using resource invariants. This specification can be used as a starting point to verify our implementation of a deque.

4. METHODOLOGY

To verify the Lace framework, the algorithm is implemented in PVL. The main classes that are necessary for this include: *Task*, *Deque* and *Worker*.

These classes have been implemented first. Next, all necessary permissions were added. To be able to update or access the value of a class field write and read permissions should be granted respectively. After these specifications were verified correctly, verification logic has been written that allows us to verify the functional correctness of the deque.

To understand what verification logic needs to be added, the goal has to be specified. What exactly does it mean for a work-stealing deque to function correctly? In essence, the deque functions correctly when the result is the same as the result from sequential execution. For verification, we should ensue a more strict definition:

> A deque functions correctly when the result of syncing a task is equal to the expected value of the matching spawn.

To verify this, we specify the following assertion after syncing a task:

$$assert\ task.done\ \&\&\ task.result == task.expected;$$

We define the expected result when spawning a task and assert that the result after syncing is equal to this value. This ensures that a sync is matched with the correct spawn and overwriting the task or its fields could fail the verification.

In the following sections, we provide a short explanation for the implemented classes. The description is not as comprehensive, because the implementation is essentially the same as the Lace framework described by Van Dijk [16]. Additionally, we discuss the specification of the verification constructs and provide the verification result. Finally, we discuss certain challenges that were encountered while working with VerCors. The source code can be used as a reference [1].

4.1 Implementation and specification

4.1.1 Task

The *Task* class implements simple behaviour for executing an algorithm in parallel. A newly created task is added to the deque of a worker. This worker is the owner of this task. Then a task can be executed by the worker or stolen by a thief. A task keeps track of a boolean field stating whether it is *done*, an *expected* value and a *result* of the task's execution algorithm. The *execute* method executes the function that implements this algorithm and sets the *done* field to true.

An example of such an algorithm could be calculating the n_{th} number in the Fibonacci sequence as defined by the following equation:

$$fib(n) = (n < 2)\ ?\ n\ :\ fib(n-1) + fib(n-2) \quad (1)$$

In a task, this would be implemented as:

```
if (n < 2) {
  result = n;
} else {
  Task t1 = new Task(n - 1);
  Task t2 = new Task(n - 2);
  worker.spawn(t1);
  worker.spawn(t2);
  worker.sync();
  worker.sync();
```

[1]the implementation of Lace's deque in PVL can be found at: `https://github.com/campoe/lace-vercors`

```
    result = t1.result + t2.result;
}
done = true;
```

Each recursively created subtask calculates their own Fibonacci number, used by the parent task. However, it needs to be verified that the deque functions correctly for any implemented algorithm. Therefore, a function is defined that is called by the *execute* method and directly calculates the result. It is assumed that this function does not affect the verification result. Hence, we use the Fibonacci algorithm from Equation 1. The *expected* value is set to the n_{th} Fibonacci number when spawning a new task. The *execute* method also sets the *result* field to the n_{th} Fibonacci number. Then we only need to verify that the task contains the *expected* result when finished. The *Correct* resource invariant has been defined that makes this assertion:

```
  resource Correct() =
    Value(done) ** Value(n) ** Value(
        result) ** Value(expected) **
    (done ==> (expected == result));
```

In order for this assertion to pass, the value of *n*, *expected* or *result* should not have been changed in the meantime. If, for example, *n* has been updated, the wrong Fibonacci number will be calculated.

If the task is not done yet, the value of *result* does not matter. Furthermore, the task should be locked on to be able to write to *done* and *result*. This ensures that not only either field can have write access; i.e. both fields **can** be written to within a locking block. Another advantage is that no other thread can access these fields. Hence, in the *execute* method, both *result* and *done* are set before other threads are able to see these updates; ensuring that *result* contains a value when *done* is true.

4.1.2 Deque

The deque of the Lace framework is a non-blocking split task deque. The deque has methods for *pushing*, *peeking* and *popping*. The *peek* method can be used to get the current front task. Additionally, it returns a boolean indicating whether the task is stolen or not. On another note, the *pop* method does not remove the top task, but this will be overwritten the next time a *push* instruction is performed.

For verification, some useful *resource invariants* have been defined. To access the elements, *PArray* gives access to all tasks and with *PIArray* a specific task can be accessed:

```
resource PArray(frac p) =
  Value(tasks) ** tasks != null ** Value(
      size) **
  (\forall* int j; 0 <= j && j < size;
      Perm(tasks[j], p));

resource PIArray(int i, frac p) =
  Value(tasks) ** tasks != null ** Perm(
      tasks[i], p);
```

For *PIArray* it is necessary that the index of the task needs to be less than the head, because all other tasks have been popped off by the *pop* method. All popped off tasks have an index more than or equal to the *head*. Therefore, the popped off tasks do not exist as stated in the invariants defined by Van Dijk [16]. New tasks can only be written to the array at the head and will overwrite popped off tasks.

The *steal* method makes use of the *PIArray* resource invariant to steal a shared task that has not been stolen yet. The *compare-and-swap* or *CAS* operation on the tail makes sure to increment the tail. It is an atomic operation that compares the value in memory with an expected value and, only if these are the same, puts a new value in memory. Using a lock, write permission is granted on the *tail*. This means that the current thread temporarily takes full write permission and all other threads cannot access the *tail*. Therefore, it can be safely updated when its value is equal to the expected value. If the comparison fails, or the split point was updated in the meantime, it cannot be ensured anymore that there are any unstolen shared tasks left.

The deque has a privately used method called *shrinkShared*. In a weakly consistent memory model, a memory fence should be included at the specified position. Otherwise, the result will be that a task might be stolen, but the method returns false as an update to *tail* is not detected yet. Therefore, the owner of the task will assume that it needs to execute the task itself. The stolen task is executed by the owner and popped off the deque. Another task pushes a new subtask on the deque which overwrites the stolen task. After completion, the thief writes the result to the new task which is (presumably) incorrect. Consequently, the memory fence cannot be left out in a weakly consistent memory model to ensure that this problem does not occur. As we assume a strongly consistent memory model, we can omit this memory fence.

From the task's *done* field, it can be verified whether a task has been executed (**at least** once). In essence, it is not necessary to prove that a task has been executed **exactly** once. As long as it can be proven that the result is correct.

4.1.3 Worker

The worker is basically a wrapper for the deque to make it interact in a work-stealing context. Each worker has its own deque and is able to execute its own tasks and steal tasks from other workers. For simplification, the victim that a worker steals from can be manually set. Therefore, victim selection is not a concern anymore.

The worker can *spawn* new tasks in its own deque. This essentially executes the *push* method of the deque. A spawned task needs to be synced; i.e. the task needs to be executed and then removed from the deque. Therefore, the framework should be used correctly and each *spawn* is matched with a *sync*. The *spawn* increments the *head* of the deque while the *sync* decrements it again when calling the *pop* method.

Two scenarios can be expected when this is incorrectly used:

1. **Spawning too often**
   Spawn requires that the *head* is less than the size of the task array. Otherwise, the new task cannot be pushed onto the deque. Spawning too often before syncing might therefore result in an index overflow error.

2. **Syncing too often**
   Sync requires that the *head* is more than 0. Other-

wise, decrementing the *head* would result in an invalid (negative) index. Therefore, we cannot have more executions of *sync* than of *spawn*.

As a result, each *spawn* that is executed needs to be followed by a *sync* operation. Therefore, the *head* needs to be a valid index before and after executing a *spawn* or *sync*.

The worker contains a *run* method which is called when forking the worker. Therefore, each worker is run on its own thread. A work-stealing program would start out with a single task called the *root*. The main worker (defined by an identifier of 0) executes the *root* which spawns new subtasks that can be stolen by other workers. These workers keep stealing from each other until the *root* task is finished.

4.2 Verification

The most important verification step is done in the worker. The invariant that should be verified is contained in the *Correct* resource of the task. This resource invariant states that a task that is finished executing, contains the correct result. This has been verified in the *sync* method by asserting that the peeked task ends up done and the result is equal to the value *expected* by the matching spawn. The same assertion is done in the *run* method of a worker thread for the *root* task. The verification passes and therefore asserts that any spawned task is executed correctly after syncing.

4.2.1 Validation

Unfortunately, a proof of the functional correctness of the deque cannot directly be induced from a passing verification. In other words, the proof can be unsound nevertheless. It is possible for an unsound verification to pass. Therefore, we are required to validate the specification. This can be done by introducing bugs in the specification. We can adjust the implementation of a method to violate its contract, or invert the boolean value of assertions to test their validity. The verification is invalidated if it passes nonetheless. Furthermore, we can move around a false assertion to debug where the specification becomes unsound. For example, we included a false assertion - $assert false$ - after unfolding a resource invariant. As the verification passed nevertheless, we can conclude the specification is unsound. We moved the assertion before the unfold statement, resulting in the verification to fail. Therefore, we discovered that the problem had to do with the resource invariant.

4.3 Assumptions and limitations

Another reason why the verification does not induce a proof is because some assumptions have been made which should be validated as well. First of all, we assume a strongly consistent memory model. Therefore, we can expect that loads and stores are not reordered and the memory fence in the *shrinkShared* method is not required.

Additionally, we expect that the task implements a deterministic algorithm. Hence, we implemented the Fibonacci sequence. We assume that the algorithm does not affect the verification result, but this should be validated by testing the verification for other algorithms. Furthermore, it is uncertain whether VerCors truly verifies the execution of a root task by multiple workers. Therefore, a *main* method should be introduced. A root task, executed by N workers, should calculate the n_{th} Fibonacci number.

Furthermore, VerCors has some limitations. First of all, we cannot define constant values. Therefore, we need to assume that the framework is used correctly. The n and *expected* fields in a task, for example, should not be updated after they have been initially set. Secondly, a thief is not supposed to directly call *pop* or *push* on a victim, and a worker cannot *steal* from itself. A possible approach to verify this is by restraining the method calls to specific workers by a thread identifier. Finally, we assume that a task is not overwritten before it has been executed and *sync* pops off the task. In other words, *pop* should only be called within *sync*. VerCors should be exploited to see whether such verification is feasible.

In addition, the implemented framework is not a direct copy of the Lace framework. The compare-and-swap operation, used in the *steal* method, should be executed on the *tail* and *split* point in one atomic operation. However, we have not taken this into account in the current implementation, because we expect the behaviour to be the same. The *split* point is compared after the *tail* has been incremented, hence another thread can update the *tail* or *split* point between these two operations. However, as the CAS operation on the *tail* is already done, we can ensure that thieves cannot steal the same task simultaneously. Nevertheless, more validation is required to ensure that an update on the *split* point does not affect the verification result. Additionally, an atomic tuple could be created that would incorporate the compare-and-swap of both variables in a single atomic operation.

Some other challenges, with regard to VerCors, were encountered. As an example, some methods required a resource invariant which gave read permission on a variable for which also write permission was necessary. Although, VerCors passed verification, invalidating some assertions does not fail the verification. For example, the *push* method ensures that the *head* is incremented once. However, incrementing the *head* twice did not fail on the postcondition. As a solution, we removed these permissions from the resource invariant, and added the necessary read and write permissions to the method contracts.

A more important issue was encountered when verifying that each task is executed and contains the correct result. It could not be verified that the *done* and *result* field are set at the same time and are only set in the *execute* method. VerCors does not assume that the provided code is the only code that can be executed. Therefore, it is possible to set *done* to *true* without setting the *result* correctly. Consequently, we specified the *Correct* resource invariant, ensuring that any task that is done contains the result of the function call.

4.3.1 Global invariants

In his dissertation, Van Dijk specifies five invariants [16]. Some of these invariants have been verified. For example, the first invariant - a task x **exists** iff $x <$ head - has been verified by wrapping all access to the tasks with the *PIArray* resource invariant. This invariant is used to read a task with an index less than the *head*. If write access is required, then the index must be equal to the *head*. The only place where a task is written is in the push method which adds a new task that will be considered to **exist**. Furthermore, the *steal* method accesses a task. It can be asserted that the index of the task is less than the *head*. However, due to time limitations, most of the other invariants could not be verified. In a perfect world, these invariants can be ensured to always hold. Nonetheless, it is uncertain whether this can be achieved in VerCors. A possible approach could be to define a resource invariant that encapsulates all the invariants. Each method should then have this resource invariant as a pre- and postcondition.

5. CONCLUSION AND DISCUSSION

We implemented the deque from the Lace framework in VerCors' own specification language called PVL. The correctness of the deque should be verified. Although, there is a formal paper proof, an automated proof using a verification tool is crucial. Therefore, we verified the correctness of the deque using a deductive verifier called VerCors. The verification has been limited to assume a strongly consistent memory model.

The specification in PVL passes the verification by VerCors. Therefore, we have decent confidence that the expected result from a spawned task is correctly returned by the matched *sync*. However, we had to make some assumptions, because of bounded time and limitations regarding VerCors. Some challenges were resolved, but proper validation is necessary to justify our assumptions and come to a valid proof.

Furthermore, it cannot be asserted that the *shrinkShared* method returns true for a stolen task in a weakly consistent memory model, because this proof is only valid for a strongly consistent memory model. A memory fence needs to be added to be able to prove this. Unfortunately, there was not enough time to be able to come to this proof. Therefore, we propose directions for future work.

5.1 Future work

All in all, to show that the deque used by the Lace framework functions correctly in a strongly consistent memory model, the specification should be tested for unexpected behaviour. A possible approach would be to introduce a bug by inverting boolean assertions and verifying that the specification passes nevertheless.

Furthermore, the assumptions we made should be validated. A main function should be introduced to verify that workers indeed correctly execute a task together. Ideally, this can be verified for a more general case than the Fibonacci algorithm. In addition, it should be verified that a thief cannot directly call *pop* or *push* on a victim, and that a worker cannot *steal* from itself. A possible solution is to restrain the method calls to specific workers by a thread identifier. Moreover, it should be verified that *sync* is the only method to overwrite tasks, but it is uncertain whether VerCors can verify this. Finally, the compare-and-swap operation should be validated. An atomic tuple could be implemented if the current implementation is invalid.

Additionally, the global invariants specified by Van Dijk [16] should ideally be verified. A possible approach is to create a resource invariant that encapsulates all these invariants. This resource invariant should then be included in the pre- and postconditions of every method.

The next step is to rewrite the existing specification for a weakly consistent memory model, because most computer architectures nowadays do not use a strongly consistent memory model. This would at least require to add a memory fence. After verification passes and a proof has been established, the validity should be checked without the memory fence. The verification tool should then detect a bug and invalidate the verification.

6. REFERENCES

[1] `https://fizalihsan.github.io/technology/work-stealing.png`.

[2] `https://i.stack.imgur.com/czGx1.png`.

[3] A. Amighi, S. Blom, and M. Huisman. Vercors: A layered approach to practical verification of concurrent software. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 495–503, Feb 2016.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.

[5] S. Brookes and P. W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, Aug. 2016.

[6] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[7] K. Faxén. Efficient work stealing for fine grained parallelism. In *2010 39th International Conference on Parallel Processing*, pages 313–322, Sep. 2010.

[8] J.-C. Filliâtre. Deductive software verification. *Int. J. Softw. Tools Technol. Transf.*, 13(5):397–403, Oct. 2011.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[11] R. Hähnle and M. Huisman. 24 challenges in deductive software verification. In G. Reger and D. Traytel, editors, *ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements*, volume 51 of *EPiC Series in Computing*, pages 37–41. EasyChair, 2017.

[12] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 69–80, New York, NY, USA, 2013. ACM.

[13] W. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification.* Dissertation. University of Twente, 2019.

[14] A. Podobas, M. Brorsson, and K.-F. Faxen. A comparison of some recent task-based parallel programming models. In *Proceedings of 3rd Workshop on Programmability Issues for Multi-Core Computers.* Swedish Insitute of Computer Science, 2010.

[15] C. G. Ritson and S. Owens. Benchmarking weak memory models. *SIGPLAN Not.*, 51(8):24:1–24:11, Feb. 2016.

[16] T. van Dijk. *Sylvan : Multi-core decision diagrams.* Dissertation. University of Twente, 2016.

[17] T. van Dijk and J. C. van de Pol. Lace: Non-blocking split deque for work-stealing. In *Euro-Par 2014: Parallel Processing Workshops*, pages 206–217, Cham, 2014. Springer International Publishing.

[18] D. B. Wagner and B. G. Calder. Leapfrogging: A portable technique for implementing efficient futures. *SIGPLAN Not.*, 28(7):208–217, July 1993.