university of groningen

faculty of science and engineering

Concurrent Programming Using Multicore OCaml

July 25, 2024

University of Groningen

Bachelor's Thesis

To fulfill the requirements for the degree of
Bachelor of Science in Computing Science
at University of Groningen under the supervision of
Prof. Jorge A. Perez
and
Dr. Dan Frumin

Stephanos Faustmann (s4644417)

Contents

Page

Abstract **3**

1 Introduction **4**
1.1 Motivation 4
1.2 Backround 4
1.3 Proposal 5
1.3.1 Research Questions 5

2 Implementation **6**
2.1 Methodology 6
2.1.1 Tools and Technologies 6
2.2 Plain Linked List 7
2.3 Coarse-Grained 8
2.3.1 Concept 8
2.3.2 Implementation 8
2.4 Fine-Grained 9
2.4.1 Concept 9
2.4.2 Implementation 9
2.5 Lock-Free 11
2.5.1 Concept 11
2.5.2 Implementation 12
2.6 Testing 15

3 Benchmarking and Results **16**
3.1 Approach 16
3.2 Experiments 16
3.2.1 Experiment 1 16
3.2.2 Experiment 2 19
3.2.3 Conclusion 22

4 Future Work **23**

Bibliography **24**

Abstract

Concurrent programming has become a significant component when it comes to utilizing the capabilities of modern multicore processors. In this project, we aim to delve into the scope of concurrent programming in OCaml. In order to study Coarse-Grained, Fine-Grained and Non-Blocking synchronization, we will implement a linked list. The concurrent operations applied on the linked list will be node addition, node deletion and functionality to check if a node is contained in the linked list. We assess the synchronization approaches in terms of efficiency under different ratios of these operations. Moreover, we benchmark execution time and how well each algorithm scales in terms of contention. Through the insights gained from this study, we aim to contribute to future related work on synchronization in OCaml. We expect the project's outcome to provide valuable insights into optimizing concurrent programming strategies.

1 Introduction

1.1 Motivation

The growing complexity and interconnectedness of systems have made concurrency in computing science increasingly fundamental. Traditionally, speedup had been achieved by increasing clock speeds and, more recently, by adding multiple processing cores to the same chip [1]. With the particular usage of a functional programming language named OCaml, we delve into concurrency with the utilization of its multicore extension. Recent advancements in OCaml have provided us with an intriguing foundation to explore and understand. While harnessing these advancements, we aim to implement a concurrent linked list in several different ways, which will be referred to as synchronization approaches. These approaches include coarse-grained, fine-grained and non-blocking synchronization techniques.

Several factors motivated this research topic. Firstly, the lack of existing literature regarding the comparison of performance in concurrent linked list implementations in OCaml is notable. This is not the case in other functional programming languages such as Haskell [2]. Consequently, this research paper aims to fill this gap to some extent. Moreover, given the widespread adoption of OCaml across multiple industry companies [3], there is a strong motivation to further investigate this programming language through our project. Furthermore my academic background in courses such as Parallel Programming and Algorithms and Data Structures have spurred a deep interest in this particular topic. This interest is magnified when the acquired knowledge is applied in the context of functional programming.

Concluding, we believe that the diverse design methodologies and the advent of the multicore update in OCaml present a compelling area for development.

1.2 Backround

Presenting and implementing a concurrent data structure in multicore OCaml relies on the intersection of functioning programming paradigms and multithread computing concepts. In this subsection, we aim to analyze the characteristics OCaml has to offer in relation to concurrent programming approaches and to elaborate on synchronization methods in data structures.

The ”recent” multicore extension (16th December 2022) marked a breakthrough in the world of OCaml. With OCaml 5.0, a completely new runtime system with support for shared parallelism and effect handlers was introduced [4]. Effect handlers allow for non-local control flow mechanisms such as generators, async/await, lightweight threads and coroutines to be composably expressed [5].

Shared parallelism in OCaml 5.0 enables multiple domains to execute in parallel, sharing the same memory space. This is a significant enhancement over the previous versions of OCaml, which were limited to concurrency through cooperative threading without true parallelism. Hence the new runtime’s support for effect handlers and shared parallelism further extends the capabilities of concurrency patterns.

Performing synchronized access to shared resources using multiple threads, leads to an overall speedup in execution time [6] and offers much more creative solutions to modern computer science problems. Specifically, the management of shared resources such as lists, is an important area of focus. Diverse list algorithms offer different approaches [7], which are also mentioned in the methodology subsection. Beyond lists, the OCaml community has developed concurrent data structures, including stacks and queues, which offer invaluable resources for our research. Noteworthy examples include the lock-free data structures [8] and the use of the Saturn library, which provides implementations of

concurrent queues in OCaml [9].
It is important to mention that there are challenges in our implementation. Even though the introduction of shared memory and effect handlers in OCaml made synchronization of threads possible, there exist some limitations to this day. We should consider effect handlers in OCaml 5.0 experimental as they are not supported as a language feature with new syntax (even though this is likely to change in the future) [10]. Moreover, Ocaml's lack of built-in concurrency support, unlike Erlang [11], necessitates greater manual effort and poses more restraints to the user. Furthermore, the way OCamls garbage collector is constructed with stop the world pauses [12], can lead to performance bottlenecks in highly concurrent scenarios. This is not the case for instance in Java, which contains a garbage collector able to avoid this issue [13].

1.3 Proposal

We propose several synchronization implementations of a particular data structure in multicore OCaml. This study aims to unravel the implications and complications of various multi thread programming techniques within a functional programming paradigm. Apart from the analysis of coarse-grained, fine-grained and non-blocking synchronization methods, we also aim to evaluate and compare their performance across a number of dimensions such as execution time and how well each algorithm scales under contention. Moreover we study performance based on different input of operations.
The proposed method will make use of a particular data structure, namely a linked list, integrating concurrency into three main linked list methods: node insertion, node deletion and the contains method, which is responsible for checking if a node exists in a linked list. We chose the linked list because many implementations exist in other languages [14][7], setting a solid foundation for us to be inspired by and build upon.
This proposal leads to the following research questions :

1.3.1 Research Questions

- How do different synchronization approaches affect overall linked list performance in multicore OCaml?
- What are the challenges when implementing different synchronization approaches in multicore OCaml for linked lists?
- How do we deal with the common issues that arise in parallel and concurrent computing in our implementations?

With the methods we describe in the following section, we expect these research questions to lead to the following outcomes:

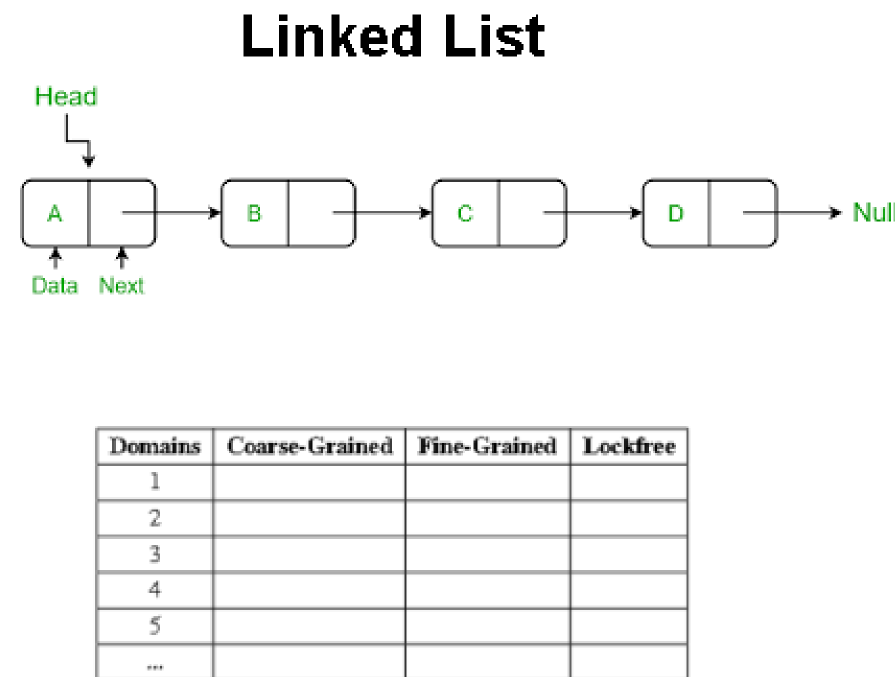
- Analysis and graphical representations/tables of execution time and contention of the synchronization approaches
- Cross comparison of performance between lock-based and lock-free synchronization methods with different inputs of operation ratios.
- Strategies to minimize and manage race conditions, ensuring data integrity across multiple threads.

2 Implementation

In the following section we give a brief introduction of the methodology we used in our research. Moreover we discuss each approach in more detail, highlighting their concept and implementation details.

2.1 Methodology

This section describes the methodology and resources we used in our research. We focused on existing literature to fill our theoretical gaps. In order to understand how different synchronization approaches work and the foundations of multicore OCaml, we used several resources. The main sources used in our research are ”Retrofitting parallelism onto OCaml” [12] and ”The art of multiprocessor programming” [7]. The following image illustrates the components of the project.



Domains	Coarse-Grained	Fine-Grained	Lockfree
1			
2			
3			
4			
5			
...			

Figure 1: Visual illustration of components.

We sourced the image in Figure 1 from the GeeksforGeeks website [15].

2.1.1 Tools and Technologies

Apart from theory, we also set up a multicore OCaml development environment equipped with standard tools for concurrent programming. This includes the configuration of the OCaml environment to support the external libraries such as kcas [16]. We also used modules such as Atomic and Mutex to assist in our different implementations [17] [18]. Additionally, we integrated benchmarking tools in order to facilitate comparative analysis.
Our research was written in LaTex, using the online editor - Overleaf. We used a Github repository to store the foundation code of the linked list. Additionally, we employed a script using dune for evaluating the diverse implementations. Lastly, in order to run the code in the multicore environment we used a personal laptop.

2.2 Plain Linked List

In this subsection we talk about the linked list implementation, which will act as a foundation for all of our synchronization approaches. Each node in the linked list will contain data and a reference to the next node, adhering to the description of the data structure as shown below.

```
type 'a node = {
  value : 'a;
  key : int;
  mutable next : 'a node option;
}
```

While the key field is the node values hash code, the value field correspoonts to the nodes value and the mutable next field is a pointer to an optional node. All nodes are sorted in ascending order of keys, providing an efficient way to detect when an item is absent. We enhanced this structure with capabilities for node removal and addition, as demonstrated in Figures 2 and 3 [19][20], respectively.

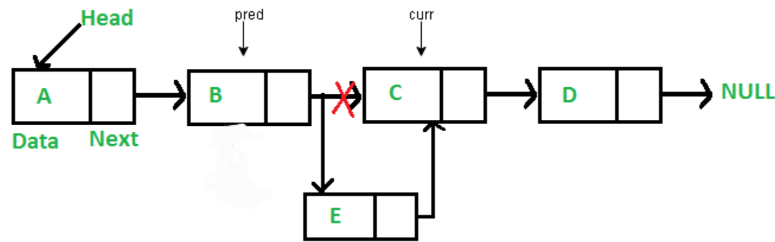


Figure 2: Node Addition

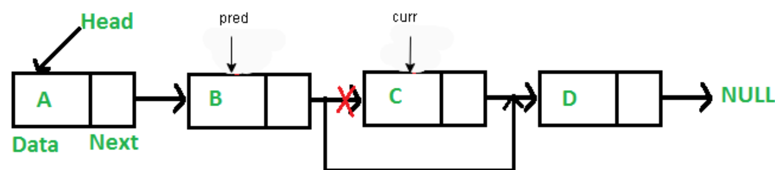


Figure 3: Node Deletion

In Figure 2, a thread adding a node E uses two variables: curr is the current node, and pred is its predecessor. The thread moves down the list comparing the keys for curr and E. If a match is found, the item is already present, so it returns false. If curr reaches a node with a higher key, the item is not in the set so we set E's next field to curr, and pred's next field to E. On the other hand, in Figure 3, to delete curr, the thread sets pred's next field to curr's next field.
Additionally our linked list has functionality to check if an element is contained within the list, which is also depicted below
In Figure 4 the hash value of value 3 is compared with each key field of the linked list throughout the traversal. If a match is found, the function call returns true, else if while traversing we reach an element with a bigger hash key value, that means the value 3 is not in the list and we return false.

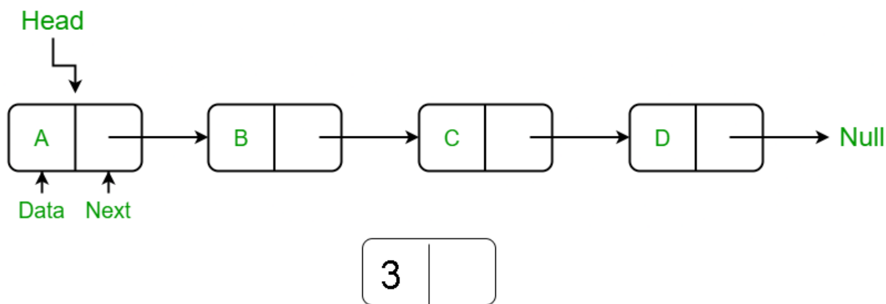


Figure 4: Node Contains

Furthermore, the data structure encompasses two different types of nodes, sentinel nodes and non sentinel nodes. The sentinel nodes act as the head and the tail of the list, containing the min int and max int value in key field respectfully. These two nodes are constant and will never be removed. On the other hand non sentinel nodes are the nodes in between the head and the tail of the list which are mutable. The sentinel nodes can also be seen in the creation of the linked list in our code.

```
let create_linkedlist () : 'a linkedlist =
  let sentinel1= {value= min_int;key= min_int;next= None } in
  let sentinel2= {value= max_int;key= max_int;next= None } in
  sentinel1.next<- Some sentinel2;
  {
    firstnode= sentinel1;
    lastnode= sentinel2;
  }
```

While the plain linked list provides a straightforward and efficient implementation, it serves as a fundamental structure for building more advanced synchronization implementations. The subsections below aim to explore and elaborate on these synchronization methods

2.3 Coarse-Grained

2.3.1 Concept

Coarse-grained synchronization uses a single mutex lock [18] to control access to the entire data structure, ensuring that only one thread can modify the linked list at any given time. The way it works is by locking the entire linked list during any operation that modifies it. This ensures that no other thread can access or modify the list while the lock is held, preventing this way any race conditions. The linearization point for any method call that acquires a lock is the instant the lock is acquired.

2.3.2 Implementation

The usage of the mutex lock can also be seen when creating the linked list .

```
let create_linkedlist () : 'a linkedlist =
  let sentinel1= {value= min_int;key= min_int;next= None } in
  let sentinel2= {value= max_int;key= max_int;next= None } in
  sentinel1.next<- Some sentinel2;
  {
```

```
    firstnode= sentinel1;
    lastnode= sentinel2;
    lock= Mutex.create();
  }
```

Additionally the following code depicts the usage of the mutex lock in our implementation.

```
let additem linkedlist value =
  Mutex.lock linkedlist.lock;
  (* ... *)
  Mutex.unlock linkedlist.lock;

let removeitem linkedlist value =
  Mutex.lock linkedlist.lock;
  (* ... *)
  Mutex.unlock linkedlist.lock;

let contains linkedlist value =
  Mutex.lock linkedlist.lock;
  (* ... *)
  Mutex.unlock linkedlist.lock;
```

2.4 Fine-Grained

2.4.1 Concept

Fine grained synchronization improves upon the coarse-grained synchronization by allowing greater concurrency. Instead of locking the entire linked list, we use a mutex lock [18] for each node in order to control access to individual nodes or sections of the list. This approach increases overall throughput and reduces contention by allowing multiple threads to operate on different parts of the list simultaneously.

2.4.2 Implementation

The code snippet bellow shows modified node structure to include a lock at each node.

```
type 'a node = {
 value: 'a;
 key: int;
 lock: Mutex.t;
 mutable next: 'a node option;
  }
```

Moreover, since we are using a lock per node, the "create_linkedlist" method is changed to adhere to these standards

```
let create_linkedlist () : 'a linkedlist =
  let sentinel1= {value= min_int;key= min_int;lock= Mutex.create();next =
      None } in
```

```
3   let sentinel2= { value = max_int; key= max_int; lock= Mutex.create (); next =
        None } in
4   sentinel1.next <- Some sentinel2;
5   {
6     firstnode= sentinel1;
7     lastnode= sentinel2;
8   }
```

The controlled access to individual sections of the list can also bee seen in the ”additem”, ”removeitem” and ”contains” method, in which we lock nodes in pairs and release them after performing the necessary operations.

```
1  let additem linkedlist value =
2      let key= Hashtbl.hash value in
3      let pred= linkedlist.firstnode in
4      Mutex.lock pred.lock;
5      let rec find_and_insert pred curr_opt =
6        match curr_opt with
7        | Some curr ->
8          Mutex.lock curr.lock;
9          if curr.key< key then (
10           Mutex.unlock pred.lock;
11           find_and_insert curr curr.next
12         ) else if curr.key= key then (
13           Mutex.unlock curr.lock;
14           Mutex.unlock pred.lock;
15           false
16         ) else (
17           let new_node = { value= value; key= key; next= curr_opt; lock=
                Mutex.create () } in
18           pred.next <- Some new_node;
19           Mutex.unlock curr.lock;
20           Mutex.unlock pred.lock;
21           true
22         )
23       | None ->
24         Mutex.unlock pred.lock;
25         false
26     in
27     find_and_insert pred pred.next
```

```
1  let removeitem linkedlist value =
2      let key= Hashtbl.hash value in
3      let pred= linkedlist.firstnode in
4      Mutex.lock pred.lock;
5      let rec find_and_remove pred curr_opt =
6        match curr_opt with
7        | Some curr ->
8          Mutex.lock curr.lock;
9          if curr.key< key then (
10           Mutex.unlock pred.lock;
```

```
          find_and_remove curr curr.next
        ) else if curr.key= key then (
          pred.next<- curr.next;
          Mutex.unlock curr.lock;
          Mutex.unlock pred.lock;
          true
        ) else (
          Mutex.unlock curr.lock;
          Mutex.unlock pred.lock;
          false
        )
      | None ->
        Mutex.unlock pred.lock;
        false
    in
    find_and_remove pred pred.next
```

```
let contains linkedlist value =
    let key= Hashtbl.hash value in
    let pred= linkedlist.firstnode in
    Mutex.lock pred.lock;
    let rec find pred curr_opt =
      match curr_opt with
      | Some curr ->
        Mutex.lock curr.lock;
        if curr.key< key then (
          Mutex.unlock pred.lock;
          find curr curr.next
        ) else if curr.key= key then (
          Mutex.unlock curr.lock;
          Mutex.unlock pred.lock;
          true
        ) else (
          Mutex.unlock curr.lock;
          Mutex.unlock pred.lock;
          false
        )
      | None ->
        Mutex.unlock pred.lock;
        false
    in
    find pred pred.next
```

2.5 Lock-Free

2.5.1 Concept

Lock-free synchronization aims to enhance concurrency by eliminating the need for locks entirely. Instead of using mutexes, this approach relies on atomic operations to manage concurrent access to

the data structure. In a Lock-free linked list implementation, nodes contain an atomic marked variable which in addition with the nodes next field is treated as a single atomic unit. The reason for this is because we want to ensure that the nodes fields cannot be updated, after that node has been logically or physically removed from the list. A thread logically removes a node by setting the mark bit in the node's next field, and shares the physical removal with other threads when performing add() or remove(). As each thread traverses the list, it cleans up the list by physically removing any marked nodes it encounters.

2.5.2 Implementation

In our implementation, we utilize atomic markable references by using kcas [16] to handle synchronization. This involves using the AtomicMarkable module, containing markable_CAS operation to safely update node references without locks, ensuring that the linked list remains consistent even under concurrent modifications. Moreover operations such as get_marked and get_reference are used, which return the marked value of the node and the reference value of the node in question respectively. The node structure is modified to include an atomic markable reference to the next node.

The implementation above can bee seen in the code snippet below:

```
module AtomicMarkable = struct
type 'a markable = {
ref : 'a Kcas.Loc.t;
mark : bool Kcas.Loc.t
}
let markable_CAS r exRef exMark newRef newMark =
let tx ~xt =
let _ = Xt.compare_and_set ~xt r.ref exRef newRef in
Xt.compare_and_set ~xt r.mark exMark newMark
in Xt.commit { tx }
let get_mark r = Loc.get r.mark
let get_reference r = Loc.get r.ref
let get_marked r =
(Loc.get r.ref, Loc.get r.mark)
let make_markable value mark =
{ ref = Loc.make value; mark = Loc.make mark}
end

type 'a node = {
 value: 'a;
  key: int;
  next : ('a node) AtomicMarkable.markable option
}
```

When it comes to creating the sentinel nodes and creating new nodes in general, we use the AtomicMarkable.make_markable operation, which creates an atomic markable reference as shown below.

```
let create_linkedlist () : 'a linkedlist =
let sentinel1 = {value= min_int;key= min_int; next = None } in
let sentinel2 = {value= max_int;key= max_int; next = None } in
```

```
sentinel1.next <- Some (AtomicMarkable.make_markable sentinel2 false);
{
firstnode = sentinel1;
lastnode = sentinel2;
}
```

In the additem and removeitem functions we decided to factor out functionality, in order to adhere to the book and the lock-free algorithm implementation [7]. The factored out functionality can be seen by adding a find_window function as shown below.

```
let find_window (linkedlist : 'a linkedlist) key: 'a window =
  let  retry () =
    let pred = ref linkedlist.firstnode in
    let curr = ref (match !pred.next with
      | Some nxt -> get_reference nxt
      | None -> raise (Invalid_argument "find_window: next node is None")
        ) in
    let marked = ref false in
    let continue_traversal = ref true in
    while !continue_traversal do
      try
        while true do
          match !curr.next with
          | None -> raise Exit
          | Some nxt ->
            let current_value,current_mark= get_marked nxt in
            marked := current_mark;
            let succ = ref  current_value in
            while !marked do
              let snip = markable_CAS (match !pred.next with Some p -> p
                 | None -> raise Exit) !curr false !succ false in
              if not snip then raise Exit;
              curr:= !succ;
              let current_value,current_mark= get_marked (match !curr.
                 next with Some c -> c | None -> raise Exit) in
              marked := current_mark;
              succ := current_value;
            done;
            if !curr.key>=key then (
              continue_traversal := false;
              raise Exit
            );
            pred := !curr;
            curr := !succ;
        done
      with Exit ->
        continue_traversal := false
    done;
    let p = !pred in
    let c = !curr in
    { pred = p ; curr = c }
```

```
    in
    retry ()
```

The reason we used the find_window is inorder to help with navigation. The method accepts the head of the list and a head node and a key a, and traverses the list, seeking to set pred to the node with the largest key less than a, and curr to the node with the least key greater than or equal to a. As thread A traverses the list, each time it advances currA, it checks whether that node is marked. If so, it calls markable_CAS to attempt to physically remove the node by setting predA's next field to currA's next field. This call tests both the field's reference and Boolean mark values, and fails if either value has changed. A concurrent thread could change the mark value by logically removing predA, or it could change the reference value by physically removing currA. If the call fails, A restarts the traversal from the head of the list, otherwise the traversal continues[7].

```
let additem linkedlist value =
let key= Hashtbl.hash value in
let rec loop () =
let find = find_window linkedlist key in
let pred = find.pred in
let curr = find.curr in
if curr.key= key then
false
else
let node = make_node value (Some (AtomicMarkable.make_markable curr false))
    in
if AtomicMarkable.markable_CAS (match pred.next with Some p -> p | None
   -> raise Exit) curr false node false then
true
else
loop ()
in
loop ()

let removeitem linkedlist value =
let key= Hashtbl.hash value in
let rec loop () =
let find = find_window linkedlist key in
let pred = find.pred in
let curr = find.curr in
if curr.key <> key then
false
else
let succ = AtomicMarkable.get_reference (match curr.next with Some c -> c
    | None -> raise Exit) in
let snip = AtomicMarkable.markable_CAS (match curr.next with Some c -> c
   | None -> raise Exit) curr false succ true in
if not snip then loop ()
else
let _ = AtomicMarkable.markable_CAS (match pred.next with Some p -> p |
   None -> raise Exit) curr false succ false in
true
```

```
in
loop ()
```

The contains method does not participate in the cleanup of logically removed nodes. However it checks if the node we are looking for is contained within the list and whether it has been marked or not.

```
let contains linkedlist value =
let marked = ref false in
let key= Hashtbl.hash value in
let curr = ref (AtomicMarkable.get_reference (match linkedlist.firstnode.
   next with Some f -> f | None -> raise Exit)) in
while !curr.key<key do
let current_value, current_mark = AtomicMarkable.get_marked (match !curr.
   next with Some c -> c | None -> raise Exit) in
curr := current_value;
marked := current_mark;
done;
!curr.key= key&& not !marked
```

2.6 Testing

Concurrent programming introduces various challenges, such as race conditions, deadlocks, and data corruption. To ensure the correctness of our implementation and to mitigate these issues, we decided to thoroughly test each approach. We can compile and test our code using the ”dune runtest” command, which executes all test files on our current implementations. For more detailed analysis, we can check the outputs of a specific file by running the command ”dune exec ./test/test_”implementation name”.exe”. Given the often unpredictable nature of concurrent operations, we opted to avoid assertions when testing operations in parallel and instead use print statements to verify the results. This approach allows us to closely monitor the behavior of the code under concurrent conditions and ensure its reliability.

3 Benchmarking and Results

3.1 Approach

Due to the noticeable lack of pre-existing researches for concurrent linked list benchmarking in OCaml, we decided to build upon available resources in other languages. Our approach was inspired by the research paper ”Comparing the performance of concurrent linked-list implementations in Haskell” [2].
The benchmarking approach described in the paper involved generating test data consisting of an initial list of 3000 elements and 8 lists of 3000 operations with a ratio of 2 finds to 1 delete to 4 inserts. Each benchmark run was measured by loading the test data into memory, generating the initial linked list, and then starting the clock before creating 8 threads, each performing one of the pre-generated lists of operations. When all threads finished, the clock was stopped, and the elapsed time was reported. The average of 3 runs of each benchmark was taken [2].
We decided to expand upon this approach by allowing more flexible benchmarking options. In our benchmarking, we enable the user to set custom ratios for additem, removeitem, and contain operations. Moreover, instead of assigning an operations list to each thread by default, we divide the initial operations list into partitions. Additionally, users can set custom amounts for the number of repetitions, elements and operations. Furthermore due to the significant time it takes for a domain to spawn in OCaml, we decided to use a barrier which waits for all threads to spawn before starting concurrent execution.
Below is an illustrative example of the degree of flexibility we have in our benchmarking:

```
Enter the number of elements:
Enter the number of list operations:
Enter the number of iterations:
Enter the ratio of additions (0.0 - 1.0):
Enter the ratio of deletions (0.0 - 1.0):
Enter the ratio of contains (0.0 - 1.0):
Do you want to test a sequence of domains (yes/no):
```

3.2 Experiments

It is noteworthy mentioning that we are not only comparing absolute execution time of each implementation, but also how well each algorithm scales in terms of contention and metrics manipulation. For our first experiment, we decided to change the number of elements in the list, keeping the other metrics exactly the same. We decided to opt for 3000 operations, 15 iterations and an even ratio across the 3 operations. The results can be seen below in Table 1 and Figure 5 for a linked list with 3000 elements and in Table 2 and Figure 6 for a linked list with 100 elements.

3.2.1 Experiment 1

Benchmark 1: 3000 elements

- Number of elements: 3000
- Number of list operations: 3000

- Number of iterations: 15
- Ratios:
  - Additions: 0.33333333
  - Deletions: 0.33333333
  - Contains: 0.33333333

Domains	CoarseGrained	FineGrained	LockFree
2	0.030970	0.044222	0.060142
3	0.035326	0.033804	0.047661
4	0.033655	0.026787	0.036993
5	0.032754	0.023084	0.029730
6	0.032915	0.030821	0.022314
7	0.031441	0.030599	0.019964
8	0.028902	0.032517	0.017936
9	0.029223	0.034549	0.017357
10	0.027191	0.034418	0.016146
11	0.029613	0.039109	0.015589

Table 1: Benchmark 1 results for different implementations across various domains

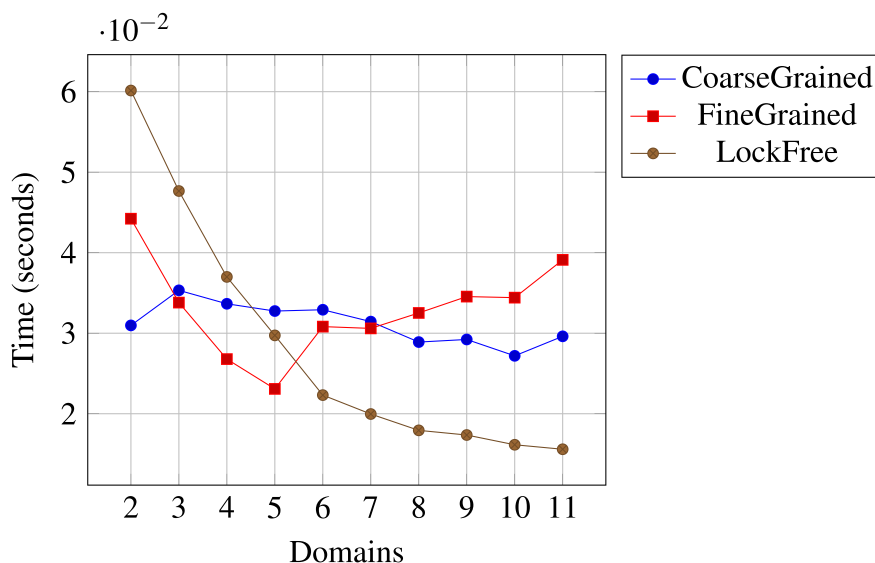


Figure 5: Performance graph of Table 3

Benchmark 2: 100 elements

- Number of elements: 100

- Number of list operations: 3000
- Number of iterations: 15
- Ratios:
  - Additions: 0.33333333
  - Deletions: 0.33333333
  - Contains: 0.33333333

Domains	CoarseGrained	FineGrained	LockFree
2	0.005038	0.009520	0.005728
3	0.006337	0.008099	0.004572
4	0.007224	0.007356	0.003831
5	0.007557	0.007188	0.003154
6	0.011936	0.013491	0.004068
7	0.012542	0.011954	0.003832
8	0.014245	0.013241	0.003833
9	0.013526	0.013200	0.003504
10	0.013088	0.013268	0.003371
11	0.013372	0.013490	0.003361

Table 2: Benchmark 2 results for different implementations across various domains

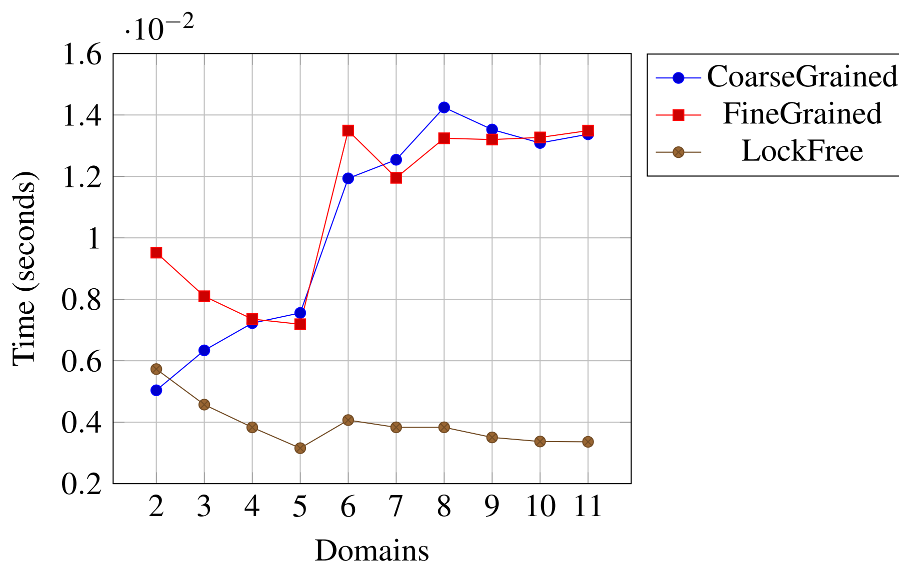


Figure 6: Performance graph of Table 2

From the results of the first experiment, it is evident that the lock-free implementation is the most efficient and scalable solution for handling parallel operations in both datasets. By using 3000 elements, the differences in performance between the implementations is more noticeable. The lock-free

implementation shows significant performance gains with increasing domains, demonstrating its superior ability to manage contention and parallelism. On the other hand, the coarse-grained approach has the least amount of overall speedup, showing limited scalability due to high contention and reduced parallelism. Moreover, in the fine-grained approach, speedup is observed for up to 5 domains in both benchmarks. Concluding, the fine-grained approach outperforms the coarse-grained approach in terms of speedup in both cases, but the benefits of reduced contention are more evident with the larger dataset.
With 100 elements, although the contention is less severe, the coarse-grained implementation still does not scale well, highlighting its inefficiency in handling parallel operations. The speedup in the fine-grained approach is significant with fewer domains, but the performance gains plateau as the domain count increases due to the smaller amount of elements. The lock-free implementation consistently outperforms the others in both scenarios, but the performance improvement is more substantial with 3000 elements. This indicates that the lock-free approach efficiently manages contention and parallelism, regardless of the workload size, making it the most efficient and scalable solution for both small and large datasets.

3.2.2 Experiment 2

In our second experiment, we decided to vary both the number of operations and the operation ratios. We aimed to determine if any algorithm's performance changes with different operation ratios. Increasing the number of operations to 9000 was intended to help us understand how the differences in operations affect performance. Therefore, we used 9000 operations and adjusted the ratios in both benchmarks, as shown below.

Benchmark 3:

- Number of elements: 3000
- Number of list operations: 9000
- Number of iterations: 15
- Ratios:
  - Additions: 0.5
  - Deletions: 0.4
  - Contains: 0.1

Domains	CoarseGrained	FineGrained	LockFree
2	0.129047	0.161767	0.244729
3	0.141547	0.124854	0.187765
4	0.136595	0.097755	0.144720
5	0.137125	0.085311	0.120772
6	0.130621	0.101442	0.087080
7	0.122571	0.105900	0.075700
8	0.122576	0.107811	0.067502
9	0.119679	0.111649	0.062242
10	0.114058	0.114413	0.058788
11	0.116295	0.122124	0.062897

Table 3: Benchmark 3 results for different implementations across various domains

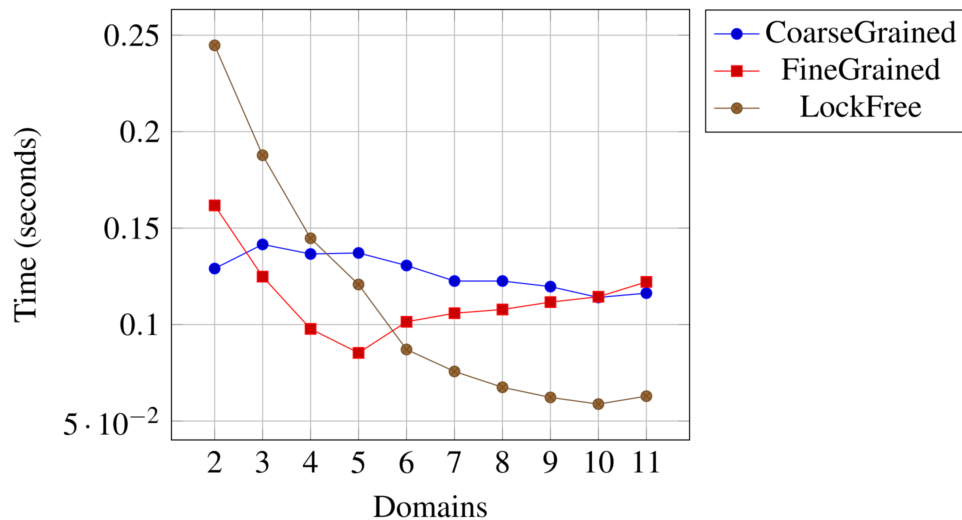


Figure 7: Performance graph of Table 3

Benchmark 4:

- Number of elements: 3000
- Number of list operations: 9000
- Number of iterations: 15
- Ratios:
  - Additions: 0.33333333
  - Deletions: 0.33333333
  - Contains: 0.33333333

Domains	CoarseGrained	FineGrained	LockFree
2	0.115959	0.143806	0.212581
3	0.126814	0.113421	0.164036
4	0.126929	0.089772	0.129655
5	0.123509	0.076160	0.107634
6	0.114848	0.093900	0.076207
7	0.115994	0.098172	0.067843
8	0.115279	0.101927	0.058895
9	0.110433	0.103519	0.054976
10	0.105056	0.107147	0.052091
11	0.107647	0.112359	0.053356

Table 4: Benchmark 4 results for different implementations across various domains

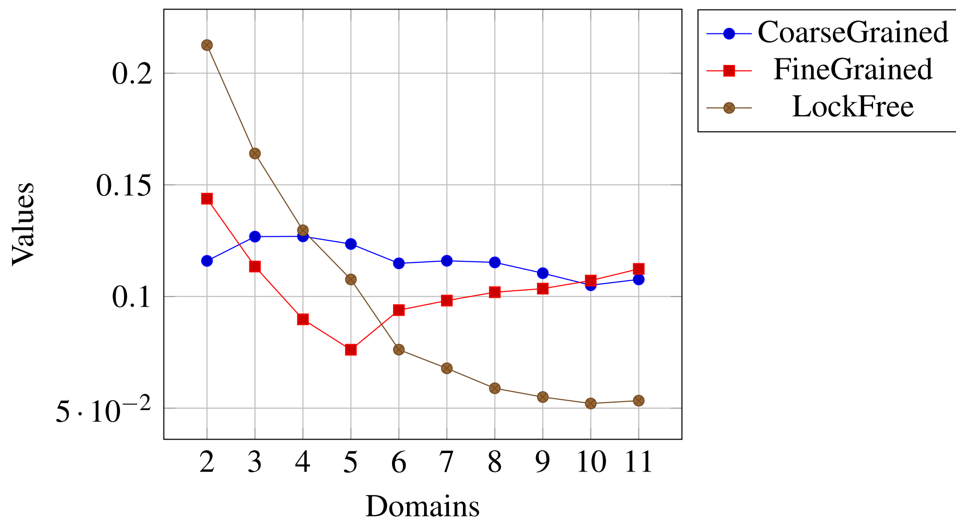


Figure 8: Benchmark 4 results for different implementations across various domains

As expected, the execution time of all implementations is less when the ratios are more balanced. The reasoning behind this is due to the fact that pointer manipulation takes on average more time than reading a value within a list. Moreover the reason the lock-free implementation is not the fastest up until 6 domains is because of the implementation. When removing or adding an element to a list, the algorithm traverses the list and physically removes all logically removed marked nodes, which makes additions and deletions more time-consuming compared to the coarse-grained and fine-grained implementations.

Additionally, the coarse-grained and fine-grained implementations benefit from pointer manipulations, leading to faster execution times at smaller domain sizes. As the domain size increases, the fine-grained implementation particularly shows a significant improvement, suggesting better scalability and efficiency in handling higher operation loads with more granular locking mechanisms.

3.2.3 Conclusion

In both experiments, the lock-free algorithm scales the best in terms of contention as the number of domains increases. This makes sense from an implementation perspective. In coarse-grained synchronization, there is contention and reduced parallelism because no more than one thread can bypass a mutex lock at a time, causing other threads to wait their turn to acquire the lock. This can become a bottleneck in a highly concurrent environment as threads must wait for the lock to be released before proceeding.

In fine-grained synchronization, contention is reduced compared to coarse-grained synchronization. The speedup over the coarse-grained approach is due to the fine-grained approach allowing concurrent manipulation of multiple domains on the linked list simultaneously, which causes the algorithm to scale better. However, there are still limitations in the fine-grained approach. Mutex locks acquired to traverse the list can cause contention in highly concurrent scenarios, as threads wait for permission to acquire an unused mutex lock.

The lock-free approach significantly improves parallelism and throughput by allowing multiple threads to operate on the linked list without waiting for locks to be released. This results in a more efficient utilization of resources, especially in highly concurrent environments.

It is important to mention that the limited speedup after a certain number of domains for all implementations can be explained by Amdahl's Law. Amdahl's Law states that the potential speedup of a program through parallelization is limited by the portion of the program that must remain serial. It highlights that even with an infinite number of processors, the maximum achievable speedup is constrained by the serial fraction of the workload [21].

These findings highlight the importance of selecting the appropriate implementation based on the specific workload characteristics and operation ratios. The lock-free implementation is advantageous in high-concurrency environments with balanced operations, while the coarse-grained and fine-grained implementations provide faster responses when using less domains and more operations. This comparative analysis underscores the necessity of tailoring the choice of algorithm to the particular demands of the application to optimize performance effectively.

4 Future Work

Due to time constraints we decided to implement and benchmark three linked list algorithms. However it is possible to expand this project to benchmark even more algorithms such as lazy synchronization and optimistic synchronization. We have taken potential future work into account and ensured that the code is expandable and reusable, making it easy to add more implementations. The current structure of the benchmarking framework allows for a high degree of flexibility, enabling a wide range of experiments and benchmarks depending on the research focus. This setup provides ample room for imagination and creativity in further experimentation and benchmarking.

Bibliography

[1] N. Sudha, “Multicore processor — architecture and programming,” in *2015 19th International Symposium on VLSI Design and Test*, pp. 1–2, 2015.

[2] M. Sulzmann, E. S. Lam, and S. Marlow, “Comparing the performance of concurrent linked-list implementations in haskell,” *SIGPLAN Not.*, vol. 44, p. 11–20, Oct 2009.

[3] “Ocaml in industry.” https://v2.ocaml.org/learn/companies.html. Accessed: 21.03.2024.

[4] “OCaml Releases.” https://ocaml.org/releases. Accessed: 24.02.2024.

[5] K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy, “Retrofitting effect handlers onto ocaml,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 206–221, Association for Computing Machinery, 2021.

[6] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. O’Reilly Media, 2nd ed., 2013.

[7] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[8] P. Chambart, “Lockfree: A lock-free data structures library for ocaml.” https://github.com/chambart/lockfree, 2024. Accessed: 26.03.2024.

[9] OCaml Multicore Team, “Saturn: Concurrent queue implementations for multicore ocaml.” https://github.com/ocaml-multicore/saturn, 2024. Accessed: 26.03.2024.

[10] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and et al., “The ocaml system release 5.1: Documentation and user’s manual,” intern report, Inria, 2023. ⟨hal-00930213v10⟩.

[11] Erlang/OTP Team, “Concurrent programming in erlang.” https://www.erlang.org/doc/getting$_{s}tarted/conc_{p}rog.html.Accessed$: 24.02.2024.

[12] K. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, and A. Madhavapeddy, “Retrofitting parallelism onto ocaml,” *Proc. ACM Program. Lang.*, vol. 4, Aug 2020.

[13] T. Printezis and D. Detlefs, “A generational mostly-concurrent garbage collector,” *SIGPLAN Not.*, vol. 36, p. 143–154, Oct 2000.

[14] P. Tang, P.-C. Yew, and C.-Q. Zhu, “A parallel linked list for shared-memory multiprocessors,” in *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*, pp. 130–135, 1989.

[15] GeeksforGeeks, “Applications of linked list data structure,” n.d.

[16] OCaml Multicore Team, “Kcas: Lock-free data structures for multicore ocaml.” https://github.com/ocaml-multicore/kcas, 2024. Accessed: 26.06.2024.

[17] OCaml Team, “Atomic module.” https://v2.ocaml.org/api/Atomic.html, 2024. Accessed: 26.06.2024.

[18] OCaml Team, “Mutex module.” https://v2.ocaml.org/api/Mutex.html, 2024. Accessed: 26.06.2024.

[19] GeeksforGeeks, “Python program for inserting a node in a linked list.” https://www.geeksforgeeks.org/python-program-for-inserting-a-node-in-a-linked-list/, 2024. Accessed: 2024-07-15.

[20] GeeksforGeeks, “Java program for deleting a node in a linked list.” https://www.geeksforgeeks.org/java-program-for-deleting-a-node-in-a-linked-list/, 2021. Accessed: 26.03.2024.

[21] S. Eyerman and L. Eeckhout, “Modeling critical sections in amdahl’s law and its implications for multicore design,” *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 362–370, 2010.