Synchronization Views for Event-loop Actors

Joeri De Koster
Vrije Universiteit Brussel
jdekoste@vub.ac.be

Stefan Marr
Vrije Universiteit Brussel
smarr@vub.ac.be

Theo D'Hondt
Vrije Universiteit Brussel
tjdhondt@vub.ac.be

Abstract

The actor model has already proven itself as an interesting concurrency model that avoids issues such as deadlocks and race conditions by construction, and thus facilitates concurrent programming. The tradeoff is that it sacrifices expressiveness and efficiency especially with respect to data parallelism. However, many standard solutions to computationally expensive problems employ data parallel algorithms for better performance on parallel systems.

We identified three problems that inhibit the use of data-parallel algorithms within the actor model. Firstly, one of the main properties of the actor model, the fact that no data is shared, is one of the most severe performance bottlenecks. Especially the fact that shared state can not be read truly in parallel. Secondly, the actor model on its own does not provide a mechanism to specify extra synchronization conditions on batches of messages which leads to event-level data-races. And lastly, programmers are forced to write code in a continuation-passing style (CPS) to handle typical request-response situations. However, CPS breaks the sequential flow of the code and is often hard to understand, which increases complexity and lowers maintainability.

We proposes *synchronization views* to solve these three issues without compromising the semantic properties of the actor model. Thus, the resulting concurrency model maintains deadlock-freedom, avoids low-level race conditions, and keeps the semantics of macro-step execution.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Design, Languages

Keywords Actor Model, Synchronization, Data Parallelism

1. Introduction

Because of the multicore evolution, parallel programming is no longer only useful for high performance computing (HPC) applications but is also useful in the desktop and embedded world. In contrast to HPC, where speedup is generally gained by splitting up a single problem and processing the input data in parallel, we assume that desktop applications are generally more task driven. That means, desktop, or more general user-centric applications have in general less parts that are data parallel but possible parallelism comes from distinctly different activities that need to be done in the problem domain.

Thus, the starting hypothesis of our research is that such application benefit more from a language model that provides good abstractions for expressing task driven parallelism in a safe way. The actor model [1, 2] is such a model. This model has mainly been used in a distributed setting (Erlang [3], SALSA [4], AmbientTalk [5]) where the architectural benefits of its concurrency model are more important than the speed gained by running multiple actors in parallel.

However, user-centric applications often have to deal with certain subproblems that are inherently data parallel. Examples are large data-intensive spread-sheets or search operations where the use of data parallel approaches could reduce the latency towards the user. Thus, task-driven parallelism should not be the only method of parallelism provided by a programming language, otherwise the application can not fully exploit the available parallelism of such data driven problems.

The goal of this paper is to argue for a computational model that combines a safe task-based parallelism model with the ability to use a restricted number of data-driven optimizations.

From that perspective, the actor model is a good starting point. It is free of low-level data races and in its original inception deadlock-free by construction. Those two properties are delivered by the actor model because it adheres to three fundamental rules:

- Actors do not share mutable state
- Actors communicate only asynchronously
- Actors are scheduled *fairly*, i. e., no actor is permanently starved

We however believe that it is the first two rules that place the largest restriction on the expressiveness of the actor model. A consequence of the first rule is that data-parallel algorithms that read from and write to a shared resource are impossible to express efficiently within an actor program. A consequence of the second rule is that communication with a remote shared resource[1] requires most of the time the use of some request-response idiom which is expressed in most languages using *continuation-passing style* (CPS) and results in less maintainable code (cf. Sec. 2).

Our solution is an extension of the actor model so that it, in addition to expressing task parallelism, also becomes useful for expressing data parallelism within an application. This extension has been crafted in such a way that the extended model keeps the same guarantees, such as race condition-freedom, deadlock-freedom and macro-step semantics, as the original model. We introduce synchronization views as a way of synchronizing access to a remote resource and demonstrate that this extension is useful when implementing data-parallel algorithms.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
Copyright © 2012 ACM 978-1-4503-1160-1/12/02...$10.00

[1] A resource is remote for an actor if that resource is located in the memory space of another actor. Having different remote actors does not necessarily imply distribution.

2. The problem: Accessing non-local shared state

The main problem of the actor model is that share state is traditionally represented by an additional independent actor which encapsulates that shared state or resource. However, this leads to the following three problems:

No parallel reads. State which is conceptually shared can never be read truly in parallel because all accesses to that state are sequentialized by the event queue of the encapsulating actor.

No synchronization conditions. The traditional actor model does not allow to specify extra synchronization conditions on different events since the order in which events from different senders are handled is nondeterministic. There are some solutions for this problem but as explained later in this section they often only solve part of the problem.

Continuation-passing style enforced. Using a distinct actor to represent conceptually shared state implies that this resource can not be accessed directly from any other actor since all communication happens asynchronously within the actor model. Thus, the programmer needs to explicitly handle a request-response situation, which usually forces the programmer to employ CPS.

3. Views as a synchronization mechanism

Most of the identified problems stem from the fact that an actor cannot have synchronous access on the state or part of the state of another remote actor. Our *view* abstraction solves that issue. Views are a synchronization mechanism that allows one actor to have synchronous access to multiple objects or parts of objects owned by other remote actors. There are two kinds of views, a shared and an exclusive view which mimic multiple reader, single writer access as a synchronization strategy. We added three new primitives to access shared state from within our language:

```
<far-reference>.whenExclusive(<closure>)
<far-reference>.whenShared(<closure>)
whenSharedAndExclusive(<far-reference>+,
                       <far-reference>+,
                       <closure>)
```

The only restriction imposed on the usage of these primitives is that the `far-reference` they access has to be a reference to a remote shallow object. This can be either an object with no scope such as an isolate [5] or a shallow primitive datatype such as an array. This restriction is imposed to guarantee race-condition freeness.

In the listing below, we give an example where sending an `increase` message to the actor will schedule an event that increases the counter in the cell when that cell becomes available for exclusive access. We can access the cell synchronously from within the closure we provided to the `whenExclusive` primitive. The body of that closure is executed in a separate turn if and only if the cell object becomes available for exclusive access. This means that the `get` and `set` method of the cell are executed atomically, which would not have been possible without using views or changing the implementation of the cell object. If we want to read the same value multiple times or read different values synchronously, we can do so for the duration of that turn. We no longer have to employ CPS or futures to read and/or write values from and to a shared resource.

Currently, only multiple reader, single writer access is provided as a synchronization strategy. We also prioritize writers to prevent their starvation. We chose this locking strategy to limit the expressiveness of our primitives in favor of more concise abstractions.

```
cell: isolate({
    c: 0;
    get()@reader:
        c;
    set(n):
        c := n});

act: actor({
    increase(cell):
        cell.whenExclusive(
            lambda(c):
                c.set(c.get() + 1));
```

View requests on a shared object are scheduled by a view-scheduler. These view-schedulers exist on a per-object basis and are created when the first view on that object is requested. This prevents our abstraction to have any overhead on the system when it is not used. Requested views are put in a queue and when the resource associated with that view-scheduler becomes available the next request is handled.

4. Conclusion

The engineering benefits of semantically coarse-grained synchronization mechanisms in general [6] and the restrictions of the actor model [7] have been recognized by others. In particular the notion of *view*-like constructs has been proposed before.

The main contribution of our view-abstractions is to address the inefficient and often confusing (CPS) access to shared state in the actor paradigm by allowing controlled synchronous access on shared state. The advantages of this system over the traditional event-loop model are threefold. Firstly we avoid the continuation passing style of programming when accessing shared state. Secondly we allow the programmer to introduce extra synchronization constraints on groups of messages and lastly we are able to model true parallel reads.

Acknowledgments

Joeri De Koster and Stefan Marr are supported by doctoral scholarships granted by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

References

[1] G. Agha. Actors: a model of concurrent computation in distributed systems. AITR-844, 1985.

[2] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In Proceedings of the 3rd international joint conference on Artificial intelligence, pages 235245. Morgan Kaufmann Publishers Inc., 1973.

[3] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. Concurrent programming in erlang. 1996.

[4] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. ACM SIGPLAN Notices, 36(12):2034, 2001.

[5] T. Van Cutsem, S. Mostinckx, E. Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In Chilean Society of Computer Science, 2007. SCCC07. XXVI International Conference of the, pages 312. Ieee, 2007.

[6] B. Demsky and P. Lam. Views: Object-inspired concurrency control. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pages 395404. ACM, 2010.

[7] R. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, pages 1120. ACM, 2009.