Science of Computer Programming 80 (2014) 52–64

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model

Christophe Scholliers [a,*,1], Éric Tanter [b,2], Wolfgang De Meuter [a]

[a] *Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, Elsene, Belgium*

[b] *PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Avenida Blanco Encalada 2120, Santiago, Chile*

A R T I C L E I N F O

Article history:
Received 3 April 2012
Received in revised form 26 November 2012
Accepted 23 March 2013
Available online 11 April 2013

Keywords:
Actors
Concurrency
Monitors
Efficiency

A B S T R A C T

While the actor model of concurrency is well appreciated for its ease of use, its scalability is often criticized. Indeed, the fact that execution *within* an actor is sequential prevents certain actor systems to take advantage of multicore architectures. In order to combine scalability and ease of use, we propose Parallel Actor Monitors (PAMs), as a means to relax the sequentiality of intra-actor activity in a structured and controlled way. A PAM is a modular, reusable scheduler that permits one to introduce intra-actor parallelism in a local and abstract manner. PAM allows the stepwise refinement of local parallelism within a system on a per-actor basis, without having to deal with low-level synchronization details and locks. We present the general model of PAM and its instantiation in the AmbientTalk language. Benchmarks confirm the expected performance gain.

1. Introduction

The actor model of concurrency [1] is well recognized for the benefits it brings for building concurrent systems. Actors are *strongly encapsulated* entities that communicate with each other by means of asynchronous message passing. Data races are prevented by design in an actor system because data cannot be shared between actors, and actors process messages sequentially. However, these strong guarantees come at a cost: efficiency.

The overall parallelization obtained in an actor system stems from the parallel execution of multiple actors. Upon the reception of a message, an actor executes a task which can only manipulate data local to the actor. Therefore, either data is encapsulated in an actor and only one task can manipulate that data or the data is distributed over multiple actors in order to exploit parallelism. Because of the strong *data-task entanglement* in the actor model, the actor programmer is *forced* to partition her data in order to exploit parallelism. In general, the structure and choice of algorithms strongly depend on the structure of the underlying data [2]. When each actor possesses a partitioning of the data, global operations over the data must be implemented by a well defined protocol between those actors. These protocols have to be carefully encoded to guarantee properties like sequentiality and consistency, important properties that can be taken for granted within a single actor. The actor programmer needs to encode these properties manually as a side effect of trying to exploit parallelism. The implementation of global data operations is low-level, error-prone, and potentially inefficient. Moreover, restructuring large amount of data at runtime by transferring the data from one actor to another, involves high overheads. In conclusion, data-task entanglement ensures important properties at a local level but at the same time forces the programmer to partition

* Corresponding author.
E-mail addresses: cfscholl@vub.ac.be (C. Scholliers), etanter@dcc.uchile.cl (É. Tanter), wdmeuter@vub.ac.be (W. De Meuter).

1 Funded by a doctoral scholarship of the Institute for the Promotion through Science and Technology in Flanders (IWT-Vlaanderen).

2 Partially funded by FONDECYT Project 1110051.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.03.011

her data when trying to implement data-level coarse grained parallelism [3]. The partitioning of data in order to exploit parallelism leads to complex and inefficient code for global operations when coordination is required between the actors that encapsulate the partitioned data.

Let us further illustrate the problem by means of an example. Consider a group of actors, of which one is a dictionary actor. The other actors are clients of the dictionary: one actor does updates (writer), while the others only consult the dictionary (readers). The implementation of the dictionary with actors is easy because the programmer does not need to be concerned with data races: reads *and* writes to the dictionary are ensured to be executed in mutual exclusion. The programmer is sure that no other actor could have a reference to the data encapsulated in the dictionary actor. The asynchronous nature of the actor model also brings distribution transparency as the dictionary code is independent from the physical location of the actors.

However, when the number of readers increases the resulting application performs badly precisely because of the benefits of serial execution of requests to the dictionary actor: there are no means to process read-only messages in parallel and thus the dictionary actor becomes the bottleneck. In order to scale the dictionary, the programmer is thus forced to partition his data over a group of dictionary actors, for example one per letter. As explained, this introduces accidental complexity for global operations over the dictionary. A previous simple operation, such as counting all the words in the dictionary ending at "ing", has to be implemented by a protocol, *e.g.* block all actors that hold parts of the dictionary, ask them to perform this operation on their local data and wait for their answer, and finally release the actors.

The problems of data-task entanglement within the actor model has been acknowledged by many of the current actor implementations as many implementations compromise the strong encapsulation properties for efficiency [4]. However, such solutions are both unsafe and ad-hoc as we show in the next section. In order to disentangle task-level parallelism from data partitioning in a structured and high-level manner we propose the use of *Parallel Actor Monitors* (PAMs). In essence, a PAM is a *scheduler* that expresses a coordination strategy for the *parallel execution of messages within a single actor*. Since with a PAM, messages can be processed in parallel within the same actor, the programmer is no longer forced to partition her data in order to exploit parallelism.

There are four main contributions in applying the PAM model in order to solve the problems of data-task entanglement introduced by traditional actor systems.

1. **Efficiency.** A PAM makes it possible to take advantage of parallel computation for improved scalability. Benchmarks of our prototype implementations suggest speedups that are almost linear to the number of processors available (Section 8).
2. **Modularity.** A PAM is a modular, reusable scheduler that can be parameterized and plugged into an actor to introduce intra-actor parallelism without modification of the original code. This allows generic well-defined scheduling strategies to be implemented in libraries and reused as needed. By using a PAM programmers can separate the coordination concern from the rest of the code.
3. **Locality.** Binding a PAM to an actor *only* affects the parallel execution of messages inside that single actor. The scheduling strategy applied by one actor is completely transparent to other actors. This is because a PAM preserves the strong encapsulation boundaries between actors.
4. **Abstraction.** A PAM is expressed at the same level of abstraction as actors: the scheduling strategy realized by a PAM is defined in terms of a message queue, messages, and granting permissions to execute. A PAM programmer does *not* refer explicitly to threads and locks. It is the underlying PAM system that takes responsibility to hide the complexity of allocating and handling threads and locks for the programmer.

The next section gives an overview of the closest related work and shows why current approaches are not sufficient. Section 3 presents our parallel actor monitors in a general way, independent of a particular realization. Sections 5–7 given an overview of our implementation of PAM on top of AmbientTalk. We show canonical examples as well as more complex coordination strategies and a concrete use-case of PAM. Section 8 evaluates PAM against current actor systems, and provides an assessment of the implementation through a set of benchmarks. Section 9 concludes.

2. Related work

When ABCL [5] introduced state in the previously functional actor model one of the major design decisions for synchronization was the following:

"*One at a time: An object always performs a single sequence of actions in response to a single acceptable message. It does not execute more than one sequence of actions at the same time.*"

Since then it has been one of the main rules in stateful actor languages such as Erlang [6], Akka and Scala [7], Kilim [8], ProActive [9], E [10], Salsa [11] and AmbientTalk [12]. In all these languages execution of parallel messages within a single actor is disallowed by construction, *e.g.* every actor has only one thread of control and data cannot be shared between actors. As seen before, this leads to scalability issues when resources have to be shared among a number of actors. With this wild growth of actor languages it is not surprising that we are not the first ones to observe that the actor model is too strict [13]. There a number of alternatives used in actor-based systems to overcome this limitation.

First, actor languages that are built on top of a thread-based concurrency system can allow an "escape" to the implementing substrate. For instance, AmbientTalk [12] supports symbiosis with Java [14], which can be used to take