Parallel Actor Monitors

Christophe Scholliers [*]
Vrije Universiteit Brussel
Pleinlaan 2, Elsene, Belgium
cfscholl@vub.ac.be

Éric Tanter [†]
PLEIAD Laboratory
DCC / University of Chile
etanter@dcc.uchile.cl

Wolfgang De Meuter
Vrije Universiteit Brussel
Pleinlaan 2, Elsene, Belgium
wdemeuter@vub.ac.be

ABSTRACT

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 (PAM), 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 to introduce intra-actor parallelism in a local and abstract manner. PAM allow 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. Initial benchmarks confirm the expected scalability 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. Because there is no shared data and actors process messages sequentially, there can be no data races in an actor system. However, these strong guarantees come at a cost: efficiency.

Let us first illustrate the actor model by means of an example (Figure 1). The example consists of four actors, one of which is a dictionary actor. The three other actors are client of the dictionary: one actor does updates, while the others only consult the dictionary. The overall concurrency obtained in an actor system stems from the concurrent execution of multiple actors. Each actor has only *one* thread of execution which dequeues the pending messages of its inbox and processes them *one by one*. Message passing between actors is purely asynchronous. Responding to a message is done either by sending another asynchronous message, or, if supported by the actor system, by resolving a future [5].

The implementation of the dictionary example 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. However, the resulting application performs badly precisely because of the enforced serial execution of requests to the dictionary actor: there are no means to process read-only messages in parallel. In order to obtain scalability in such a scenario, some form of *intra-actor concurrency* is needed: that is, it may be worthwhile to consider relaxing the strong rules of the actor model.

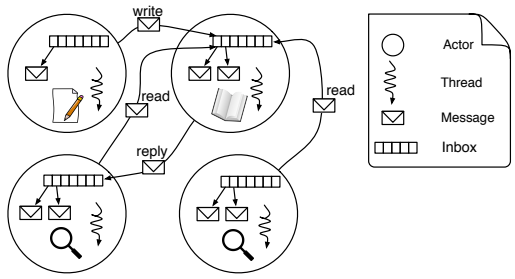

Figure 1: Actor model: Dictionary example

Although we are not the first ones to observe that the actor model is too strict, current alternatives are unsatisfying for being too ad-hoc and unsafe. Some actor languages, which are built on top of a thread-based concurrency system, allow some sort of "escaping" to the implementing substrate. For instance, AmbientTalk [6] supports symbiosis with Java [8], which can be used to take advantage of multithreading. However, such a backdoor reintroduces the traditional concurrency problems and forces the programmer to think in two different paradigms. Another approach is to introduce heterogeneity in the system by allowing actors to coexist with non-encapsulated, shared data structures. This is the case for instance of the ProActive actor-based middleware for Java [2]. A ProActive implementation of the dictionary example[1] would make the dictionary a "naked" data structure, that actors can access freely, concurrently. Avoiding data races is then done by suggesting client actors to request access through a coordinator actor (with methods such as `enterRead`, `exitWrite`, etc.), which implements a typical multiple-readers/single-writer strategy. A major issue with this approach is that the model does not *enforce* clients to use the coordinator actor: nothing prevents an actor to access the shared data structure directly, thereby compromising thread safety.

We propose the use of *parallel actor monitors* (PAM) in order to provide the programmer with intra-actor concurrency, in a structured and high-level manner. In essence, a

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

[†]Partially funded by FONDECYT projects 11060493 & 1090083.

[1]http://proactive.inria.fr/index.php?page=reader_writers

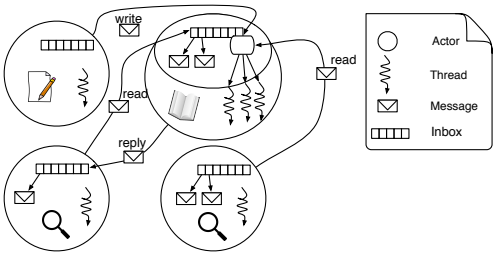

Figure 2: PAM model: Dictionary example

PAM is a *scheduler* that expresses a coordination strategy for the parallel execution of requests within a single actor. Figure 2 sketches the implementation of the dictionary example with a PAM. The dictionary actor has been changed by plugging in a generic, reusable PAM that implements the typical multiple-reader/single-writer coordination strategy.

There are four main advantages of using a PAM to overcome the bottleneck introduced by traditional actor systems.

1. **Efficiency.** A PAM makes it possible to take advantage of parallel computation for improved scalability. For the dictionary example, initial benchmarks of our prototype implementation in AmbientTalk suggest speedups that are almost linear to the number of processors available (Section 4).

2. **Modularity.** A PAM is a modular, reusable scheduler that can be parametrized and plugged in an actor without modification of the original code. This allows generic well-defined scheduling strategies to be implemented in libraries and reused as needed. There are no intrusive changes needed in order to introduce intra-actor concurrency in an existing application.

3. **Locality.** Binding a PAM to an actor *only* affects the concurrency of that single actor. Other actors are not affected in any way, other than by feeling they are interacting with a more responsive actor. This is because a PAM does not break the strong encapsulation boundaries between actors.

4. **Abstraction.** A PAM is expressed at the level of abstraction of actors: the scheduling strategy realized by a PAM is defined in terms of a message queue, messages, and granting permissions to execute. A PAM does *not* explicitly refer to threads and locks. It is the responsability of the PAM system to hide the complexity of allocating and handling several threads.

This paper first presents parallel object monitors in a general way (Section 2), independent of a particular realization. Section 3 then overviews how our implementation of PAM on top of AmbientTalk is used to express some canonical examples. Section 4 details the implementation of PAM in AmbientTalk, and provides an initial assessment of the implementation through a set of benchmarks. Section 5 concludes.

2. PARALLEL ACTOR MONITOR

A parallel actor monitor, PAM, is a low-cost, thread-less, *scheduler* controlling parallel execution of messages within an actor. Recall that an actor encapsulates a number of passive objects, accessed from other actors through asynchronous method calls. A PAM is therefore a *passive object* that controls the synchronization aspect of objects living within an actor, whose functional code is not tangled with the synchronization concern.

With this work, it is our objective to bring the benefits of the model of *parallel object monitors* (POM) [4] to actor programming. It is therefore unsurprising that the following operational description and guarantees of a parallel actor monitors closely ressemble that of POM. POM is formulated in a thread-based, synchronous world: PAM is its adaptation to an actor-based, purely asynchronous model.

2.1 Operational Description

A PAM is a monitor defining a *scheduling method* responsible for specifying how messages in an actor queue should be scheduled, possibly in parallel. A PAM also defines a *leaving method* that is executed by each thread once it has executed a message. These methods are essential to the proposed abstraction, making it possible to reuse functional code as it is, adding necessary synchronization constraints externally. An actor system that supports PAM allows the definition of schedulers and the binding of schedulers to actors.

Figure 3 illustrates the operation of a PAM in more detail. The figure displays an actor runtime system hosting a single actor and its associated PAM, as well as a thread pool, responsible for allocating threads to the processing of messages. Of course, several actors and their PAMs can live within the runtime system. When an asynchronous call is performed on an object hosted by an actor (1), it is put in the actor queue as a message object (2). Messages remain in the queue until the scheduling method (3) grants them permission to execute (4).

The scheduling method can trigger the execution of several messages. All selected messages are then free to execute *in parallel*, each one run by a thread allocated by the thread pool of the runtime system (5). Note that, if allowed by the scheduler, new messages can be dispatched before a first batch of selected messages has completed. Parallel execution of selected messages in PAM is in sharp contrast with the traditional actor model, where messages are executed sequentially by the unique thread of the actor [10].

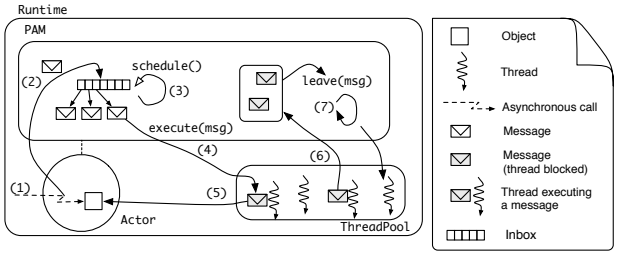

Figure 3: Operational sketch of PAM

Finally, when a thread has finished the execution of its associated message (6), it has to run the leaving method before leaving the PAM (7). To run the leaving method, a thread may have to wait for the scheduler monitor to be free (a PAM *is* a monitor): invocations of the scheduling and leaving methods are always safely executed, in *mutual exclusion*. Before leaving the monitor, a thread may have to execute the scheduling method again. The fact that a thread

spends some time scheduling requests for other threads (recall that the scheduler itself is a passive object) allows for a more efficient scheduling by avoiding unnecessary thread context switches.

2.2 Illustration

To further illustrate the working of a PAM, let us consider an actor whose PAM implements a simple join pattern coordination: when both a message `a` and a message `b` have been received by the actor, both can proceed in parallel. Otherwise, the messages are left in the queue. Figure 4 shows a *thread diagram* of the scenario. A thread diagram pictures the execution of threads according to time by picturing the call stack, and showing when a thread is active (plain line) or blocked waiting (dash line). The diagram shows two threads T1 and T2, initially idle, available for the activity of the considered actor. The state of the actor queue is initially empty.

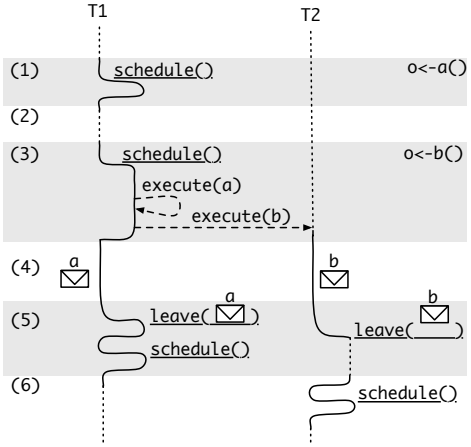

Figure 4: A simple join pattern coordinated by a PAM. ***(underlined method calls are performed in mutual exclusion within the scheduler)***

When a message `a` is received in the queue, T1 runs the scheduling method (1). Since there is no message `b` in the queue, nothing happens, T1 remains idle (2). When a message `b` is received, T1 runs again the `schedule` method (3). This time, both messages `a` and `b` are found in the queue, so they are both dispatched in parallel. First `a` is dispatched, then `b`. T1 finishes the execution of the schedule method, while T2 starts processing the `b` message. Then, both T1 and T2 are executing, in parallel, their respective messages (4). When T1 finishes processing `a`, it calls the `leave` and then the `schedule` methods, in mutual exclusion (5). (Note that the `schedule` method is only called at this point if there are pending messages in the queue.) Meanwhile, T2 also finishes processing its message but has to wait until the PAM is free in order to execute both methods itself (6).

2.3 Guarantees

An actor system supporting PAM should provide a number of guarantees to programmers. These guarantees are summarized below:

PAM Guarantees

1. *Any asynchronous messages send to an object encapsulated by a actor which is bound to a PAM is guaranteed to be scheduled by this PAM.*
2. *The schedule and leave method of a PAM are guaranteed to be executed in a thread-safe manner within the PAM, but in parallel with the messages being executed*
3. *The scheduling method is guaranteed to be executed if a message may be scheduled and guaranteed not to be executed when there are no pending messages*
4. *As soon as the scheduler method instructs the execution of a message it will be executed in parallel with the PAM or as soon as the schedule method is finished*
5. *When a message has finished processing it is guaranteed that it will call the leave method once, and the schedule method at most once*

2.4 Binding and Reuse

A scheduler is a passive object that defines a scheduling strategy. To be effective, it must be *bound* to an actor. After binding a scheduler to a specific actor, all the guarantees listed above hold. In order to enhance reuse of schedulers, it is important for a scheduler to be as independent as possible from the actors it can be bound to. More precisely, the scheduler has to abstract over the actual message *names* to coordinate. For example, a reusable reader-writer scheduler should work not only with dictionaries whose methods have particular names, and should also be used for actors encapsulating other data structures.

In order to support the definition of abstract, reusable schedulers, PAM adopts the notion of *message categories*, also found in POM. A scheduler is defined in terms of categories, rather than actual message names. This makes it possible to define for instance a reader-writer scheduler by just using a reader and a writer message category. Actual message names are then associated to specific categories at *scheduler binding time*. In the case of a dictionary, a query message would belong to the reader category, while a put message would belong to the writer category. The use of categories is further illustrated in the following section.

3. CANONICAL EXAMPLES

In this section, we first introduce our implementation of PAM in AmbientTalk, from the point of a view of a programmer. To illustrate its use, we then give the implementation of two classical schedulers and how to bind them to actors in a program. The first example is purely didactic, since it re-introduces sequentiality within an actor: a standard mutual exclusion scheduler. The second example is the classical multiple-reader/single-writer scheduler.

Although we have implemented more canonical examples with PAM (*e.g.* dining philosophers) these two problems clearly show the typical use of PAM. Note that because PAM is a reincarnation of POM within actors, it also allows the implementation of more advanced coordination mechanisms, like guards [7] and chords [3, 4].

3.1 PAM in AmbientTalk

executeLetter(Letter)	executeOldestLetter()
executeAll()	is:taggedWithCategory:
executeYougest()	Category(type)
excuteOlderThan(*,*)	listIncomingLetters
executeAllOlderThan(*,*)	annotateMethod:with:on:
executeYoungerThan(*)	bindScheduler:on:
executeAllYougerThan(*)	

Table 1: PAM API

AmbientTalk [6] is an actor language with an event loop concurrency model adopted from E [9]. In this model, an actor encapsulates a number of passive objects, which can be referenced by objects living in other actors. Communication between objects can only be conducted by sending asynchronous messages. A PAM takes care of the scheduling of these messages. Note that AmbientTalk is a *prototype-based* language, that is, objects are created ex-nihilo or by cloning existing objects, rather than by instantiating classes.

A PAM in AmbientTalk is created by using the *scheduler:* constructor function. Every scheduler in AmbientTalk has to implement at least a *schedule* and a *leave* method. Inside a PAM the programmer can examine the inbox of the actor, which contains *letters*. A letter contains a *message* and a *receiver*. The receiver is the (passive) object that lives inside the actor to which the message was sent. The API of PAM (Table 1) supports a number of methods to execute letters, *i.e.* to trigger the processing of the message contained in the letter. It is possible to execution a single letter or a group of letters. `executeLetter`, `executeOldestLetter` and `executeYoungestLetter` return a boolean indicating wether a letter was effectively triggered for execution or not. The other execution methods all return an integer indicating the number of letters triggered for execution. The PAM API also includes methods to declare and introspect message categories, as well as to bind a given scheduler to an actor. The rest of this section illustrates their use.

3.2 Mutual Exclusion

Listing 1 shows a simple mutual exclusion scheduler, for the sake of illustration. `mutex` is a constructor function that returns a new scheduler when applied. The scheduler keeps track of whether a message is currently being processed through an instance variable `working`, initialized to `false`. The `schedule` method only triggers a message execution if it is not working already. If so, it executes the oldest letter in the inbox, if any. Its state is updated to the result of invoking `executeOldestLetter`, which indicates if a message was actually triggered or not. The `leave` method changes the state of the scheduler accordingly after a message has been processed. Finally the scheduler is instantiated and bound to the dictionary actor by the `bindScheduler` method. Note that the definition of the scheduler is simple and does not deal with low-level synchronization and notification details.

```
def mutex() { scheduler: {
 def working := false;
 def schedule() {
  if: !(working) then: {
   working := super.executeOldestLetter();
  };
 };
 def leave(letter) { working := false; };
 };
};

bindScheduler: mutex() on: dictionaryActor();
```

Listing 1: Mutual exclusion PAM

```
def RWSched() { scheduler: {
 def R := Category();
 def W := Category();
 def writing := false;
 def readers := 0;
 def schedule() {
  if: !(writing) then: {
   def executing := super.executeAllOlderThan(R,W);
   readers := readers + executing;
   if: (readers == 0) then: {
       writing := super.executeOldest(W);
   };
  };
 };
  def leave(letter) {
   dispatchCategory: letter as:
   [[R, { readers := readers - 1 }],
    [W, { writing := false }]];
 };
};};
```

Listing 2: Reader/Writer example

3.3 Parallel dispatch

The reader-writer scheduler for coordinating the parallel access to a shared data structure is shown in Listing 2. Like before, `RWSched` is a constructor function that, when applied, returns a fresh scheduler. The scheduler defines two method categories readers (`R`) and writers (`W`). In order to keep track of how many readers and writers are executing, the scheduler maintains two variables `writing` (boolean) and `readers` (integer). When the scheduler is executing a write letter, no further message can be processed. In case the scheduler is not executing a write letter, the scheduling method triggers the parallel execution of all the read letters that are older than the oldest write letter, if any, using `executeAllOlderThan(R,W)`. This call returns the number of dispatched readers, used to update the `readers` state. If there were no reader letters to process (older than the oldest writer), the scheduler dispatches the oldest writer using `scheduleOldest(W)`. This method returns true if the processing of a letter was actually dispatch, false otherwise. Note that this scheduler uses a typical strategy, and could easily be modified to give priority to writers, for instance. Finally, the `leave` method updates its state according to which message has finished executing, by either decreasing the number of readers, or turning the `writer` flag to false. Note the use of a convenience syntax for dispatching on message categories.

In order to use this scheduler with a dictionary actor, one should just annotate the methods of the dictionary with the appropriate categories, and then instantiate and bind the scheduler to the dictionary (Listing 3).

```
annotateMethod: 'get with: RWSched.R on: dictionary;
annotateMethod: 'put with: RWSched.W on: dictionary;
bindScheduler: RWSched() on: dictionaryActor();
```

Listing 3: Instantiating and binding

4. MICRO-BENCHMARKS

We now report on micro-benchmarks of our PAM implementation for AmbientTalk. The aim is to measure both the overhead of PAM by contrasting plain AmbientTalk with PAM using a mutual exclusion scheduler (Listing 1), and to measure the speedup obtained by using the reader-writer PAM scheduler (Listing 2). The results depicted on Figure 5 were obtained on an Intel Core 2 Duo with a processor speed of 1.8 GHz running Mac OS X (10.5.8). We measured the processing time of reading one hundred items from a dictionary actor of varying size. For each measurement, we take the average of 30 tests, discarding the extremes. These results show the low overhead of PAM ($< 6\%$) and the expected speedup (1.9x for the 32K and 150K dictionary), taking almost full advantage of the 2 cores. In addition, we have performed preliminary tests with a two-dualcore machine running Linux. For a dictionary size of 32K we obtain a 3x speedup, although it seems the overhead of PAM plays a bigger role. To date, we have not been able to obtain isolated access to the quadcore machine, so we have not been able to analyze further these preliminary results yet. This should be addressed in the very near future.

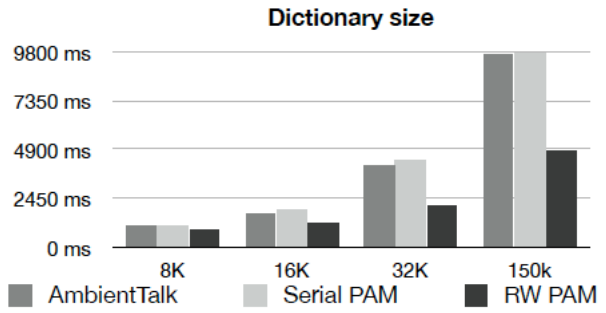

Figure 5: PAM benchmarks.

5. CONCLUSION

In order to address the strict restriction of sequentiality inside actors, we have proposed the model of Parallel Actor Monitors. Using PAM, there can be intra-actor concurrency, thereby leading to better scalability by making it possible for a single actor to take advantage of real concurrency offered by the underlying hardware. PAM offers a particularly attractive alternative to introduce concurrency inside actors, because it does so in a modular, local, and abstract manner: modular, because a PAM is a reusable scheduler; local, because only the internal activity of an actor is affected by using a PAM; abstract, because the scheduler is expressed in terms of messages, queues and granting permission to execute. Although more in-depth measurements are in order, in particular with quadcore machines (or more), initial benchmarks confirm the expected speedup.

6. REFERENCES

[1] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[3] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, Sept. 2004.

[4] D. Caromel, L. Mateu, G. Pothier, and É. Tanter. Parallel object monitors. *Concurrency and Computation—Practice and Experience*, 20(12):1387–1417, Aug. 2008.

[5] A. Chatterjee. Futures: a mechanism for concurrency among objects. In *Proc. of the 1989 ACM/IEEE conf. on Supercomputing*, pages 562–567. ACM Press, 1989.

[6] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In D. Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. GOTOP Information Inc., 1996.

[9] M. Stiegler. The E language in a walnut. *www.skyhunter.com/marcs/ewalnut.html*, 2004.

[10] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. MIT Press, 1990.