J Supercomput (2018) 74:1485–1496
https://doi.org/10.1007/s11227-017-2233-1

Actor model of Anemone functional language

Paweł Batko[1] · Marcin Kuta[2]

Published online: 8 January 2018

Abstract This paper describes actor system of a new functional language called *Anemone* and compares it with actor systems of Scala and Erlang. Implementation details of the actor system are described. Performance evaluation is provided on sequential and concurrent programs.

Keywords Compiler construction · Concurrent programming · Actors · Threads · Message passing · Garbage collector

1 Introduction

Creating a new language is a complex task, requiring a lot of time and human resources. Creating a function language is even more difficult, as it contains many high-level concepts, e.g., higher-order functions or closures. This work presents design and implementation of *Anemone*—fully useful, secure, functional language with user-friendly syntax, which can be enriched with new components and optimized in future.

Anemone [5] supports concurrent programming model based on actors communicating via messages. Polymorphic type system of *Anemone* supports error detection at compile time and encourages code reuse. Complete type inference disposes a programmer of defining explicit type signatures. *Anemone* functions are first class citizens [1]

✉ Marcin Kuta
mkuta@agh.edu.pl

Paweł Batko
pbatko@virtuslab.com

1 VirtusLab, Smoleńsk 21/15, 31-108 Kraków, Poland

2 Department of Computer Science, AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland

of the language, and memory is automatically managed with garbage collector module. Mechanism of external functions gives access to functions and libraries written in C. *Anemone* compiler is based on the LLVM infrastructure, which generates high quality code for many computer architectures [8]. Created language draws inspiration from many mechanisms present in various programming languages, including Scala, Erlang, Haskell, and ML. Actor system integrated directly into language, safe programming style encouraged by immutable variables, coherent type system with subtyping, pattern matching mechanism, and compiler retargetability make Anemone attractive compared to other languages.

The aims of this work were the following:

1. Implementation of a functional language compiler, analysis of such languages, and ways of their implementation.
2. Convenient model of concurrent processing based on actors mechanism and message passing.
3. Performance comparison of the implemented model with existing mechanisms in other languages.
4. Equipping *Anemone* with polymorphic type system, full type inference, and algebraic data types in order to provide expressiveness, static guarantees of correctness, at the same time preserving code clarity and conciseness.

Figure 1 outlines the most important phases performed during compilation of a source program written in *Anemone*.

Figure 2 shows main modules of *Anemone* runtime system and dependencies between them.

2 Related work

Threads are a low-level abstraction of concurrent programming, but using them is difficult because they share memory. Secure, concurrent access to shared memory requires synchronization with a critical section along with structures like semaphores [6], mutexes or critical sections. A programmer has to pay attention to avoid race conditions, starvation, dead locks or live locks.

Actor model [2,7] offers several advantages over thread model. Actors intuitively model real world, and writing reliable applications using actors is simple. Asynchronous communication between actors leads to better utilization of CPU time, as idle cycles are not wasted while waiting for an answer. Separation between actors makes programs easier to understand. The whole actor system can be understood by analyzing each actor behavior in isolation. Actors are well suited for problems requiring high performance, responsiveness or multiscaling [9], as they can delegate tasks to workers. Actors are also lightweight, and thousands of instances can be created, which is impossible for system threads. With actor model, synchronization mechanisms present in thread model are no more needed. As actors do not share a global state, critical sections are neither needed. Taking into account above advantages, actor model has been adopted as concurrency model in *Anemone*.

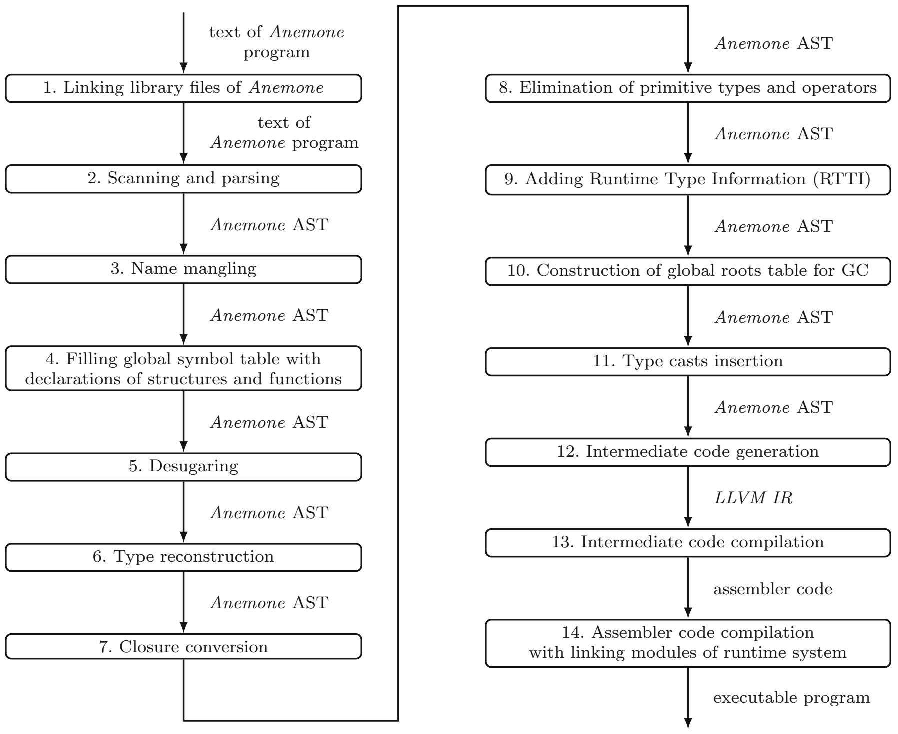
text of *Anemone* program
1. Linking library files of *Anemone*
text of *Anemone* program
2. Scanning and parsing
Anemone AST
3. Name mangling
Anemone AST
4. Filling global symbol table with declarations of structures and functions
Anemone AST
5. Desugaring
Anemone AST
6. Type reconstruction
Anemone AST
7. Closure conversion
Anemone AST
8. Elimination of primitive types and operators
Anemone AST
9. Adding Runtime Type Information (RTTI)
Anemone AST
10. Construction of global roots table for GC
Anemone AST
11. Type casts insertion
Anemone AST
12. Intermediate code generation
LLVM IR
13. Intermediate code compilation
assembler code
14. Assembler code compilation with linking modules of runtime system
executable program

Fig. 1 Compilation phases of *Anemone*

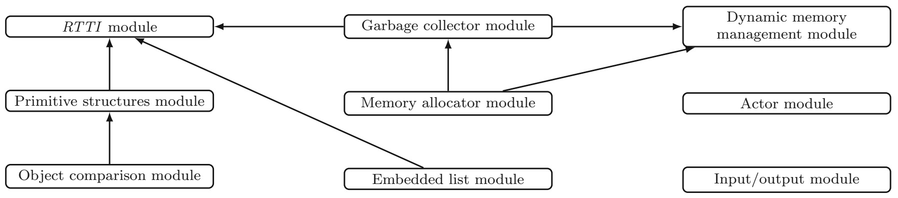

Fig. 2 Main modules of *Anemone* runtime system and dependencies between them

2.1 Actors in Erlang and Scala

Erlang [3,4] implements actors at a language level and calls them processes. Erlang processes are lightweight, as they are implemented at the level of Erlang virtual machine and do not involve threads or processes of a operating system. Each actor is assigned its own dynamic memory, and garbage collection is performed for each actor independently. Sent messages are copied between heaps, which prevents sharing them by two or more actors. Actor scheduling is a duty of the Erlang virtual machine, and no synchronization mechanism, e.g., semaphores, is needed. Erlang actor system ensures scalability (one actor needs only 300 bytes of memory) and high reliability of written programs [4].

```
ping(N, Pong) ->
    Pong ! {self(), ping},
end




pong() ->
    receive
        {From, ping} ->
            From ! pong,
            pong();
    end.
```
(a) Erlang implementation

```
class Ping extends Actor {
  val state: Int = 1
  def receive = {
    case Ping =>
        pong ! PingMsg
  }
}

class Pong extends Actor {
  def receive = {
    case PingMsg =>
        sender ! PongMsg
  }
}
```
(b) Scala implementation

Fig. 3 Two actors *ping* and *pong* implemented in Erlang and Scala. Erlang actors are implemented as *ping* and *pong* functions, while Scala actors are represented with instances of *Ping* and *Pong* classes. In both cases, messages are sent with ! operator and received within `receive` block. The code does not comprise start-up of the actor system

Figure 3a presents two actors, *ping* and *pong*, defined as functions in Erlang. It is worth noting code conciseness, built-in message sending operator, a block for receiving messages, and message identification with pattern matching.

Scala provides actors implementation in the Akka library [10]. Message passing is done in shared memory, if actors run within one Java virtual machine. If actors perform on different virtual machines, messages are serialized before sending them. Garbage collection applies to all the actors from a given JVM instance. Actors are scheduled by the Akka library. Akka does not impose immutable messages and avoiding global state although strongly recommends them.

Figure 3b presents actor definitions with Akka. In contrast to Erlang, Akka actors are defined as classes. Pattern matching, message sending operator, and a block for receiving messages are exploited similarly as in Erlang. Actor state is represented as a field of an actor instance. Continuity of actor work is ensured by Akka, so an actor does not have to call itself, as in Erlang.

Features of actor models of Erlang, Scala and *Anemone* are summarized in Table 1.

3 Actor model of Anemone

Anemone does not provide a programmer low-level model of threads, which, although very expressive, does not fit needs of a high-level, secure functional language.

Actor model is more restrictive than thread model, as actors communicate only through message passing. It makes manual synchronization redundant and allows to write programs more easily, due to imposed style of communication and message identification. Continuity of actor work is provided in *Anemone* by internal implementation, which, similarly to Akka, calls actor function for each new message. Similarly to Erlang, *Anemone* models actors as functions, which can accept an initial state and pass it through. Message passing model in *Anemone* is similar to Akka, as they both use shared memory to implement it.

Table 1 Comparison of concurrency models of Erlang, Scala and Anemone

	Erlang	Scala	Anemone
Actors support	In language	In Akka library	In language
Actors specification	As functions	As classes	As functions
Communication	Asynchronous	Both synchronous and asynchronous	Asynchronous
Message passing	No shared state	Shared memory	Shared memory
Pattern matching	Yes	Yes	Yes
Dynamic creation of actors	Yes	Yes	Yes
Garbage collection	On single threads	Global VM lockup	Polling

Actor model of *Anemone* characterizes with:

– asynchronous communication
– possibility of creating many actors in one thread
– possibility of dynamic creation of new actors
– complete integration with the garbage collector
– modeling actors as functions, similarly to Erlang
– actor control similar to Akka
– message handling through pattern matching.

Implementation of many actors within one system thread was especially challenging, as it required their proper scheduling. Construction of the garbage collector which correctly and effectively cooperates with multithreaded actors architecture was equally challenging.

Figure 4 presents a complete program implementing two actors which communicate with each other via messages. Function `createActorSystem(2)` creates an actor system on the basis of two system threads, and next, actors *ping* and *pong* are created. Function `sendFromOutside` starts activities of actors, i.e., sending messages. Actors exploit pattern matching to identify received messages, function `sendMsg` to send messages, and maintain the state identifying the address of a second actor.

4 Implementation of actors system

Anemone models concurrent computing with the actor module, which provides abstraction for convenient work on multicore architectures. Figure 5 presents detailed architecture of actor module. Actor module consists of the following submodules:

– `Mailbox module` defines how received messages are stored and is responsible for actors adding and removing.
– `Actor management module` defines architecture of a single actor.
– `Threads module` is responsible for management of OS threads
– `Dispatcher module` is responsible for running actors on system threads

```
fun ping(state, msg) {
    var otherActorId = state in {
        match msg {
            | s :: String => {
                printStr(s)
                sendMsg(otherActorId, "fromPing")
                nap(1)
                state
            }
            | otherActorId :: ActorId => {
                sendMsg(otherActorId, "fromPing - first")
                otherActorId
}   }   }   }

fun pong(state, msg) {
    var otherActorId = state in {
        match msg {
            | s :: String => {
                printStr(s)
                sendMsg(otherActorId, "fromPong")
                nap(1)
                state
}   }   }   }

fun main_fun() {
    createActorSystem(2)
    var pingActorRef = createActor(ping, 0),
    pongActorRef = createActor(pong, pingActorRef) in {
        sendFromOutside(pingActorRef, pongActorRef)
}   }
```

Fig. 4 Creation and starting actor system

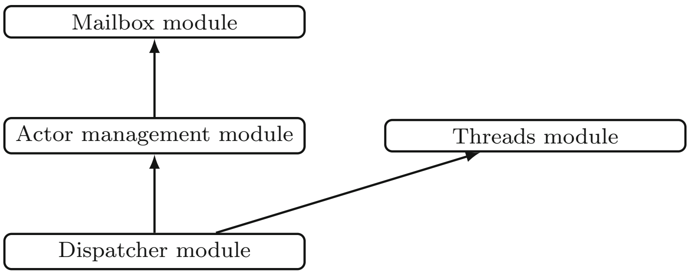

Fig. 5 Main submodules of module of actors and message passing

In *Anemone* using actors is possible due to actor module. Actor library is an interface to this module from *Anemone*. Figure 6 presents functions to handle actors. The actors library provides the following functions to manipulate actors:

- `createActorSystem` creates an actor system running on a given number of system threads.

Fig. 6 Functions of actors library

```
type:: (double) -> unit
fun createActorSystem(n)

type:: (('s, 'm) -> 's, 's) -> ActorId
fun createActor(f, a)

type:: (ActorId, 'a) -> unit
fun sendMsg(to, msg)

type:: (('s, 'm) -> 's) -> unit
fun become(f)
```

Fig. 7 Structure of actor module describing an actor thread of *Anemone*

```
struct __athread_t {
        int64_t thread_id;
        int64_t actors_array_capacity;
        int64_t actors_array_occupied;
        int64_t current_actor_idx;
        actor_t** actors_array;
};
typedef struct __athread_t athread_t;
```

– `createActor` creates a new actor in the actor system. Two parameters define actor behavior and an initial state of actor. Function returns the id of a created actor.
– `killActor` returns an object, which sent to an actor or passed as its new state terminates its action.
– `sendMsg` sends a message, specified in the second parameter, to an actor defined by the first parameter. Function can be called only by an actor.
– `become` can be called only by an actor. The function takes as its argument a function determining actor new behavior.

4.1 Creating actor system

Creating actor system is equivalent to creating a number of threads and initialization of their data structures. The actor system of *Anemone* distinguishes two kinds of threads: system threads and actor threads. Each actor thread contains a system thread and a number of actors.

Function `createActorSystem` creates a new actor system with a given number of threads. In particular, it creates a table of `athread_t` structures, describing actor threads (Fig. 7). Each `athread_t` structure corresponds to one system thread and many actors (`actors_array`).

System threads are created and managed with the help of POSIX *pthreads* library. Function `athread_main_fun` creates system threads and defines their behavior. Function `athread_main_fun` works in the following steps:

– initializes of structures describing an actor thread.
– waits for the first actor in this thread.
– takes the next actor assigned to this actor thread.

Fig. 8 Structure of actor module describing a single actor of *Anemone*

```
struct __actor_t {
        closure_t *user_cls;
        void *state;
        uint64_t actor_id;
        mailbox_t *mailbox;
};
typedef struct __actor_t actor_t;
```

```
fun pong(state, msg) {
    printStr("Pong: " ^ msg)
    sendMsg(pingActorId, "PONG" ^ msg)
    state
}
```

Fig. 9 An example of a function defining actor behavior. Function pong takes as its arguments a current actor state and a new message. It sends a message of type *string* to an actor identified by *pingActorID*. The function returns a new state (in this case unchanged) as a result of its call

– starts handling at most max_msg_handled_per_actor messages in the mailbox of the actor. Variable max_msg_handled_per_actor is tunable.
– removes an actor from the system in the following cases:
  – an actor received a message being a result of killActor call
  – an actor returned a new state equal to the return value of killActor call.
– serves next actor, if it is present.
– terminates, if no actors are present in this actor thread.

4.2 Implementation of an actor and message passing

Actors and message passing are implemented in the actor module in C. Figure 8 presents the structure of the actor module used by the actor management module. It consists from the following fields:

– user_cls—a closure describing actor behavior,
– state—actor state, it is passed as a parameter to a closure describing actor behavior,
– actor_id—actor id in the actor management module,
– mailbox—a pointer to the mailbox of an actor.

An actor mailbox stores messages sent to a given actor. Each message has a sender, a receiver, and the content. Messages are processed under the *FIFO* regime. Sending a message does not incur copying the whole message but only a lightweight copying of a pointer to the message, as message passing is done in shared memory.

At any time, actor thread can run at most one actor. Figure 9 presents a simple actor definition. Function sendMsg, responsible for sending a message to another actor, does not specify explicitly the sender of a message. The sender (currently running actor) is identified due to the local context of the actor thread, implemented by the actor module.

Figure 10 presents functions of the actor management module, responsible for associating actor threads with contextual data. Function threads_setspecific

```
void* threads_getspecific() {
    return pthread_getspecific(THREAD_KEY);
}

void threads_setspecific(void* data) {
    pthread_setspecific(THREAD_KEY, data);
}
```

Fig. 10 Getting and setting local state of a given actor thread. Function `threads_getspecific` gets local state of a actor thread, while `threads_setspecific` sets it. Key `THREAD_KEY` defines local data of a given actor thread. Both functions are implemented in *pthreads*

is used in the main loop of `athread_main_fun` to set thread context to current actor. When function responsible for sending messages is called, it can access local context of a thread to get the id of the actor. The advantage of such a solution is a support for many actors in one system thread through changing value of thread context and conciseness of function `sendMesg`.

In addition to local context setters and getters, implementation of actor module uses synchronization primitives—mutexes and conditional variables of *pthreads*. Correct and effective realization of the actor system on the basis of above mechanisms is a duty of the runtime system of *Anemone*.

4.3 Scheduling many actors in one system thread

The actor module can create a huge number of actors, significantly exceeding the number of running threads of the operating system, as an *Anemone* actor is defined by a lightweight data structure (several hundred bytes). An actor thread associates a system thread with many actors. Actor scheduling in *Anemone* is a duty of the actor module and is based on the number of received messages. Each running actor may receive no more than `max_msg_handled_peractor` messages. If an actor receives all the messages from its mailbox (but not exceeding the `max_msg_handled_per_actor` threshold), the next actor belonging to the same actor thread will be scheduled. Above scheduling algorithm promotes implementation of actors as quickly performing functions. Time-consuming tasks should be delegated by an actor to its child workers. The drawback of this solution is that some system threads will be blocked by these heavy computations. A separate pool of threads with asynchronous communication could be a remedy to this problem and future extension of *Anemone*.

4.4 Pattern matching

With pattern matching supported by *Anemone*, defining actors and their behavior is easier.

Figure 11 presents an actor defined with pattern matching. Message `msg`, received by an actor, is matched to the first pattern, `Bar(b)`, where `Bar` denotes a data type defined in *Anemone*, and `b` denotes a field of this data type. If the first match fails, pattern `f :: Foo` will be checked. The second pattern will be matched if `msg` is

Fig. 11 Usage of pattern matching in definition of *Anemone* actor

```
fun anActor(state, msg) {
    match msg {
        | Bar(b)    => { ... }
        | f :: Foo => { ... }
    }
}
```

Table 2 Execution time of the program computing 20th element of the Fibonacci sequence

Language	Execution time (s)	Ratio
Java (1.7.0_60)	0.45	3.09
Scala (2.10.4)	0.75	1.87
Anemone	1.41	1.00

of type `Foo`. Identifier `f` introduces variable `f`, which refers to matched object, `msg`, and has type `Foo`.

5 Experiments

To assess quality of *Anemone* implementation and its runtime system, performance tests have been conducted and execution time of programs written in *Anemone*, Scala and Java have been compared. For each language, the arithmetic mean of ten measurements with *time* command was reported. The experiments were performed under dual-core Intel Core i3-2310M CPU with 2.10 GHz clock and Linux Ubuntu 13.10. Heap size for Scala and Java was set to default values. For *Anemone* heap size was set to 100 KiB and the threshold triggering a collection to 0.8.

The first experiment, which assessed quality of generated code and efficiency of the runtime system, compared time performance of a sequential program (computation of 20th element of the Fibonacci sequence) written in Java, Scala and *Anemone*. Results in Table 2 show that implementation in Scala was two times faster, while implementation in Java three times faster than *Anemone* implementation. It can be partly attributed to memory organization in *Anemone*. Language performance is significantly influenced by efficiency of the memory allocation module. While *Anemone* is a new language, memory allocation algorithm in JVM used by Scala and Java has been fine-tuned for many years. On the other hand, *Anemone* is a language compiled to machine code and does not bear overhead of starting its virtual machine.

The second experiment assessed efficiency of the *Anemone* actor system. Implementations in *Anemone* and Akka of two actors communicating with each other were compared. Actors sent in total 10,000 messages.

Table 3 presents results of this performance test. Implementation in Akka turned out to be six times faster than implementation in *Anemone*. Efficiency of the memory allocation module of *Anemone* could have significant impact on results.

Table 3 Execution time of the program creating simple actor system

Language	Execution time (s)	Ratio
Scala (2.10.4) + *Akka* (2.2-M3)	1.86	6.12
Anemone	11.41	1.00

Table 4 Statistics of memory allocator and collector for the program computing 22th term of the Fibonacci sequence. Part 1

Heap size (KiB)	Allocated (KiB)	Collected (KiB)	Application time (s)	Collector time (s)
100	17,781	17,714	3.54	0.08
1000	17,781	17,498	34.34	0.05

Table presents total allocated and collected memory, application time (collector time not included) and collector time for the program computing 22th term of the Fibonacci sequence for different heap sizes

Table 5 Statistics of memory allocator and collector for the program computing 22th term of the Fibonacci sequence. Part 2

Heap size (KiB)	Allocation speed (KiB/s)	Collection speed (KiB/s)	Number of collections
100	5019	216,454	241
1000	517	344,489	22

Table presents speed of memory allocation and collection as well as the number of performed collections for the program computing 22th term of the Fibonacci sequence

5.1 Efficiency tests of memory allocator and collector

The third experiment examined efficiency of the dynamic memory management module of *Anemone*.

Tables 4 and 5 present statistics about garbage collector for heap size of 100 and 1000 kB. During the program run, there was allocated and collected approximately 17 MB memory. Efficiency of the garbage collector module was determined by dividing total collected memory by total time of application execution. Measured efficiency was 200 and 300 MB per sec, respectively, which seems a very good results. It is worth noting, that increasing heap size reduced the number of memory collections and increased efficiency by 50%.

Table 5 presents also memory allocation speed, determined by dividing total allocated memory by total time of application execution. Allocation speed was much lower than collection speed, which can be easily explained by the fact that beside allocation, proper computations were performed by the application.

However, implemented memory allocator does not scale well with the increase in the heap size. With the tenfold increase of heap size, total time of application increased ten times, and memory allocation speed decreased ten times (Table 5). The performance drop can be caused by implementation of the search of free chunks of memory through linear heap scan.

Optimization of this aspect of the dynamic memory management module could significantly improve the results.

6 Conclusions

Model and implementation of actor system of a new function language, *Anemone*, have been described. *Anemone* was developed within a limited period of time. Proposed language was few times slower than Scala or Java, which is a very good result, taking into account that latter languages have been already developed and optimized for a long period of time by large teams of experts.

Acknowledgements This research was financed by AGH University of Science and Technology Statutory Fund.

References

1. Abelson H, Sussman GJ, Sussman J (1996) Structure and interpretation of computer programs, 2nd edn. MIT Press/McGraw-Hill, Cambridge
2. Agha GA (1990) ACTORS—a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence. MIT Press, Cambridge
3. Armstrong J (2007) Erlang—software for a concurrent world. In: Ernst E (ed) ECOOP 2007—Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30–August 3, 2007, Proceedings, vol. 4609, p 1
4. Armstrong J (2010) Erlang. Commun ACM 53(9):68–75
5. Batko P (2014) A compiler for functional language with support for message passing. Master's thesis, Department of Computer Science, AGH University of Science and Technology, Krakow **(in Polish)**
6. Dijkstra EW (1965) Cooperating sequential processes, technical report EWD-123. Technical report Eindhoven University of Technology
7. Hewitt C, Bishop P, Steiger R (1973) A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73, pp 235–245
8. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), pp 75–88
9. Rycerz K, Bubak M (2016) Using Akka actors for managing iterations in multiscale applications. In: Wyrzykowski R, Deelman E, Dongarra J, Karczewski K, Kitowski J, Wiatr K (eds) Proceedings of the 11th International Conference on Parallel Processing and Applied Mathematics, PPAM 2015, pp 332–341. https://doi.org/10.1007/978-3-319-32149-3
10. Wyatt D (2013) Akka concurrency. Artima Incorporation, New York