CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **00**:1–14
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A Design Pattern for Optimizations in Data Intensive Applications using ABS and JAVA 8

V. Serbanescu[1]*, K. Azadbakht[1], F. de Boer[1], C. Nagarajagowda[1], B. Nobakht[1]

[1] *Centrum Wiskunde & Informatica, Science Park 123 1098 XG Amsterdam Netherlands*

SUMMARY

Cloud environments have become a standard method for enterprises to offer their applications by means of web-services, data management systems or simply renting out computing resources. In our previous work we presented how we can use a modeling language together with the new features of JAVA 8 to overcome certain drawbacks of data structures and synchronization mechanisms in parallel applications. We extend this solution into a design pattern that allows application-specific optimizations in a distributed setting. We validate this integration using our previous case study of the Prime Sieve of Eratosthenes and illustrate the performance improvements in terms of speed-up and memory consumption. Copyright © 2014 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Cloud computing, programming models, distributed applications, formal methods, application-level optimization, design patterns, memory management

1. INTRODUCTION

Cloud computing has proven to be an essential concept in developing large scale applications for several domains of research and industry. Examples of these domains include designing real-time applications for embedded systems [1], reputation-based web-services[2], data aggregation in wireless networks [3, 4], high performance computing [5] [6] and delivering cloud services at several levels through SaaS or PaaS [7, 8, 9]. However, distributed characteristics such as scalability, performance and replication over a large number of resources can prove to be a very complex effort [10]. When it comes to designing applications, the Java language [11] is one of the mainstream object oriented programming languages in terms of usability and ease of learning. Java has both a concurrent model[12] and remote method invocation standard[13] that both require following a certain pattern in programming that that give the user little flexibility in terms of designing a loosely-coupled parallel and distributed application.

In our previous work [14] which was published in [15] we presented a solution to develop distributed applications in JAVA using several features introduced in JAVA 8 and integrated it into an API offering the user an actor-based model for programming. Our focus was to use abstract actors that can be extended to run services of distributed applications[16]. Our API had very limited practical support in distributed applications and was mainly used for designing parallel applications

*Correspondence to: Centrum Wiskunde & Informatica, Science Park 123 1098 XG Amsterdam Netherlands. E-mail: vlad.serbanescu@cwi.nl

†Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (`http://www.envisage-project.eu`). Partly funded by the EU project FP7-612985 UPSCALE: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (`http://www.upscale-project.eu`).

without an actual degree of distribution. We also had several issues with a large amount of redundant communication that was done between actors, as our research was mainly focused on the ease of use of the API and observing drawbacks of certain implementations directly at application level. We extended the current API with the possibility of actors sending remote messages through lambda expressions without the complexity of writing code for sockets or remote method invocations. The main scientific contribution of this paper is an extended API that provides a design pattern which significantly reduce memory consumption and maximize throughput in parallel and distributed applications data-intensive applications.

Our previous contribution was mainly focused on reducing code size and improving readability compared to other programming models while maintaining superior performance in a highly parallelizable application, but restricted to on machine. We presented this a solution named the ABS-API library[17]. The objectives in this extended work now cover design patterns, the ability to analyze code, integrate optimizations into the library and extend to the deployment of applications in a distributed environment. First of all we introduce the **ABS-API-Remote** which simplifies the communication between actors, allowing them to run and communicate on different machines and validate this solution by running the case study on multiple machines on the SurfSara[18] cluster. We present a new method to identify actors based on the machine's location, send remote messages via lambda expressions and optimize communication between actors that have access to a shared memory. Furthermore we generalize API with a new notion for actors called a **Concurrent Actor Group** which allows them to operate on this shared memory based on their containing VM and allows the user to design optimizations for his application based on this notion. The rest of our paper is structured as follows: Section 2 presents an overview of the ABS language and the reasons for developing the ABS-API. Section 3 discusses our previous work on the API followed by the extension added in our current research work. Section 4 presents a comparison between the previous implementation of the case-study and the implementation using our current extension. In section 5 we formulate the design pattern in our ABS-API-Remote. We follow with discussion on related work for actor-based models in section 6 and draw our conclusions and propose future work in section 7.

2. THE ABS LANGUAGE

The backbone for our research is the Abstract Behavioral Specification language (ABS) introduced in [19], as part of the HATS European Project [20] which is continued by the ENVISAGE European Project[21]. ABS offers programmers a syntax similar to JAVA with classes and interfaces for encapsulation, but includes several extra features such as a functional programming model, the possibility to issue asynchronous method calls and control synchronization with futures as well as cooperative scheduling of method invocations inside concurrent (active) objects. Instantiating an object in ABS creates an Actor with its member fields and methods which define its behavior and interactions with other actors. Actors run in parallel and communicate via messages generated through asynchronous method calls which are stored inside the called actor's queue. In ABS synchronization is handled by *"await"* and *"get"* statement which can be applied to futures or boolean conditions and force a method to suspend until the variable is verified. An actor processes messages from its queue in a FIFO order and calls its scheduler whenever a running method suspends. This feature combination results in a concurrent object-oriented model which is inherently compositional. An example of our case study, the Prime Sieve of Eratosthenes, written in ABS is presented in Listing 1.

The program first creates the interface *ISieve* which contains the method *sieve()* for sieving all numbers present in a partition that are divisible by the argument *n* by setting in the **current_Map** data structure the value of a key to **false**, where the keys represent the candidate numbers found in each partition. A second interface, *IGen* contains the method, *run_par()*, for retrieving the next prime number in the first partition and sending an asynchronous method call to all partitions using the construct *!sieve(n)*. The methods are implemented in the classes *Sieve* and *Generator* respectively.

Listing 1: Case Study written in ABS

```
interface ISieve {
  Bool sieve(Int n);
}

interface IGen{
  Unit run_par();
}

class Generator (Int limit , Int elem) implements IGen{

/*
  initializing partitions
*/

//method for sieving using the current prime number. To be run in parallel by all processors.
  Unit run_par(){
    while ((current_Map!=EmptyMap)&&(n*n<limit)){

        Int k=1;

      while(k<elem){
        Fut<Bool> f = lookupUnsafe(processors_Map , k)!sieve(n);
        k=k+1;
      }

      if(lookupUnsafe(current_Map ,n)==True){
        primes = Cons(n,primes);
        current_Map = removeKey(current_Map , n);

        Int first = n*n;
        Int j= first;
        while (j<last){
          current_Map=put(current_Map ,j , False);
          j = j+(2*n);
        }

      }
      n=n+2;
    }
  }
}

class Sieve(Int p, Int size , Int pe) implements ISieve {

/*
Initialization block for each partition
*/

  Bool sieve(Int the_prime){

    Int n = the_prime;
    Int first = n*n;
    Int j;

    /*
    compute the first candidate and store it in variable j
    */

    while (j<=last){
      current_Map=put(current_Map ,j , False);
      j = j+(2*n);
    }


    return True;

  }
}

{ // Main block:
  IGen g=new Generator(50000,1);
  Fut<Unit> f = g!run_par();
  await f?;
}
```

The *Generator* class also contains an initialization phase in which the candidate numbers are split into partitions and stored in a *processors_Map*.

ABS is essentially a simpler view of the MPI [22]. Each actor can be viewed as a separate node who sends messages to other actors with a less complicated syntax. Also the actors are identified and objects like in JAVA and not by rank like in MPI and can have distinct behavior definitions. As can be seen in Listing 1, asynchronous method calls are modeled by the statement ”*actor!method()*”. These statements return a dynamically generated reference that is stored in a Future variable (”*Fut¡Bool¿ f*”) . An important observation is the last line of code ”*await f?*”, which blocks the current method, but allows the next method in the actor’s queue to be scheduled and run. This statement is different from ”*f.get*” of the JAVA language which blocks the entire thread. This statement is also available in ABS and is the equivalent of blocking the entire actor.

The ABS code can be compiled into several executable and simulation languages like Maude, Erlang and Haskell as well as JAVA [23]. A significant issue appeared when generating JAVA code for asynchronous method calls, as these required in [23] the creation of heavyweight JAVA Threads for each call and was functional on only one machine. These issues were behind the ABS-API solution that we will describe in the next section.

3. THE ABS-API LIBRARY

The ABS-API [17] was primarily developed to mitigate the issues of translating ABS code into JAVA code. In this section we focus on the features introduced in JAVA 8 that allow us to have an efficient and easy to use implementation of the API. The goal of this API was to have an intuitive mapping between ABS objects and JAVA threads. To this end we used the Defender Methods to define a default behavior for an interface in the ABS-API and optionally override this behavior to suit a specific function. We also needed an intuitive mapping of the asynchronous method calls identified by ”*actor!method()*” constructs. Our solution wraps abstract method definitions into lambda expressions and is usable as long as there are no subsequent synchronous method calls inside the method’s body. The two interfaces that model parallel execution in JAVA, Runnable [24] and Callable [25] are now functional interfaces as of JAVA 8 and can be modeled into lambda expressions and stored as messages in an Actor’s queue. A sketch of the API class diagram is shown in Figure 1

The Central interface of the ABS-API library is the Actor Interface. This interface has the default behavior of it Actor equivalent in ABS: to poll its queue member for a lambda expression and execute it. In our previous work, a **Parallel Actor** was uniquely identified by its particular String name and deployed in a local context, where it could send messages via lambda expressions to other actors registered in the same context. Its configuration is illustrated in Figure 2. Each actor was modeled to have it’s own queue of messages and run them on a single process. The library is now extended to ABS-API-Remote and has added support for identifying actors by a URI object (the ”*NS*” field in Figure 1) which is defined by an IP and port allowing both local and remote communication. Actors are now aware of their location, as well as other actors identified by the same IP, allowing for optimizations depending on whether two actors work on shared or distributed memory. We refer to this optimization as **Location Aware Actors** or **Distributed Actors**. The Actor interface provides a set of default methods, namely the *run* and *send* methods. The member queue however is restricted to one data structure per IP name in the application. Each actor that has the same IP in its identification key has a reference to this queue. The default ”*run*” method behaves very similar as in the previous solution the only difference being that it polls a message from the shared queue, before checking its type and executing the message correspondingly. The default ”*send*” method is the direct translation the ”*actor!method()*” statement and stores the lambda expression given as an argument in the corresponding shared queue and returns a Future[26] to control synchronization.

In order to communicate with each other, all actors have to be initialized by one and the same object that implements *Context* interface. This object is also responsible for instantiating, for each actor, a *QueuedInbox* object in which all lambda expressions are stored as messages in order for the Actor to function properly. In the extended version, the same inbox is used for actors registered in the same conext that share the same IP in their *NS* field. Finally, because objects that implement the actor interface can vary significantly, yet still communicate with each other, they implement the *Reference* interface. This interface serves multiple purposes such as comparing actors for location

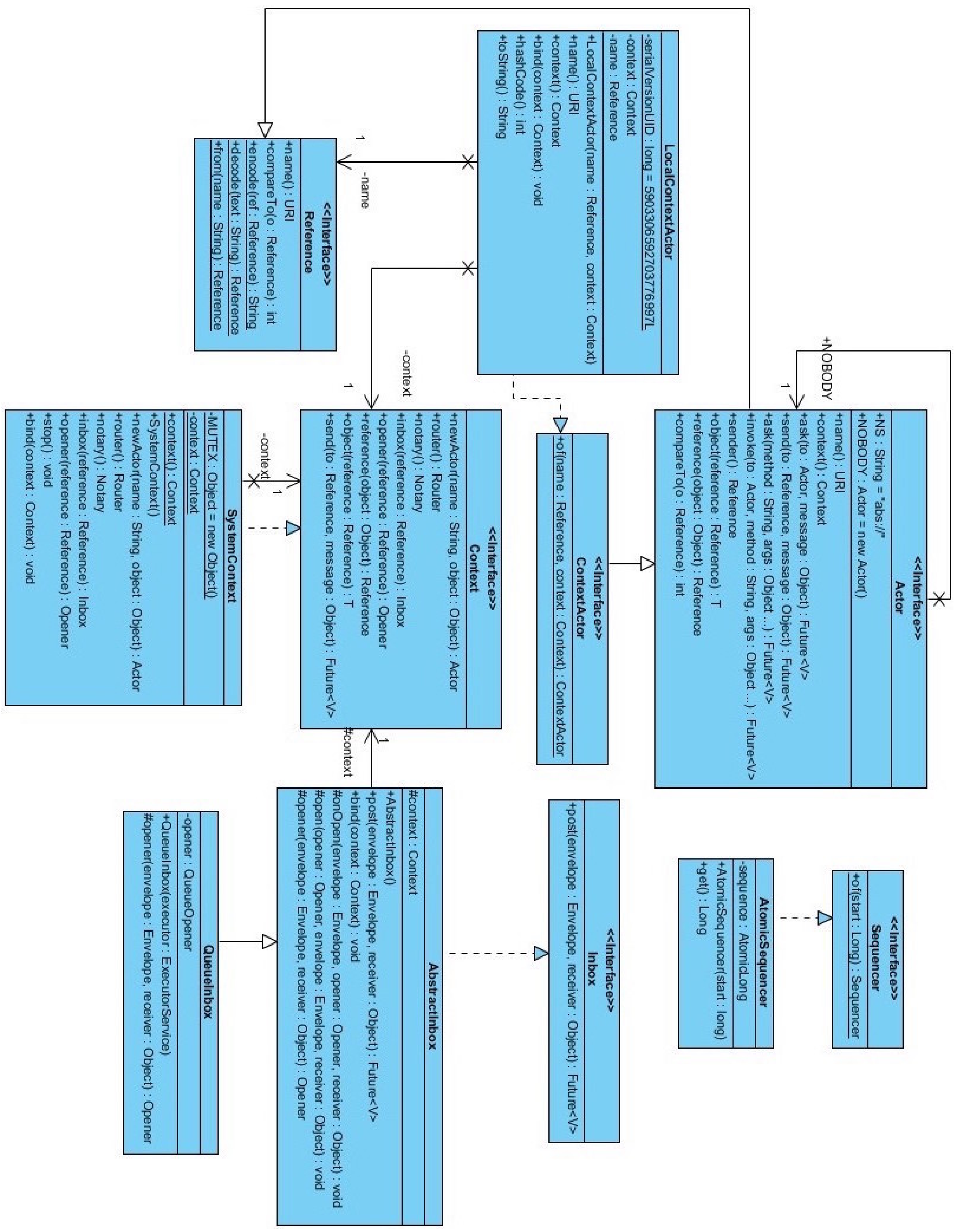
LocalContextActor
-serialVersionUID : long = 5903306592703776997L
-context : Context
-name : Reference
+LocalContextActor(name : Reference, context : Context)
+name() : URI
+context() : Context
+bind(context : Context) : void
+hashCode() : int
+toString() : String
<<Interface>> Reference
+name() : URI
+compareTo(o : Reference) : int
+encode(ref : Reference) : String
+decode(text : String) : Reference
+from(name : String) : Reference
-name
1
-context
1
<<Interface>> Context
+newActor(name : String, object : Object) : Actor
+router() : Router
+notary() : Notary
+inbox(reference : Reference) : Inbox
+opener(reference : Reference) : Opener
+reference(object : Object) : Reference
+object(reference : Reference) : T
+send(to : Reference, message : Object) : Future<V>
SystemContext
-MUTEX : Object = new Object()
-context : Context
+context() : Context
+SystemContext()
+newActor(name : String, object : Object) : Actor
+router() : Router
+notary() : Notary
+inbox(reference : Reference) : Inbox
+opener(reference : Reference) : Opener
+stop() : void
+bind(context : Context) : void
<<Interface>> ContextActor
+of(name : Reference, context : Context) : ContextActor
<<Interface>> Actor
+NS : String = "abs://"
+NOBODY : Actor = new Actor()
+name() : URI
+context() : Context
+ask(to : Actor, message : Object) : Future<V>
+send(to : Reference, message : Object) : Future<V>
+ask(method : String, args : Object ...) : Future<V>
+invoke(to : Actor, method : String, args : Object ...) : Future<V>
+sender() : Reference
+object(reference : Reference) : T
+reference(object : Object) : Reference
+compareTo(o : Reference) : int
+NOBODY
1
#context
1
AbstractInbox
#context : Context
+AbstractInbox()
+post(envelope : Envelope, receiver : Object) : Future<V>
+bind(context : Context) : void
#onOpen(envelope : Envelope, opener : Opener, receiver : Object) : void
#open(opener : Opener, envelope : Envelope, receiver : Object) : void
#opener(envelope : Envelope, receiver : Object) : Opener
QueueInbox
-opener : QueueOpener
+QueueInbox(executor : ExecutorService)
#opener(envelope : Envelope, receiver : Object) : Opener
<<Interface>> Inbox
+post(envelope : Envelope, receiver : Object) : Future<V>
AtomicSequencer
-sequence : AtomicLong
+AtomicSequencer(start : long)
+get() : Long
<<Interface>> Sequencer
+of(start : Long) : Sequencer

Figure 1. ABS-API Class Diagram

purposes, identifying destination actors for the *send* method and creating a new *QueuedInbox* for an actor.

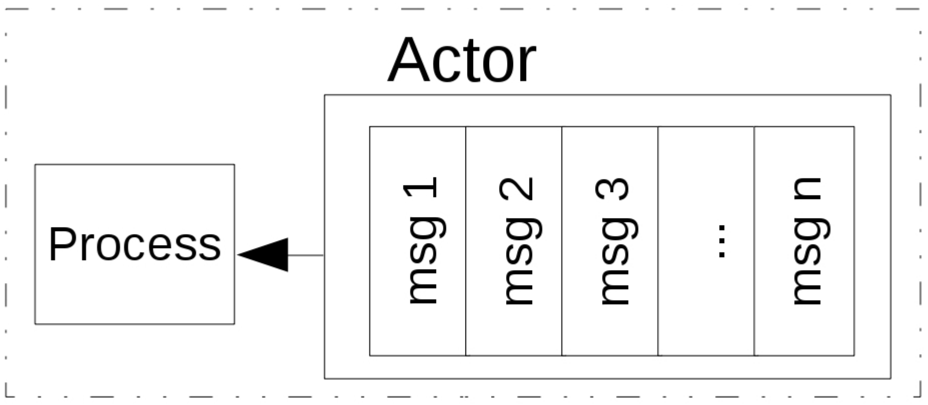

Figure 2. Actor Model

The main features of the ABS-API-Remote library are:

- An actor is identified by a URI object and is aware of its location with regards to the other actors in the system
- Sending a message is done by either placing the lambda expression in the shared queue if the actor is local or sending the message to a different VM if the actor is remote.
- An object during the processing of a method should have a context reference (defined by the URI) to the sender of a message in order to reply to the request.

All these characteristics must co-exist without requiring any modification of the intended interface, for an object to act like an actor.

4. CASE STUDY

Using the ABS-API we studied and implementation of a parallelized version of the Sieve of Eratosthenes [27, 28]. We illustrated the benefit of using the Java language to program in an actor-based model, significantly reducing code size, making implementation easier and allowing much larger data sets to be processed compared with other actor based programming languages. With a large enough data set available we now run the algorithm in a distributed environment and compare the Parallel Actors with the Distributed Actors. The Sieve of Eratosthenes also allows us to illustrate the benefit of identifying actors that run on the same machine, reducing the number of remote method invocations. We continued our work in this paper by comparing these two models using our extended library, where actors are aware of their location and how they interact with other actors.

We used the same partitioning algorithm to evenly distribute the sequence of candidate numbers on each actor. The actors were deployed on two separate VMs to ensure both distributed and parallel computation. We imposed the restriction that the number of partitions must be equal or less than $\lceil n/\lfloor\sqrt{n}\rfloor\rceil$, where n is the target number such that the first partition will have all the prime numbers required in the sieving process. The first actor is responsible for sending a newly found prime number in its partition to all other partitions for sieving as well as executing the sieving process itself. Upon reaching the end of its partitions the first actor is also responsible for awaiting on all the *futures* created in order to collect the final result. Therefore all the actors on the same VM as the first actor will benefit from the optimization, as they can read and write the numbers in a shared queue, while the rest of the actors will require remote method invocation. This is where the location-aware actors affect the performance of the program.

We maintained the two optimizations used previously, the BitSet data structure and omitting even numbers in the partition generation. These two optimizations clearly improve results and therefore needed to be applied before comparing our model to other implementations which have at least these optimizations. The shared queue optimization of Prime Sieve was implemented using existing JAVA data structures such as blocking queue, locks and semaphores which in turn provided an optimal solution for controlling the execution of threads with shared data in the prime sieve implementation using the ABS-API-Remote. The implementation of the shared queue using the ABS-API-Remote was thread safe and the queue capacity and the memory consumption was efficiently handled using the blocking queue data mechanism. We tested our solution on the SurfSara[18] cluster using a 16 CPU machine 2.13 GHz (Intel Xeon E7 ”Westmere-EX”) with 128GB of memory. We present a performance comparison between the two approaches in Figure 3 followed by a memory profiling of the application run in Figures 4 and 5. Finally we see the overhead in number of classes loaded by regular actors in comparison to location-aware actors in Figures 6 and 7.

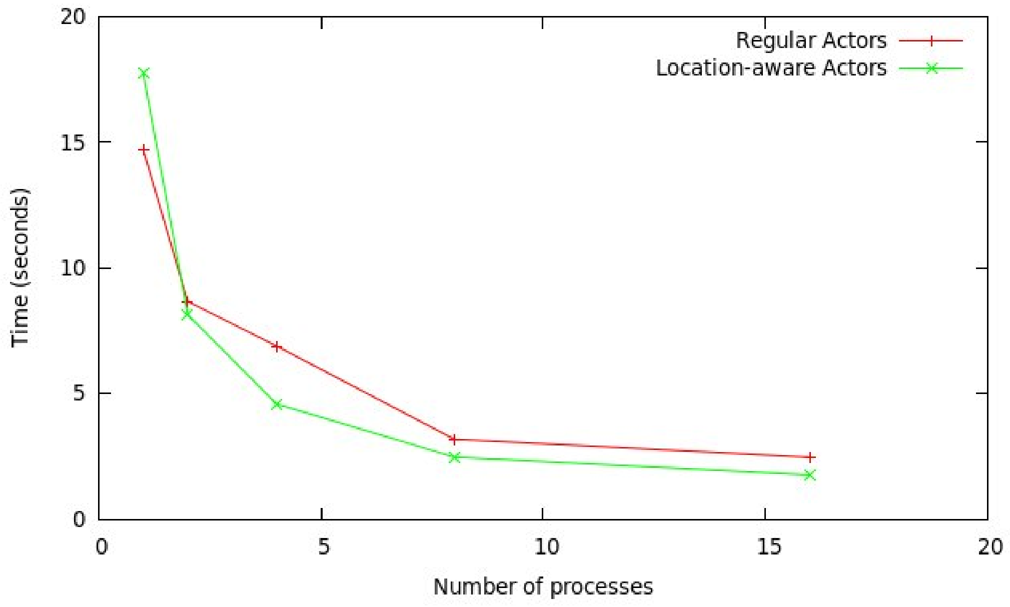

Figure 3. Comparison between location-aware actors and regular actors for a Data Set of 10^9 candidate numbers

With just two standard optimizations, we obtained instant results for candidates up to 10^8 and 2.4 seconds when testing with 10^9 candidates with regular actors and 1.7 seconds with location-aware actors. The profiling tests were conducted using the YourKit Java Profiler[29]. We observe the benefit of eliminating half of the remote method invocations in terms of non-heap memory overhead which is 2MB lower in Figure 5 (9MB) than in Figure 4 (11MB). This is due to the fact that the call stack is significantly lower when data does not need to be transmitted remotely. Furthermore the number of loaded classes for communication purposes between Distributed Actors is 216 lower than the number of classes loaded by Parallel Actors. When comparing our results with the best algorithm for this problem it is still slower, with the record program finishing for a target of 10^9 in 0.26 seconds. The source code for this optimization increased to 45K which is still significantly lower than 505K that fastest program has. The most important result that we want to draw from this study is offering the user a generalization of this optimization by proposing a design pattern presented in the next section.

We reiterate the performance measurements of other concurrent programming languages that use the Actor-based model approach [30] for computing the primes using Sieve of Eratosthenes Algorithm. Existing concurrent programming languages such as Akka [31], Erlang and Go Programming Language are compared with the distributed actor based model approach. For each of

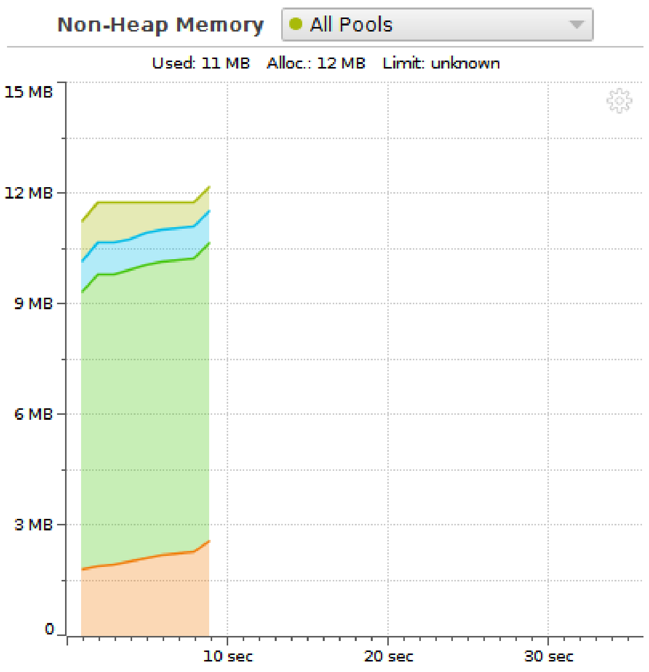

Figure 4. Non-heap Memory Usage of Parallel Actors

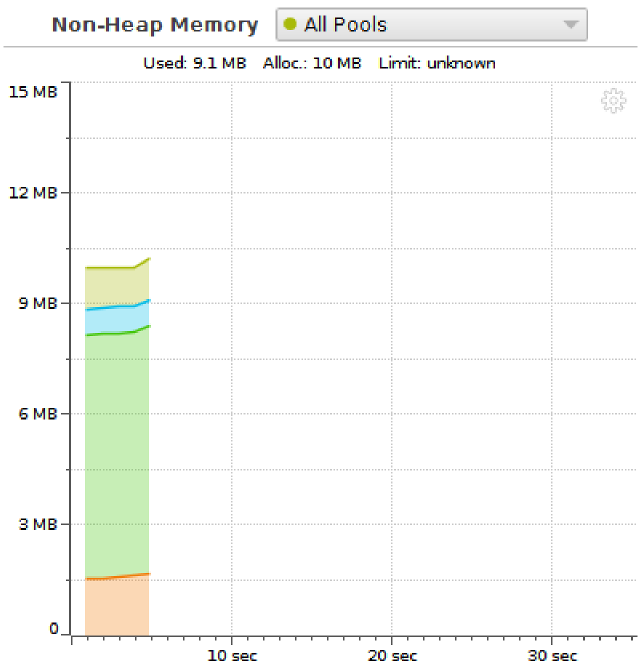

Figure 5. Non-heap Memory Usage of Distributed Actors

these algorithms we took the results presented in the documentation without running the examples with the same setup as they clearly proved inefficient for numbers reaching 10^9. In the existing implementation for Sieve of Eratosthenes Algorithm in Ruby using JRuby and Akka [32, 33], both the controller and model actors are defined as distinct classes. The message sent between the actors is a list with a leading symbol and a payload contained in the remainder of the list. The model only considers a value prime if it does not equal or divide evenly into any previously

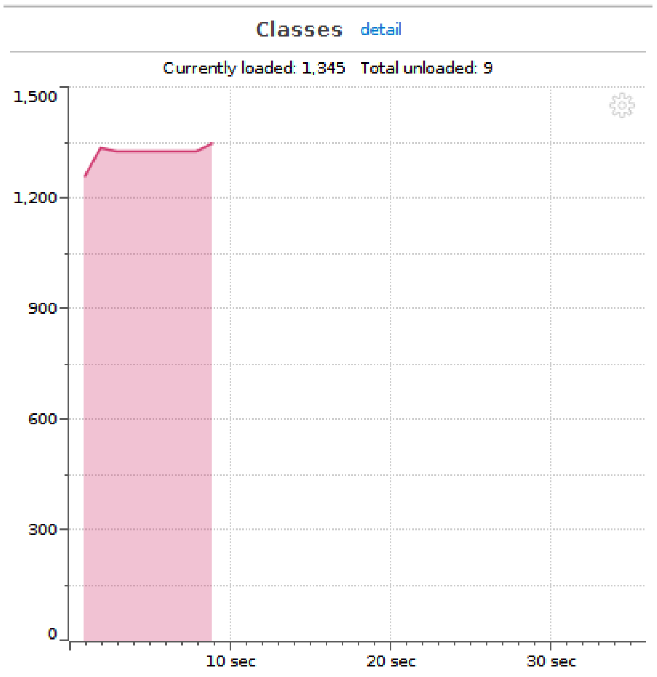

Figure 6. Total Classes Loaded by Parallel Actors

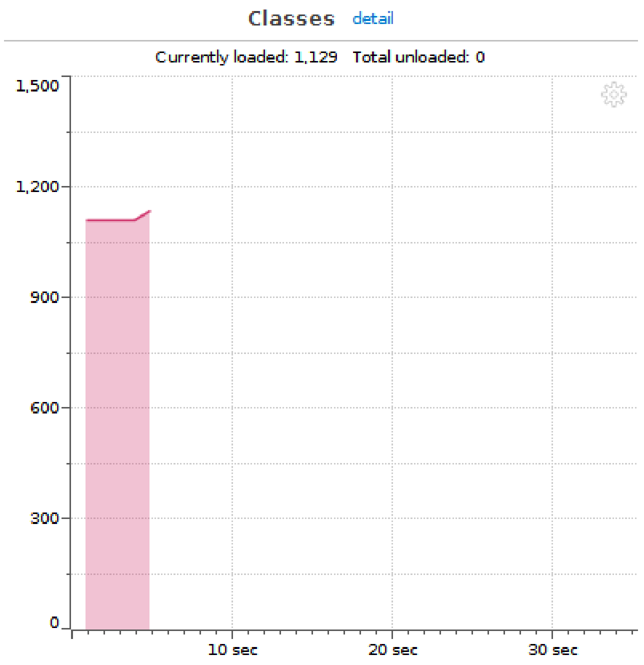

Figure 7. Total Classes loaded by Distributed Actors

observed primes. The results output for 10^4 candidate numbers in 77.114 seconds. Erlang [34] is a functional language, which extends to its native actors support. This language was very important as we have a backend for ABS written in Erlang. The Sieve of Eratosthenes algorithm implemented using Erlang calculates primes until 10^6 in 3.6 seconds. Finally, we made a comparison with the algorithm implemented using the Go programming language. Go programming language [35, 27] is a compiled language that combines some of the syntax of C with some more dynamic aspects to form a next generation systems programming language. The Sieve of Eratosthenes algorithm implemented using Go programming language uses the wheel factorization optimization technique to complete for 10^7 numbers in 12 seconds. From this we draw the conclusion that none of the Actor-based languages can support an upper bound larger than 10^7, therefore our ABS-API-Remote model performs much better compared to other actor models.

5. DATA GROUPS: OPTIMIZATION USING SHARED MEMORY

Although it has achieved novel advantages from actor model and asynchronous message passing (which are mentioned in previous sections), there are some downsides because of the nature of broadcasting mechanism, one of which is broadcasting the same message to multiple actors which can share the same memory. This has both computation overhead, because of redundant repetitive actions for broadcasting message, and memory overhead, because of redundant queuing the same message by multiple actors. This problem is shown in Figure 8 where we eliminate any additional computations.

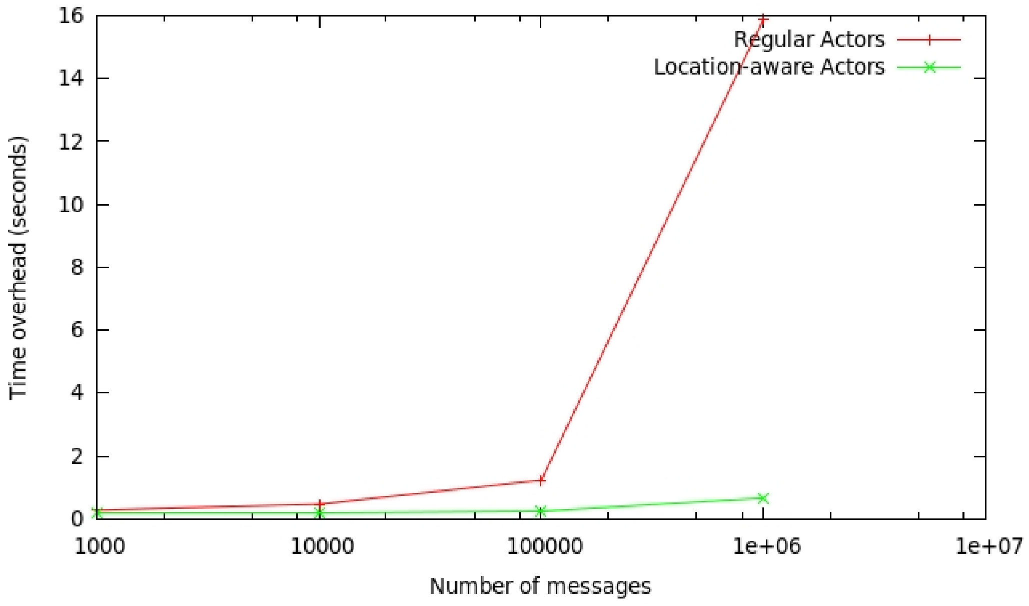

Figure 8. The message passing overhead without any additional computations

Having this problem in mind, there is a solution for such typical data- centered producer-consumers problems which solves the above-mentioned disadvantages, that is exploiting shared memory. Instead of sending one specific message to multiple actors which contains a specific type of data, the generator actor can generate the data just once and put it in a shared memory. Each consumer actor also needs a pointer to the shared memory which points to the next element to be processed. It is simple to understand how shared memory mechanism improves efficiency through eliminating the redundant computation and memory usage.

The advantages of shared memory motivated us to introduce new notion called **Concurrent Actor Group**. A general configuration of group has been illustrated in Figure 9.

Data Group is an abstract notion that puts together actors so that they process (or consume) the data in the shared memory concurrently. One of the most significant features of Data Group is that it can be looked at as one solid entity in higher abstraction level. This advantage leads to ease of use, and understandability of design and code. Furthermore, using shared memory brings about low coupling of the system implementation since generator communicates only with group's shared memory instead of multiple consumer actors.

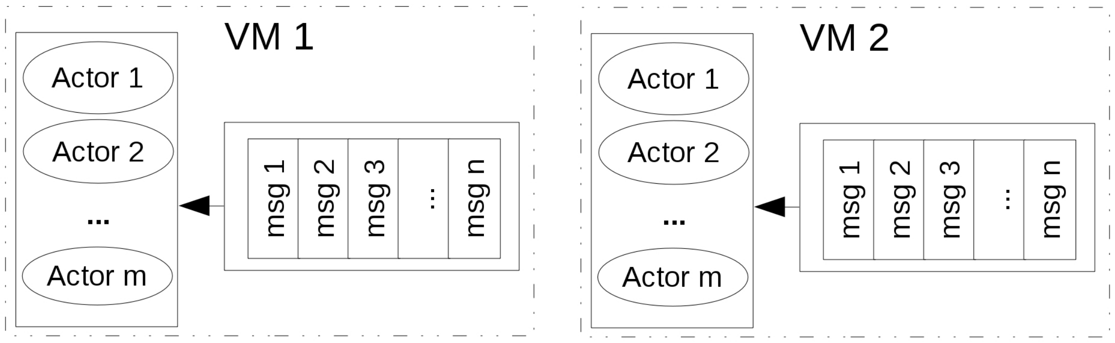

Figure 9. The generalization of the location-aware optimization

Therefore, we have extended ABS API with Concurrent Actor Group as a new notion which can be used for specific spectrum of data-centered producer-consumers problems in order to enjoy the above mentioned advantages than broadcasting mechanism. We observed this difference in Figure3. This new extension will cover several parameters that affect the operation and semantics of a group. For example consumers can be reproducible actors (which have same initial states) or not; they proactively try to have access to messages in shared memory or they passively wait for a scheduler to decide and provide them with messages; the shared memory's message is targeted whether to **all** processes, to **any** of them or to the **specific** one of them; what the type of shared memory data structure is; and what the policies of ordering the data in shared memory are. These features of group which are customizable show further capabilities of groups. As a matter of fact, the messages from different generators can be reordered based on their priority or content.

6. RELATED WORK AND DISCUSSION

The motivation behind the ABS-API-Remote and this case study was the difficulty encountered when generating code from the ABS modeling language. This language has been validated in multiple simulation environments for its quality and performance impact [36, 37], but in terms of actual executable code we experienced very basic performance penalties in both the JAVA and Erlang backends. Before implementing the classic partitioning algorithm we tried implementing the sieving process as a pipeline, where each actor contains a prime number. The spawned actors would then form a chain topology through which new candidate numbers are passed and sieved by checking divisibility with any generated prime number stored in one actor. If the candidate passed through the entire topology, a new *prime* actor is spawned and added to the configuration. This algorithm fits really well with the lightweight threads of the Erlang backend of ABS, but still the program would run out of memory or perform really slow for numbers larger than 10^8. Furthermore, we discovered how costly it is to create new actors using the existing JAVA backend. It was with this variation of the algorithm that we discovered that our existing ABS backends required a common frontend from which to generate code and this was the basis for both the ABS-API development and its current extension ABS-API-Remote. It was also a significant start for research in the Code Generation direction that we have in the ENVISAGE Project.

Another research question that we raised was the difference between two ABS constructs for synchronization: *await f?* and *f.get()*. The first statement allows the actor to schedule other method invocations in its queue to run on its thread, while the second blocks the entire actor. The development of ABS-API took this issue into consideration for the Actor interface: the *f.get()* statement is mapped easily to JAVA Futures, but the *await f?* statement only works in a purely asynchronous program, as the current method needs to be wrapped in a lambda expression before being send to the Actor's queue making it impossible to save the call stack of several synchronous method calls. It is possible to use JAVA's Executor Service to assign a thread pool to each actor, but then we would have the risk of running out of memory as explained in the previous paragraph. It is also important to study the possibility of using a solution equivalent to co-routine and continuations from functional programming languages. A major challenge in language design is still efficient implementations of co-routines in mainstream languages which don't support the continuations featured by functional languages[38].

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented an extension of the the actor-based ABS modeling language in Java 8 which enables optimizations for actors running on the same virtual machine and allows them to identify each other based on their VM location. We have used the API to model a simple distributed application and underlined the performance improvement of this extension. We also showed the impact on memory that these location-aware actors have. Finally we generalized this extension to the **Concurrent Actor Group** notion for designing parallel and distributed applications.

For future work we want to use the ABS-API-Remote and location-aware actors for satisfying more requirements of Cloud Computing users than just memory management. We want to implement scheduling and load balancing strategies for big data applications and control these strategies at the application level. To do this we will implement the computer-aided drug-design application presented in [39]. The application involves running an executable that processes a piece of data called a *ligand* and determines if is a drug candidate. The main requirement for this application is that it should be able to process libraries that may contain hundreds of thousands of ligands and be scalable. On top of that the user running the application needs to access output files on demand and have an overview of partial and complete results in a secure environment as data should not be publicly available.

Using the extended library and general **Group** notion we implemented a first version of a test scenario that connects to the SurfSara cluster with a predetermined number of started VMs. We deployed an actor for each core available on a VM, with each actor responsible for running the executable with one ligand. From this point we will attempt to analyze several scheduling strategies depending on the load of each actor and the resources available on each VM. We want to test the granularity of data retrieval operations, how these affect overall runs and how we can adapt or specialize certain actors on each VM for responding to user requests. Furthermore we want to see how data locality affects the runs, how many ligands should be in a message sent to a group of actors on a VM depending on how fast the ligands are processed and implement a load balancing algorithm for all the VM in the test scenario. Finally we want to compare our results with existing MapReduce and Grid implementations of this application.

ACKNOWLEDGEMENT

Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (`http://www.envisage-project.eu`). Partly funded by the EU project FP7-612985 UPSCALE: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (`http://www.upscale-project.eu`).

The authors would like to thank the reviewers for their for their time and expertise, constructive comments and valuable insight.

REFERENCES

1. Geoffray N, Thomas G, Folliot B, Clément C. Towards a new isolation abstraction for osgi. *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, ACM, 2008; 41–45.
2. Serbanescu VN, Pop F, Cristea V, Achim OM. Web services allocation guided by reputation in distributed soa-based environments. *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, IEEE, 2012; 127–134.
3. Cheong E, Lee EA, Zhao Y. Viptos: a graphical development and simulation environment for tinyos-based wireless sensor networks. *SenSys*, vol. 5, 2005; 302–302.

4. Serbanescu V, Pop F, Cristea V, Antoniu G. Architecture of distributed data aggregation service. *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, 2014; 727–734, doi:10.1109/AINA.2014.89.
5. Pop F, Dobre C, Cristea V. Evaluation of multi-objective decentralized scheduling for applications in grid environment. *Proceedings of 2008 IEEE 4th International Conference on Intelligent Computer Communication and Processing, Cluj-Napoca, Romania, Published by IEEE Computer Society, ISBN: 978-1-4244-2673-7*, 2008; 231–238.
6. Karmani RK, Shali A, Agha G. Actor frameworks for the jvm platform: a comparative analysis. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ACM, 2009; 11–20.
7. Pierre G, Stratan C. ConPaaS: a platform for hosting elastic cloud applications. *IEEE Internet Computing* September-October 2012; **16**(5):88–92.
8. Nicolae B, Antoniu G, Bougé L, Moise D, Carpen-Amarie A. Blobseer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.* February 2011; **71**:169–184, doi:http://dx.doi.org/10.1016/j.jpdc.2010.08.004. URL http://dx.doi.org/10.1016/j.jpdc.2010.08.004.
9. Thain D, Tannenbaum T, Livny M. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience* 2005; **17**(2-4):323–356.
10. von Laszewski G, Wang F, Lee H, Chen H, Fox GC. Accessing multiple clouds with cloudmesh. *Proceedings of the 2014 ACM international workshop on Software-defined ecosystems*, ACM, 2014; 21–28.
11. Von Laszewski G, Foster I, Gawor J, Lane P. A java commodity grid kit. *Concurrency and Computation: practice and experience* 2001; **13**(8-9):645–662.
12. Lea D. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.
13. Pitt E, McNiff K. *Java. rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.
14. Serbanescu V, Nagarajagowda C, Azadbakht K, de Boer F, Nobakht B. Towards type-based optimizations in distributed applications using abs and java 8. *Adaptive Resource Management and Scheduling for Cloud Computing*, Pop F, Potop-Butucaru M (eds.). Lecture Notes in Computer Science, Springer International Publishing, 2014; 103–112, doi:10.1007/978-3-319-13464-2_8. URL http://dx.doi.org/10.1007/978-3-319-13464-2_8.
15. Pop F, Potop-Butucaru M. *Adaptive Resource Management and Scheduling for Cloud Computing*, *Lecture Notes in Computer Science(LNCS)/ Theoretical Computer Science and General Issues*, vol. 8907. Springer, 2015.
16. Hewitt C. Procedural embedding of knowledge in planner. *IJCAI*, 1971; 167–184.
17. Nobakht B, de Boer FS. Programming with actors in java 8. *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*. Springer, 2014; 37–53.
18. https://surfsara.nl/ ; .
19. Johnsen EB, Hähnle R, Schäfer J, Schlatte R, Steffen M. Abs: A core language for abstract behavioral specification. *Formal Methods for Components and Objects*, Springer, 2012; 142–164.
20. http://www.hats-project.eu/node/113 ; .
21. Albert E, de Boer F, Hähnle R, Johnsen EB, Laneve C. Engineering virtualized services. *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, ACM, 2013; 59–63.
22. Chorley MJ, Walker DW. Performance analysis of a hybrid mpi/openmp application on multi-core clusters. *Journal of Computational Science* 2010; **1**(3):168–174.
23. Schaefer J. *A Programming Model and Language for Concurrent and Distributed Object-Oriented Systems*. Verlag Dr. Hut. ISBN: 978-3-86853-833-5.
24. http://docs.oracle.com/javase/6/docs/api/java/lang/Runnable.html ; .
25. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html ; .
26. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html ; .
27. O'NEILL ME. The genuine sieve of eratosthenes. *Journal of Functional Programming* 2009; **19**(01):95–106.
28. Bokhari SH. Multiprocessing the sieve of eratosthenes. *Computer* 1987; **20**(4):50–58.
29. http://www.yourkit.com/docs/ ; .
30. Imam SM, Sarkar V. Integrating task parallelism with actors. *ACM SIGPLAN Notices*, vol. 47, ACM, 2012; 753–772.
31. Haller P. On the integration of the actor model in mainstream technologies: the scala perspective. *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, ACM, 2012; 1–6.
32. https://travis-ci.org/fujin/sieve_actors/jobs/2149495 ; .
33. https://github.com/heuristicfencepost/sieve_actors ; .
34. Armstrong J, Virding R, Wikström C, Williams M. Concurrent programming in erlang 1993; .
35. Balbaert I. *The Way To Go: A Thorough Introduction to the Go Programming Language*. IUniverse, 2012.
36. Albert E, de Boer FS, Hähnle R, Johnsen EB, Schlatte R, Tarifa SLT, Wong PYH. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. *Service Oriented Computing and Applications* 2014; **8**(4):323–339, doi:10.1007/s11761-013-0148-0. URL http://dx.doi.org/10.1007/s11761-013-0148-0.
37. Johnsen EB, Schlatte R, Tarifa SLT. Modeling resource-aware virtualized applications for the cloud in real-time ABS. *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, 2012; 71–86, doi:10.1007/978-3-642-34281-3_8. URL http://dx.doi.org/10.1007/978-3-642-34281-3_8.
38. Haynes CT, Friedman DP, Wand M. Continuations and coroutines. *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, ACM: New York, NY, USA, 1984; 293–298, doi:10.1145/800055.802046. URL http://doi.acm.org/10.1145/800055.802046.

39. Jaghoori MM, Altena AJV, Bleijlevens B, Olabarriaga SD. A grid-enabled virtual screening gateway. *Science Gateways (IWSG), 2014 6th International Workshop on*, IEEE, 2014; 24–29.