Towards Type-Based Optimizations in Distributed Applications Using ABS and JAVA 8

Vlad Serbanescu(✉), Chetan Nagarajagowda, Keyvan Azadbakht, Frank de Boer, and Behrooz Nobakht

Centrum Wiskunde and Informatica, Amsterdam, The Netherlands
{vlad.serbanescu,nagaraja,k.azadbakht,frb,behrooz.nobakht}@cwi.nl

Abstract. In this paper we present an API to support modeling applications with Actors based on the paradigm of the Abstract Behavioural Specification (ABS) language. With the introduction of JAVA 8, we expose this API through a JAVA library to allow for a high-level actor-based methodology for programming distributed systems which supports the programming to interfaces discipline. We validate this solution through a case study where we obtain significant performance improvements as well as illustrating the ease with which simple high and low-level optimizations can be obtained by examining topologies and communication within an application. Using this API we show it is much easier to observe drawbacks of shared data-structures and communications methods in the design phase of a distributed application and apply the necessary corrections in order to obtain better results.

Keywords: Cloud computing · Programming models · Distributed applications · Formal methods · Optimization

1 Introduction

The Java language is one of the mainstream object oriented programming languages that supports a programming to interfaces discipline. It has evolved into a platform to design and implement standards in several domains of both research and industry, along with supporting its community with new language features and standards. With application reaching exascale dimensions in terms of data volumes and requiring a lot of computing power, focus has increased in researching numerous libraries and frameworks with an attempt to provide distribution and concurrency at the level of Java language. However, it is widely recognized that the thread-based model of concurrency in Java that is a well-known approach is not appropriate for realizing distributed systems because of its inherent synchronous communication model. A powerful concept on the other hand is the event-driven actor model of concurrency introduced in [9] which allows many applications to extend these actors to suit their behaviour. Examples of these domains include designing embedded systems [5], wireless sensor networks

F. Pop and M. Potop-Butucaru (Eds.): ARMS-CC 2014, LNCS 8907, pp. 103–112, 2014.
DOI: 10.1007/978-3-319-13464-2_8

[4], distributed web-services [19], multi-core programming [12,18] and delivering cloud services through SaaS or PaaS [14,17]. Furthermore, it provides the basis for increasingly popular languages in parallel and distributed computing like Scala [8]. However, such a language uses an explicit mechanism at application level to support message passing and handling, which diminishes the general object-oriented approach of method look-ups that forms the basis of programming to interfaces.

We introduce Java 8 API [15] to program distributed systems and to formalize actor-based programming which implies asynchronous message passing together with the evergrowing object-oriented software engineering approach. Using asynchronous message passing and a corresponding actor programming methodology which abstracts invocation from execution (e.g. thread-based deployment), we want to fully support and emphasize the programming to interfaces discipline. The main research question of this paper is to demonstrate that using this API, several type-based optimizations can be achieved at the design phase as well as detecting possible bottlenecks in distributed applications using the simple example of The Sieve of Eratosthenes [3,16]. This is the first step in researching how to use type-systems to automate optimizations in parallel and distributed applications.

2 The ABS Language

Our starting point for the actor programming model assumed in this paper is the Abstract Behavioral Specification language (ABS) introduced in [11]. ABS offers programmers several features such as asynchronous method calls, futures to control these calls, interfaces for encapsulation and cooperative scheduling of method invocations inside concurrent (active) objects. Specifically any object created in ABS represents an actor with encapsulated data. Similar to JAVA, their behaviour and state is defined by implementing interfaces with their corresponding methods. Thus they interact by making asynchronous calls to these methods which generate messages that are pushed into a queue specific to each actor. An actor progresses by taking a message out of its queue and processing it by executing its corresponding method. This feature combination results in a concurrent object-oriented model which is inherently compositional. The simplicity of ABS results from the fact that each actor is viewed as a separate processor making it very suitable for modeling distributed applications similar to MPI [6], with the added benefit of specifying a distinct behaviour for each actor without the connectivity issue. Finally asynchronous method calls use futures as dynamically generated references to return values.

3 The ABS-API Library

In this section we focus on the features in Java 8 that allow us to have an efficient and easy to use implementation of the actor model in ABS. First, methods in an interface are declared as Defender Methods using the **default** keyword. This

allows actors to have a default behaviour and optionally override this behaviour to suit a specific function. For instance, in Java 8 java.util.Comparator provides a default method named reversed() that creates a reversed-order comparator of the original one. Such default method eliminates the need for any implementing class to provide such behavior by inheritance. Second, the introduction of Java Functional Interfaces and lambda expressions is a fundamental change in Java 8. All interfaces that contain only one abstract method are now functional interfaces that at runtime can be turned into lambda expressions. This means that the same lambda expression can be statically cast to a different matching functional interface based on the context. This is a fundamental new feature in Java 8 that facilitates application of functional programming paradigm in an object-oriented language. This API makes use of these new features available in JAVA 8 because many of the interfaces found in the Java libraries are now marked as functional interfaces, most important of which in this context are java.lang.Runnable and java.util.concurrent.Callable. This means that a lambda expression can replace an instance of Runnable or Callable at runtime by JVM. Therefore a lambda expression equivalent of a Runnable or a Callable can be treated as a queued message of an actor and executed. Finally, Java Dynamic Invocation and execution with method handles enables JVM to support efficient and flexible execution of method invocations in the absence of static type information. This feature introduces a new API, available through java.lang.invoke.MethodHandles that allows translation of a lambda expression in Java 8 at runtime to be executed by JVM. Furthermore, this feature has been validated performance-wise over anonymous inner classes and the Java Reflection API. Thus, lambda expressions are compiled and translated into method handle invocations rather reflective code or anonymous inner classes.

The ABS-API library has a fundamental interface namely the Actor Interface. Using an interface for an actor allows an object to preserve its own interfaces, and also it allows for multiple interfaces to be implemented and composed. A Java API for the implementation of ABS models should have the following main features. First, one actor should be able to asynchronously send an arbitrary message in terms of a method invocation to a receiver actor. Second, sending a message can optionally generate the equivalent of an ABS future that the sending actor can use to refer to the return value. Finally, an object during the processing of a message should have a context reference to the sender of a message in order to reply to the message via another message. All these characteristics must co-exist without requiring any modification of the intended interface, for an object to act like an actor. The Actor interface provides a set of default methods, namely the run and send methods, as well as a queue that supports concurrent features of Java API 5. On one hand, the default run method takes a message from the queue, checks its type and executes the message correspondingly. On the other hand, the default (overloaded) send method stores the sent message in the corresponding queue. As mentioned before, in ABS we use futures to control synchronization. In the ABS-API we model messages that are expected to return a result as instances of Callable and a future is created

by the send method which is returned to the caller, while those messages that need to run in parallel without a future reference to the outcome are modeled as instances of Runnable.

4 Case Study

Using our solution, we present in this section a parallelized implementation of the Sieve of Eratosthenes [3,16]. We aim to illustrate the benefit of using the Java language to program in an actor-based model while at the same time showing the performance improvement compared to other actor models, the benefit of observing certain behaviours in the programming phase, as well as showing that the actor-based model still performs well when compared to implementations that apply low-level optimizations. Generating prime numbers is a key factor in authentication algorithms. With distributed applications running on several cloud environments, the need to authenticate securely and transparently without a sizable overhead is constantly increasing. At the same time our case study is perfect for modeling partitions as actors as well as making it easy to simulate an application that can work on a multi-core platform using a shared memory or a distributed platform where communication between actors is key. The Sieve of Eratosthenes also allows us to illustrate several optimizations that result from the actor based model, as well as how certain well known optimizations are easy to apply in this model without significantly increasing the code size and therefore the design phase of a distributed application.

To model the algorithm using actors we use the well-known partitioning parallel algorithm and represent each partition as an actor. In this algorithm, the numbers are partitioned into smaller sequences of numbers with the same size. Based on this algorithm, the size of each partition must be equal or greater than(except probably for the last partition) $\lfloor\sqrt{n}\rfloor$, and the number of partitions must be equal or less than $\lceil n/\lfloor\sqrt{n}\rfloor\rceil$, where n is the target number. Following the above-mentioned constraints, the first partition contains all the prime numbers required to sieve, therefore the first actor in the model will be responsible for sending asynchronous messages to the others that will invoke the sieving process. With asynchronous messages written as regular method calls in Java, there is a significant improvement in the ease of programming compared to a similar solution that uses specialized directives like in MPI.

We decided to implement a data structure optimization, and therefore use a BitSet data structure and also half the amount of processing work by eliminating even numbers. These two optimizations clearly improve results and therefore needed to be applied before testing our model to other implementations which have at least these optimizations. We tested our solution on the SurfSara [13] cluster using a 16 CPU machine with 128 GB of memory. A small example of a sieve invocation and using a future to synchronize on the result for checking the correctness of the prime numbers found at the end of the program is given in the following.

```
for (Actor s : actors) {
//sieving process invocation for a new prime number
Future<Object> r = this.send(()->{s.sieve(prime)});
futures.add(r);
}
```

Our main result in this paper is that with just the two standard optimizations, we obtained instant results for candidates up to 10^8 and 2.6 s when testing with 10^9 candidates. We decided to compare our results to the fastest sieving algorithm that further has cache-friendly memory management, wheel factorization and segmented sieve [20]. Our model is only 10 times slower with the record program finishing for a target of 10^9 in 0.26 s. What we want to emphasize however is that the source code size for this record implementation is 505 K compared to 30 K, the size of the Actor-based model. This significantly improves the ease of programming even in a simple distributed application. Further comparisons with other Actor-based models will be discussed in the following section.

4.1 Type-Based Optimization

As discussed before, we aim to use this API to observe certain drawbacks or bottlenecks from the programming phase of the application. In this simple example it is easy to observe that the number of asynchronous messages sent between actors is very high. With the API exposed in the Java language we can easily use a shared data structure to eliminate the messages sent corresponding to each prime number used to sieve the partitions. While this is something very trivial, what we actually aim to extract from this ABS-API is the possibility to detect and automate such optimizations depending on the application that is modeled. We want to be able to analyze several applications which can be CPU-intensive, IO-intensive or with multiple memory accesses and be able to detect performance penalties just like the one above.

5 Experimental Methodologies and Results

The development of multicore CPUs rapidly provides a bigger need for parallel and concurrent programming. Currently there are multiple open source frameworks such as Akka, Erlang, Scala, Finagle, Storm, Hadoop, Ruby, Go Language, Hive and Pig available for distributed parallel and concurrent programming. Further Akka, Finagle, Storm and MapReduce are different elegant solutions for distributed computing and are based on functional programming languages. Pig programs are more complex, and can be compiled into an execution plan consisting of several stages of MapReduce jobs, some of which can run concurrently. Further Pig and Hive are script based data flow languages and thus more volatile and harder to debug during programming and provides a higher level of abstraction for MapReduce programming that is similar to SQL, but it is procedural code, not declarative. They can be extended with User Defined

Functions (UDFs) written in Java, Python, JavaScript, Ruby, or Groovy and includes tools for data execution but was not ideal for implementing the Sieve of Eratosthenes case demonstrated in the proposed paper.

Also there are various actor oriented libraries and languages in the existing techniques for implementing some variant of actor semantics and are based on object oriented programming languages. The actor oriented languages includes but are not limited to Erlang, SALSA, E language and AXUM that are based on message passing. Further one of the important programming models based on message passing is the Actor model. The Actor model is an inherently concurrent model based on asynchronous message passing. Moreover the Actor based model includes many important features such as encapsulation, fair scheduling, location transparency, locality of references which makes the actor model a suitable programming model for distributed parallel and concurrent programming. ABS is a concurrent, object-oriented modeling language that features functional data types. The ABS model uses asynchronous method calls, interfaces for encapsulation, and cooperative scheduling of method activations inside concurrent objects. In specific the ABS language is a class-based object-oriented language that features algebraic data types and side effect-free functions. Also Actors [22] implement a shared-nothing model for concurrency. A model represents a fragment of the state of sieve, specifically some subset of the primes discovered currently in the existing techniques. Further the existing open source frameworks are compared with the ABS model proposed in the paper for performance by implementing the Sieve of Eratosthenes case using the ABS API.

Currently there is a plurality of concurrent programming languages that use the Actor-based model approach for computing the primes using Sieve of Eratosthenes Algorithm. The results obtained from the existing concurrent programming languages such as Scala, Erlang and Go Programming Language are compared with the Actor based model approach implemented for Sieve of Eratosthenes Algorithm in the proposed paper. As discussed in the previous section, the Actor-based model approach proposed in the paper generates primes at 2.6 s until 10^9 on 16 CPU machines using Sieve of Eratosthenes Algorithm.

In the existing implementation for Sieve of Eratosthenes Algorithm in Ruby using JRuby and Akka [23], 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 observed primes. The Sieve of Eratosthenes algorithm implemented in Ruby using JRuby and Akka computes primes until 10^4 in 77.114 s. This method was not effective and the performance was really slow once we got past an upper bound of about 10,000 numbers.

This follows a similar implementation [21] for the Sieve of Eratosthenes Algorithm in Erlang where tuples are used for sending messages instead of lists. The Sieve of Eratosthenes algorithm implemented using Erlang calculates primes until 10^6 in 3.6 s. This method was effective for calculating primes until 10^6, but the performance was really slow for higher numbers between the range of 10^7 to 10^9.

Further there are other actor-based languages, like Scala which closely follows the object-oriented model of programming though it has many functional programming features included to support message passing and handling but this method diminishes the general object-oriented approach of method look-ups that forms the basis of programming to interfaces.

Further in the existing technique, the Sieve of Eratosthenes Algorithm is implemented using the Go programming language. Go programming language [2] 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. One interesting feature of the Go language is the built-in multithreading feature which is based on channels and goprocesses. For the Sieve of Eratosthenes implementation using GoLangauge, each time a candidate makes it through the sieve and is returned as a new prime number. Further a new goroutine is created to check future candidates and reject them if they divide evenly by the new prime. The implementation is not useful as a standalone application since it includes no termination condition and also there are other disadvantages in the model as each goroutine knows only about the prime it contains and the channel where candidates should be sent if they pass. Once the goroutine is created its state does not change and also new state is added by creating a new goroutine for a newly-discovered prime and the state is never deleted. Moreover once a prime is discovered, removing it from consideration is non-sensical due to all states being completely distributed and no entity in the system knows about all discovered primes. The Sieve of Eratosthenes algorithm implemented using Go programming language calculates primes until 10^7 in 1 m 33.62 s. This method was effective for calculating primes until 10^7 and could be further optimized using the Wheel Factorization optimization technique which in turn provided better time performance and calculated primes in 12 s for primes until 10^7.

Using the approach of the Go Programming Language, we tried the same modeling in ABS. We created an object which generated candidate numbers and created new objects with new found primes. The numbers were then sent through asynchronous messages to objects containing primes up until the last object which spawned a separate object with the newly found prime. Each object operated as a separate thread which verified candidate numbers and discarded them. In this manner we discovered that the JAVA backend of ABS was extremely costly performance-wise when sending asynchronous messages and creating new objects. Even after buffering several prime numbers into the same object and balancing the verification load we still obtained very slow results compared even to a naive sequential approach. These results are what prompted us to develop the ABS-API as a layer of translation between ABS and the JAVA backend to both reduce code size and improve performance.

We also looked at some optimizations possible from the API perspective. One interesting optimization was using instruction level parallelism by sieving with more than one prime number at a time. While this is a very trivial task for this application, we want to investigate the possibility of adjusting the overall work of an actor as a load balancing technique implemented directly in the

coding phase using the ABS-API. Furthermore we want to introduce a notion of location-awareness to our ABS-API such that actors know when they can communicate using just reads and writes from a shared data structure and when actual asynchronous messages need to be passed in between them depending on the machine that the actors run on. This memory-management optimization is be a significant benefit to Cloud-distributed applications.

6 Related Work

Our Java ABS-API solution was constructed after looking at several works of research and development in the domain of actor modeling and implementation in different languages [10]. We discuss a few languages at the level of modeling and implementation with more focus on Java and JVM-based efforts. Erlang [1] is a programming language used to build massively scalable soft real-time systems with requirements on high availability. It is a functional language, which extends to its native actors support. Its runtime system has built-in support for concurrency, distribution and fault tolerance. Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. The processes in Erlang communicate using message passing instead of shared variables, which removes the need for locks, but makes all synchronization explicit. Scala [8] is both a functional and object-oriented language that unifies thread-based and event-based programming model to fill the gap for concurrency programming. Like Java it provides the same features for handling concurrency, but it is not possible to manage and schedule priorities on messages sent to other actors. We also compared our results to Akka [7] implementation of the actor model. This toolkit allows to build highly concurrent, distributed, and fault tolerant event-driven applications on the JVM based on actor model.

7 Conclusions

In this paper, we discussed an implementation of the actor-based ABS modeling language in Java 8 which uses the basic object-oriented mechanisms, principles of method look-up and programming to interfaces. We have used the API to model a simple distributed application that remains performant without applying specific optimization and fares much better than other actor-based models. We also showed the functionality of using Java to program distributed applications as well as making it possible to detect possible optimizations at the design phase.

The underlying modeling language has an executable semantics and supports a variety of formal analysis techniques, including deadlock and schedulability analysis. Further it supports a formal behavioral specification of interfaces to be used as contracts. As discussed in Sect. 4.1 our research will focus on using type systems to automate optimization, extend our solution to identify resource usage of programs and communication topologies and apply a corresponding optimization table from which to eliminate drawbacks and bottlenecks during

code generation. Our future work will also focus on modeling more difficult distributed applications at testing important cloud features such as reliability, resource-provisioning, multitenancy and scalability. We also aim to automatically generate ABS models from Java code which follows the ABS design methodology. Model extraction allows industry level applications be abstracted into models and analyzed for different goals such as deadlock analysis and concurrency optimization.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Morgan Kaufmann, San Francisco (1993)
2. Balbaert, I.: The Way to Go: A Thorough Introduction to the Go Programming Language. IUniverse, Bloomington (2012)
3. Bokhari, S.H.: Multiprocessing the sieve of Eratosthenes. Computer **20**(4), 50–58 (1987)
4. Cheong, E., Lee, E.A., Zhao, Y.: Viptos: a graphical development and simulation environment for TinyOS-based wireless sensor networks. In: SenSys, vol. 5, pp. 302–302 (2005)
5. Geoffray, N., Thomas, G., Folliot, B., Clément, C.: Towards a new isolation abstraction for OSGi. In: Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems, pp. 41–45. ACM (2008)
6. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-passing Interface, vol. 1. MIT press, Cambridge (1999)
7. Haller, P.: On the integration of the actor model in mainstream technologies: the scala perspective. In: Proceedings of the 2nd edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, pp. 1–6. ACM (2012)
8. Haller, P., Odersky, M.: Scala actors: unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2), 202–220 (2009)
9. Hewitt, C.: Procedural embedding of knowledge in planner. In: IJCAI, pp. 167–184 (1971)
10. Imam, S.M., Sarkar, V.: Integrating task parallelism with actors. In: ACM SIGPLAN Notices, vol. 47, pp. 753–772. ACM (2012)
11. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
12. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, pp. 11–20. ACM (2009)
13. https://surfsara.nl/
14. Nicolae, B., Antoniu, G., Bougé, L., Moise, D., Carpen-Amarie, A.: Blobseer: next-generation data management for large scale infrastructures. J. Parallel Distrib. Comput. **71**, 169–184 (2011). http://dx.doi.org/10.1016/j.jpdc.2010.08.004
15. Nobakht, B., de Boer, F.S.: Programming with actors in Java 8. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014, Part II. LNCS, vol. 8803, pp. 37–53. Springer, Heidelberg (2014)

16. O'Neill, M.E.: The genuine sieve of Eratosthenes. J. Funct. Program. **19**(01), 95–106 (2009)
17. Pierre, G., Stratan, C.: ConPaaS: a platform for hosting elastic cloud applications. IEEE Internet Comput. **16**(5), 88–92 (2012)
18. Pop, F., Dobre, C., Cristea, V.: Evaluation of multi-objective decentralized scheduling for applications in grid environment. In: Proceedings of 2008 IEEE 4th International Conference on Intelligent Computer Communication and Processing, pp. 231–238. IEEE Computer Society, Cluj-Napoca, Romania (2008). ISBN: 978-1-4244-2673-7
19. Serbanescu, V.N., Pop, F., Cristea, V., Achim, O.M.: Web services allocation guided by reputation in distributed SOA-based environments. In: 2012 11th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 127–134. IEEE (2012)
20. Sieve, F.: http://primesieve.org
21. Tasharofi, S.: Efficient testing of actor programs with non-deterministic behaviors. Ph.D. thesis, University of Illinois at Urbana-Champaign (2014)
22. http://heuristic-fencepost.blogspot.nl/2012/02/ruby-and-concurrency-maintaining-purity.html
23. http://absurdfarce.github.io/blog/2012/01/05/ruby-and-concurrency-design-with-actorsandakka/