Benchmarking Communication in Actor- and Agent-Based Languages

(Extended Abstract)

Rafael C. Cardoso
FACIN–PUCRS
Porto Alegre - RS, Brazil
rafael.caue@acad.pucrs.br

Jomi F. Hübner
DAS–UFSC
Florianópolis - SC, Brazil
jomi@das.ufsc.br

Rafael H. Bordini
FACIN–PUCRS
Porto Alegre - RS, Brazil
r.bordini@pucrs.br

ABSTRACT

This paper presents some results of communication benchmarks used to compare the performance of one agent-oriented and two actor-oriented programming languages. The experiments include an existing benchmark for traditional programming languages as well as two new variants of that benchmark. We selected Erlang and Scala to represent actor languages, and Jason to represent agent languages. We discuss here a summary of the result for those three experimental scenarios for each of the three languages and the respective result analysis in regards to time, memory, and core usage.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*

General Terms

Experimentation, Languages, Performance

Keywords

communication benchmark; multi-agent programming; actor-based programming

1. INTRODUCTION

There is no quantitative analysis, to the best of our knowledge, of how well agent languages can perform compared to actor languages. Because the actor approach is by design lighter than agents and because each language has a different runtime environment or virtual machine, this comparison cannot be done directly. Therefore, in this paper we make use of scale factors to compare these languages.

Our motivation for this line of work is first because of the lack of benchmarks for agent programming languages specifically. The long-term goal is to arrive at benchmarks that cover the specific features of programming languages for autonomous agents. The reason for comparing agent languages to actor languages is that actor languages have the same basic principles of agent languages but have been developed for longer and therefore have had more time to have the underlying techniques improved. Future work on this line of research might lead to ideas for improving the basic mechanisms of agent languages.

Actor-oriented programming languages are based on the actor model [1]; an actor is a lightweight process that does not share state with other actors and communicates by asynchronous message passing through mailboxes. We selected Erlang [2] and Scala [5] as the actor language representatives. Agent-oriented programming languages are based on the agent model, which is effectively an extension of the actor model. While both agents and actors are lightweight processes and reactive, agents are more complex entities and typically capable of "practical reasoning" (i.e., logic-based reasoning about the best action to take at given circumstances) [3, 4]. Jason [4] is the agent language representative in this paper[1].

We also considered including JACK as a second agent language representative, but we refrained from doing all the experiments with that language because of two main reasons: first, unlike the other selected languages, JACK is commercial software; and second, the results in the first two scenarios showed that it seems not to take advantage of multiple cores, so it cannot be compared to the other 3 languages used in these experiments.

2. COMMUNICATION BENCHMARKING

The Computer Language Benchmarks Game (`http://shootout.alioth.debian.org/`) provides performance evaluation for approximately twenty four languages using various benchmark problems. Although they evaluate the performance on computers with multiple cores, the tasks and most of the languages are not appropriate for concurrent programming. A Python script is available in that website which performs repeated measurements of elapsed time, resident memory usage, and CPU load for each core.

The scenarios described below focus on the message passing aspect of communication, testing the support for asynchronous message passing and concurrency of each language; both of these features are essential for actor- and agent-based languages. The experiments were run on an Intel®Core™i5-2400 CPU @ 3.10GHz (4 physical cores, no HyperThreading) machine with 4GB of DDR3 RAM, running the operating system Ubuntu 12.04.1 64 bits, and using Jason 1.3.9, Erlang R14B04 erts 5.8.5, and Scala 2.9.1.

The first scenario is a simple case of passing a token N times through a *ring* of "workers" (i.e., agents, processes, or actors, depending on the language). Each program in this scenario should: create a ring of 500 linked workers (named 1 to 500); pass a to-

[1]The code for all experiment scenarios used in this paper is available at `http://tinyurl.com/b5rjoc9`.

Appears in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013)*, Ito, Jonker, Gini, and Shehory (eds.), May, 6–10, 2013, Saint Paul, Minnesota, USA.

ken to worker 1; each worker passes the token to its neighbouring worker; the program halts when the token has been passed (between any two workers) N times.

In Scenario 2, we added more tokens and allowed them to be passed concurrently. So rather than passing only one token, at the start of a run 50 tokens are distributed around the ring using $I * (W/T)$, where I is the number of the current token to be sent, W is the total number of workers, and T is the total number of tokens. All 50 tokens must have been passed N times each in order for a run in Scenario 2 to end.

As a follow up to Scenario 2, in Scenario 3 N is fixed at 500 and besides those 50 tokens from Scenario 2, now a new type of token is introduced in the ring; we call the new tokens as "type 1". The 50 tokens as in Scenario 2 are now referred to as "type 2" and they work exactly the same way as in that scenario. We created 1,000 type 1 tokens, two per worker, to simulate the "mundane tasks" that the workers would be doing normally while waiting for the "special" type 2 tokens to arrive. When a worker receives a token type 2, as soon as it realises it, the worker should pass that token directly to the next worker in the ring. A run of Scenario 3 ends when all 50 type 2 tokens have been passed 500 times each. This scenario measures not only concurrency in communication but also reaction time to higher priority messages.

We ran experiments for the three scenarios varying N from 500 to 50m with an order of magnitude increase between each configuration (i.e., 6 variations on N), and again with the number of workers (W) varying from 50 to 5k, this time with N fixed at 5m, for measurements of resident memory. Erlang was the fastest language in regards to elapsed time in all scenarios, followed by Scala in Scenario 1. In Scenario 2, Scala lost performance and Jason seized to second position. As an example, we show the values in seconds for Scenarios 1 and 2 with $N = 50m$: in the first scenario Jason took 572, Erlang 14, and Scala 211; in the second scenario Jason took 3927, Erlang 252, and Scala 6066. Finally for Scenario 3, $N = 500$, Jason took 12, Erlang 0.5 and Scala 4 seconds.

3. ANALYSIS OF THE RESULTS

Jason and Scala had an even distribution of core load, while Erlang uses for the most part only one core. However we must consider that Scenario 1 does not require concurrency, so it is acceptable that mostly one of the cores is used. Moving to the CPU-load for Scenario 2, we can see that Erlang starts to use all of the 4 cores evenly, with both Erlang and Jason using a lot more of core load than in Scenario 1, but Scala remains on the 30% mark, which explains some of its poor performance regarding elapsed time in Scenario 2. This might have been caused by an inadequate underlying runtime management of multi-core usage, unless there are relevant configuration parameters that we failed to find out.

Where memory was considered, Jason and Erlang had the expected increase in used memory with the increase in the number of workers. Jason has the highest memory usage of the three languages, which was predictable since each agent has a more complex internal structure than an actor. Surprisingly, Erlang did not show any difference regarding memory usage from Scenario 1 to Scenario 2, and while Scala showed the expected results for Scenario 1, when observing Scenario 2 the memory usage was all over the place, again probably because of the poor core usage.

We use scale factors to compare the languages: scale factors represent the proportional increase in time when scaling up the experiment configurations such as number of token passes, and denotes the degradation of performance. For example, a scale factor of 9.12 was obtained for Scala in Scenario 1, when going from 5m to 50m token passes; this was calculated by dividing 211.36s, its elapsed time in 50m, by 23.184s, its elapsed time in 5m. Next, we show the difference between the scale factor from N = 500k to 5m and the scale factor from N = 5m to 50m in Scenario 1: Scala had the lowest, 2.21 (i.e. the scale factor from configuration 5m to 50m, 9.12, minus the scale factor from 500k to 5m, 6.91) compared to 2.5 for Erlang and 2.38 for Jason. However when looking at the values for Scenario 2, Erlang takes the lead with a difference between configurations of only 0.33 compared to 0.45 of Jason and 2.2 of Scala. Regardless of the scenario being concurrent or not, Jason manages to follow closely the performance of the actor languages, and even surpasses them by a large margin when considering the scale factors for passing from a non-concurrent to a concurrent scenario, having a scale factor 3 times lower than Erlang, and 5 times lower than Scala. The scale factors for Scenario 3 shows Scala as the most reactive language, followed by Jason and then Erlang.

Having its own virtual machine and runtime environment execution certainly helps Erlang to achieve the performance presented in this paper, although we cannot attest to how far it may affect its overall performance. Clearly there are advantages in using Java and the JVM for Jason and Scala, but this poses a limit to the performance that they can achieve.

Where scaling was considered, Scala made justice to its name and did better in almost all the cases, while Erlang stood distant as the one with significantly better performance than the other two languages at both elapsed time and memory used in the current scenarios, which is expected as it was created for industrial development. Jason, as a representative from a "heavier" paradigm, did not disappoint, following closely on both aspects, scalability and performance, and even excelling at some aspects in comparison to the two actor languages, showing that agent-oriented programming languages can perform surprisingly close to its predecessors as far as communication is concerned.

4. CONCLUSION

Future work includes running the experiments reported here on machines with a higher number of cores, and analysing new aspects such as giving priority to some of the tasks assigned by communication. Furthermore, we intend to benchmark other agent programming languages, and expect to work on a more qualitative comparison based on the principles of programming languages. There has been very little research on benchmarking for agent programming languages, so we expect to report various other results and analyses in the near future, and we also expect to see similar efforts for the variety of existing agent programming languages.

5. ACKNOWLEDGEMENTS

We are grateful for the support given by CNPq (grant numbers 307924/2009-2 and 307350/2009-6) and by CAPES.

6. REFERENCES

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, 2009.

[4] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.

[5] P. Haller and F. Sommers. *Actors in Scala*. Artima Incorporation, USA, 2012.