Investigating the Scalability Limits of Distributed Erlang

Amir Ghaffari

The University of Glasgow
School of Computing Science, G12 8QQ, UK
Amir.Ghaffari@glasgow.ac.uk
http://www.release-project.eu

Abstract

With the advent of many-core architectures, scalability is a key property for programming languages. Actor-based frameworks like Erlang are fundamentally scalable, but in practice they have some scalability limitations.

The RELEASE project aims to improve the scalability of Erlang on emergent commodity architectures with 10^5 cores. This paper investigates the scalability limits of distributed Erlang on up to 150 nodes by using *DE-Bench*. We discuss the design and implementation of DE-Bench, a scalable peer-to-peer benchmarking tool for distributed Erlang.

Our benchmarking results demonstrate that the frequency of global commands limits the scalability of distributed Erlang. There is also a common belief that distributed Erlang does not scale on large architectures with hundreds of nodes and thousands of cores. We provide evidence against this belief and show that distributed Erlang scales linearly up to 150 nodes and 1200 cores with relatively heavy data and computation loads when no global command is made.

Measuring the latency of commonly-used distributed Erlang commands reveals that the latency of *rpc* calls rises as cluster size grows. Our results also show that server processes like *gen_server* and *gen_fsm* have low latency and good scalability.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Distributed programming

Keywords Scalability; Benchmarking; Distributed Erlang

1. Introduction

The trend toward horizontally scalable architectures such as clusters, grids, and clouds will continue because they offer scalable hardware platform in a cost-effective way [1]. These scalable infrastructures typically consist of loosely-connected commodity servers in which node and network failures are common. To take full advantage of such architectures, the need for reliable scalable programming paradigms is essential.

Erlang'14, September 05, 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-3038-1/14/09 . . . $15.00.
http://dx.doi.org/10.1145/2633448.2633449

Recently, Erlang has become a popular platform to develop large-scale distributed applications, e.g. *Facebook chat backend*, *T-Mobile advanced call control services*, *Ericsson AXD301 ATM switch*, and *Riak DBMS* [2], [3]. This popularity is due to a combination of factors, including data immutability, share-nothing concurrency, asynchronous message passing based on the actor model, process's location transparency, and fault tolerance [4].

However, in practice the scalability of Erlang is constrained by aspects of the language and virtual machine [5]. For instance, our measurement shows that Riak 1.1.1 does not scale beyond 60 nodes because of overloaded single supervisor processes [6].

The RELEASE project aims to scale the Erlang's radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on emergent commodity architectures with 10^5 cores [5]. The RELEASE consortium works to scale Erlang at the virtual machine, language level, infrastructure levels, and to supply profiling and refactoring tools.

At the language level we aim to scale distributed Erlang, thus identifying the scalability bottlenecks of distributed Erlang is a key requirement before proceeding to next steps. For this purpose, we needed a benchmarking tool to perform the necessary measurements on a large-scale architecture with hundreds of nodes and thousands of cores. Since there was not such a tool to fulfil our requirement, we have designed and implemented *DE-Bench* [7]. DE-Bench, which stands for "Distributed Erlang benchmark", is a scalable peer-to-peer benchmarking tool that measures the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes. In the rest of the paper, we will explain how DE-Bench has been developed (Section 2) and what results have been achieved by employing it (Section 3). Finally, conclusions and future works are discussed (Section 4).

2. How Does the Benchmark Work?

2.1 Platform

The benchmark was carried out on the Kalkyl cluster at UPPMAX (Kalkyl has now been decommissioned and replaced with Milou) [8]. The Kalkyl cluster consists of 348 nodes with 2784 64-bit processor cores which are connected via 4:1 oversubscribed DDR Infiniband fabric. Nodes have 24GB RAM memory and 250 GB hard disk. The Kalkyl cluster is running Scientific Linux 6.0, a Red Hat Enterprise Linux. Each node comprises Intel quad-core Xeon 5520 2.26 GHz processors with 8MB cache. In this report, to avoid confusion with Erlang nodes (Erlang VM), we use the term *host* to refer to the Kalkyl nodes (physical machines). Erlang version R16B has been used in all our experiments.

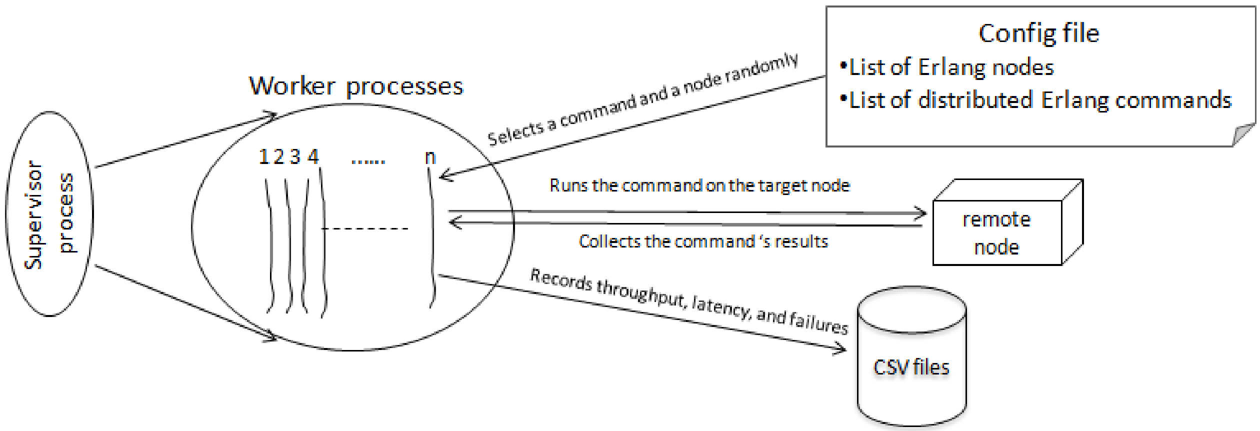

Figure 2: DE-Bench's Internal Workflow

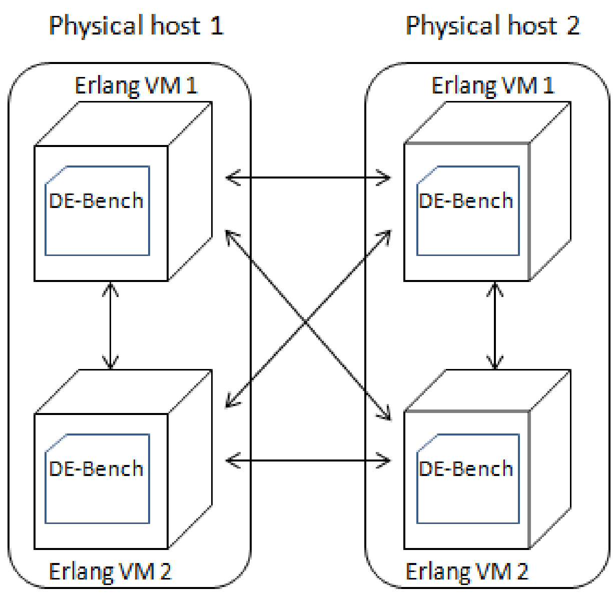

Figure 1: 2 hosts and 2 Erlang VMs per each host

2.2 Hosts and Nodes Organization

The same as an ordinary distributed Erlang application, our benchmark consists of a number of Erlang Virtual Machines (Erlang VMs) communicating with each other over a network. The benchmark is run on a cluster of hosts and there can be multiple Erlang VMs on each host, however, each Erlang VM runs only one instance of DE-Bench. For example, Figure 1 depicts a cluster with 2 hosts and 2 Erlang nodes (Erlang VMs) per host. As shown, a node can communicate with all the other nodes in the cluster regardless of whether nodes are located on the same host or not. DE-Bench follows a peer-to-peer model in which all nodes perform the same role independently, and so there is no specific node for coordination or synchronisation. The peer-to-peer design of DE-Bench improves scalability and reliability by eliminating central coordination and single points of failure.

2.3 The Design and Implementation of DE-Bench

To evaluate the scalability of distributed Erlang, we measure how adding more nodes to a cluster of Erlang nodes would increase the throughput. By *throughput* we mean the total number of successfully executed distributed Erlang commands per experiment. DE-Bench is based on Basho Bench, an open source benchmarking tool for Riak NoSQL DBMS [9].

One interesting feature of Erlang is its support for failure recovery. Processes in an Erlang application can be organised into a hierarchical structure in which parent processes monitor failures of their children and are responsible for their restart [10]. DE-Bench uses this feature to provide a fault-tolerant service. Figure 2 represents the internal workflow of DE-Bench. Initially, a supervisor process runs a number of worker processes in parallel on a node. The number of worker processes on each node is definable through a configuration file. As each host in the Kalkyl cluster has 8 cores, we run 40 worker processes on each node to exploit available cores. The supervisor process supervises all the worker processes and keeps them alive by restarting them in case of failure. A worker process randomly selects an Erlang node and a distributed Erlang command from the configuration file and runs that command on the selected node. There are three kinds of commands in DE-Bench:

- *Point-to-Point* (P2P): In P2P commands, a function with tunable argument size and computation time is run on a remote node. Figure 3 depicts the argument size and computation time for a P2P command. As the figure shows, firstly, a function with argument size *X* bytes is called. Then, a non-tail recursive function is run on the target node for *Y* microseconds. Finally, the argument is returned to the source node as result. P2P commands include *spawn*, *rpc*, and synchronous calls to server processes, i.e. *gen_server* or *gen_fsm*.
- *Global commands*: When a global command is run, all the nodes in a cluster get involved, and the result will be ready once the command runs successfully on all nodes. Global commands such as *global:register_name* and *global:unregister_name* are defined in the OTP *global* module.
- *Local commands*: In local commands such as *register_name*, *unregister_name* and *whereis_name*, just the local node gets involved and there is no need to communicate with other nodes in the cluster. The command *whereis_name* is a look up in the local name table regardless of whether it is from the *global* module or not.

After running a command, the latency and throughput of that command is measured and recorded in appropriate CSV files. The CSV files are stored on the local disk of each node to avoid disk access contention and network communication latency.

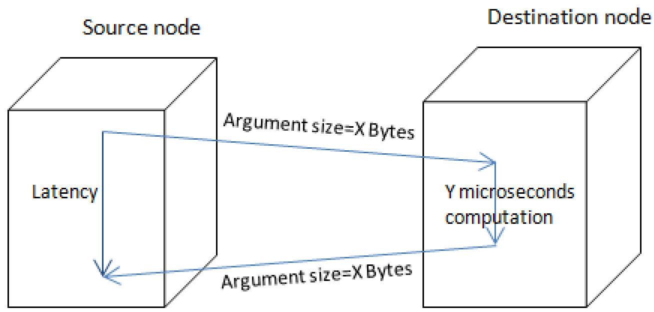

Figure 3: Argument size (X) and computation time (Y) in a P2P command

DE-Bench is extensible and one can easily add new commands into DE-Bench through the *de_commands.erl* module. At the time of writing this paper, the following commands are defined and measurable in DE-Bench:

1. P2P commands:
   (a) *spawn(Node, Fun)*: a function is called at a remote node with tunable argument size and computation time as depicted in Figure 3. Since spawn is an asynchronous call, the elapsed time is recorded after receiving an acknowledgment from the remote node.
   (b) rpc*(Node, Fun)*: synchronously calls a function at a remote node with tunable argument size and computation time.
   (c) *server process call*: makes a synchronous call to a generic server process (gen_server) or a finite state machine process (gen_fsm) by sending a request and waiting for the reply.
2. Global commands:
   (a) *global:register_name(Name, Pid)*: globally associates a name with a pid. The registered names are stored in name tables on every node in the cluster.
   (b) *global:unregister_name(Name)*: removes a globally registered name from all nodes in the cluster.
3. Local commands:
   (a) *register_name(Name, Pid)*: associates a name with a process identifier (pid).
   (b) *unregister_name(Name)*: removes a registered name, associated with a pid.
   (c) *whereis(Name)*: returns the pid registered with a specific name.
   (d) *global:whereis(Name)*: returns the pid associated with a specific name globally. Although, this command belongs to *global* module, it falls in local commands because it does a lookup in the local name table.

These commands are not used at the same rate in a typical distributed Erlang application. For instance, P2P commands such as *spawn* and *rpc* are the most commonly used ones and global commands like *register_name* and *unregister_name* are used much less than the others. Thus, to generate more realistic results, we can use each command with a different ratio.

In Erlang, a process identifier (pid) only can be registered once, otherwise an exception is thrown. To prevent the exception, three internal states are defined for a worker process to ensure that after registering a process, all necessary commands like *whereis_name* and *unregister_name* will be executed afterward. Figure 4 shows the states that a worker process follows to avoid duplicate registration exception. As shown, P2P commands do not change the current state (*state1*). The commands *whereis_name(Name)* and *unregister_name(Name)* are ignored unless they come after *register_name(Name, Pid)*. After running a *register_name(Name, Pid)* command, both *whereis_name(Name)* and *unregister_name(Name)* will be run respectively. To avoid name clashes, a timestamp function is used to generate globally unique names for processes in a cluster.

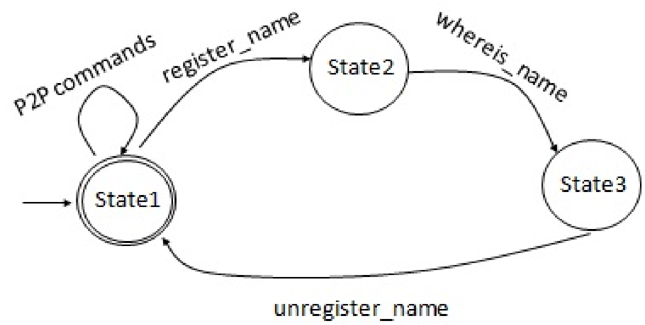

Figure 4: Worker Process' Internal States

3. Benchmarking Results

In this section, we employ DE-Bench to measure the scalability of distributed Erlang from different perspectives. In the scalability benchmark, we measure the throughput for different sizes of Erlang clusters and observe how adding more Erlang nodes to a cluster affects the throughput.

The benchmark is conducted on 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100-node clusters and measures the throughput by counting successful operations over the duration of an experiment. The duration of an experiment is specified in the configuration file. All the experiments in this paper run for 5 minutes. There is one Erlang VM on each host and as always one DE-Bench instance on each VM. After the end of each experiment, the generated CSV files from all participating nodes are collected and aggregated to find out the total throughput and failures. For example, for benchmarking a 70-node cluster, 70 instances of DE-Bench are run simultaneously and consequently they will generate 70 CSV files which need to be aggregated to find out the total throughput of the 70-node cluster. To provide reliable results, all experiments are run three times and the middle values are represented in diagrams.

We will measure following aspects of the scalability of distributed Erlang:

1. **Global Commands**: As illustrated previously in Section 2.3, in global commands all nodes in the cluster get involved. This feature of global commands could make them a bottleneck for scalability. To find out the effects of global commands on the scalability of distributed Erlang, we run the measurements with different percentages of global commands.
2. **Data Size**: As shown in Figure 3, the argument size of P2P commands is tunable. To understand the effect of data size on the scalability and performance of an Erlang cluster, we run the benchmark with different argument sizes.

3. **Computation Time**: As with argument size, the computation time of P2P commands is also tunable in DE-Bench (Figure 3). We investigate the effect of computation time on both scalability and performance of distributed Erlang.

4. **Data Size & Computation Time**:

   There is a common belief that distributed Erlang does not scale in a large distributed environment with hundreds of nodes and thousands of cores. To assess this belief, we measure how distributed Erlang scales up to 150 nodes and 1200 cores with relatively heavy data and computation loads.

5. **Server Process**: There are two popular types of server process in Erlang/OTP: generic server processes (*gen_server*) and finite state machine processes (*gen_fsm*). This section will inspect the scalability of these server processes, and try to find out whether the server processes are bottlenecks for the scalability of distributed Erlang.

3.1 Global Commands

To find out how global commands affect the scalability of distributed Erlang, we run the benchmark with different frequencies of global commands. The following commands are used in the measurement:

1. P2P commands: *spawn* and *rpc* with 10 bytes argument size and 10 microseconds computation time
2. Global commands: *global:register_name* and *global:unregister_name*
3. Local commands: *global:whereis(Name)*

Figure 5 shows how frequency of global commands limits the scalability of distributed Erlang. As we see from the diagram, scalability becomes more limited as more global commands are used. For example, when 0.01 percentage of global commands are used (the dark blue curve) , i.e. 1 global command per 10000 P2P commands, distributed Erlang doesn't scale beyond ≈60 nodes.

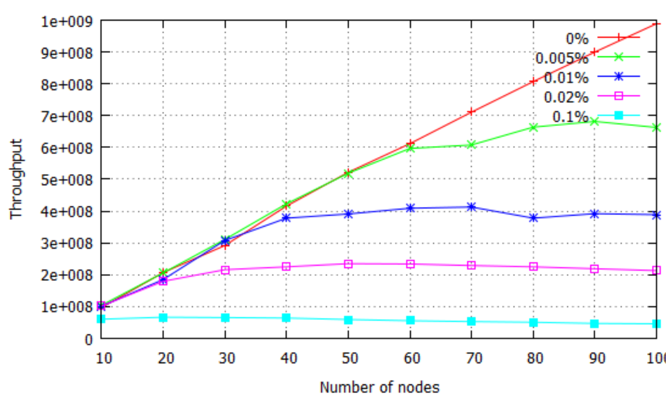

Figure 5: Scalability vs. Percentage of Global Commands

Figure 6 represents the latency of all the commands that we used in this measurement. The diagram reveals that the latency of both global commands, i.e. *global:register_name* and *global:unregister_name*, increases dramatically when cluster size grows. For example, we see from the figure that a globally name registration on a 100-node cluster takes ≈20 seconds which is a considerably long time. The registered names are stored in name tables on every node and these tables are strongly consistent, which means that an update is considered complete only when all nodes have acknowledged it [11]. This lock mechanism for updating the replicated information becomes a bottleneck on large clusters. As we see from Figure 6, the other commands' latencies (i.e. *spawn*, *rpc*, and *whereis*) are very low and negligible in comparison with the global ones.

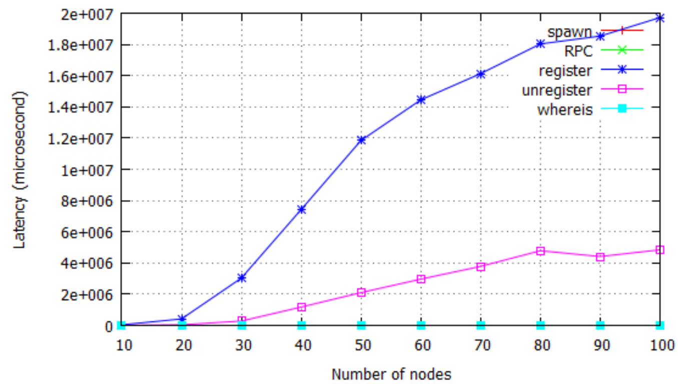

Figure 6: Latency of Commands

3.2 Data Size

This section studies the effect of data size on the scalability of distributed Erlang. As shown in Figure 3, P2P commands have two configurable parameters, i.e computation time and argument size. In this benchmark, the computation time is constant while the argument size of P2P commands change for different experiments. The following commands are used in this benchmark:

- P2P commands, i.e. *spawn* and *rpc*, with 10, 100, 1000, and 10000 bytes argument size and 10 microseconds computation time

Figure 7 represents the scalability of distributed Erlang for different data sizes. The diagram shows that as argument size increases, the performance and scalability decrease. For example the best scalability belongs to 10 bytes data size (the red curve) and the worst scalability belongs to 10K bytes data size (the pink curve).

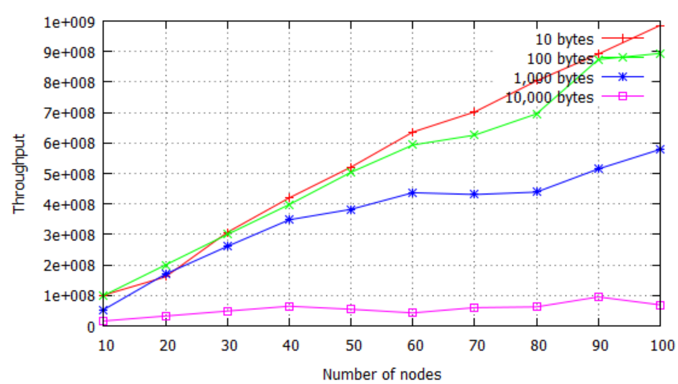

Figure 7: Scalability vs. Data Size

3.3 Computation Time

This section tries to find out how computation time affects the scalability of distributed Erlang. In this benchmark, the argument size of P2P commands is constant while the computation time changes for different experiments. The following commands are used in the benchmark:

- P2P commands, i.e. *spawn* and *rpc*, with 10 bytes argument size and 10, 1000, and 1000000 microseconds computation time.

Figure 8 represents the scalability of distributed Erlang for different computation times. The diagram shows that as we increase the computation time, performance and scalability degrade. The best scalability achieved for 10 microseconds computation time, and 1 second computation time shows the worst scalability. This is expected because for larger computation time, worker processes should wait longer for their response and consequently spend most of the time idly.

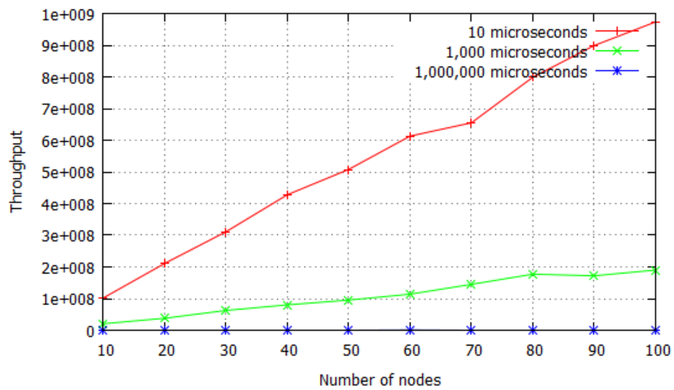

Figure 8: Scalability vs. Computation Time

3.4 Data Size & Computation Time

Previously, we have seen the individual effects of data size and computation time on the scalability of distributed Erlang (Sections 3.2 and 3.3). In this section we aim to discover how distributed Erlang scales when both data size and computation time are relatively large.

The following commands are used in the benchmark:

- P2P commands, i.e. *spawn* and *rpc*, with 1000 bytes argument size and 1000 microseconds computation time.

We chose 1000 bytes for the argument size because it is a relatively large size for an inter-process communication. Also, running a non-tail recursive function for 1000 microseconds can be considered as a relatively computation-intensive function. Accessing more than 100 nodes on the Kalkyl cluster is difficult because it's a highly demanded and busy cluster. But we could run this benchmark up to 150 nodes and 1200 cores (8 cores per each node) to see how distributed Erlang scales on that size.

Figure 9 represents the benchmark results. As we see from the figure, distributed Erlang scales linearly up to 150 nodes under relatively heavy data and computation loads when no global command is made.

However, this doesn't mean that all P2P commands have the same scalability and performance. Figure 10 depicts the latency of *spawn* and *rpc* commands and it shows that the latency of *spawn* is much less in comparison with *rpc*. In the next section, we will discuss why *rpc* doesn't scale well.

3.5 Server Process

Our experience with Riak 1.1.1 shows how an overloaded server process could limit the scalability of a distributed Erlang application [6]. This section investigates the scalability of two common server processes in Erlang/OTP: generic server processes (*gen_server*) and finite state machine processes (*gen_fsm*). As mentioned in Section 2.3, a server process call is a P2P command.

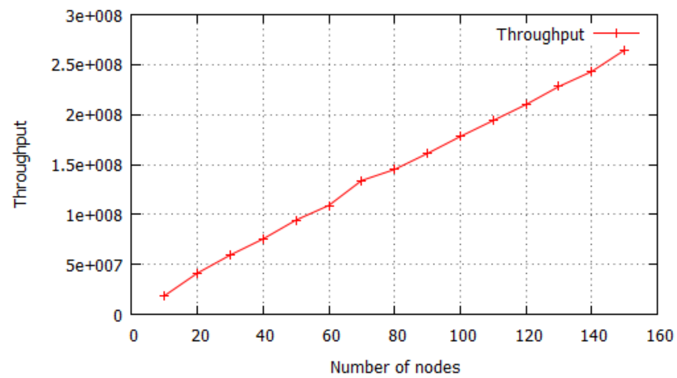

Figure 9: Scalability vs. Data Size & Computation Time

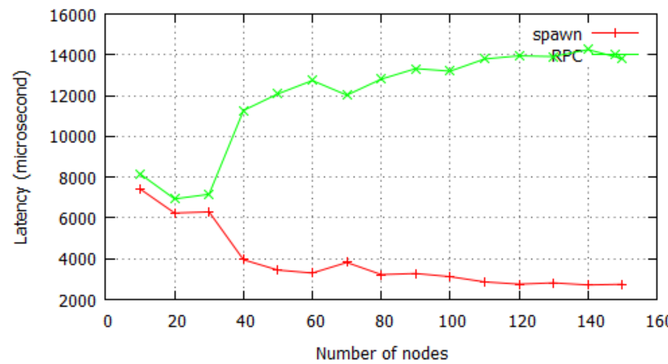

Figure 10: Latency of P2P commands

However, in this benchmark, we use all kinds of P2P commands, i.e. server process calls and non-server process calls such as *spawn* and *rpc*. Using all kinds of P2P commands makes us able to compare the scalability of server process calls with that of non-server process calls. The following commands are used in the benchmark:

- Non-server process P2P commands, i.e. *spawn* and *rpc*, with 10 bytes argument size and 1 microsecond computation time.
- Server process synchronous calls, i.e. *gen_server* and *gen_fsm*, with 10 bytes argument size and 1 microsecond computation time.

Figure 11 compares the scalability of distributed Erlang with different percentages of server process calls, i.e. 1% (red line), 50% (green line), and 100% (blue line). For example, when server process call is 1% (the red line), the other 99% of the calls are non-server process, i.e. *spawn* and *rpc*.

We see from the figure that as more server calls are used, the scalability improves. For instance, the best scalability is achieved when all calls are server process (the blue line) and the worst scalability occurs when 1% of calls are server process calls (the red line).

We also depict the latency of each command individually to understand which commands' latency increase when the cluster size grows. Figures 12 and 13 present the latency of commands that we used in the benchmark for 1% and 50% of server process calls. As the figures show, the latency of *rpc* calls rises when cluster size grows. However, the latency of the other commands such as *spawn*, *gen_server*, and *gen_fsm* do not increase as cluster size grows. We see that server processes scale well if they are used properly, i.e. not becoming overloaded as we experienced for Riak 1.1.1 [6].

To find out why *rpc*'s latency increases as the cluster size grows, we need to know more about *rpc*. Figure 14 shows how an *rpc* call is handled in Erlang/OTP. There is a generic server process (gen_server) on each Erlang node which is named *rex*. This process is responsible for receiving and handling all *rpc* requests that come to an Erlang node. After handling the request, generated results will be returned to the source node. In addition to user applications, *rpc* is also used by many built-in OTP modules, and so it can be overloaded as a shared service. In contrast with *rpc*, *spawn* is an asynchronous call and the request is handled by a newly-generated process on the target node. This feature makes *spawn* more scalable in comparison with *rpc*.

Alternatively, one also can implement one's own *gen_server* process to handle incoming requests synchronously. This approach reduces the possibility of overloading the process, since one's application is the only client for that server process.

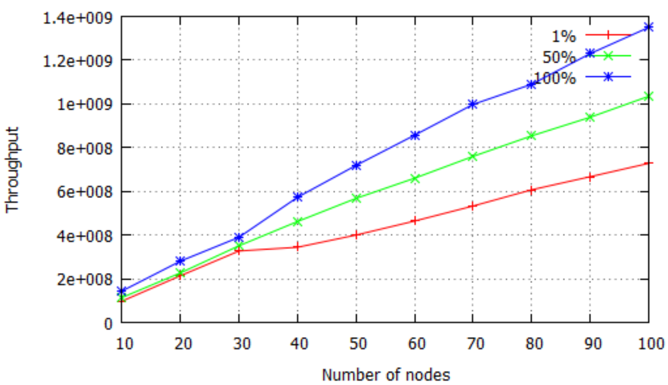

Figure 11: Scalability vs. Percentages of Server Process Calls

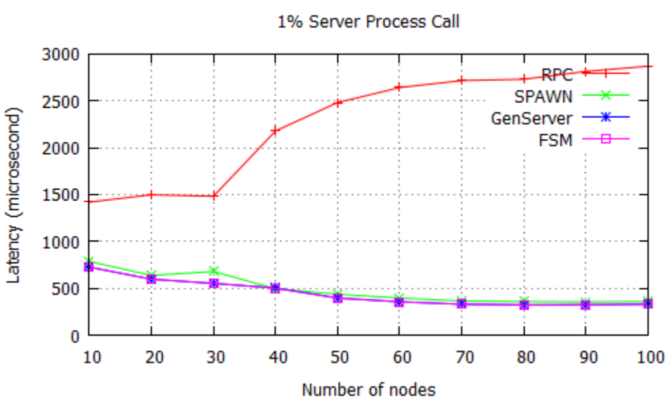

Figure 12: Latency of Commands for 1% Server Process Call

4. Conclusions and Future Works

This paper has investigated the scalability limits of distributed Erlang by employing DE-Bench. We have presented the design, implementation, and deployment of a scalable peer-to-peer benchmarking tool to measure the throughput and latency of distributed Erlang commands on a cluster of Erlang nodes.

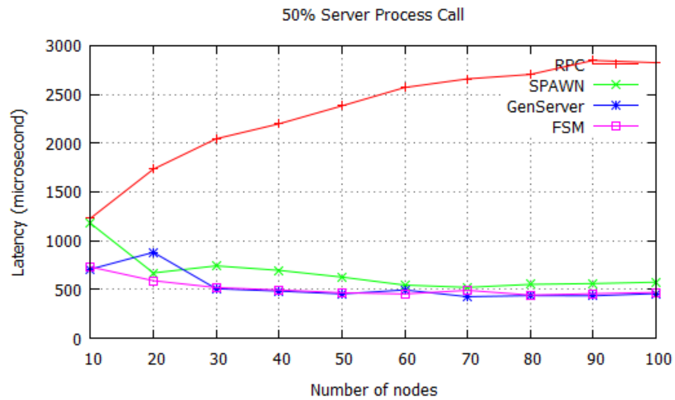

Figure 13: Latency of Commands for 50% Server Process Call

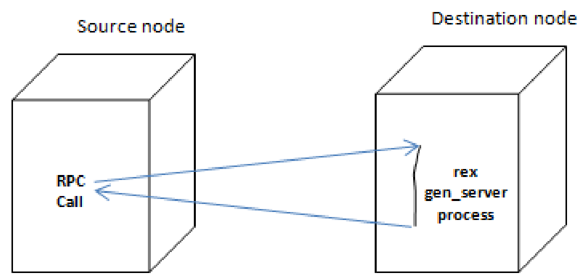

Figure 14: *rpc* call in Erlang/OTP

We have demonstrated that global commands are bottlenecks for the scalability of distributed Erlang (Figure 5). In ongoing work we are developing techniques to improve this limitation [12].

We have also measured the scalability of distributed Erlang with a relatively large data and computation size. We observed from Figure 9 that distributed Erlang scales linearly up to 150 nodes when no global command is made. Our results reveal that the latency of *rpc* calls rises as cluster size grows (Figure 10). This shows that *spawn* scales much better than *rpc* and using *spawn* instead of *rpc* in the sake of scalability is advised. Moreover, we have shown that server processes scale well and they have the lowest latency among all P2P commands (Figures 11, 12, and 13).

As future work, we are currently developing other scalable benchmarking applications to run on a larger architecture (i.e. Blue Gene/Q system).

A. Code Availability

DE-Bench's source code and the deployment scripts are publicly available on GitHub at https://github.com/amirghaffari/DEbench.

Acknowledgments

This research has been supported by the European Union grant IST-2011-287510 RELEASE.

The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under Project p2012172.

I thank Phil Trinder and Kenneth MacKenzie from Glasgow University for their insightful comments.

References

[1] Naidila Sadashiv and S. M Dilip Kumar. Cluster, Grid and Cloud Computing: A Detailed Comparison. The 6th International Conference on Computer Science & Education (ICCSE), Singapore, 2011.

[2] Ericsson AB. Who uses Erlang for product development?, 2012. URL http://www.erlang.org/faq/introduction.html

[3] Basho Technologies. Riak, 2014. URL http://basho.com/riak/

[4] Joe Armstrong. Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf, 1st edition, 2007. ISBN 193435600X.

[5] Olivier Boudeville, Francesco Cesarini, Natalia Chechina, Kenneth Lundin, Nikolaos Papaspyrou, Konstantinos Sagonas, Simon Thompson, Phil Trinder, and Ulf Wiger. RELEASE: a high-level paradigm for reliable large-scale server software. In Proceedings of the Symposium on Trends in Functional Programming, St Andrews, UK, 2012.

[6] Amir Ghaffari, Natalia Chechina, Phil Trinder, and Jon Meredith. Scalable Persistent Storage for Erlang : Theory and Practice. In Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, Boston, USA, 2013.

[7] Amir Ghaffari. DE-Bench, A Benchmark Tool for Distributed Erlang, 2014. URL https://github.com/amirghaffari/DEbench

[8] SNIC-UPPMAX. Clusters at UPPMAX, 2012. URL http://www.uppmax.uu.se/hardware

[9] Basho Technologies. Basho Bench, 2014. URL http://docs.basho.com/riak/latest/ops/building/benchmarking/

[10] Ericsson AB. Global Name Registration Facility, 2013. URL http://www.erlang.org/doc/man/global.html

[11] Ericsson AB. Supervisor, 2014. URL http://www.erlang.org/doc/man/supervisor.html

[12] Natalia Chechina, Phil Trinder, Amir Ghaffari, Rickard Green, Kenneth Lundin, and Robert Virding. The Design of Scalable Distributed Erlang. In Proceedings of the Symposium on Implementation and Application of Functional Languages, Oxford, UK, 2012.