The Design of Scalable Distributed Erlang

Natalia Chechina[1], Phil Trinder[1], Amir Ghaffari[1], Rickard Green[2], Kenneth Lundin[2], and Robert Virding[3]

[1] Heriot-Watt University, Edinburgh, EH14 4AS, UK
[2] Ericsson AB, 164 83 Stockholm, Sweden
[3] Erlang Solutions AB, 113 59 Stockholm, Sweden
`http://www.release-project.eu`

Abstract. The multicore revolution means that the number of cores in commodity machines is growing exponentially. Many expect 100,000 core clouds/platforms to become commonplace, and the best predictions are that core failures on such an architecture will become relatively common, perhaps one hour mean time between core failures. The RELEASE project aims to scale Erlang to build reliable general-purpose software, such as server-based systems, on massively parallel machines.
In this paper we present a design of Scalable Distributed (SD) Erlang – an extension of Distributed Erlang functional programming language for reliable scalability. The design focuses on three aspects of Erlang scalability: 1) scaling the number of Erlang nodes by eliminating transitive connections and introducing scalable groups (s_groups), 2) managing process placement in the scaled networks by introducing semi-explicit process placement, and 3) preserving Erlang reliability model.

Keywords: Erlang, functional programming, scalability, multi-core systems, massive parallelism

1 Introduction

General-purpose software is predominately written in mainstream programming languages, firmly planted in legacy software concepts, and the software industry is struggling for better ways to parallelise these languages. Current shared-memory technologies like OpenMP often work well for small scale problems, but applications are rapidly experiencing the inherent limitations of these technologies. MPI works well for large scale computations with a regular process structure and independent computations, for example classical High-Performance Computing (HPC) problems like computational fluid dynamics. However many general purpose applications have significant data dependencies, or irregular process structures. Simultaneously, increasing the number of cores increases the likelihood of failures: a system with 10^5 cores might experience a core failure every 50 minutes in addition to any other failures [31]. Handling partial failures in massively-parallel applications inevitably introduces dependencies between components and calls for coordination logic that cannot easily be expressed using current programming language technologies.

Erlang [11] is a functional programming language. Its concurrency-oriented programming paradigm is novel in being very high level, predominantly stateless, and having both parallelism and reliability built-in rather than added-on. Building on this success, the user uptake of Erlang is exploding around the world and shifting from its telecom base into other sectors. Currently Erlang/OTP has inherently scalable computation and reliability models, but in practice scalability is constrained by the transitive sharing of connections between all nodes and by explicit process placement. The former implies that the virtual machine maintains data structures quadratic in the number of nodes and the latter makes constructing large dynamic or irregular process structures challenging. Moreover programmers need support to engineer applications at this scale and existing profiling and debugging tools do not scale, primarily due to the volumes of trace data generated.

We target reliable scalable general purpose computing on stock heterogeneous platforms. Our application area is that of general server-side computation, e.g. a web or messaging server. This form of computation is ubiquitous, in contrast to more specialised forms such as traditional high-performance computing. Moreover, this is computation on stock platforms, with standard hardware, operating systems and middleware, rather than on more specialised software stacks on specific hardware.

To extend the Erlang concurrency-oriented paradigm to large-scale reliable parallelism (10^5 cores) we propose an extension to the Erlang language, Scalable Distributed Erlang (SD Erlang), for reliable scalability. Key goals in scaling the computation model are to provide mechanisms for controlling locality and reducing connectivity, and to provide performance portability. The goal in scaling the reliability model is to preserve Erlang's sophisticated and effective reliability mechanisms of first class processes and supervision behaviours in the presence of locality and connectivity controls. The SD Erlang name is used only as a convenient means of identifying the extensions we propose: we expect the extensions to become standard Erlang in the future.

The rest of the paper is organised as follows. We start with an overview of related work and background information (Section 2). For a language to scale in-memory and persistent data structures, together with computation must scale (Section 3). The primary Erlang in-memory data structure Erlang Term Storage (ETS) is implemented inside Erlang Virtual Machine (VM), so any scalability issues will be addressed by the VM team of the RELEASE project. In terms of persistent data structures we believe that such databases as Riak [3] and Cassandra [15] will be able to meet the target scalability requirements. The computation scalability is a language level issue and we address it by eliminating transitive connections of Erlang nodes and introducing a semi-explicit process placement (Sections 4). Finally, we discuss the design validation exemplars (Section 5) and provide conclusion together with the future work (Section 6).

2 Related Work

This section covers related work and provides background information. First, we discuss typical hardware architectures we might expect in the next 4-5 years in Section 2.1. Actor languages and Erlang functional programming language are covered in Sections 2.2 and 2.3 respectively. Finally, the RELEASE project overview is provided in Section 2.4.

2.1 Architecture Trends

To make predictions in computer science even for the next 4-5 years is a hard job as the field is very young and moves forward much faster than any other branch of science. However, to understand limitations of today Erlang we need to get an idea of typical hardware architectures that will use SD Erlang in the next few years. Currently the main factors that shape trends in computer architectures are as follows: memory size/bandwidth, energy consumption, and cooling. Below we provide a brief analysis of these factors and discuss their impact on the development of the hardware architectures.

Memory size/bandwidth. As the number of cores goes up the memory bandwidth goes down, and the larger number of cores shares the same memory the larger memory is required. DRAM-based main memory systems are about to reach the power and cost limit. Currently, the main two candidates to replace DRAM are Flash memory and Phase Change Memory (PCM). Both types are much slower than DRAM, i.e. 2^{11} and 2^{17} processor cycles respectively for a 4GHz processor in comparison with 2^9 processor cycles of DRAM, but the new technologies provide a higher density in comparison with DRAM [26].

Energy consumption and cooling are the main constrains for the high core density. Moreover, the cooling is also a limitation of the silicon technology scal-

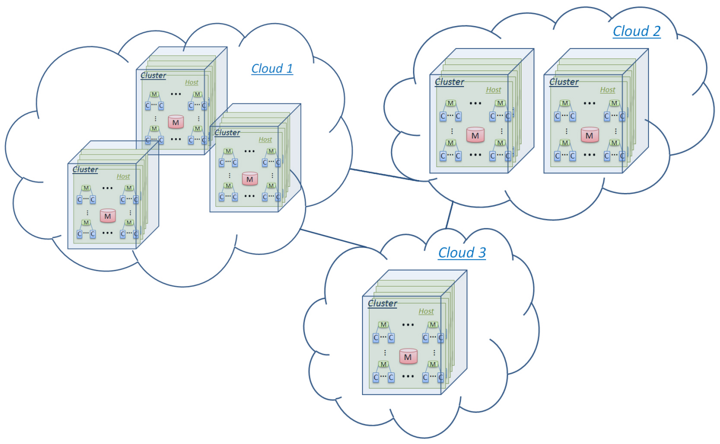

Fig. 1. A Typical Server Architecture

ing [37]. To save energy and maximize compute performance supercomputers exploit small and simple cores apposed to large cores. Many architectures, especially HPC architectures, exploit GPUs [10]. GPUs accelerate regular floating point matrix/vector operations. The high throughput servers that RELEASE targets do not match this pattern of computation, and GPUs are not exploited in the server architectures we target. The air cooling might be replaced by one of the following technologies: 2D and 3D micro-channel cooling [17], phase-change cooling [22], spot cooling [36], or thermal-electric couple cooling [29].

From the above we anticipate the following typical server hardware architecture. A host will contain ∼4–6 SMP core modules where each module will have ∼32–64 cores. Analysis of the Top 500 supercomputers that always lead the computer industry illuminating the next 4–5 year computer architecture trends allows us to assume that ∼100 hosts will be grouped in a cluster, and ∼1–5 clusters will form a cloud. Therefore, the trends are towards a NUMA architecture. The anticipated typical server architecture is presented in Figure 1.

2.2 Actor Languages and Frameworks

An actor model was introduced in 1973. Actors are independent entities that asynchronously exchange messages, and perform actions depending on these messages. There is a number of actor-based languages and frameworks such as Akka [2], Axum [20], E [28], Erlang [11], Kilim [30], Ptolemy [27], SALSA [34], Scala [4], and Smalltalk [16]. In this section we cover only Scala and Akka. Erlang is discussed in Section 2.3.

Scala is a statically typed programming language that combines features of both object-oriented and functional programming languages [4]. Scala has been designed to interact with mainstream platforms such as Java and C#. Currently, Scala is implemented on Java and .NET platforms. Scala has a pure oriented model similar to Smalltalk [16] where values are objects, and operations are messages. Operator names are treated as identifies, and identifies between two expressions are treated as method calls. Scala is also a functional language due to treating functions as values. The language supports such functions as nested, anonymous, curried, and higher order functions. Scala supports parameterisation, abstract members, and classes to model Erlang type actors [24].

Akka is an event driven middleware framework to build reliable distributed applications [2]. Akka is implemented in Scala. A fault tolerance in Akka is implemented using similar to Erlang 'Let it crash' philosophy and supervisor hierarchies [33]. An actor can only have one supervisor which is the parent supervisor but similar to Erlang actors can monitor each other. Due to possibility to create an actor within a different JVM two paths can be used to reach an actor: logical and physical. Logical path follows parental supervision links toward the root. Physical actor path starts at the root of the system where the actual actor object resides, and never spreads over multiple JVMs. Akka uses remote to local approach via optimisation [35]. In Akka multiple shreds can execute actions on shared memory. Like Erlang Akka does not support guaranteed delivery. A cluster support is planned to be introduced in Akka.

2.3 Erlang

Erlang is a functional general purpose concurrent programming language designed in 1986 at Ericsson computer science laboratory [11]. Erlang was influenced by a number of languages such as ML [21], Miranda [32], ADA [19], and Prolog [38]. Erlang was designed to meet requirements of distributed, fault-tolerant, massively concurrent, and soft-real time systems. Erlang is a dynamically typed language. Distributed Erlang was introduced to allow autonomous Erlang Virtual Machines (VMs) to work together when they are situated either on the same or different computers. In Erlang a collection of processes work together to solve a particular problem. The processes are lightweight and communicate with each other by exchanging asynchronous messages [39].

Erlang concurrency differs from the most other programming languages in that concurrency is handled by the language and not by the operating system [8]. Some of the principles of the Erlang philosophy are as follows. *Share nothing* implies that isolated processes do not share memory and variables are not reusable, i.e. once a value is assigned it cannot be changed. *Let it crush* is a non-defensive approach that lets failing processes to crash, and then other processes detect and fix the problem. The approach also provides clear and compact code [7].

2.4 RELEASE Project

The RELEASE project aims to scale the radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines. Concurrency-oriented programming is distinctive as it is based on highly-scalable lightweight processes that *share nothing.* The trend-setting concurrency-oriented programming model we use is Erlang/OTP. Erlang/OTP provides high-level coordination with concurrency and robustness built-in: it can readily support 10,000 processes per core, and transparent distribution of processes across multiple machines, using message passing for communication. Moreover, the robustness of the Erlang distribution model is provided by hierarchies of supervision processes which manage recovery from software or hardware errors.

Currently Erlang/OTP has inherently scalable computation and reliability models, but in practice scalability is constrained by the transitive sharing of connections between all nodes and by explicit process placement. Moreover programmers need support to engineer applications at this scale and existing profiling and debugging tools do not scale, primarily due to the volumes of trace data generated. In the RELEASE project we tackle these challenges working at three levels: evolving the Erlang virtual machine, evolving the language to Scalable Distributed (SD) Erlang, developing a scalable Erlang infrastructure.

3 SD Erlang Design Overview

In this section we provide our vision of requirements for Erlang scalability and principles behind design decisions taken in Section 4. We believe that for SD

Erlang to scale in-memory and persistent data structures together with computation must be taken into account.

Scalable in-memory and persistent data structures. When an application scales it requires support from in-memory and persistent data storages to handle a large number of processes and data. Therefore, in-memory and persistent data structures need to be able to scale to the same magnitude as the application that implores them (Section 3.1).

Scalable computation. From the analysis of the typical Erlang exemplars in Section 5 we have identified two main scalability issues. These are a fully connected network together with transitive connections and explicit placement. A fully connected network and transitive connections prevent a network of Erlang nodes to scale because it becomes not feasible to maintain a good performance in a network of more than a hundred of nodes. Whereas an explicit placement requires a programmer to be aware of all Erlang nodes in the network for every process which is again not feasible (Section 4).

To design SD Erlang we came up with two types of principles: general and reliable scalability. The general principles include our view on the aspects of the language that we want to preserve and implementation level of the modifications. The general principles are as follows.

- *Preserving the Erlang philosophy and programming idioms.*
- *Minimal language changes*, i.e. minimizing the number of new constructs but rather reusing of existing constructs.
- *Working at Erlang level* rather than VM level as far as possible.

The reliable scalability principles include concepts that we want to either preserve or avoid when scaling Erlang. They are as follows.

- *Avoiding global sharing*, i.e. global names, bottlenecks, and using groups instead of fully connected networks.
- *Introducing an abstract notion of communication architecture*, e.g. locality/affinity and sending disjoint work to remote hosts.
- *Avoiding explicit prescription*, e.g. replacing spawning on named node with spawning on group of nodes, and automating load management.
- *Keeping the Erlang reliability model* unchanged as far as possible, i.e. linking, monitoring, supervision.

3.1 Scalable Data Structures

Scalable In-Memory Data Structures. The primary Erlang in-memory data structure is Erlang Term Storage (ETS). ETS is a data structure to associate keys with values, and is a collection of Erlang tuples, i.e. tuples are inserted and extracted from an ETS table based on the key. An ETS is memory resident and provides large key-value lookup tables. Data stored in the tables is transient. ETS tables are implemented in the underline runtime system as a BIF inside the Erlang VM, and are not garbage collected. ETS tables are stored in a separate storage area not associated with normal Erlang process memory. The size of ETS tables depends on the size of a RAM. An ETS table is owned by the process that has created it and is deleted when a process terminates. The process can transfer the table ownership to another local process. The table can have the following read/write access [12]: 1) *private*, i.e. only the owner can read and write the table, 2) *public*, i.e. any process can read and write the table, 3) *protected*, i.e. any process can read the table but only the owner can write it.

ETS tables provide limited support for concurrent updates [11]. That is inserting a new element may cause a rehash of elements within the table; when a number of processes write or delete elements concurrently from the table the following outcomes are possible: a runtime error, `bad arg` error, or undefined behaviour, i.e. any element may be returned. As the number of SMP cores increases the number of Erlang nodes and processes also increase. This can lead to either a bottleneck if the table is private/protected or undefined outcomes if the table is public. ETS tables are implemented inside the Erlang VM, and hence any scalability issues will be addressed by the VM team of the RELEASE project.

Scalable Persistent Data Structures. We have analysed a number of DataBase Mangament Systems (DBMSs) for Erlang such as Mnesia [23], Riak [3], CoachDB [5], and Cassandra [15]. Below we have summarised the main principles and desirable features required for highly available and scalable databases. We believe that such DBMSs as Riak and Cassandra will be able to meet the target scalability requirements [25].

Fragmenting data across distributed nodes. 1) Decentralized approaches are preferable as they show a better throughput by spreading the load over a large number of servers and increase availability by removing a single point of failure; 2) The placement of replicas should be handled systematically and automatically, i.e. location transparency. 3) A node departure or arrival should only affect the node immediate neighbours whereas other nodes remain unaffected.

Replicating data across distributed nodes. 1) A decentralized model such as P2P is desirable. 2) An asynchronous replication where consistency is sacrificed to achieve higher availability, i.e. from the CAP theorem [18] a database cannot simultaneously guarantee consistency, availability, and partition-tolerance.

Partition tolerance, i.e. a system continues to operate despite of connection loss between some nodes. We anticipate the target architecture to be loosely coupled; therefore, partition failures are highly expected. Again from the CAP theorem by putting stress on availability we must sacrifice strong consistency to achieve partition-tolerance and availability.

4 Scalable Actor Computation

We have identified two main Distributed Erlang issues that prevent scalability, these are transitive connections and explicit placement. Transitive connections are the reason of inability to scale beyond a hundred of nodes, whereas explicit placement is a restriction that prevents a programmer to easily manipulate the available nodes.

In this section we introduce extension of the Distributed Erlang – Scalable Distributed (SD) Erlang – to effectively operate when the number of hosts, cores, nodes, and processes scales. We start with a discussion of scalability limitations of Distributed Erlang in Section 4.1. To scale a network of Erlang nodes we introduce scalable groups (s_groups) in Section 4.2. S_groups aim to eliminate transitive connections, i.e. nodes have transitive connections with nodes of the same s_group and non-transitive connections with other nodes. To manage process placement we introduce semi-explicit placement and `choose_node/1` function in Section 4.3. So that a process can be spawned, for example, to an s_group or a particular communication distance.

4.1 Distributed Erlang Scalability Limitations

Figure 2 illustrates Erlang's support for concurrency, multicores and distribution. A blue rectangle represents a host with an IP address, and a red arc represents a connection between nodes. Multiple Erlang processes may execute in a node, and a node can exploit multiple processors, each having multiple cores. Erlang supports single core concurrency as a core may support as many as 10^8 lightweight processes [1]. In the Erlang distribution model a node may be on a remote host, and this is almost entirely transparent to the processes. Hosts need not be identical, nor do they need to run the same operating system.

Erlang currently delivers reliable medium scale concurrency, supporting up to 10^2 cores, or 10^2 distributed memory processors. However, the scalability of a distributed system in Erlang is constrained by the transitive sharing of connections between all nodes and by explicit process placement. The transitive sharing of connections between nodes means that the underlying implementation needs to maintain data structures that are quadratic in the number of processes, rather than considering the communication locality of the processes. While it is possible to explicitly place large numbers of processes in a regular, static way (as for conventional HPC computations), explicitly placing the irregular or dynamic processes required by many servers and other general purpose applications is far more challenging. Erlang/OTP has a world leading language level reliability. The challenge is to maintain this reliability at massive scale.

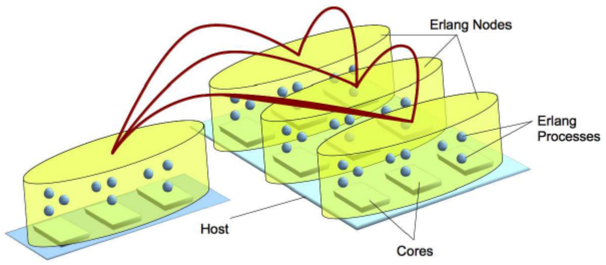

Fig. 2. Connections in s_groups

4.2 Network Scalability

To allow scalability of networks of nodes the existing scheme of transitive connection sharing in Distributed Erlang should be changed as it is not feasible for a node to maintain connections to tens of thousands of nodes, i.e. the larger the network of Erlang nodes the more 'expensive' it becomes on each node to keep up-to-date replications of global names and global states, and periodic checking of connected nodes. Instead we propose to use overlapping scalable groups (s_groups) where nodes would have transitive connections within their s_group and non-transitive connections with nodes of other s_groups. The idea of s_groups is *similar* to the existing in Distributed Erlang hidden global_groups in the following: 1) each s_group has its own name space; 2) transitive connections are only with nodes of the same s_group. The *differences* with hidden global_groups are in that 1) a node can belong to an unlimited number of s_groups, and 2) information about s_groups and nodes is not globally collected and shared.

In SD Erlang nodes with no asserted s_group membership belong to a notional group *G0* that follows Distributed Erlang rules and allows backward compatibility with Distributed Erlang. By the backward compatibility we mean that when nodes run the same VM version they may use or not use s_groups and still be able to communicate with each other. Therefore, s_groups are not compulsory but rather a tool a programmer may use to scale a network of nodes.

Types of s_groups. To allow programmers flexibility and provide an assistance in grouping nodes we propose s_groups to be of different types, i.e. when an s_group is created a programmer may specify parameters against which a new s_group member candidate can be checked. If a new node satisfies an s_group restrictions then the node becomes the s_group member, otherwise the membership is refused. The following parameters can be taken into account: communication distance, security, available code, specific hardware and software requirements. A programmer may also introduce his/her own s_group types on the basis of some personal preferences. The information about specific resources can be collected by introducing node self awareness, i.e. a node is aware of its execution environment and publishes this information to other nodes.

We also propose the following features: *a)* a node can establish a direct connection with any other node, *b)* nodes can have short lived connections, and *c)* a host can have an unlimited number of nodes. We do not consider any language constructs to provide programmers control over cores, i.e. the lowest level a programmer may control in terms of where a process can be spawned is a node.

s_group Functions. We propose a number of functions to support s_group employment, some of them are listed below. The functions may be changed during the development. The final implementation will be decided during actual SD Erlang code writing and will depend on the functions that programmers find useful.

1. Creating a new s_group, e.g.
   ```
   new_s_group(S_GroupName, [Node]) -> ok | {error, ErrorMsg}
   ```

2. Adding new nodes to an existing s_group, e.g.
   `add_node_s_group(S_GroupName, [Node]) -> ok | {error, ErrorMsg}`
3. Monitoring all nodes of an s_group, e.g.
   `monitor_s_group(S_GroupName) -> ok | {error, ErrorMsg}`
4. Listing nodes of a particular s_group, e.g.
   `s_group_nodes(S_GroupName) -> [Node] | {error, ErrorMsg}`
5. Connecting to all nodes of a particular s_group, e.g.
   `connect_s_group(S_GroupName) -> [boolean() | ignored]`
6. Disconnecting from all nodes of a particular s_group, e.g.
   `disconnect_s_group(S_GroupName) -> boolean() | ignored`

Example. Assume we start six nodes A, B, C, D, E, F, and initially the nodes belong to no s_group. Therefore, all these nodes belong to notional group $G0$ (Figure 3(a)). Fist, on node A we create a new s_group $G1$ that consists of nodes A, B, and C, i.e. `new_s_group(G1, [A, B, C])`. Note that a node belongs to group $G0$ only when this node does not belong to any s_group. When nodes A, B, and C become members of an s_group they may still keep connections with nodes D, E, F but now connections with these nodes are non-transitive. If connections between nodes of s_group $G1$ and group $G0$ are time limited then the non-transitive connections will be lost over some time (Figure 3(b)). Then on node C we create s_group $G2$ that consists of nodes C, D, and E. Nodes D, and E that now have non-transitive connections with node F may disconnect from the node using function `disconnect_s_group(G0)`. Figure 3(c) shows that node C does not share information about nodes A and B with nodes D and E. Similarly, when nodes B and E establish a connection they do not share connection information with each other (Figure 3(d)).

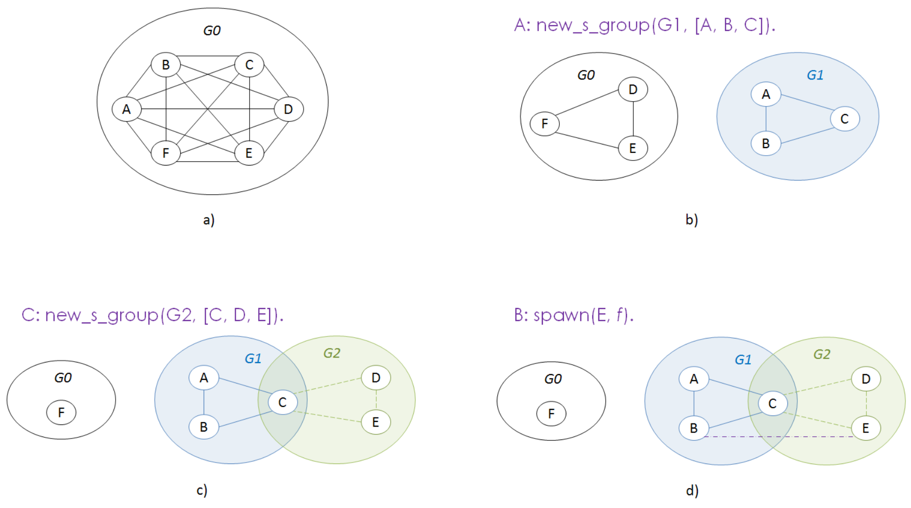
G0
A
B
C
D
E
F
a)
A: new_s_group(G1, [A, B, C]).
G0
G1
b)
C: new_s_group(G2, [C, D, E]).
G0
G1
G2
c)
B: spawn(E, f).
G0
G1
G2
d)

Fig. 3. Connections in s_groups

4.3 Semi-Explicit Placement

For some problems, like matrix manipulations, optimal performance can be obtained on a specific architecture by explicitly placing threads within the architecture. However, many problems do not exhibit this regularity. Moreover, explicit placement prevents performance portability: the program must be rewritten for a new architecture, a crucial deficiency in the presence of fast-evolving architectures. We propose a dynamic semi-explicit and architecture aware process placement mechanism. The mechanism does not support the migration of processes between Erlang nodes. The semi-explicit placement is influenced by Sim-Diasca process placement [14] and architecture aware models [9].

In Sim-Diasca a computing load is induced by a simulation and needs to be balanced over a set of nodes. By default, model instances employed by Erlang processes are dispatched to computing nodes using a round-robin policy. The policy proved to be sufficient for most basic uses, i.e. a large number of instances allows an even distribution over nodes. However, due to bandwidth and latency for some specifically coupled groups of instances it is preferable for a message exchange to occur inside the same VM rather than between distant nodes. In this case, a developer may specify a placement hint when requesting a creation of instances. The placement guarantees that all model instances created with the same placement hint are placed on the same node. This allows the following: a) to co-allocate groups of model instances that are known to be tightly coupled, b) to preserve an overall balancing, and c) to avoid model level knowledge of the computing architecture. In [9] to limit the communication costs for small computations, or to preserve data locality, the authors proposes to introduce communication levels and specify the maximum distance in the communication hierarchy that the computation may be located. Thus, sending a process to level 0 means the computation may not leave the core, level 1 means a process may be located within the shared memory node, level 2 means that process may be located to another node in a Beowulf cluster, and level 3 means that a process may be located freely to any core in the machine.

For the SD Erlang we propose that a process could be spawned either to an s_group, to s_groups of a particular type, or to nodes on a given distance. From a range of nodes the target node can be picked either randomly or on the basis of load information.

Load Management. When a node is picked on the basis of load an important design decision is the interaction between two main load management components, i.e. information collection and decision making. The components can be either merged together and implemented as one element or implemented independently from each other. We propose to implement information collection and decision making as one element, i.e. a load server. Its responsibility will be collecting information from the connected nodes and deciding where a process can be spawned when a corresponding request arrives. It seems that one load server per node is an appropriate number of load servers per node for SD Erlang. In this case the decisions are made within the node (in comparison with one load server per s_group, a host, and a group of hosts) and load information redundancy level is not too high (in comparison with one per group of processes and a multiple number of load servers per node).

chose_node. We propose to introduce a new function `chose_node/1`. The function will return a node ID where the process should be spawned. The node will be picked on the basis of restrictions identified by the programmer, e.g. s_groups, s_group types, minimum/maximum/ideal communication distances. The function can be written in SD Erlang as follows.

```
chose_node(Restrictions) -> node()
Restrictions = [Restriction]
Restriction = {s_group_name, S_GroupName}
| {s_group_type, S_GroupType}
| {min_dist, MinDist :: integer() >= 0}
| {max_dist, MaxDist :: integer() >= 0}
| {ideal_dist, IdealDist :: integer() >= 0}
```

We deliberately introduce `Restrictions` as a list of tuples to allow the list of restrictions to be extended in the future. A process spawning may look as follows:

```
start() ->
TargetNode = chose_node([{s_group, S_Group}, {ideal_dist, IdealDist}]),
spawn(TargetNode, fun() -> loop() end).
```

4.4 Summary.

To enable scalability of network of nodes we propose a *new s_group library* for Erlang. 1) Grouping nodes in s_groups where s_groups can be of different types, and nodes can belong to many s_groups. 2) Transitive connections between nodes of the same s_group and non-transitive connections with all other nodes. Direct non-transitive connections are optionally short lived, e.g. time limited.

To enable *semi-explicit placement* and load management we propose the following constructs. 1) Function `chose_node(Restrictions) -> node()` where the choice of a node can be restricted by a number of parameters, such as s_groups, s_group types, and communication distances. 2) The nodes may be picked randomly or on the basis of load. We assume that when a process is spawned using semi-explicit placement it is a programmer responsibility to ensure that the prospective target node has the required code. If the code is missing an error is returned.

5 Design Validation Exemplars

To validate SD Erlang design presented in Section 4 we have done a theoretical validation of five Erlang applications: Sim-Diasca [14], Orbit, Mandelbrot set, Moebious, and Riak [3]. Here, we only discuss Moebious. The description of the remaining applications is presented in [25].

Moebius is a continuous integration system recently developed by Erlang Solutions. Moebius aims to provide users an automated access to various cloud

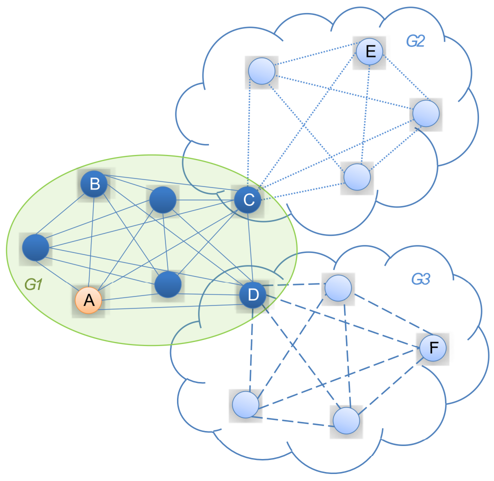

Fig. 4. Mandelbrot Set Node Grouping

providers such as Amazon EC2. The system has two types of nodes: master node and moebius agents. The master node collects global information and makes decisions to start and stop nodes. The moebius agents are located on the utilised nodes and periodically send state information to the master node. Currently, moebius agents are only connected to the master node via HTTP but in the future there are plans to move Moebius to SD Erlang and build a hierarchical master structure.

A top level Moebius algorithm is as follows. A user is asked to indicate the requirements, e.g. hardware configuration, software configuration, and a description on how the initial master node should start the remaining nodes in the cloud (Figure 4). Thus, a hierarchical organization of nodes can be easily set from top to bottom. Additional s_groups from nodes of different levels can also be formed if needed. An SD Erlang Moebius may require the following. 1) The s_groups may be grouped on the basis of different factor such as communication locality, security, and availability of a particular hardware or software; therefore, custom s_groups types are required. 2) Nodes and s_groups will dynamically appear and disappear depending on the user current requirements. 3) Moebius master nodes most probably will be organised in a hierarchical manner, so nodes will not need to directly communicate with each other. 4) The number of s_groups most probably will be much less than the number of nodes.

Table 1 provides a summary of the exemplar requirements for scalable implementations. Thus, s_groups may be either static, i.e. once created nodes rarely leave and join their s_groups, or dynamic, i.e. nodes and s_groups are constantly created and deleted from the network. S_groups may be formed on the basis of locality (Sim-Diasca and Mandelbrot set), Hash table (Orbit), Preference list (Riak), or programmers' and users' preferences (Moebius). *'Yes/No'* indi-

No	**Property**	**Sim-Diasca**	**Orbit**	**Mandel-brot set**	**Moebius**	**Riak**
			s_groups			
1	**Static/Dynamic**	Static	Static	Static	Dynamic	Dynamic
2	**Grouping**	Locality	Hash table	Locality	Multiple	Preference list
3	**Custom types**	Yes	No	No	Yes	No
			General			
4	**Number of nodes and s_groups**	$N_g << N_n$	$N_g << N_n$	$N_g << N_n$	$N_g << N_n$	$N_g >= N_n$
5	**Short lived connections**	Yes	Yes	No	No	Yes
6	**Semi-explicit placement**	Yes	No	Yes	No	No

Table 1. Exemplar Summary

cates whether an application requires a particular feature. For instance, some scalable exemplars require custom s_group types, short lived connections, and semi-explicit placement. Such applications like Riak may have the number of s_groups compatible with the number of nodes.

6 Conclusion and Future Work

This paper presents the design of Scalable Distributed (SD) Erlang: a set of language-level changes that aims to enable Distributed Erlang to scale for server applications on commodity hardware with at most 10^5 cores. The core elements of the design are to provide scalable in-memory data structures, scalable persistent data structures, and a scalable computation model. The scalable computation model has two main parts: scaling networks of Erlang nodes and managing process placement on large numbers of nodes. To tackle the first issue we have introduced s_groups that have transitive connections with nodes of the same s_group and non-transitive connections with nodes of other s_groups. To resolve the second issue we have introduced semi-explicit placement and `choose_node/1` function. Unlike explicit placement a programmer may spawn a process to a node from a range of nodes that satisfy predefined parameters, such as s_group, s_group type, or communication distance.

Erlang follows a functional programming idiom of having a few primitives and building powerful abstractions over them. Examples of such abstractions are algorithm skeletons [13] that abstract common patterns of parallelism, and behaviour abstractions [6] that abstract common patterns of distribution. We plan to develop SD Erlang behaviour abstractions over primitives presented in Sections 4.2 and 4.3. We expect the behaviours to become apparent during the work on the case studies and scalable infrastructure.

Acknowledgements. We would like to thank all our colleagues who work on the RELEASE project. This work has been supported by the European Union grant RII3-CT-2005-026133 'SCIEnce: Symbolic Computing Infrastructure in Europe', IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and by the UK's Engineering and Physical

Sciences Research Council grant EP/G055181/1 'HPC-GAP: High Performance Computational Algebra and Discrete Mathematics'.

References

1. *Erlang/OTP Efficiency Guide, System Limits*, 2011. http://erlang.org/doc/efficiency_guide/advanced.html#id67011.
2. Akka: Event-driven middleware for Java and Scala, July 2012. http://www.typesafe.com/technology/akka.
3. Basho documentation, July 2012. http://wiki.basho.com.
4. The Scala programming language, July 2012. http://www.scala-lang.org.
5. J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide.* O'Reilly Media, 1st edition, 2010.
6. J. Armstrong. *Making Reliable Distributed Systems in the Presence of Sofware Errors.* PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.
7. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
8. J. Armstrong. Erlang. *Commun. ACM*, 53:68–75, 2010.
9. M. Aswad, P. Trinder, and H.-W. Loidl. Architecture aware parallel programming in Glasgow parallel Haskel (GpH).
10. S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.
11. F. Cesarini and T. Simon. *Erlang Programming: A Concurrent Approach to Software Development.* O'Reilly Media Inc., 1st edition, 2009.
12. M. Christakis and K. Sagonas. Static detection of race conditions in Erlang. In M. Carro and R. Peña, editors, *Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer Berlin / Heidelberg, 2010.
13. M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.
14. EDF. The sim-diasca simulation engine, July 2012. http://sim-diasca.com/.
15. T. A. S. Foundation. Apache Cassandra, 2012. http://cassandra.apache.org/.
16. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
17. W. H., M. Stan, S. Gurumurthi, R. Ribando, and K. Skadron. Interaction of scaling trends in processor architecture and cooling. In *SEMI-THERM '10*, pages 198–204. IEEE Computer Society, 2010.
18. D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon highly available key-value store. In *SIGOPS '07*, pages 205–220, New York, NY, USA, 2007. ACM Press.
19. J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, J. G. P. Barnes, O. Roubine, and J.-C. Heliard. Rationale for the design of the Ada programming language. *SIGPLAN Not.*, 14(6b):1–261, 1979.
20. Microsoft. *Axum Programmer's Guide*, 2009. http://www.microsoft.com.
21. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.
22. S. Murthy, Y. Joshi, and W. Nakayama. Orientation independent two-phase heat spreaders for space constrained applications. *Microelectronics Journal*, 34(12):1187–1193, 2003.

23. H. Nilsson, C. Wikström, and E. T. Ab. Mnesia - an industrial DBMS with transactions, distribution and a logical query language. In *CDSAA'96*, Kyoto, Japan, 1996.
24. M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, 2004.
25. R. Project. Scalable reliable sd erlang design, July 2012. http://release-project.eu/documents/D3.1_main.pdf.
26. M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *SIGARCH Comput. Archit. News*, 37:24–33, 2009.
27. H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08*, pages 155–179, Berlin, Heidelberg, 2008. Springer-Verlag.
28. J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the E programming language. *ACM Trans. Program. Lang. Syst.*, 15(3):494–534, 1993.
29. G. Snyder, M. Soto, R. Alley, D. Koester, and B. Conner. Hot spot cooling using embedded thermoelectric coolers. In *SEMI-THERM '06*, pages 135–143. IEEE Computer Society, 2006.
30. S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP '08*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.
31. A. Trew. Parallelism and the exascale challenge. Distinguished Lecture. St Andrews University, 2010.
32. D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *FPCA '85*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
33. Typesafe Inc. *Akka Documentation: Release 2.1 - Snapshot*, July 2012. http://www.akka.io/docs/akka/snapshot/.
34. C. A. Varela, G. Agha, W.-J. Wang, T. Desell, K. E. Maghraoui, J. LaPorte, and A. Stephens. *The SALSA Programming Language: 1.1.2 Release Tutorial*. Rensselaer Polytechnic Institute, Troy, New York, 2007. http://www.cs.rpi.edu/research/groups/wwc/salsa/tutorial/main.pdf.
35. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin / Heidelberg, 1997.
36. E. Wang, L. Zhang, L. Jiang, J. Koo, J. Maveety, E. Sanchez, K. Goodson, and T. Kenny. Micromachined jets for liquid impingement cooling of VLSI chips. *Microelectromechanical Systems*, 13(5):833–842, 2004.
37. J. Warnock. Circuit design challenges at the 14nm technology node. In *DAC '11*, pages 464–467, New York, NY, USA, 2011. ACM.
38. D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog – the language and its implementation compared with Lisp. *SIGPLAN Not.*, 12(8):109–115, 1977.
39. C. Wikström. Distributed programming in Erlang. In *PASCO'94*, pages 412–421, 1994.