Performance Portability through Semi-explicit Placement in Distributed Erlang

Kenneth MacKenzie
School of Computer Science, University of St. Andrews
kwxm@inf.ed.ac.uk

Natalia Chechina
School of Computing Science, University of Glasgow
Natalia.Chechina@glasgow.ac.uk

Phil Trinder
School of Computing Science, University of Glasgow
Phil.Trinder@glasgow.ac.uk

Abstract

We consider the problem of adapting distributed Erlang applications to large or heterogeneous architectures to achieve good performance in a portable way. In many architectures, and especially large architectures, the communication latency between pairs of virtual machines (nodes) is no longer uniform.

We propose two language-level methods that enable programs to automatically adapt to heterogeneity and non-uniform communication latencies, and both provide information enabling a program to identify an appropriate node when spawning a process. We provide a means of recording *node attributes* describing the hardware and software capabilities of nodes, and mechanisms that allow an application to examine the attributes of remote nodes. We provide an abstraction of *communication distances* that enables an application to select nodes to facilitate efficient communication.

We have developed open source libraries that implement these ideas. We show that the use of attributes for node selection can lead to significant performance improvements if different components of the application have different processing requirements. We report a detailed empirical investigation of non-uniform communication times in several representative architectures, and show that our abstract model provides a good description of the hierarchy of communication times.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

Keywords Erlang, distributed computation, placement, attribute, metric space.

1. Introduction

Applications on large distributed systems encounter issues that do not arise in smaller systems, including the following.

- The individual machines comprising the system may not all be the same: they may have differing amounts of RAM, different software installed, and so on.
- Communication times may be non-uniform: it may take considerably longer to send a message from machine A to machine B than it does to send a message from machine C to machine D. This is particularly important in large distributed applications, where communication times may begin to exceed the time required for individual processes to carry out their computations, and may dominate execution time.

These factors will make it difficult to deploy applications, especially in a *portable* manner. A programmer may be able to use system-specific knowledge to decide where to spawn processes so as to enable an application to run efficiently, but if the application is then deployed on a different system, or if the structure of the system changes as virtual machines (nodes) fail or new nodes are added, this knowledge could become useless. This problem could become especially pernicious if the deployment strategy is built into the code of the application.

To address these difficulties, we propose a notion of *semi-explicit placement*, where the programmer selects nodes on which to spawn processes based on run-time information about the properties of the nodes and of the overall system rather than selecting nodes explicitly based on system-specific knowledge. For example, if a process performs a lot of computation one would like to spawn it on a node with a lot of computation power, or if two processes are likely to communicate a lot then it would be desirable to spawn them on a pair of nodes which communicate quickly.

We have implemented two Erlang libraries which address the problems outlined above. The first deals with *node attributes*, which describe the properties of individual Erlang nodes and the physical machines on which they run. The second deals with a notion of *communication distances* which models the communication times between nodes in a distributed system. Our libraries are open source, and are available from `https://github.com/release-project/portability-libs/`.

We describe the theory, implementation, and validation of these ideas in Sections 2 and 3. Some issues which our initial experiences have brought to light we discuss in Section 4. Related work is discussed in Section 5, and then we conclude with a summary and some possibilities for further work in Section 6.

The work described here was carried out as part of the RELEASE project, whose overall aim was to improve scalability for distributed Erlang: see `http://www.release-project.eu/` for more information.

2. Node Attributes

This section describes an implementation of a library for managing attributes for Erlang nodes, and also a `choose_nodes/2` function which makes use of these attributes to select nodes with certain

Erlang '15, September 4, 2015, Vancouver, British Columbia, Canada.

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

properties. The implementation is contained in a library called `attr`.

2.1 Design and Implementation

2.1.1 Attributes

There are a large number of properties which may be of interest when selecting an Erlang node on which to spawn a process. We divide these into *static* and *dynamic* attributes.

Static attributes describe properties of a node which are not expected to change during the lifetime of an Erlang application. Some possibilities are

- The operating system type and version.
- The amount of RAM available.
- The number of cores used by the virtual machine (VM).
- Availability of other hardware features such as specialised floating point units or GPUs.
- Availability of software features, such as particular libraries. A specific example of this is that the Erlang `crypto` library requires a C library from OpenSSL versions later than 0.9.8. We have used a number of platforms where sufficiently recent OpenSSL versions have not been installed, and this leads to runtime failures of Erlang applications which use functions from `crypto`.
- Access to shared filesystems. One reason for this might be that an application may wish to use Erlang's DETS tables (which are stored on disk), and thus if a number of VMs all wish to access the same table then they must all be able to access the same filesystem. This may be possible if all of the VMs are running in machines in the same cluster, but not if they are running on different clusters.

On the other hand dynamic attributes describe properties which are expected to vary during program execution; for example:

- The load on the physical machine as a whole including other users' processes.
- The number of Erlang processes running in the VM.
- The amount of memory which is currently available.
- Whether a particular type of Erlang process is currently running on the machine. One might want to spawn a process on a VM which is already running some other type of process, or one might wish to avoid competing with a CPU-hungry process.

2.1.2 Propagation Strategy

One of the fundamental properties of attributes is that (in our use-cases at least) they should be available to other VMs, so that a suitable machine can be selected to spawn a process which has special requirements. The question then arises of how attributes should be propagated through a network. The technique adopted in the present (prototype) implementation is to equip each Erlang node with a small server which maintains a database of attributes. When a process wishes to select another node to spawn a process on, it queries the nodes it is interested in and asks for the values of the attributes involved in the selection criterion. The servers on these nodes return the attribute values to the original node, which then makes a choice according to the information which it has received. We discuss the merits and demerits of this approach in Section 2.3.1, and suggest extensions and alternatives.

2.1.3 The Attribute Server

Attributes are stored in an *attribute table* on each node. Our current implementation uses an ETS table for this, although Erlang's recently-introduced *maps* might be a more lightweight alternative. The attribute table contains two types of entry:

- Static attributes are stored as name-value pairs, for example, `{num_cpus, 4}` or `{kernel_version, {3,11,0,12}}`. The first entry is an Erlang atom, and the second is an arbitrary Erlang term (typically a number, string, or tuple).
- Dynamic attributes are represented by tuples of the form `{dynamic, {`*M,F*`}}`. Here M is the name of a module and F is the name of a zero-argument function in M. When a dynamic attribute is looked up, the function *M:F*() is evaluated and its result is returned as the value of the attribute.

  We also allow dynamic attributes of the form `{dynamic,{`*M,F,A*`}}` where A is a list of arguments.

The attribute table is managed by a process which is registered with the local name `attr_server`. Remote nodes can request information about attributes by evaluating a term of the form

```
{attr_server,Node} ! {self(),{report,Key,AttrNames}}.
```

Here `AttrNames` is a list of attribute names and `Key` is an Erlang reference (see `make_ref/0`) which is used to match responses with requests. When the server receives a request of this form, it looks up all of the specified attributes and sends back a message containing the attribute names and their values. If an attribute cannot be found, or if there is some problem in evaluating a dynamic attribute, the atom `undefined` is returned as the value of the attribute.

2.1.4 Populating the Attribute Table

How do the attributes get into the attribute table? The attribute server can be started by calling `attr:start(ConfigFile)` where `ConfigFile` is a string containing the name of a configuration file, which in turn contains an Erlang list of attributes. For example, a configuration file might contain

```
[{num_cpus, 4},
 {hyperthreading, 2},
 {cpu_speed, 2994.655},
 {mem_total, 3203368},
 {os, "Linux"},
 {kernel_version, {3,11,0,12}},
 {num_erlang_processes, {dynamic,
    {erlang, system_info, [process_count]}}}].
```

The final entry here is a dynamic attribute which evaluates `erlang:system_info(process_count)`: this returns the total number of processes existing on the node at the current point in time.

We also provide functions `insert_attr/2`, `update_attr/2` and `delete_attr/1` which can be used to modify attributes during program execution; one specific use of these would be for a node to advertise the fact that it is running a particular type of process, but see Section 4 below for a discussion of a potential problem with this strategy.

This scheme is completely extensible. Users can define arbitrary static and dynamic attributes. For dynamic attributes, they can even use functions from their own libraries (although caution should be exercised here since an attribute which takes a long time to calculate could slow things down; also, an attribute whose evaluation never terminates would cause the server to become locked up).

Discovering static attributes automatically. Many static properties of the system can be discovered automatically, for example, by examining system files. The `attr` library includes a mechanism for specifying such attributes in the configuration file by means of terms of the form `{automatic, {`*M,F*`}}` and `{automatic, {`*M,F,A*`}}`, where the given functions are evaluated *once* just after

the configuration file is read, with the resulting values being stored in the attribute table in the same way as normal static attributes.

Non-instantaneous attribute values. When a dynamic attribute is queried, the related function is called and the result is returned. This will typically return the value of the attribute *at a particular moment*, whereas in some cases it might be desirable to have a value averaged over some time period. For example, it might be useful to know the average number of Erlang processes running on a node during the last five minutes. In some cases, the attribute may in fact be implemented by calling some system function which already performs such averaging: for example, the built-in `loadavg15` attribute (Table 2). In other cases, a user might want to implement their own non-instantaneous attributes. This can in fact be done using the `automatic` attributes described above: one could specify a function which would spawn a process which would then run at regular intervals and use the `update_attr/2` function to modify the current value of the relevant attribute. This is not, however, how we intend `automatic` attributes to be used, and it might be worth extending the `attr` library to include explicit support for non-instantaneous attributes. Note that Folsom [2] and Exometer [10] both support means for collecting information over extended periods of time.

2.1.5 Built-in Attributes

As mentioned above, many system properties can be discovered automatically, for example, by examining files in the `proc` filesystem on Linux, or by using functions such as `erlang:system_info/1`, `erlang:statistics/1`, or functions in the `os` Erlang kernel library.

As an experiment, we have implemented a small number of useful attributes of this form which are automatically loaded when the attribute server is started. Most of these are kept in a library called `dynattr`. The built-in attributes are loaded before the contents of the configuration file, and will be overridden by user-defined versions. The attribute server can also be started without a configuration file, by calling `attr:start()`; in this case, only the built-in attributes will be loaded. One can also call `attr:start(nobuiltins)` or `attr:start(nobuiltins, ConfigFile)` to omit the built-in attributes.

The current built-in attributes are described in Tables 1 and 2.

This is just a sample implementation for experimental purposes. However, it does show that quite a large range of properties can be expressed by attributes. Note however that many of the attributes (in particular system load) are found by consulting files in the Linux `proc` filesystem. This definitely will not work on Windows (the atom `undefined` will be returned), and perhaps not on other Unix implementations where the precise format of the files may differ.

2.1.6 Querying Attributes

The `attr` library also contains a function called `request_attrs/2` which can be used to query a list of nodes for the values of specified attributes. This is done by a call of the form `request_attrs (Nodes, AttrNames)` where `Nodes` is a list of node names and `AttrNames` is a list of attribute names, for example:

```
request_attrs([vm1@osiris, vm1@bwlf01, vm2@bwlf02],
                [loadavg1, cpu_speed])
```

The function returns a list of the form `[NodeName, [{AttrName, AttrValue}]]`.

2.1.7 Choosing Nodes

We have used the `request_attrs/2` function to implement a simple `choose_nodes/2` function. This takes a list of nodes and a list of predicates which those nodes must satisfy. For example:

```
choose_nodes(Nodes, [{cpu_speed, ge, 2000},
   {loadavg5, le, 0.6}, {vm_num_processors, ge, 4}])
```

The function calls `request_attrs/2` to get the values of the required attributes on the specified VMs, then evaluates the predicates (discarding attributes whose values are `undefined`) and returns the subset of the nodes for which all of the predicates are satisfied.

We currently provide two types of predicates. The first carries out comparisons of attribute values against constants: `{AttrName, `*op*`, Const}`. We currently have the usual six comparison operators: `eq, ne, lt, le, gt,` and `ge`. These correspond to the Erlang operators `==, /=, <, =<, >,` and `>=`, respectively. These operators can compare any two Erlang terms, although you sometimes have to be careful; for example, `[1,2,3,4] < [1,2,4]` but `{1,2,3,4} > {1,2,4}`. Note also that we have used `==` and `/=` instead of `=:=` and `=/=` so that we get the expected results when comparing floats and integers.

The second type of predicate checks boolean values: we can say `{AttrName, true}` and `{AttrName, false}` (or `{AttrName, yes}` and `{AttrName, no}`).

This is a fairly minimal predicate grammar, implemented here as a proof of concept. It should suffice for many purposes, but it would not be hard to extend it if necessary by adding disjunction, for example.

2.2 Experimental Validation

As a simple validation experiment, we ran a modified version of one of our benchmark programs: multilevel Ant Colony Optimisation (ML-ACO) [11]. We hope to perform more extensive validation on more complex programs at a later date.

Ant Colony Optimisation (ACO) [7] is a metaheuristic inspired by the foraging behaviour of real ant colonies which is used for solving combinatorial optimisation problems. Our implementation is specialised to solve a scheduling problem called the Single Machine Total Weighted Tardiness Problem [18]. In the basic single-colony ACO algorithm, a number of artificial ants independently construct candidate schedules guided by problem-specific heuristics with occasional random deviations influenced by a structure called the *pheromone matrix* which contains information about choices of paths through the solution space which have previously led to good solutions. After all of the ants have produced solutions, the best solution is selected and used to update the pheromone matrix. A new generation of ants is then created which constructs new solutions guided by the improved pheromone matrix, and the process is repeated until some halting criterion is satisfied. In our implementation, the criterion is that some predetermined number of generations have been completed. The algorithm is naturally parallelisable, with one process for each ant in the colony. Increasing the amount of parallelism (i.e., the number of ants) does not lead to any speedup, but does lead to an improvement in the *quality* of the solution.

In the distributed setting, even more concurrency can be exploited by having several colonies which occasionally share pheromone information. In addition to increasing the number of ants exploring the solution space, distribution also gives the possibility of having colonies with different parameters: for example, some colonies might have more randomness in their search, making it easier to escape from locally-optimal solutions which are not globally optimal.

In the context of Erlang, one can have a number of nodes with one colony per node. Furthermore, we can connect colonies together in various different topologies [20], providing us with a variety of communication patterns.

As a specific example, our multilevel ACO application is structured as a tree: see Figure 1. There is a single *master node* M, and a number of *submaster nodes* S and *colony nodes* C. The colony

Attribute name	Value
`os_type`	This calls `os:type()`, which returns a pair such as `{unix, linux}` giving the family (either `unix` or `win32`) and type of the operating system.
`os_version`	This calls `os:version()` which will return a tuple or string containing the OS version.
`otp_release`	This calls `erlang:system_info(otp_release)` to get the OTP version.
`vm_num_processors`	This calls `erlang:system_info(logical_processors_available)` to get the number of processors available to the VM. This may be less than the total number of processors on the physical machine if the VM is restricted to use some subset of the processors.
`mem_total`	Total memory on the system (in kB), found in `/proc/meminfo`.

Table 1: Built-in static attributes

Attribute name	Value
`cpu_speed`	Current speed (in GHz) of the first CPU , as found in `/proc/cpuinfo`.
`mem_free`	Current free memory (kB) , from `/proc/meminfo`.
`loadavg1`	System load average over last minute, from `/proc/loadavg`. The value is a float between 0 and 1: see `man proc` and `man uptime` for details.
`loadavg5`	Load average over last 5 minutes.
`loadavg15`	Load average over last 15 minutes.
`kernel_entities`	Numbers of Linux scheduling entities (processes/threads), from `/proc/loadavg`. This returns a pair `{R,E}`, where `R` is the number of currently runnable entities and `E` is the total number of entities on the system.
`num_erlang_processes`	Number of Erlang processes currently existing on the VM, from `erlang:system_info(process_count)`.

Table 2: Built-in dynamic attributes

nodes independently construct solutions to the input problem, and after a certain number of iterations report their solutions to a submaster node on the level above. Each submaster chooses the best solution from its children and passes that to the level above, and so on. Eventually, the master node selects the best solution from its children, which is the best solution from among all of the colony nodes. This solution is then sent back down the tree to the colony nodes, which use it to update their pheromone matrices for future searches. This process is repeated a number of times, after which the master reports its current best solution and the application terminates.

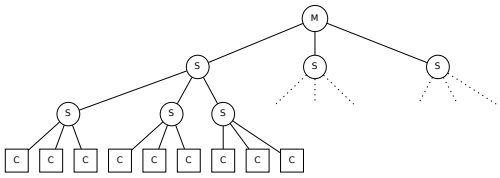

Figure 1: Multilevel ACO

The colony nodes perform a considerable amount of mathematical computation (and themselves have many ant processes constructing solutions concurrently), but the master and submaster nodes do not do much work. It would therefore seem reasonable to run colonies on VMs with lots of processors and master and submaster nodes on colonies with fewer processors.

2.2.1 Attribute-aware ML-ACO

We modified ML-ACO so that it used attributes to spawn submasters on small Erlang nodes (at most 4 processors) and colonies on large ones (more than 4 processors). This was run on 256 compute nodes in EDF's Athos cluster; three machines each had 24 small VMs running (one pinned to each core) and the other 253 had a single large VM. We ran the modified ML-ACO version with the VMs presented in random order, gradually increasing the load on the VMs by increasing the number of ants in each colony from 1 to 80. For each number of ants, we recorded the mean execution time over 5 runs, firstly using attributes for placement and then without attributes. In the latter case, processes were just spawned on nodes in the random order in which they were presented to the application.

Figure 2 shows the resulting execution times. We see that the program performs substantially better when attributes are used for placement. This is unsurprising, since when attributes are not used, colony nodes will often be placed on Erlang VMs which are only using one core instead of 24. These colonies will take much longer to execute than ones on VMs using lots of cores, and this slows the entire program down. This is confirmed by Figure 3, which shows the ratio of execution times without attributes to those with attributes. For small numbers of ants, the performance of the attribute-unaware version is similar to the attribute-aware version, but the ratio become progressively worse as the number of ants (and hence the number of concurrent processes) in the VM increases. We expect that the ratio would asymptotically approach 24 with very large numbers of ants.

This is admittedly a rather artificial example, since the effect of introducing small VMs is quite predictable. However, it does demonstrate that the use of node attributes can improve performance in a heterogeneous network, and that this can be done without any information about the network being coded into the program. We plan to test the use of attributes with a large and complex program such as Sim-Diasca [8] (one of the RELEASE project's use-cases) but we have not yet done this at the time of writing.

2.3 Discussion

2.3.1 Attribute Propagation Strategy

We have adopted a very simple strategy here: when a node wants to know the value of an attribute on another node, it just asks for it. Another approach would be to have nodes broadcast their attributes to all the other nodes which they are connected to. We argue that our present approach has several advantages:

- Information is only transmitted when it is required, and only to nodes that require it. It is possible that in a real system, only a limited number of nodes would actually be spawning remote

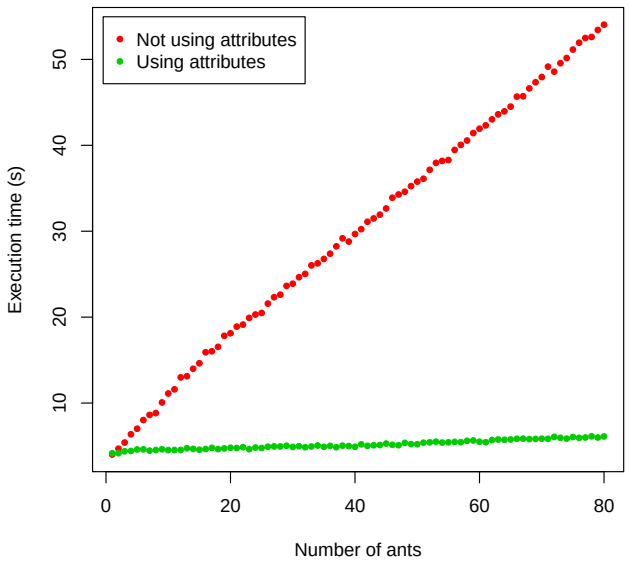

Figure 2: Execution times with and without attributes

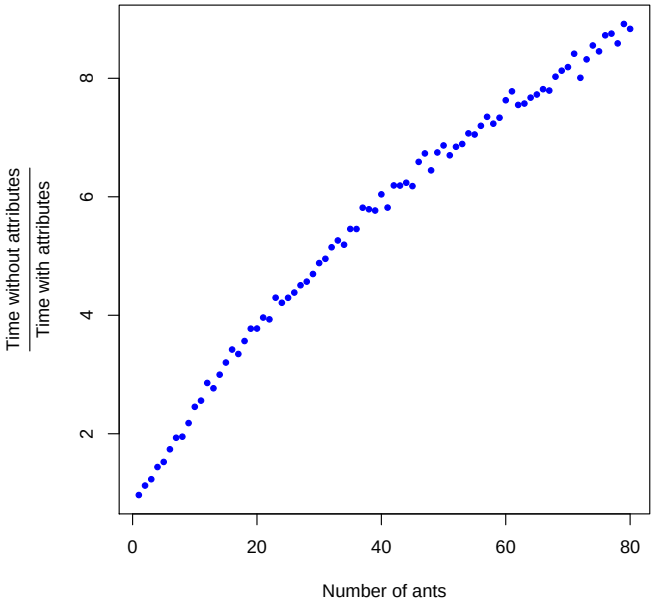

Figure 3: Ratio of execution times

processes, and they would be the only ones which would need to receive attribute information. Perhaps one could have a model where nodes have to register to receive attribute information

- Only those attributes which are required are computed and transmitted. We presumably don't want thousands of nodes broadcasting their load average across an entire system once a minute if it is seldom required.
- In this scheme, information about dynamic attributes is always fresh. If attribute information has to be broadcast then it will sometimes be out of date unless it is broadcast regularly, which might lead to too much network activity. Also, some overhead is incurred in finding the value of dynamic attributes, and this would be adding extra load to machines if we had to calculate dynamic attributes at frequent intervals.

On the other hand, there are some disadvantages to requesting attributes at the time when a process is spawned:

- If many processes are being created, there will be a lot of requests to the attribute servers, adding load to the target machines. There is a tension between this factor and the danger of flooding the network with too many broadcast attributes. Like other functions that require communication between multiple nodes, the `choose_nodes/2` and `request_attributes/2` functions must be used with care.
- If there is a lot of latency in the network or just between the machine which is making the request and a single member of the list of machines it is querying, then `request_attrs/2` and hence `choose_nodes/2` will take a long time every time it is called. This could slow things down considerably in comparison to the case where information is broadcast regularly.

It may be possible to find some middle ground by caching responses to the `request_attrs/2` function, e.g. in an ETS table. Once a static attribute has been retrieved once, it would never be necessary to ask for it again (although one could end up with attribute information belonging to defunct nodes, so it might be worth invalidating all of the static attributes once every few hours). Dynamic attributes could be saved with a timestamp, and if they have been in the cache too long then we could ask for their value again. The time for which an attribute is valid could vary with the attribute: we would wish to discard the 1-minute load average quite quickly, but could keep the 15-minute load average for longer.

Given more time, we would have liked to implement a system where attributes are broadcast across the network as well, and to see how this method compares with the one we have already implemented. It is possible that the second approach would be more effective in certain situations, but this would depend on the properties of the application.

2.3.2 Reliability

The present implementation is fairly simplistic, and makes no pretence to reliability. If a node's attribute server crashes for some reason then there is no attempt to restart it. There are also problems in querying such nodes. At the moment, `request_attrs/2` times out if it has to wait too long: if nothing happens for 5 seconds after receiving its most recent message, it just times out and returns whatever attributes it has already received. If an attribute server has gone down then `request_attrs/2` will take at least 5 seconds every time it requests information from a list of nodes which contains a bad one. On the other hand, timing out runs the risk of missing a response from a live node which is taking a long time to respond.

See Section 4 below for some related points.

2.3.3 Attributes and s_groups

In earlier publications [5, 26] we have described and implemented s_groups, which partition the address space of distributed Erlang applications and help to improve scalability by by avoiding the default behaviour of having connections between every pair of nodes.

In this paper we have not considered the interaction of node attributes and s_groups; however it should not be too difficult to integrate the two. The current `choose_nodes/2` function takes a list of Erlang nodes as its first argument, so one can easily supply it with the members of an s_group, obtained using the `s_group:own_nodes/0` function for example. One could modify `attr:choose_nodes/2` to take an s_group name as a parameter, but this would make it quite tightly coupled to the `s_group` library, which is not part of the standard Erlang/OTP distribution; our current `attr` library can be used with any Erlang version.

3. Communication Distances

In many modern distributed computer installations, the inter-node communication infrastructure has some kind of hierarchical structure. Within a particular organisation, machines in a cluster may be connected together by a high-speed network, and this network may in turn be connected to other machines within the organisation via a network with slower communication. If the network spans several sites in different geographical locations, then communication between sites may be slower still. Connecting to machines in a different organisation may introduce further delays.

On a smaller scale, communication between processing units within an SMP machine may be similarly hierarchical: inter-processor communication rates may depend on the level of cache which two processors share, or whether the processors are located on the same socket.

In this section we describe an implementation of an abstract model of communication times which can be used for process placement in systems with this type of hierarchical communication structure. We use an idea originating in [17].

Suppose we have a collection of Erlang nodes. A useful way to think about inter-node communication times is to think of the nodes as points in a space and to regard communication times as *distances* between these points. An appropriate mathematical model is the notion of a *metric space* (see [12, 6.12] or [28, 2.15], for example).

3.1 Metric and Ultrametric Spaces

Definition. A *metric space* is a set X together with a function

$$d : X \times X \to \mathbb{R}^+ = \{x \in \mathbb{R} : x \geq 0\}$$

such that, for all $x, y, z \in X$,

$$\begin{aligned} &\text{(i) } d(x,y) = 0 \text{ if and only if } x = y \\ &\text{(ii) } d(x,y) = d(y,x) \\ &\text{(iii) } d(x,z) \leq d(x,y) + d(y,z) \end{aligned}$$

The inequality (iii) is called the *triangle inequality*. If we replace (iii) with

$$\text{(iii}') \; d(x,z) \leq \max\{d(x,y), d(y,z)\}$$

then we obtain the definition of an *ultrametric space* [15] (and (iii$'$) is called the *ultrametric inequality*). It is not hard to see that every ultrametric space is a metric space. □

Metric spaces give a very general model of distances, and admit generalisations of many concepts from standard geometry. One specific concept we will make use of is the *closed disc*.

Definition. Let X be a metric space. For $x \in X$ and $r \in \mathbb{R}^+$, the *closed disc of radius r with centre x* is

$$D(x,r) = \{y \in X : d(x,y) \leq r\}.$$

□

3.2 Trees and Ultrametric Spaces

Given a tree with an arbitrary amount of branching, we can define a metric (in fact, an ultrametric) on its set of leaves by

$$d(x,y) = \begin{cases} 0 & \text{if } x = y \\ 2^{-\ell(x,y)} & \text{if } x \neq y. \end{cases}$$

where $\ell(x,y)$ is the length of the longest subpath which is shared by the paths from the root to x and y. We leave it as an exercise to show that d is in fact an ultrametric.

Referring to the tree in Figure 4 we have

$$\begin{aligned} d(b,c) &= 2^{-2} = \frac{1}{4} \\ d(b,f) &= 2^{-1} = \frac{1}{2} \\ d(b,k) &= 2^{-0} = 1 \end{aligned}$$

and so on.

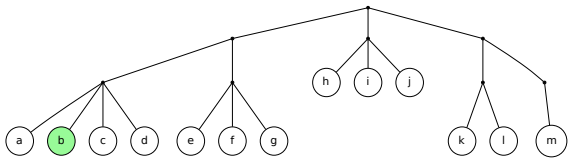

Figure 4

Similarly, some closed discs around b are

$$\begin{aligned} D(b, 0.3) &= \{a, b, c, d\} \\ D(b, 0.8) &= \{a, b, c, d, e, f, g\} \\ D(b, 1) &= \{a, b, c, d, e, f, g, h, i, j, k, l, m\} \end{aligned}$$

(see Figure 5, for example).

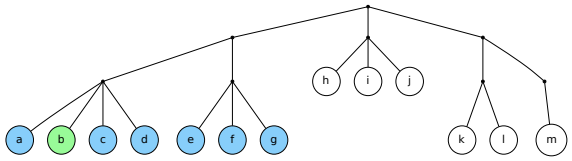

Figure 5: $D(b, 0.8)$

This suggests why this particular metric might be useful for studying communication distances: given a (computing) node in some hierarchical communication system, the various closed discs contain nodes which are in the same subclusters at various levels, and hence which might be expected to have similar inter-node communication times.

3.2.1 Implementation

We have implemented a simple Erlang library which carries out calculations using ultrametric distances. It takes as input a tree (represented as an Erlang term) which describes the structure of a network of Erlang VMs, and then uses closed discs to describe VMs

at various distances. The library includes a `choose_nodes/2` function similar to that in the attribute library: one can select machines by making calls of the type

```
choose_nodes (Nodes, {dist, le, 0.2})
```

or

```
choose_nodes (Nodes, {dist, gt, 0.8})
```

to get machines which are close or far away, respectively. Eventually we intend to merge this library with the attribute library.

3.3 Comparing the Model with Reality

Our model is very abstract, and we claim that this is in fact an advantage. Given a computer network with a hierarchical communication structure, we can draw a tree which reflects the gross structure of the communication hierarchy and use its abstract metric properties to reason about communication times, *without having to know details about the physical structure of the network, including actual communication times*.

The question arises of whether our model might be *too* abstract. If we make decisions based on the abstract hierarchical structure of the network, can we be sure that they bear a reasonable correspondence to real communication times?

In an attempt to answer this question, we have carried out some empirical studies of Erlang communication times on real-life systems. Our technique has been to look at the time taken for messages to pass between pairs of nodes in distributed Erlang systems and then to use statistical techniques to study how nodes cluster together as determined by communication times. We can then compare the outcome of the clustering process with our abstract view of the system to see how they correspond.

We do not expect actual communication times to satisfy the metric space axioms precisely. For example, messages sent from a node to itself will not be sent instantaneously (see axiom (i)); however, it is plausible that such messages will be significantly faster than messages between different nodes. Similarly, communication times will probably not be precisely symmetric (axiom (ii)), but we would expect messages times from node A to node B to be very similar to those from B to A. These expectations are confirmed by the data from the experiments described below.

3.3.1 Empirical Validation

We have implemented a small Erlang application with two components:

- A server which waits for messages and then replies immediately to the sender.
- A client which sends a large number of messages to the server and then calculates the average time between sending a message and receiving a reply, using the functions in the Erlang `timer` module.

Given a network of machines, we run the client and server on every pair of machines to determine average communication times. We then apply statistical methods to detect *clusters* within the network.

We view the machines in the network as points in a space and the communication times as a measure of distance between the points. This view is distinct from our earlier abstract view involving metric spaces. Here we simply have empirical data and we have no guarantee that it will satisfy any of the axioms of metric spaces. The point of our experiments is to see how closely our empirical data in fact conforms (or fails to conform) to our abstract model.

3.3.2 Cluster Analysis

To study the hierarchical structure of our results we use a technique known as *hierarchical agglomerative clustering* [9, Chapter 4], [14]. This collects data points into clusters according to how close together they are. Furthermore, the clusters are arranged hierarchically, with small clusters grouped together to form larger ones, and so on.

The basic technique is as follows:

- Start off by placing every point in a cluster of its own.
- Look for the two clusters which are closest together and merge them to form a large cluster.
- Repeat the previous step until we have only a single cluster.

This gives rise to a system of nested clusters, as illustrated in Figure 6 for a set of points in the plane (with the usual Euclidean metric).

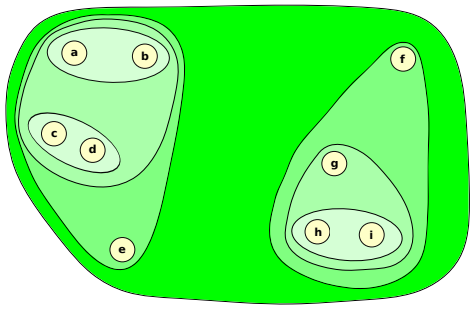

Figure 6: Hierarchical clustering

A question arises here: we know the distance between two points (that is our basic data), but how do we measure the distance between two *clusters*? Various methods can be used. For example, given two clusters A and B, any of the following could be used:

$$d(A,B) = \max\{d(a,b) : a \in A, b \in B\}$$
$$d(A,B) = \text{mean}\{d(a,b) : a \in A, b \in B\}$$
$$d(A,B) = \min\{d(a,b) : a \in A, b \in B\}$$

All of these methods (and others) are used in the clustering literature (see [9] or [14], for example), and all are useful in different situations. We have chosen the first method, which is known as the *complete linkage* method: $d(A,B) = \max\{d(a,b) : a \in A, b \in B\}$. In terms of communication times, this tells us what the worst-case communication time between a node in A and a node in B is; this is reasonable from our point of view because we wish to have upper bounds on communication time.

The hierarchical structure of clusters and subclusters can be displayed in a type of diagram called a *dendrogram*. This is a tree which has one node for each cluster, with the children of a cluster being its subclusters. Figure 7 shows the dendrogram corresponding to Figure 6. The dendrogram was obtained using the `hclust` command in the R system for statistical analysis and visualisation [23], using distances between points measured directly from Figure 6. The dendrogram describes the hierarchical nested structure seen in Figure 6, with the height of the internal nodes of the dendrogram reflecting the distances between the corresponding subclusters.

3.4 Measurements

We ran our distance-measuring Erlang application on several systems. Firstly we used it on a small scale to look at inter-processor communication times within a multicore system, then we used it

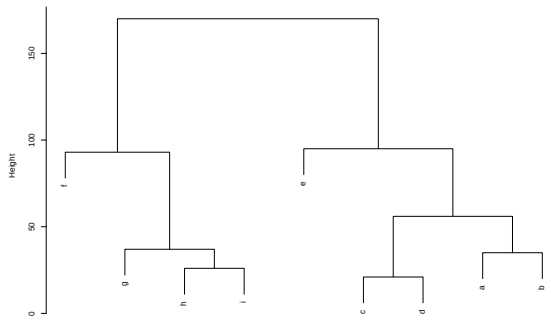

Figure 7: Dendrogram

on a larger scale to look at inter-node communication times on two sizeable networks.

Our technique was to run Erlang VMs on each of the components of the system (processing units within a single SMP machine, individual physical machines within a network), and measure the average time taken to send several messages back and forth between each pair of VMs. More precisely, our basic unit of measurement was the time taken to send 100 messages back and forth: we chose this number because the time taken for a single message can be close to the one-microsecond resolution of Erlang's `timer:tc/1,2,3` function, so it is difficult to get precise times for single messages.

For each pair of VMs, we actually measured the mean time over 100 such 100-message batches, and used that as our final data. This was an attempt to make sure that our figures were representative: with a smaller number of datapoints, there was a danger that, for example, one of the VMs might have been swapped out by the operating system when the messages arrived, adding an unusual delay. By sending 100 batches we hoped to mitigate such effects.

We also tried two different strategies. In the first, we ran a single process on VM number 1 which broadcast messages to all other VMs in parallel; once this had finished, we ran a similar process on VM number 2, and then on VM number 3, and so on. In our second strategy, we ran such processes on all VMs concurrently, so that all pairs of VMs were communicating simultaneously. We found that this gave more interesting results, because high traffic densities made irregularities in communication times more apparent.

It should be pointed out that running multiple VMs on the same physical machine is not obviously a sensible thing to do: within a single VM, inter-process messages are transmitted directly within the VM, by copying data between VM data-structures. This is something like 40 times faster than TCP/IP communication between two VMs on the same physical machine, and hence having more than one VM would appear to lead to inefficiency. However, this is not necessarily the case. If a VM does not require too many resources, then pinning it to some subset of the cores (on a single socket, for example) might enable it to benefit from the same locality effects that we have seen above, and hence to operate *more efficiently* than if it is using all of the cores. Whether or not this is desirable would depend on the application being run.

3.4.1 Forty-eight Core Machine

We ran our experiment on an AMD Opteron 6348 machine with 48 cores. The structure of the machine is shown in Figure 8 (this was obtained using the `lstopo` command included with the `hwloc` library [4]) and a dendrogram of our results in Figure 9.

The structure of the dendrogram, and hence the communication times from which it was derived, reflects the NUMA structure of the machine very closely. Erlang uses TCP/IP for inter-VM communications, and when the VMs are on the same host, this will take place via the Application layer of the TCP/IP protocol [3], where messages are transmitted within memory by the OS. This means that there will be very little overhead, so communication times are strongly affected by the cache structure of the machine.

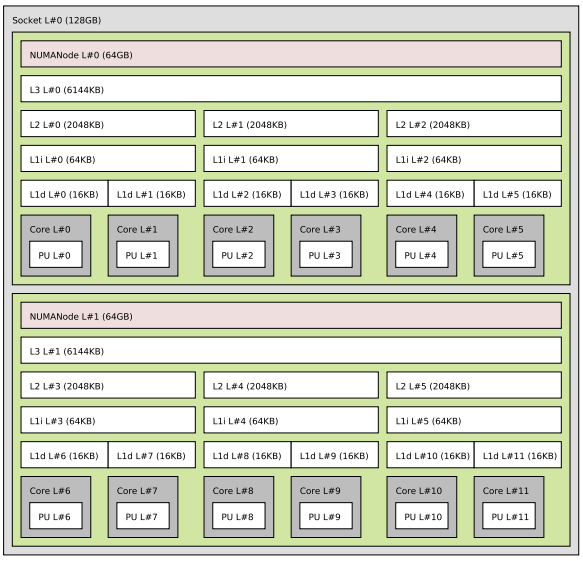

Figure 8: Forty-eight-core machine (four sockets like this)

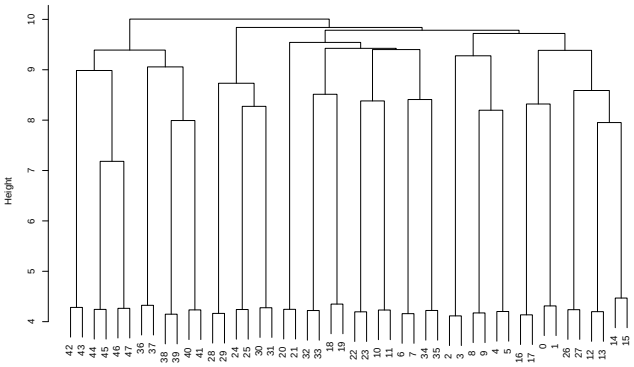

Figure 9: Dendrogram for forty-eight-core machine

We have run similar experiments on other multicore machines and in each case we obtained dendrograms which corresponded very closely to the NUMA hierarchy.

3.4.2 Communication Times in Networks

The results in the previous subsection show that hierarchical communication structures do appear in reality, and the dendrograms which we obtain correspond very closely to the tree which we would use in our metric space model, based on the physical structure of the system. However, our experiments involved Erlang VMs

pinned to individual cores of a multicore machine, a situation which is definitely not typical of distributed Erlang applications.

We also ran experiments in more realistic settings, measuring communication times between Erlang VMs running on computational nodes in distributed networks.

3.4.3 Departmental Network

We ran our tests on some machines in a departmental network at Heriot-Watt University. We used 39 machines, including a 34-node Beowulf cluster. The results are shown in Figure 10. Here, the dendrogram picks out the Beowulf cluster, however, it is less obvious what is happening with the other machines.

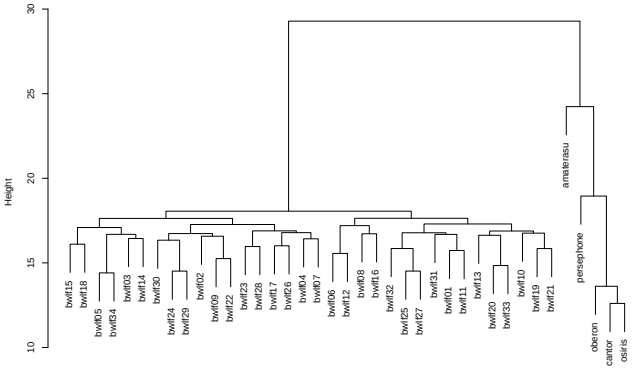

Figure 10: Dendrogram for departmental network

3.4.4 A 256-node Network

Our final test-case was a 776-node cluster at EDF; the RELEASE project was allowed simultaneous access to up to 256 nodes (6144 cores in total). Users interact with the cluster via a front-end node and initially have no access to any of the compute nodes. Access to the compute nodes is obtained via the SLURM workload manager (see `http://slurm.schedmd.com/`), either interactively or via a batch script which specifies how many nodes are required, and for how long. Jobs wait in a queue until sufficient resources are available, and then SLURM allocates a number of compute nodes, which then become accessible (via ssh, for example). The user has exclusive access to these machines, and no-one else's code will be running at the same time. Fragmentation issues mean that jobs are not usually allocated a single contiguous block of machines, but rather some subset scattered through the cluster: for example nodes 127-144, 163-180, 217-288, 487-504, 537-648, 667-684. These will typically be interspersed with machines allocated to other users. Users can request specific (and perhaps contiguous) node allocations, but it may take a long time before the desired nodes are all free at once, leading to a very long wait in the SLURM queue.

Athos was somewhat problematic. As we shall see shortly, communication times were highly non-uniform. We hypothesise that Athos communication takes place via a hierarchy of routers, but the precise structure of the cluster is not publicly available. Furthermore, SLURM tends to allocate a different set of nodes to each job, so it is difficult to get repeatable results.

Figures 11 and 12 illustrate the complicated communication structure that we have observed. Figure 11 shows a dendrogram for communication times in a 256-node SLURM allocation on the Athos cluster. We can see 9 or 10 distinct subclusters with fast intra-cluster communication, but with substantially slower communication between the subclusters. However, it's difficult to determine exactly what's going on due to the denseness of the diagram.

In an effort to make the data more comprehensible, Figure 12 shows two views of a three-dimensional plot of the communication times. The x- and y- axes show the source and target nodes (i.e., the nodes which are sending and receiving messages, respectively), and there is one point for each pair of machines, whose z-coordinate represents the mean communication time observed between those machines. These points are coloured according to the source machine, in an attempt to make the perspective views easier to interpret. It is clear that communication times are highly quantised: for some pairs of machines, the mean time taken to exchange 100 messages is on the order of 25ms, whereas for others, it is over 800ms. This is a 32-fold difference.

Remark. It is worth noting that communication times vary with the amount of network traffic. We also ran our test program (during the same SLURM job as above) with a different strategy, where one machine at a time would exchange messages with all of the others, then the next machine would do the same thing, and so on; this involves much less network traffic. The distribution of message times is much smoother: all communication between different machines takes between about 50ms and 60ms for 100 exchanges. This contrasts strongly with the situation in Figure 12, where the mean communication time is much slower (585ms, as opposed to 58ms); Oddly, some of the communication is actually *faster* in Figure 12, where about 10% of the exchanges take between 20ms and 40ms.

Discussion. The earlier figures show that Athos does have a very hierarchical communication structure (at least when there is a lot of network traffic), but we have been unable to determine exactly what that structure is. Information about the construction of the network is not available to us, but we hypothesise that there is some tree-shaped hierarchy of routers. When there is a lot of network traffic (as there was in these experiments, where all nodes were talking to all others simultaneously) this would mean that some messages would have to travel up and back down through several layers of routers, and some of the routers would become congested. There appears to be an extra complication that node names do not correspond cleanly to the hierarchical structure of the network; this would explain some of the off-diagonal areas of fast communication at the bottom of the plots in Figure 12. The situation is made even worse by the fact that we cannot observe the whole network at once: we can only look through the 256-node windows supplied by SLURM.

Despite these difficulties, the dendrogram in Figure 11 (and similar ones obtained with different SLURM allocations) shows that the communication times are strongly hierarchical, and fit our metric-space model well. This is also the case with our earlier results for communication times between cores on NUMA machines. Our model abstracts away detailed information about communication times, but the closed discs which it provides correspond strongly to the clustering structure in actual communication times, and so closed discs in the metric space provide a good description of sets of nodes which can communicate quickly. We have not yet been able to test our methods with a large application, but the large range of communication times (differing by a factor of 32 in some cases) seen in the Athos cluster suggests that careful use of information about communication distances could improve performance significantly for applications which perform a large amount of suitably localised communication.

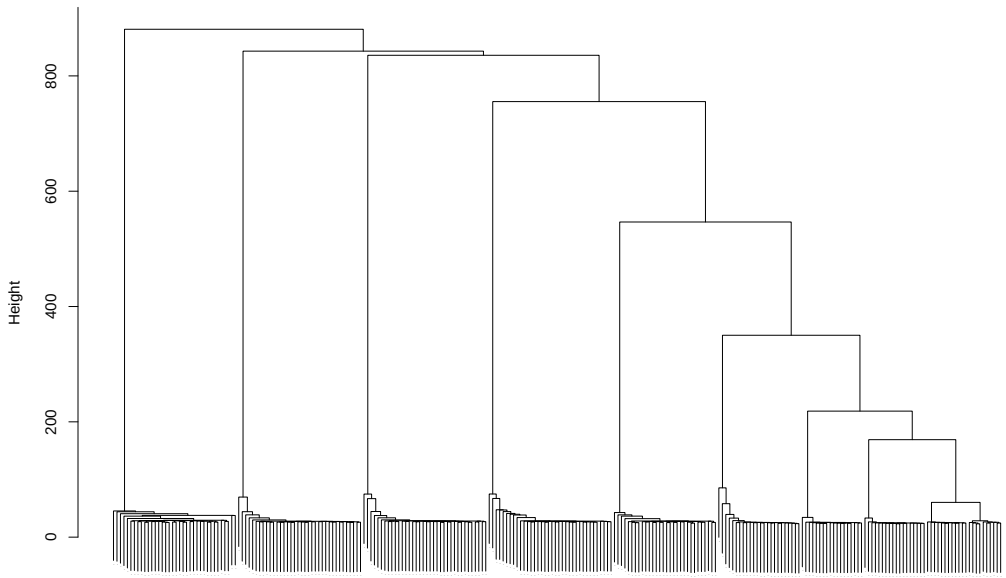

Figure 11: Dendrogram for Athos cluster

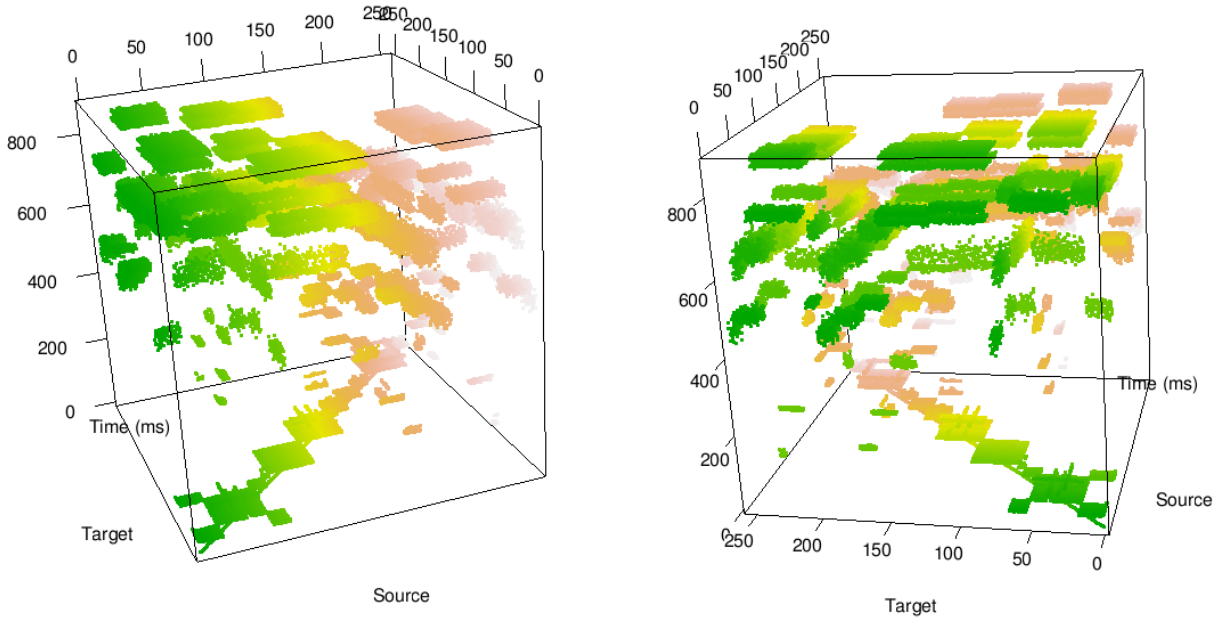

Figure 12: Athos communication times

4. Practical Issues & Future Work

The libraries for attributes and communication distances described above are prototypes, and our experience so far suggests several factors which it would be helpful to modify in libraries intended for general use.

Concrete and abstract bounds. In practice it may be somewhat difficult to know what concrete bounds to use to make a choice of nodes. For instance, at a particular time there may be no nodes satisfying `{loadavg5, lt, 0.1}` but several satisfying `{loadavg5, lt, 0.3}`. For practical use it might be worth having predicates like `{loadavg5, low}` and `{cpu_speed, high}` which would examine an entire list of candidate nodes and select the ones whose attributes are relatively "good" in comparison with the majority.

Similarly, programmers should not have to know about explicit distances. Note that distances vary with the depth of a network hierarchy: in a shallow network, the nearest node to a specified node might be at a distance of $\frac{1}{4}$, whereas in a deeper network it could be at a distance of $\frac{1}{32}$. In order to avoid this, we should provide the programmer with some abstractions such as `[very_near, near, far, very_far, anywhere]`.

Conflicting constraints. Another issue is that a programmer may supply conflicting constraints. For example, one might ask for a node which has a large number of cores and has a particular library installed, but it may not be possible to satisfy both requirements at once. One could overcome this to some extent by, for example, giving priority to the earlier constraints in a list. For debugging purposes, it could be useful to include a mode in which such conflicts are reported at runtime.

Avoiding clashes when spawning processes. There may be some danger in using attributes to select nodes. If a large number of processes are spawning other processes, there is a possibility that many of them may select the same node, and then performance will suffer. This could be mitigated to some extent by making a random choice from a list of suitable candidates, but there could be problems if there is a unique node which has some particularly desirable properties. At some point it may be necessary to spawn processes on a suboptimal node, but it is not entirely clear how to do this in a portable manner.

Fault tolerance. Our current implementation of attribute scheme is highly fault-tolerant as long as no applications dynamically modify the attributes of a node. Suppose that only static and built-in attributes are used: in the event of the attribute server crashing, it can simply be restarted and all of the node attributes will be restored; similarly, if an application process crashes then it can be restarted and the node attributes will be the same as they were before the crash. However, if an application modifies the attributes (for example, to register the fact that it is running on the node), then if the attribute server crashes, this information will be lost; if the application crashes then the attribute server could be left with stale attributes belonging to a defunct process. More research on allowing applications to modify node attributes may be needed.

Dynamic changes to network structure. Our current distance scheme depends on a static description of the network structure. In a long-running system, it is likely that nodes will leave the system, or that new ones may join. We have some preliminary ideas as to how to deal with this problem, but this will require further work.

Using attributes in practice. We have described mechanisms which allow programmers to select nodes with useful properties, but have not said anything about *how* these methods should be used in practice. Finding optimal solutions to placement problems even with a fixed number of tasks whose requirements are known in advance is already difficult (consider the (NP-hard) Quadratic Assignment Problem [22], for example), and we suspect that it would become completely intractable when tasks whose requirements may vary with time are created and destroyed dynamically. Our hope is that a programmer's intuition about the behaviour of their program will will enable them to make reasonable choices about where to spawn important processes, and that our techniques will facilitate the construction of applications with good and portable performance.

5. Related Work

Attributes. The Folsom [2] and Exometer [10] libraries provide facilities for collecting per-node metrics on running Erlang applications; the built-in and user-defined metrics are similar to our attributes, but are typically used for monitoring and analysing the behaviour of the application (usually via external tools).

At the tool, rather than language level, WombatOAM [24, 25, 27] is a commercial product for deploying and monitoring Erlang applications on large distributed systems, either physical or cloud-based. Wombat can collect *metrics* from the nodes which it is managing [24, §4.2.2]. WombatOAM metrics are properties such as sizes of run queues, numbers of processes, numbers of atoms, and so on. There are about 90 built-in metrics, and users can define their own, using Folsom or Exometer, for example. Wombat uses metrics for interactive monitoring the behaviour of Erlang nodes: someone using Wombat to manage a distributed Erlang application can ask for information about metrics, plot graphs of them, and so on.

The metrics of Folsom, Exometer, and WombatOAM are closely related to the *attributes* described below, but the crucial point about our attributes is that they are made available to other nodes in order to assist with process placement; the systems mentioned above all use metrics for monitoring run-time behaviour.

Basho's Riak Core – the framework underlying their distributed Erlang database system, Riak – uses a notion of node *capabilities* whereby nodes in a distributed system can publish information which is then propagated to other nodes [19]. This feature is intended for use during upgrades, so that, for example, one can determine which version of a protocol to use to talk to other nodes.

Further afield, information about members of distributed systems is used in Grid scheduling, where one wishes to execute a job on one or more remote machines: schedulers need information about the properties of individual machines, including hardware capabilities, current load, the amount of work queued for later execution, and so on. Deciding how to schedule jobs can then involve complex optimisation problems. This is a large field, and the most we can do here is to refer the reader to surveys such as [29] and [6].

Communication distances. In a number of systems work distribution is informed by communication topology, e.g. HotSLAW [21] and hierarchical load balancing in CHARM++ [16]. Several parallel functional languages have exploited communication topology, e.g. parallel Haskells with a two-level topology [13], and a multi-level topology [1, Chapter 5]. Our model of arbitrary communication distances is relatively sophisticated in combining multiple levels and a notion of equidistant discs, i.e. suitable targets for work distribution. The model is taken directly from Haskell distributed parallel Haskell (HdpH) [17].

RELEASE technologies. The new attribute and communication libraries are part of the Scalable Distributed Erlang technologies developed in the EU RELEASE project, but can be used with any distributed Erlang program. Moreover our language level adaption mechanisms can be combined with existing deployment tools such as WombatOAM, which was also developed in RELEASE.

6. Conclusion

We have designed, implemented, and performed a preliminary evaluation of semi-explicit work-placement mechanisms combining host/node attributes and communication distances.

Our attribute library provides a flexible and user-extensible method for Erlang nodes to advertise their properties to other nodes in a distributed system (Section 2). We have performed a validation of this which suggests that the use of attributes to select suitable nodes to spawn remote processes on can lead to significant improvements in performance (Figures 2 and 3). Attributes can be exploited without the programmer requiring any *a priori* knowledge of the configuration of the system, and hence allows one to achieve good performance in a *portable* way.

We have also proposed an abstract model of communication distances, and an empirical investigation of communication times on representative architectures including NUMA, and both large and small clusters (Section 3). Our method gives a good description of hierarchical communication structures without requiring concrete (and perhaps very complicated) information about communication times (Section 3.4). This suggests strongly that in distributed applications with non-trivial communication, our model would improve performance, again in a portable way.

In future research we hope to apply our techniques to larger and more realistic Erlang applications in order to provide a more convincing validation. We believe that the concept of communication distances could profitably be applied to distributed computation in general, and hope to experiment with other languages in addition to Erlang.

Acknowledgments

The research described here was carried out as part of the European Union-funded project 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software' (IST-2011-287510) while the first author was employed by the University of Glasgow. We would like to thank our project partners Électricité de France for allowing us access to their Athos cluster.

References

[1] M. Aswad. *Architecture Aware Parallel Programming in Glasgow Parallel Haskell (GPH)*. PhD thesis, School of Mathematical and Computer Sciences, Heriot-Watt University, August 2012.

[2] Boundary. *Folsom*. `https://github.com/boundary/folsom`.

[3] R. Braden. *RFC 1122 Requirements for Internet Hosts - Communication Layers*. Internet Engineering Task Force, Oct. 1989.

[4] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186, Feb 2010. URL `http://www.open-mpi.org/projects/hwloc/`.

[5] N. Chechina, H. Li, A. Ghaffari, S. Thompson, and P. Trinder. Improving network scalability of Erlang. *Submitted to the Journal of Parallel and Distributed Computing*, January 2015.

[6] F. Dong and S. G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, Technical report, 2006.

[7] M. Dorigo and T. Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004. ISBN 0262042193.

[8] EDF. *The Sim-Diasca Simulation Engine*, 2010. http://www.sim-diasca.com.

[9] B. Everitt, S. Landau, M. Leese, and D. Stahl. *Cluster analysis*. Wiley, 5th edition, 2011. ISBN 978-0-470-74991-3.

[10] Feuerlabs. *Exometer*. `https://github.com/Feuerlabs/exometer`.

[11] A. Ghaffari. *The Scalability of Reliable Computation in Erlang*. PhD thesis, School of Computing Science, The University of Glasgow, 2015. Under review.

[12] E. Hewitt and K. Stromberg. *Real and abstract analysis. A modern treatment of the theory of functions of a real variable*, volume 25 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1965.

[13] V. Janjic and K. Hammond. Granularity-aware work-stealing for computationally-uniform Grids. In *CCGrid 2010, Melbourne, Australia*, pages 123–134, 2010.

[14] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley and Sons, New York, 1990.

[15] M. Krasner. Nombres Semi-Réels et Espaces Ultramétriques. *Comptes Rendus de l'Académie des Sciences, Tome II*, 219:433–435, 1944.

[16] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *HPDC'12, Delft, The Netherlands*, pages 137–148, 2012.

[17] P. Maier, R. Stewart, and P. Trinder. Reliable scalable symbolic computation: The design of SymGridPar2. *Computer Languages, Systems & Structures*, 40(1):19 – 35, 2014. Special issue on the Programming Languages track at the 28th ACM Symposium on Applied Computing.

[18] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12, 1959.

[19] C. Meiklejohn. Personal communication. May 2015.

[20] M. Middendorf, F. Reischle, and H. Schmeck. Multi colony ant algorithms. *Journal of Heuristics*, 8(3):305–320, May 2002.

[21] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *PGAS 2011, Galveston Island, TX, USA*, 2011.

[22] C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0201530821.

[23] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL `http://www.R-project.org/`.

[24] RELEASE Project. Deliverable D4.3: Heterogeneous super-cluster infrastructure, December 2012.

[25] RELEASE Project. Deliverable D4.4: Capability-driven Deployment, May 2013.

[26] RELEASE Project. Deliverable D3.4: Scalable Reliable OTP Library Release, September 2014.

[27] RELEASE Project. Deliverable D4.5: Scalable Infrastructure Performance Evaluation report, February 2015.

[28] W. Rudin. *Principles of Mathematical Analysis*. International series in pure and applied mathematics. McGraw-Hill, 1976. ISBN 9780070856134.

[29] Y. Zhu. A survey on grid scheduling systems. *Department of Computer Science, Hong Kong University of science and Technology*, page 32, 2003.