Multi-level Visualization of Concurrent and Distributed Computation in Erlang

Robert Baker
School of Computing
University of Kent, UK
rb440@kent.ac.uk

Peter Rodgers
School of Computing
University of Kent, UK
P.J.Rodgers@kent.ac.uk

Simon Thompson
School of Computing
University of Kent, UK
S.J.Thompson@kent.ac.uk

Huiqing Li
School of Computing
University of Kent, UK
H.Li@kent.ac.uk

***Abstract*—This paper describes a prototype visualization system for concurrent and distributed applications programmed using Erlang, providing two levels of granularity of view. Both visualizations are animated to show the dynamics of aspects of the computation.**

At the low level, we show the concurrent behaviour of the Erlang schedulers on a single instance of the Erlang virtual machine, which we call an Erlang node. Typically there will be one scheduler per core on a multicore system. Each scheduler maintains a run queue of processes to execute, and we visualize the migration of Erlang concurrent processes from one run queue to another as work is redistributed to fully exploit the hardware. The schedulers are shown as a graph with a circular layout. Next to each scheduler we draw a variable length bar indicating the current size of the run queue for the scheduler.

At the high level, we visualize the distributed aspects of the system, showing interactions between Erlang nodes as a dynamic graph drawn with a force model. Specifically we show message passing between nodes as edges and lay out nodes according to their current connections. In addition, we also show the grouping of nodes into "s_groups" using an Euler diagram drawn with circles.

I. Introduction

Erlang applications are typically concurrent and distributed, and run on multicore platforms. By visualizing the concurrent processes involved we can convey how individual processes are performing and communicating. This means the programmer can evaluate performance, verify program correctness and gain insight into program behaviour [10]. This paper describes a prototype visualization system for Erlang applications, giving two (low and high level) views of the system.

Erlang [2], [6] is a programming language with built-in support for concurrency and distribution. Erlang models the world as sets of parallel processes that can interact only by message passing, that is without any shared memory. Erlang concurrency is directly supported in the implementation of the Erlang Virtual Machine (VM), rather than indirectly by operating system threads.

Messages between processes can consist of data of any Erlang type: as well as the basic types (integers, booleans, strings and bit strings, or binaries) these can be tuples (consisting of a number of values of different types), lists or functions. Process identifiers and symbolic names can also be communicated as (parts of) messages.

Erlang supports multi-core programming "out of the box". At startup, the Erlang VM automatically detects the CPU topology on which it is running and creates a process scheduler for each available CPU core. Each process scheduler maintains its own run queue of processes to execute. In order to balance the workload between schedulers, a process migration (work stealing) mechanism is applied periodically, so that schedulers with less work pull processes from schedulers with more work, while schedulers with more work push processes to others.

A distributed Erlang system consists of a number of Erlang VMs communicating with each other. Each VM is called an Erlang *node*. Message passing between processes on different nodes is indistinguishable from communication on the same node (at least when process identifiers (*pids*) are used for message addressing). Whilst different Erlang nodes in a distributed Erlang system can run on physically separated computers, it is also possible for multiple nodes to run on the same computer.

The default model for an Erlang distributed system is for all nodes to be connected to all other nodes, but this has negative consequences for the scalability of systems. Erlang nodes in a distributed Erlang system can be partitioned into groups, where every element in a group is connected to all others, but the fact that the groups are disjoint means that certain kinds of architectures are not possible.

One of the outcomes of the ongoing FP7 RELEASE project [3] is the notion of *s_groups*. These extend Erlang's original partition-based grouping in a number of ways. As far as the work described in this paper is concerned, the major difference between s_groups and groups is that s_groups are not necessarily disjoint, and so one Erlang node may be a member of multiple s_groups, or indeed a member of none.

This paper describes a prototype visualization system for concurrent and distributed Erlang applications. In the system we provide two granularities of view: low level and high level.

The low level view shows the concurrent behaviour of Erlang processes on a single Erlang node, focussing in particular on scheduling aspects. The schedulers are laid out in a circle: we indicate the size of their run queues by the length of bars which are drawn next to the schedulers. We also visualize the migration of Erlang concurrent processes from one scheduler to another, as work is redistributed to fully exploit the hardware: this migration is represented by animated edges between schedulers.

The high level view visualizes the distributed aspects of the system, with each graph node representing an Erlang node. We

show message passing between nodes as edges, laying nodes out according to their current connections. The grouping of nodes into s_groups is shown by surrounding each s_group by an interlinking circle. Dynamic communication behaviour is animated using a force model [7].

The high level layout is produced by first drawing an Euler diagram for the intersecting s_groups of Erlang nodes [19]. The nodes are then placed in the appropriate zone of the diagram, and a force-directed layout is applied to the communication graph. The forces in the layout include the standard spring embedder forces [7], with an additional repulsion from the edges of the circles to keep the nodes within the correct regions [16]. Whilst the number of nodes and their grouping into s_groups is fixed and does not change during execution (at least in this prototype), the communication between nodes does change over time, and the use of a force based approach allows the layout of the graph to alter dynamically in an animated way.

The research contribution of this paper is to describe the first visualization for concurrent and distributed Erlang execution. The visualization has novel features:

- The use of Euler diagrams to show groups of virtual machine nodes in concurrent programs is new.
- Whilst inspired by previous approaches [16], [7], our method of integrating Euler diagrams with graphs represents a new combination of forces.
- The two different levels of view with discrete visualizations has not been seen, to our knowledge, in previous systems for concurrent and distributed visualization.
- The combination of bar charts and circular graph layout appears to be new in the context of concurrent process visualization.

In the remainder of this paper, we first discuss related work in Section II. Section III then provides an overview of the low level visualization and animation method, whilst Section IV discusses the high level method. Finally, Section V presents our conclusions and outlines possible future work.

II. Related Work

Circular layout has been used previously for processes on a core [10]. However, in this earlier work, communication was shown by dots moving from one process to another. This requires the users attention during the movement of the dot to identify the start and ending processes, which is particularly problematic when several processes are communicating at the same time. We improve on this by using curved edges to show the communication between two processes. These edges fade over time, to prevent the display from becoming overly cluttered. We note that the use of curved edges follow the Gestalt principle of good continuation [17], and so can be justified by perceptual theories. Justification for grouping processes by surrounding groups with contours – which overlap in the high level view, and do not overlap in the low level view – is provided by the preattentive principle of closure [20].

Visualization of communication graphs in concurrent processes has been prevalent for some time. Visputer [22] has a multi-level approach to visualizing such graphs, where a single node can contain either a process or a communication subgraph. This consistent approach to the different levels, where lower levels have the same syntax as higher levels, contrasts with our approach where the visual display of the two levels is tuned to the different diagram semantics. The Meander project [21] not only visualized communication graphs, it used them as the basis for a visual concurrent programming language. The use of graph visualization for the presentation of communication graphs is widespread, see for instance a 1992 survey [11]. More recent work has applied a variety of graph drawing methods to automatically generate layouts for various graphs associated with concurrent and distributed programs [4], [14], [18].

Another approach to visualising Erlang programs is provided by Percept2 [12]. Percept2 is a tool for offline visualisation of Erlang application level concurrency and identifying concurrency bottlenecks. It utilises Erlang's built-in support for tracing to monitor events from process states. Trace events are collected and stored in a file, which can then be analysed offline. Once the analysis is done, the data can be viewed in chart form through a web-based interface.

Percept2 gives a picture of application-level parallelism, as well as how much time processes spend waiting for messages. It also exposes scheduler-related activities to end-users, and provides finer-grained profiling information about the existing parallelism of an application. It is also designed to scale well to parallel applications running on multiple cores.

VampirTrace/Vampir [15] is a tool set and a runtime library for instrumentation and tracing of software applications. It allows a variety of detailed performance properties to be collected and recorded during runtime, including function enter and leave events, MPI communication, OpenMP [1] events, and performance counters. VampirTrace provides the input data for the Vampir analysis and visualization tool. Scalasca [8] is another representative trace-based tool that supports performance optimization of parallel programs by measuring and analysing their runtime behaviours. The analysis identifies potential performance bottlenecks, in particular those concerning communication and synchronization, and offers guidance in exploring their causes. Finally, ThreadScope [9] is used for performance profiling of parallel Haskell programs. ThreadScope reads GHC-generated tracing events from a log file, and displays the thread profile information in a web-based graphical viewer.

III. Low Level Visualization

The low level visualization shows the concurrent behaviour of Erlang processes on a single Erlang node. We visualize the migration of Erlang concurrent processes from one scheduler to another and the size of the run queue on each scheduler.

The visualization system (both low level and high level components) is developed in JavaScript and rendered in SVG using the Data-Driven Documents library (D3) [5]. The visualization is re-drawn on every iteration and this gives the effect of animating the state of the schedulers to reflect the current

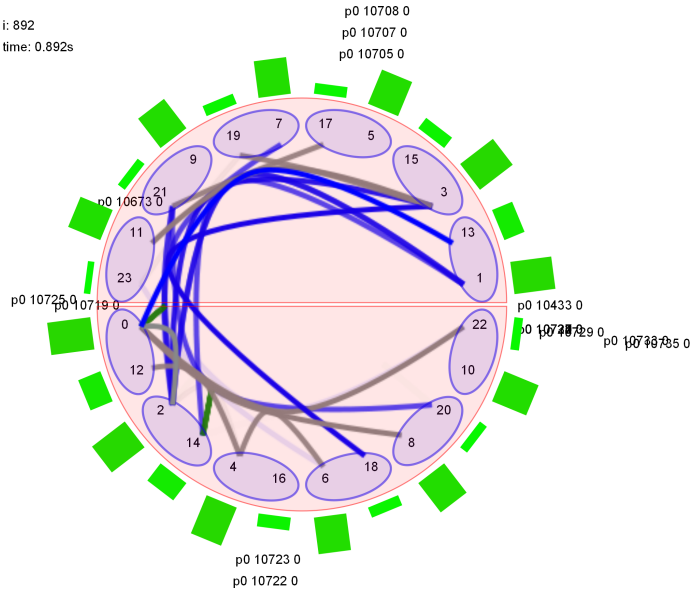

Fig. 1. Low Level Visualization

situation. This constant re-drawing is required as the system is dynamic: processes can be created, terminate or migrate between schedulers at any time.

The data used in this paper was generated from running Dialyzer [13], the Erlang static type analysis tool, on a selection of Erlang applications and observing the computation. During the running of the application, every process migration between schedulers is recorded using Erlang's built-in support for tracing, and the size of run queues is sampled every millisecond. The trace data are recorded in an Erlang file, with each trace datum represented by an Erlang tuple (that is collection of Erlang values), and data analysis is performed by an Erlang program.

The application was run on an Intel machine, running CentOS 6, with two processors, each having six physical CPU cores. Through hyper-threading technology, the OS is able to address two logical cores for each physical CPU core, therefore there are 24 logical cores in total for an Erlang node to exploit.

An example frame in the low level visualization can be seen in Figure 1. The parallel machine is split into two six-core processors, represented by the two pink semicircles. Erlang schedulers are shown as a number that represents the unique ID of the related threads. Pairs of Erlang schedulers that run on a single real core are shown grouped by the blue ellipses.

Each Erlang scheduler has an individual run queue, and this is represented by a bar radiating from the visualization. The height and colour are used to indicate the size of the run queue: larger queues (i.e. busier Erlang schedulers), are shown with longer and redder bars, whereas smaller queues are represented by shorter and greener bars. The run queue size can be displayed on screen as a value by selecting the relevant option in the interface, or by hovering a cursor over the bar or Erlang scheduler in question. It is possible that the queues can be so busy that they are not fully displayed on screen, if this is the case, the user is able to take advantage of the scalability of SVG graphics and alter the zoom level on their browser.

Processes get spawned on a particular scheduler. The list of numbers radiating from a Erlang scheduler shows the unique ID for each process spawned on that scheduler. As can be seen in Figure 1, Erlang scheduler 5 has just spawned a number of new processes.

Processes may migrate from one Erlang scheduler to another and this is shown by edges between various schedulers. These are curved to avoid occlusion with other schedulers (e.g. a migration from 10 to 18 would cross schedulers 20 and 8). The edges are coloured to indicate the three types of migration: within a single core (green), between different cores on the same processor (grey), and from a core on one processor to a core on the other (blue), although these migrations can also be deduced from the start and end position of the edge. These edges fade away after a few seconds to prevent the visualization from becoming occluded.

The entire visualization displays the current state of the trace and is animated such that a new time frame (iteration) is shown every 200 milliseconds and each time frame is 1 millisecond sample of the original trace, that is, the animation runs 200 times more slowly than the original application. This allows the user to focus on the changing state without being overwhelmed with the amount of animation. In addition, it allows enough processing time for the animations to display correctly on slower machines. The animation can be paused and resumed using the interface. A slider allows the selection of a specific time frame and permits the animation of a sequence of time frames in either direction.

Figure 2 shows three snapshots of the low level view at various stages of execution of an Erlang program. The top diagram shows the state at iteration 890, where the none of the schedulers are particularly busy, and the amount of communication between different schedulers is correspondingly low. The middle diagram, at iteration 967, shows the Erlang node in a much more active state , here the run queues are extremely uneven. The bottom diagram shows the state at iteration 3804, where the disparity in run queue size is now extreme, with three outliers that have full run queues, and others with relatively short queues.

A number of interesting phenomena are indicated by the visualizations.

- A clear consequence of hyper-threading is that there is an asymmetry between the two schedulers on a single physical core. This is evident in the visualisation in the different lengths of the two run queues on a single core (e.g. cores 0 and 12), resulting in the alternating pattern seen in the figure.
- Setting aside this intra-core issue, migration does not ensure that run queues have the same size: however this is not necessary, since all that is needed is that each core has enough work to ensure that it is fully occupied.

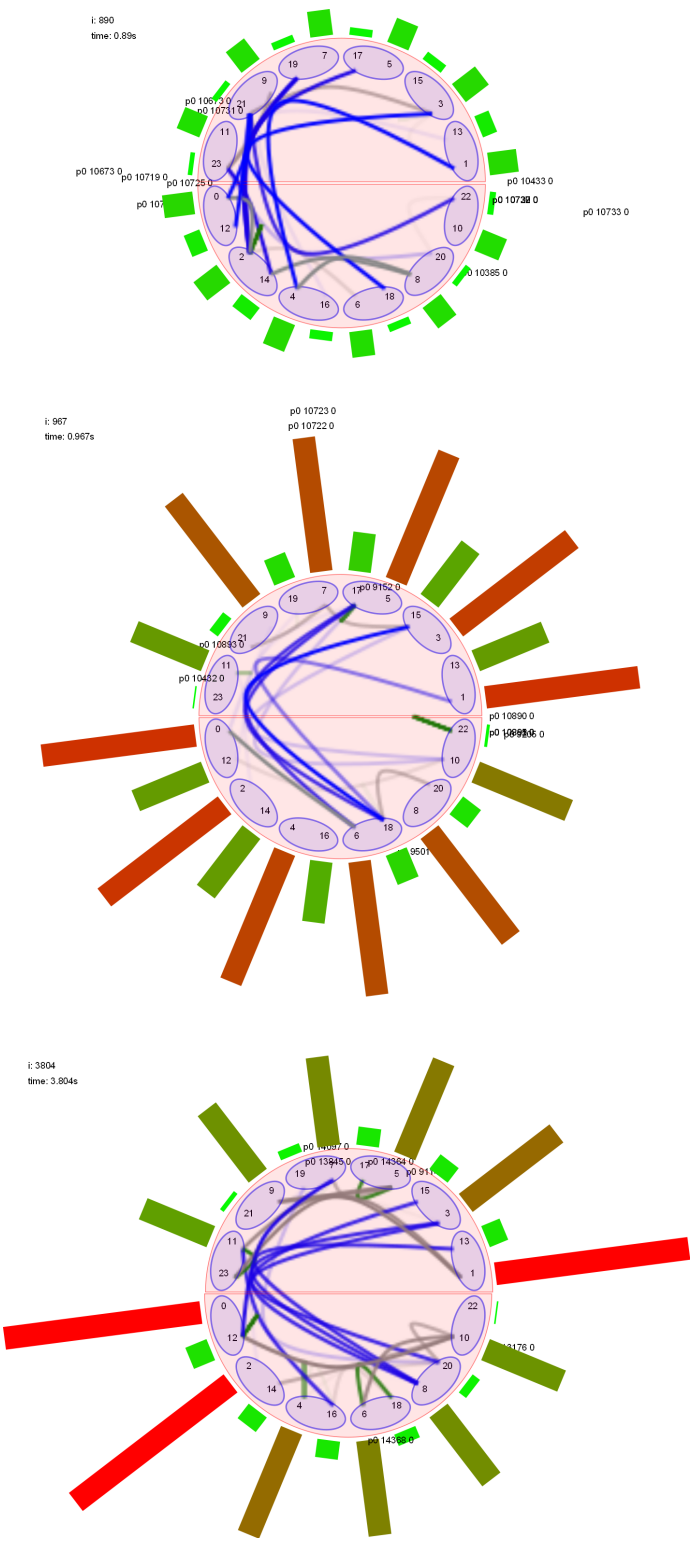

Fig. 2. Three steps in the low level view

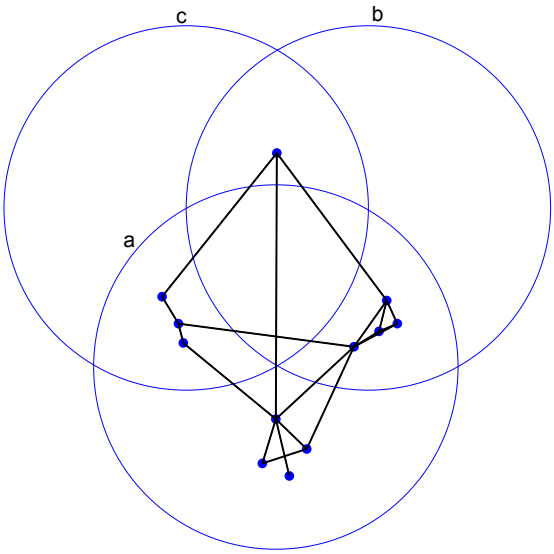

Fig. 3. High level view.

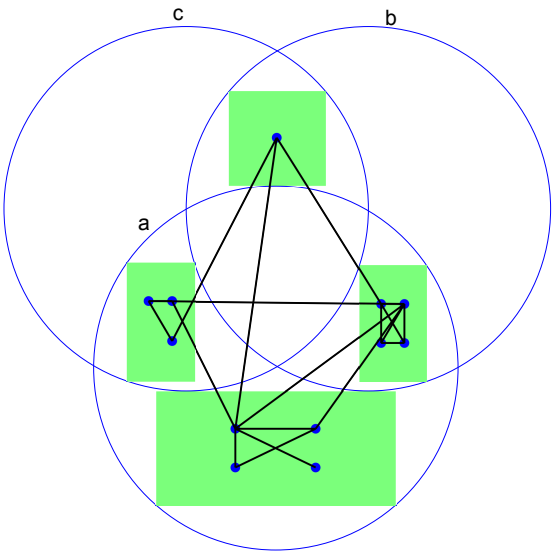

Fig. 4. Initial placement of Erlang nodes before force application.

IV. High Level Visualization

The high level visualization shows the grouping of Erlang nodes within particular s_groups and edges visualizing the communication between the nodes. Each node on this high level visualization corresponds to an instance of the low level visualization.

An example of the high level visualization is shown in Figure 3. The s_groups are drawn first, with each s_group being represented by a circle. As a node may be the member of more than one s_group, the circles typically intersect. The s_group membership is derived and is passed to code described by [19] which returns the circle centres and radii that will then be drawn in SVG. The intersecting circles form connected regions and the largest rectangle within each region is calculated (shown in green in Figure 4). Nodes that are

contained in the relevant s_groups are drawn in a grid pattern within the rectangle. The method for drawing Euler diagrams often creates extra regions, in which no node is placed. In some visualization systems (particularly those using Venn diagrams) these empty regions can be shaded to indicate that they have no elements inside. However, we regard it as self-evident that there are no elements in empty regions and so we do not add the extra syntactic complexity of shading.

A force-directed layout is then applied to the nodes to improve the layout. The forces are the standard spring embedder attraction and repulsion forces [7], with an additional force. This third force is applied to a node to keep it away from the borders of the region, and is a simplified version of the force introduced in [16]. It is proportional to the node's distance from the circumference every circle, pushing the node toward the centre of circles in which it is inside, whilst pushing the node away from the centre of circles which it is outside of. Additionally, a test is made to ensure that nodes do not leave the region to which they belong by disregarding any node movement that moves it outside the correct region (although nodes are allowed to move outside their starting rectangle).

Communication between nodes is represented by edges and these may change during the program's execution. The shading of the edges is related to the frequency of messages within the timeframe. The darker the edge, the more frequently messages are sent. Edges are removed if no communication exists between them and new edges are introduced when communication is initiated between nodes. The use of a force-directed layout method allows the visualization to be dynamic, so that the repositioning of the nodes can be shown in an animated manner as the diagram reaches a new equilibrium.

The data used in this paper was generated by running a distributed Erlang benchmark application on a collection of Erlang nodes, which were configured to form the s_groups (these s_groups were manually defined, as no s_group data is currently present in the trace data). The total wallclock runtime of the application was 222 seconds. On each participating Erlang node, Erlang's built-in support for tracing was used to record message sending events. Each such event recorded contains the name of the sending node, the name of the receiving node as well as the actual message sent. Trace data from each node were written to a file, and the data files were then collected, analyzed and synthesized to provide the visualization tool with the accumulated communication data on each iteration, which occurs every 200 milliseconds.

Figure 5 shows some steps in the animation process. The top diagram shows the visualization after 100 iterations since the start of the trace. At this point there are relatively few communication edges between Erlang nodes. In the middle diagram, an iteration later, the number of communication edges suddenly increases dramatically, indicating that the scheduling has initiated a number of communications between nodes. In terms of the visualization, the middle diagram has not had time for the forces to be applied to the nodes, and so the nodes are shown in the same position as in the top diagram. After another 99 iterations, the forces find an new equilibrium,

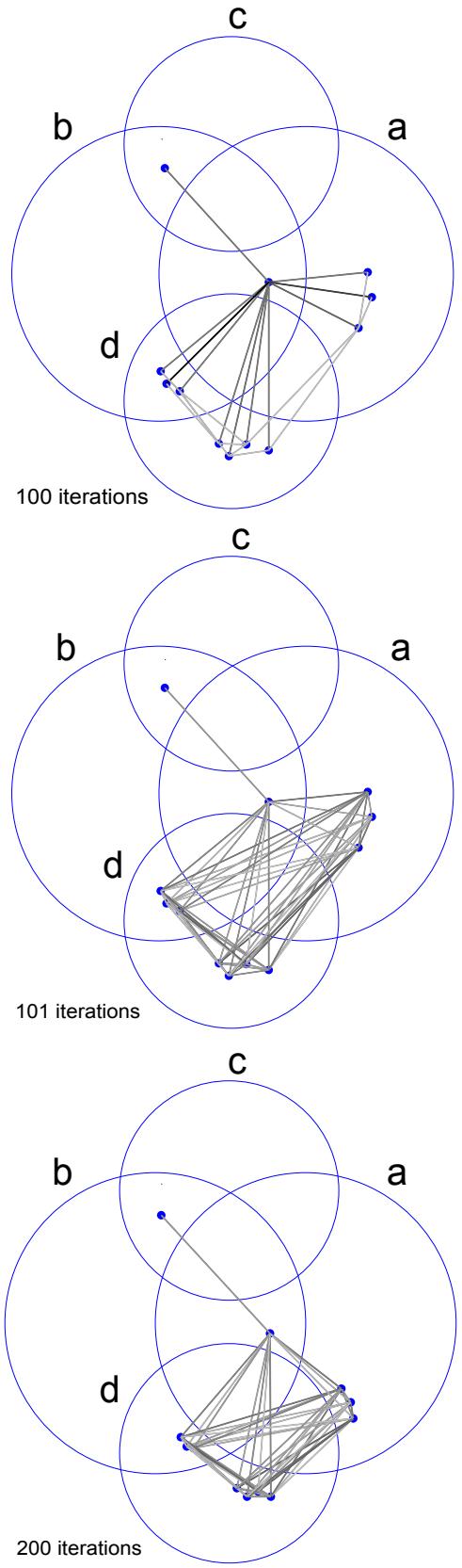

Fig. 5. Three steps in the animation of the high level view.

with the nodes in the positions as shown in the lower diagram. Here, the nodes have moved further away from the region centres, so indicating a stronger affinity to nodes outside their regions. However, the only node in s_group "c" (top circle) has been relatively unaffected by this, as the number of nodes it is communicating with has not changed.

V. Conclusion

This paper has described a two level visualization for Erlang distributed and concurrent processes. The complexity of such systems requires a mixed approach where concurrency and distribution are visualized by distinct methods.

Concurrency in Erlang nodes has been represented in a low level visualization by animating communication edges between Erlang schedulers which are shown in a circular layout. How busy each Erlang scheduler is can be seen by the length of the bar associated with that scheduler.

The distributed nature of Erlang computation is shown in a high level visualization using a dynamic graph of communication between Erlang nodes. Nodes are placed in s_groups which is shown by drawing the nodes in regions of a Euler diagram drawn with circles. The changes in communication between nodes is animated by a force-directed layout, which also maintains the nodes in their s_groups.

Informal feedback from RELEASE project members has been solicited. This has resulted in broadly positive reactions, with suggestions for alternative data that might be represented by the edges between processes or nodes. This leads to the first aspect of further work: once the system is more stable, there is potential for formal evaluation of the system to see if it is successful in its aims of:

- assisting system performance evaluation;
- aiding the verification of program correctness; and
- improving insight into program behaviour.

Given the limited number of distributed Erlang experts, and the considerable amount of time that would need to be devoted to using the visualizations to get useful information, a longitudinal study with a handful of participants might be the best mechanism for this. Before such an evaluation, the user interface will have to be improved, and the connection between the two levels of visualization needs to integrated more closely.

Other further work relates to modifying the system to deal with an increasing amount of distribution. The current high level visualization shows each Erlang node as a graphical node. Whilst this has the advantage of showing all Erlang nodes, it can become unwieldy when their number grows. To allow the system to scale we expect to cluster Erlang nodes, using mechanisms similar to those developed in [22]. When applied to distributed Erlang, a graphical node might represent an s_group, and edge affinity could be a composite measure of overlap and communication bandwidth between groups.

The system so far supports *offline* visualisation, but the same graphical approach can be used to provide an *online* – that is, real time – view. The challenge of online visualisation in this case is to provide streams of suitable aggregate data without adversely affecting the performance of the system under observation. These streams of data could be provided by the built-in Erlang tracing mechanism, or using a Unix teaching framework such as DTrace / SystemTap.

Acknowledgements This research is supported by EU FP7 collaborative project RELEASE (http://www.release-project.eu/), grant number 287510.

References

[1] The OpenMP API specification for parallel programming. http://openmp.org/wp/.
[2] Erlang/OTP. http://www.erlang.org, May 2013.
[3] RELEASE. http://www.release-project.eu/, May 2013.
[4] Benjamin Albrecht, Philip Effinger, Markus Held, Michael Kaufmann, and Stephan Kottler. Visualization of complex bpel models. In *Graph Drawing*, pages 421–423. Springer, 2010.
[5] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011.
[6] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009.
[7] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
[8] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika brahm, Daniel Becker, Bernd Mohr, and Forschungszentrum Jlich. The SCALASCA performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010.
[9] Don Jones Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *Haskell Symposium 2009*, Edinburgh, Scotland, September 2009. ACM Press.
[10] Eileen Kraemer and John T Stasko. Creating an accurate portrayal of concurrent executions. *Concurrency, IEEE*, 6(1):36–46, 1998.
[11] Eileen T Kraemer and John T Stasko. The visualization of parallel systems: An overview. 1992.
[12] H. Li and S. Thompson. Multicore Profiling for Erlang Programs Using Percept2. In *Erlang Workshop 2013*, 2013.
[13] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.
[14] J. Lonnberg, M. Ben-Ari, and L. Malmi. Visualising concurrent programs with dynamic dependence graphs. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pages 1–4, 2011.
[15] Matthias S. Mller, Andreas Knpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2007.
[16] P. Mutton, P. Rodgers, and J. Flower. Drawing graphs in Euler diagrams. In *Proceedings of 3rd International Conference on the Theory and Application of Diagrams*, volume 2980 of *LNAI*, pages 66–81, Cambridge, UK, March. Springer.
[17] A. Rusu, A.J. Fabian, R. Jianu, and A. Rusu. Using the gestalt principle of closure to alleviate the edge crossing problem in graph drawings. In *Information Visualisation (IV), 2011 15th International Conference on*, pages 488–493, 2011.
[18] Lucas Mello Schnorr, Arnaud Legrand, Jean-Marc Vincent, et al. Interactive analysis of large distributed systems with topology-based visualization. 2012.
[19] Gem Stapleton, Leishi Zhang, John Howse, and Peter Rodgers. Drawing euler diagrams with circles: The theory of piercings. *Visualization and Computer Graphics, IEEE Transactions on*, 17(7):1020–1032, 2011.
[20] Anne Treisman and Janet Souther. Search asymmetry: a diagnostic for preattentive processing of separable features. *Journal of Experimental Psychology: General*, 114(3):285, 1985.
[21] Guido Wirtz. A visual approach for developing, understanding and analyzing parallel programs. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 261–266. IEEE, 1993.
[22] Kang Zhang and Gaurav Marwaha. Visputera graphical visualization tool for parallel programming. *The Computer Journal*, 38(8):658–669, 1995.