`VisPar`: Visualising Dataflow Graphs from the *Par* Monad

Maximilian Algehed
Chalmers University of Technology, Sweden
algehed@chalmers.se

Patrik Jansson
Chalmers University of Technology, Sweden
patrikj@chalmers.se

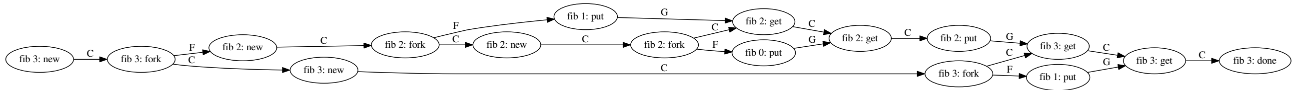


Figure 1. A teaser: the extended dataflow graph of the call *fib* 3

Abstract

We present a work in progress tool (`VisPar`) for visualising computations in the *Par* monad in Haskell. Our contribution is not a revolutionary new idea but rather a modest addition to the set of tools available for making sense of parallel programs. We hope to show that `VisPar` can be useful as a teaching tool by providing visualisations of a few examples from a course on parallel functional programming.

CCS Concepts •**Computing methodologies** → *Parallel programming languages;* •**Human-centered computing** → *Graph drawings;* •**Software and its engineering** → Domain specific languages;

Keywords Functional Programming, Parallel Programming, Visualization, Threads

ACM Reference format:
Maximilian Algehed and Patrik Jansson. 2017. `VisPar`: Visualising Dataflow Graphs from the *Par* Monad. In *Proceedings of FHPC'17, Oxford, United Kingdom, September 7, 2017,* 6 pages.
DOI: 10.1145/3122948.3122953

1 Introduction

Writing parallel programs is difficult and achieving good performance often requires expert knowledge. One reason why this is the case is that the behaviour of a parallel program is more difficult to predict and analyse than that of a conventional sequential program. There is a clear need for good tools for understanding and debugging parallel programs both in education and elsewhere.

The *Par* monad [18] provides an interface for writing deterministic task-parallel programs in Haskell [11]. The need for visualisation of *Par* monad code was noted by Marlow in his book [17, p. 60]:

> Unfortunately, right now there's no way to generate a visual representation of the data flow graph from some *Par* monad code, but hopefully in the future someone will write a tool to do that.

Code written in the *Par* monad builds an explicit dataflow graph where results of parallel computations are communicated between nodes in the graph using I-Structures [2], called *IVar*s in the *Par* monad. In this paper we present a prototype tool we call `VisPar` which provides visualisation of *Par* computations in terms of these dataflow graphs. More precisely, we make the following contributions:

- We present a tool for providing a dataflow graph as output from executing code written in the *Par* monad (Sections 3 and 6).
- We present two ways of visualising these graphs in an understandable way (Sections 3 and 5.1).
- We present a way to estimate the performance characteristics of *Par* computations from the dataflow graphs (Section 4).
- We show how the above contributions make it possible to analyse the parallel behaviour of programs and apply it to an example from an exam in a Parallel Functional Programming course at Chalmers (Section 5.2).

FHPC'17, Oxford, United Kingdom

© 2017 ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of FHPC'17, September 7, 2017*, http://dx.doi.org/10.1145/3122948.3122953.

2 The *Par* Monad

The *Par* monad provides a small interface for explicit task parallelism.

$$
\begin{aligned}
&\mathit{fork} && :: \mathit{Par}\ () \to \mathit{Par}\ () \\
&\mathit{new} && :: \mathit{Par}\ (\mathit{IVar}\ a) \\
&\mathit{put} && :: \mathit{NFData}\ a \Rightarrow \mathit{IVar}\ a \to a \to \mathit{Par}\ a \\
&\mathit{get} && :: \mathit{IVar}\ a \to \mathit{Par}\ a \\
&\quad \text{-- Derived operation} && \\
&\mathit{spawn} && :: \mathit{NFData}\ a \Rightarrow \mathit{Par}\ a \to \mathit{Par}\ (\mathit{IVar}\ a)
\end{aligned}
$$

- The *fork* primitive takes a *Par* computation and runs it in parallel with the rest of the program as a light-weight "thread". Note that the return type of *fork* :: *Par* () → *Par* () ensures that results from the new thread must be communicated through other means (using *IVar*s, as we will see below).
- The *new* primitive returns a new *IVar*, which can be used to communicate results of parallel computations between threads using the *put* and *get* primitives.
- The *put* primitive evaluates a value and puts it in an *IVar*. The *NFData a* constraint means that values of type *a* can be strictly evaluated to normal form (the kind of normal form is decided by the instance of *NFData* for *a*).
- The *get* primitive takes an *IVar* and blocks until a result is *put* into it by another thread.

The *put* and *get* operations can be performed on an *IVar* from anywhere in the *Par* computation. However, each *IVar* may only

be *put* into once. Performing multiple *put*s into the same *IVar* will result in a runtime error.

Finally, the derived *spawn* operation abstracts a common pattern found in many programs written in the *Par* monad: forking a *Par* program and communicating the result through an *IVar*.

```
spawn :: NFData a ⇒ Par a → Par (IVar a)
spawn p = do
  v ← new
  fork (p ≫= put v)
  return v
```

The run function for *Par* monad computations is a scheduler *runPar* :: *NFData a* ⇒ *Par a* → *a* which we later (in Section 6) explain and extend to also produce graphs.

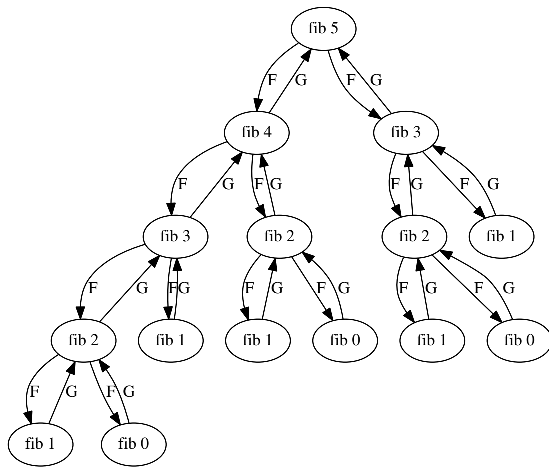


Figure 2. The simple dataflow graph for *fib* 5

As an example of a *Par* program consider the *fib* function given below, which computes the *n*th Fibonacci number in the *Par* monad.

```
fib :: Int → Par Int
fib n | n < 2     = return 1
      | otherwise = do
        lv ← spawn $ fib (n − 1)
        rv ← spawn $ fib (n − 2)
        l  ← get lv
        r  ← get rv
        return (l + r)
```

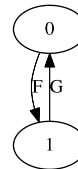
In the case where $n \geqslant 2$ we spawn two threads which will compute *fib* $(n-1)$ and *fib* $(n-2)$ in parallel and put their results in the *IVar*s *lv* and *rv*. We then obtain the results of the two recursive calls from *lv* and *rv* respectively. Finally we return the sum of the result of *fib* $(n-1)$ and *fib* $(n-2)$.

3 Visualisation

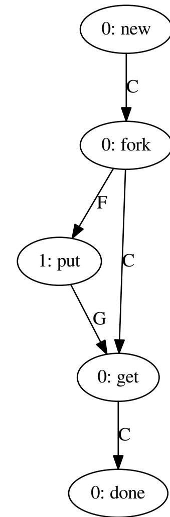
In order to provide informative visualisation we extend the *Par* monad interface with a function *forkNamed* :: *String* → *Par* () → *Par* (). This makes it possible to give forked threads names, as

```
tiny :: Par Int
tiny = do
  ivar ← new
  fork (put ivar 5)
  get ivar
```

(a) Source code of *tiny*.



(b) The simple dataflow graph



(c) The extended dataflow graph.

Figure 3. A *tiny* program (a) with its dataflow graphs (b), (c)

can be seen in the graph for the *fib* function in Figures 1 and 2. The default name is a thread ID (a counter starting from 0). Using *forkNamed* we also implement *spawnNamed* :: *String* → *Par a* → *Par* (*IVar a*) in a way very similar to the implementation of *spawn*, only using *forkNamed* instead of *fork*. We also found it useful to have a function *withLocalName* :: *String* → *Par a* → *Par a* which locally changes the name of a thread without forking. To produce the visualisation of *fib* in Figures 1 and 2 we modify the definition of *fib* in Section 2 to use *spawnNamed* instead of *spawn*:

```
fib :: Int → Par Int
fib n | n < 2     = return 1
      | otherwise = do
        lv ← spawnNamed ("fib " ++ show (n − 1)) $ fib (n − 1)
        rv ← spawnNamed ("fib " ++ show (n − 2)) $ fib (n − 2)
        l ← get lv
        r ← get rv
        return (l + r)
```

The graphs in this paper are one way of visualising parallel computations and other ways are certainly possible. Each node represents an *event* in the computation and is labeled with its thread identifier and event type. Each event corresponds to an invocation of one of the functions from the *Par* interface: *fork*, *new*, *put*, and *get*. An edge from a source event to a target event denotes a relationship between the corresponding threads:

- An **F** means the source thread forked the target thread.
- A **C** means that the target is the continuation of the source; the target event occurs after the source event in the same thread.
- A **G** means that the target event depends on the result of a *get* from an *IVar* filled by the source event.

Using our extended scheduler to run the code for *tiny* in Figure 3a gives the graph in Figure 3c. We refer to these graphs as extended dataflow graphs as they contain detailed information about the execution of the program. However, as evidenced by Figure 1, the graphs for simple parallel computations like *fib* 3 quickly become large. We therefore also provide a way to obtain simpler dataflow graphs like the one for *tiny* in Figure 3b. The "simple" graph only contains **F** and **G** edges, omitting the precise structure of the computation and events.

4 Work, Depth, and VisPar Graphs

This section briefly covers how some properties of VisPar graphs relate to the notions of work and depth, in turn related to the running time, of the program from which the graph was generated. Work, W, is defined as the total number of operations performed in a parallel computation. Depth, D, is defined as the length of the longest sequential chain of dependencies in the computation [4]. The following inequalities, due to Brent [5], relate running time, T, on P processors to work and depth:

$$max(\frac{W}{P}, D) \leqslant T < \frac{W}{P} + D \tag{1}$$

Similarly, we have that the maximum speedup, S, available by parallelisation is bounded above by [6]:

$$S \leqslant \frac{W}{D} \tag{2}$$

Using the graphs produced by VisPar we can compute lower bounds for both work and depth for an execution of a *Par* program. The work for a particular run of a *Par* program is bounded below by the number of nodes (w) in the VisPar graph. Similarly, depth is bounded below by the length (d) of the longest path from the initial (indegree zero) node to the terminal (outdegree zero) node in the graph. While this measure is only a crude approximation, it ignores the fact that a single node in the graph may represent an arbitrary, unbounded, amount of work, it does provide some insight into the behaviour of our program. Specifically, it gives a lower bound on the running time, $max(\frac{w}{P}, d)$, by equation (1). Bearing this simple connection in mind can help better understand and estimate bounds for parallel algorithms (see Section 5.2).

5 Case Studies

In this section we present two case studies which show VisPar in action.

5.1 Visualising Merge Sort

The following code for *mergeSort* implements a parallel version of the merge sort algorithm using the *Par* monad:

```
mergeSort :: (NFData a, Ord a) ⇒ Int → [a] → Par [a]
mergeSort 0 xs      = return (sort xs)
mergeSort d xs
    | length xs ⩽ 1 = return xs
    | otherwise     = do
      let (ls, rs)  =  splitAt (length xs `div` 2) xs
      lv            ← spawn (mergeSort (d − 1) ls)
      rv            ← spawn (mergeSort (d − 1) rs)
      lr            ← get lv
      rr            ← get rv
      return (merge lr rr)
```

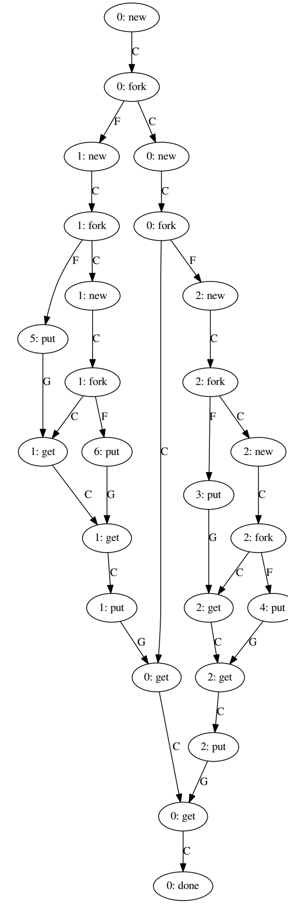


Figure 4. The extended dataflow graph for *mergeSort* 2 *xs*

The first argument controls the "depth of parallelism", giving control over granularity, *mergeSort d xs* will only spawn parallel computations to a depth of d (thus a maximum of 2^d threads). In the base case (at depth zero) *mergeSort* resorts to the sequential *sort* function from the Haskell base libraries [11]. Otherwise, at non-zero depth, the empty and singleton lists are both already sorted and can be returned as is. In the final case the list is first split into two halves, *ls* and *rs*, and sorted in parallel by spawning two recursive calls to *mergeSort* using *spawn*. (Note the decreasing argument d). Finally, the results of the two recursive calls, *lr* and *rr*, are obtained from the *IVars lv* and *rv* and combined using *merge*, which merges two sorted lists into a sorted list. Figures 4 and 5 show how VisPar renders the extended and simple dataflow graphs for *mergeSort* 2 *xs* with a long list *xs*.

While the simple graph hides a lot of the information about the implementation of *mergeSort* present in the extended graph (as it omits the ordering of operations like *new* and *put*), it makes explicit the divide and conquer nature of the merge sort algorithm in a way which is more difficult to see in the extended graph.

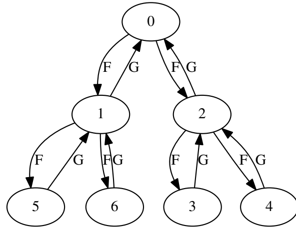


Figure 5. Merge sort dataflow graph for depth 2

5.2 Debugging *reduce*

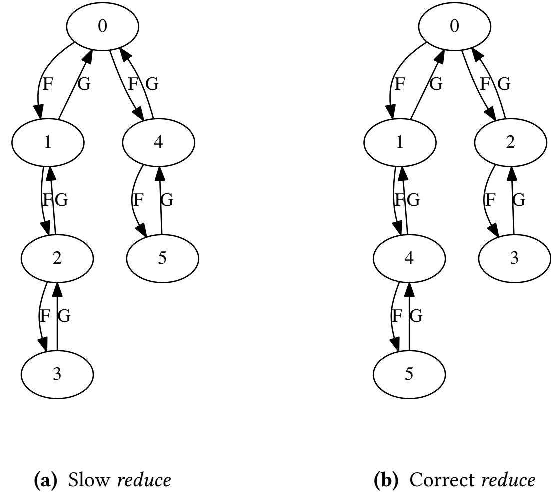
We now turn our attention to how `VisPar` visualisations can increase understanding, and ease debugging of parallel programs. We briefly cover the use of `VisPar` to tackle a recent exam question in a course on parallel functional programming at Chalmers University of Technology [21]. The original problem featured an underperforming version of the parallel *reduce* function written in Erlang [1] and has been translated to the following *Par* monad code for this paper:

```
reduce :: NFData a ⇒ (a → a → a) → [a] → Par a
reduce f [x] = return x
reduce f xs = do
  let (ls, rs) = splitAt (length xs `div` 2) xs
  rv ← spawn (reduce f rs)
  r  ← get rv
  l  ← reduce f ls
  return (f l r)
```

This implementation correctly computes the reduction of a list using an associative binary operation. However, benchmarking this function will reveal that it runs slowly, never utilising more than one core at a time. The task is to suggest a simple fix which will make the program perform well. Figure 6a shows the simple dataflow graph for the buggy version of *reduce*. From this graph alone it is certainly not obvious why the code runs slowly.

However, the extended dataflow graph in Figure 7a provides a detailed trace allowing us to study the behaviour which gives rise to the poor performance. From the graph it is evident that the entire computation is sequential, in spite of using *fork* to create a new thread, something which is not evident in Figure 6a. Recall from the discussion in Section 4 that the work and depth of this particular run of *reduce* can be approximated from this extended dataflow graph. In this case the approximation is remarkably accurate; assuming the work in the call $f\ l\ r$ is constant, both the work and depth can be computed to be 21 from the graph in Figure 7a. By equation (2) we can compute that there is no available speedup, $S \leqslant \frac{W}{D} = 1$, making our intuition from looking at the graph precise.

The problem is that the line $r \leftarrow get\ rv$ comes directly after the line $l \leftarrow reduce\ f\ ls$. This means that only one thread is active at a time while its parent thread waits for it to finish executing before continuing with the rest of the code. Fixing this error by swapping the two lines instead gives rise to the simple dataflow graph in Figure 6b. The tree shape is identical which makes it hard to see that the parallelism has increased. If we instead turn to the extended dataflow graph in Figure 7b, it clearly shows that useful parallel work can be done after the code has been fixed. The depth is now only 12 and computing the maximum speedup now gives a more encouraging upper bound $S \leqslant 1.75$. While the upper bound for the buggy version of *reduce* will be 1 regardless of the length of the input, the bound for the correct version will differ depending on the length of the input.



(a) Slow *reduce* **(b)** Correct *reduce*

Figure 6. Simple dataflow graph of a slow (a) and correct (b) *reduce* on a list of length six. Here it is hard to see the improvement.

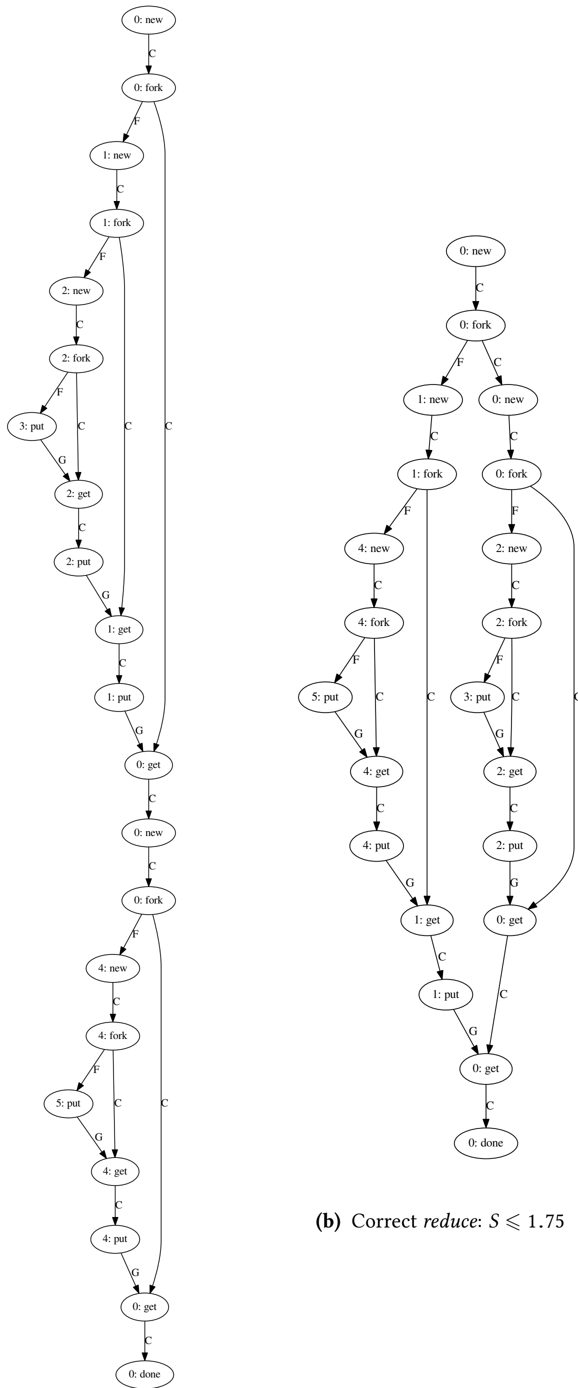
6 The Implementation of `VisPar`

This section gives a brief overview of the implementation of the visualisation presented in this paper. (The code of `VisPar` can be found online at https://github.com/MaximilianAlgehed/VisPar.)

We begin with recap of the implementation of the *Par* monad by Marlow et al. It is implemented in continuation passing style by the type **newtype** $Par\ a = Par\ ((a \rightarrow Trace) \rightarrow Trace)$ where the *Trace* type is shown below. The function *runPar* is by default implemented using a work-stealing scheduler, but as Marlow et al. point out, other schedulers can be implemented without changing the core implementation of *Par*.

```
data Trace where
  Get  :: IVar a → (a → Trace) → Trace
  Put  :: IVar a → a → Trace
  New  :: (IVar a → Trace) → Trace
  Fork :: Trace → Trace → Trace
  Done :: Trace
    -- Our added primitives
  SetName :: String → Trace → Trace
  GetName :: (String → Trace) → Trace
```

The implementation of the visualisation using numbers as names for threads is a straightforward instance of this idea. It is implemented as an extension of the work-stealing scheduler provided by Marlow et al. In this sense the visualisation is an example of the modularity of the *Par* monad. However, to implement the



(b) Correct *reduce*: $S \leqslant 1.75$

(a) Slow *reduce*: $S \leqslant 1$

Figure 7. Extended dataflow graph of a slow (a) and correct (b) *reduce* on a list of length six. Note the increase in parallelism.

forkNamed and *withLocalName* primitives a small change needs to be made to the implementation of the *Trace* data type. We add two constructors $SetName :: String \to Trace \to Trace$ and $GetName :: (String \to Trace) \to Trace$ which are used to implement the primitives $setName :: String \to Par\ ()$ and $getName :: Par\ String$. We do not expose the *getName* primitive to the programmer as it would provide a method for introspection into the behaviour of the scheduler by observing the thread name. This could be used to break determinism and referential transparency when used with a non-deterministic scheduler. Using *getName* and *setName* we can implement *withLocalName* and *forkNamed* as seen below. The implementation of *forkNamed* forks a thread which first sets its name and then executes normally. Similarly, the *withLocalName* primitive first obtains the current name, sets its name to the provided *name*, runs the computation *p*, sets the name to the original name, and finally returns the result of *p*.

```
forkNamed :: String → Par () → Par ()
forkNamed s p = fork (setName s ≫ p)
withLocalName :: String → Par a → Par a
withLocalName name p = do
  old ← getName
  setName name
  a ← p
  setName old
  return a
```

We use Erwig's functional graph library [8] to construct the dataflow graph as the computation is run. Finally we use `graphviz` [9] to render the graph (currently as a PDF).

7 Related Work

The Eden Trace Viewer [3] is a tool to visualize parallel functional program execution in the Haskell extension Eden. The `ThreadScope` tool [12] provides an interface for visualising the resource consumption of parallel programs in Haskell. VisualStudio 2010 provides similar functionality to `ThreadScope` [20]. In the functional programming domain the `percept` tool [15] for Erlang provides functionality similar to that of `ThreadScope`. Tools like these are very useful for debugging the resource consumption of parallel programs. However they do not provide a comfortable interface for visualising concrete behaviour. As such `VisPar` may be used by Haskell programmers as a complementary tool alongside other tools like `ThreadScope`. We also remark that these tools work by tracing the computation rather than relying on modifying the code (and library) being visualised. However, our technique can also be used with the standard *Par* monad to produce graphs without named threads like the one in Figure 4, provided the source code does not make use of the *forkNamed* and related primitives.

The high performance computing community has a long history of development tools for visualising and analysing parallel computations. Score-P [14] is a performance measurement runtime infrastructure that works with several HPC tools (Vampir, Periscope, Scalasca, TAU, etc.). As an example, the Vampir tool set [13], is widely used in the HPC community for visualising and analysing MPI traces. Most of these tools focus on *tracing* actual executions while our tool (VisPar) uses a custom interpretation (of the Par monad API) instead.

The gotracer and gothree.js tools [7] allow Go [10] programmers to visualise concurrent and parallel computations as interactive three dimensional animations. While we believe it would be possible to visualise our graphs in a similar way, the graphics presented in this paper are our initial experiments and providing more intuitive views is part of our ongoing work in this domain.

8 Conclusions and Future Work

The work presented in this paper is a first step towards providing visual aids for building parallel programs in Haskell. We have shown how a small extension to the *Par* monad allows us to provide useful visualisation of the dataflow graphs of parallel programs. Different visualisations of the same dataflow graph provide information and insight about different aspects of the computation. A detailed view of a graph helps us understand the fine-grained dynamic behaviour of parallel programs while a more simple view lets us clearly see the overall structure of the algorithm used. The final goal is to develop more comprehensive tools for multiple parallel programming models, possibly including approaches like Strategies [19] and Repa [16], for use by students in a course on parallel functional programming at Chalmers University of Technology. This goal provides multiple interesting avenues for future work.

While the visualisation technique presented in this paper provides an understanding of the behaviour of a parallel algorithm there is certainly room for experimentation with alternative techniques. For example, having display options for grouping several nodes into one could enable visualisation of larger graphs. Similarly, having an interactive visual environment for exploring the graphs produced by the tool could provide programmers with more in-depth understanding of the behaviour of their programs. Finally, the interface presented in this paper contains only the three primitives *forkNamed*, *withLocalName*, and *setName*. It is possible that there are other useful primitives that could be incorporated to give programmers more sophisticated debugging tools. For example, providing primitives that allow the programmer to estimate the amount of work at a node, giving more accurate estimates for the total work and depth of the program.

A different direction of future work would be to explore what the graphs can tell about *Par*-monad laws and semantics. Swapping the order of two adjacent *spawn* or of two adjacent *get* (in the implementation of *mergeSort* for example) should preserve the graph and the thread semantics.

Acknowledgments

This work was partially supported by the projects GRACeFUL (grant agreement No 640954) and CoeGSS (grant agreement No 676547), which have received funding from the European Union's Horizon 2020 research and innovation programme

The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

References

[1] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. 1996. *Concurrent Programming in ERLANG (2nd Ed.)* (2 ed.). Prentice Hall PTR.

[2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 598–632. DOI: http://dx.doi.org/10.1145/69558.69562

[3] Jost Berthold and Rita Loogen. 2007. Visualizing parallel functional program runs: Case studies with the Eden trace viewer. *Parallel Computing: Architectures, Algorithms and Applications. Advances in Parallel Computing* 15 (2007), 121–128.

[4] Guy E Blelloch. 1996. Programming parallel algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[5] Richard P Brent. 1973. The parallel evaluation of arithmetic expressions in logarithmic time. *Complexity of Sequential and Parallel Numerical Algorithms, Academic Press, New York* (1973), 83–102.

[6] Henri Casanova, Arnaud Legrand, and Yves Robert. 2008. *Parallel algorithms.* CRC Press.

[7] Ivan Daniluk. 2016. Visualizing Concurrency in Go. (2016). Blog post about Interactive WebGL visualisation of Go traces. Available from http://divan.github.io/posts/go_concurrency_visualize/.

[8] Martin Erwig. 2001. Inductive graphs and functional graph algorithms. *Journal of Functional Programming* 11, 05 (2001), 467–492.

[9] Emden R. Gansner and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE* 30, 11 (2000), 1203–1233.

[10] Robert Griesemer, Rob Pike, and Ken Thompson. 2015. The Go programming language. (2015). Available from https://golang.org/.

[11] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, and others. 1992. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices* 27, 5 (1992), 1–164.

[12] Don Jones Jr., Simon Marlow, and Satnam Singh. 2009. Parallel Performance Tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09).* ACM, 81–92. DOI: http://dx.doi.org/10.1145/1596638.1596649

[13] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. 2008. *The Vampir Performance Analysis Tool-Set.* Springer Berlin Heidelberg, Berlin, Heidelberg, 139–155. DOI: http://dx.doi.org/10.1007/978-3-540-68564-7_9

[14] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir.* Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91. DOI: http://dx.doi.org/10.1007/978-3-642-31476-6_7

[15] Huiqing Li and Simon Thompson. 2013. Multicore Profiling for Erlang Programs Using Percept2. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang (Erlang '13).* ACM, New York, NY, USA, 33–42. DOI: http://dx.doi.org/10.1145/2505305.2505311

[16] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2012. Guiding Parallel Array Fusion with Indexed Types. In *Proceedings of the 2012 Haskell Symposium (Haskell '12).* ACM, New York, NY, USA, 25–36. DOI: http://dx.doi.org/10.1145/2364506.2364511

[17] Simon Marlow. 2012. *Parallel and Concurrent Programming in Haskell.* Springer Berlin Heidelberg, Berlin, Heidelberg, 339–401. DOI: http://dx.doi.org/10.1007/978-3-642-32096-5_7

[18] Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11).* ACM, 71–82. DOI: http://dx.doi.org/10.1145/2034675.2034685

[19] Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09).* ACM, New York, NY, USA, 65–78. DOI: http://dx.doi.org/10.1145/1596550.1596563

[20] Microsoft. 2015. The Visual Studio Concurrency Visualizer. (2015). Available from https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer.

[21] Mary Sheeran and John Hughes. 2017. Parallel Functional Programming. (2017). Course web page for the PFP course at Chalmers U. of Tech.. Available from http://www.cse.chalmers.se/edu/course/DAT280_Parallel_Functional_Programming/.