# `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

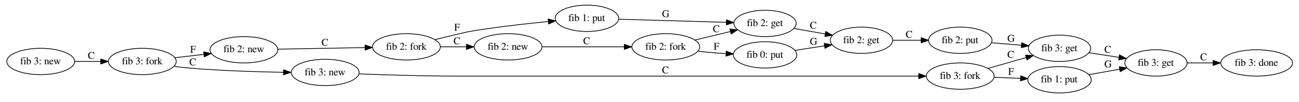


**Figure 1.** A teaser: the extended dataflow graph of the call *fib* 3

## 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 *Par*-monad computations. We hope to show that it can be useful as a teaching tool by providing visualisations of a few examples from a course on parallel functional programming.

**ACM Reference format:**
Maximilian Algehed and Patrik Jansson. 2017. `VisPar`: Visualising dataflow graphs from the *Par* monad. In *Proceedings of TODO: FHPC 2017 Workshop on Functional High-Performance Computing, Oxford, UK, September, 2017 (FHPC'17),* 5 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 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 [12] provides an interface for writing deterministic task-parallel programs in Haskell [7]. 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.

The need for visualisation of *Par* monad code was noted by Marlow in his book [11, 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.

In this paper we present a prototype tool we call `VisPar` which helps fill this gap. More precisely, we make the following contributions

- We present a tool for providing a dataflow graph as output from computations in the *Par* monad (Sections 3 and 6).
- We present two ways of visualising these graphs in an understandable way (Sections 3 and 4).
- We show how the above two contributions make it possible to analyse the parallel behaviour of programs and apply it to an example from an exam in a course Parallel Functional Programming course at Chalmers (Section 5).

## 2 The *Par* monad

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

$$
\begin{array}{ll}
\mathit{fork} & :: \mathit{Par}\ () \to \mathit{Par}\ () \\
\mathit{new} & :: \mathit{Par}\ (\mathit{IVar}\ a) \\
\mathit{put} & :: \mathit{NFData}\ a \Rightarrow \mathit{IVar}\ a \to a \to \mathit{Par}\ a \\
\mathit{get} & :: \mathit{IVar}\ a \to \mathit{Par}\ a \\
\multicolumn{2}{l}{\quad \text{-- Derived operation}} \\
\multicolumn{2}{l}{\mathit{spawn} :: \mathit{NFData}\ a \Rightarrow \mathit{Par}\ a \to \mathit{Par}\ (\mathit{IVar}\ a)}
\end{array}
$$

- 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 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 some normal form (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* to once, multiple *put*s to 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 with a single result and communicating it 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.

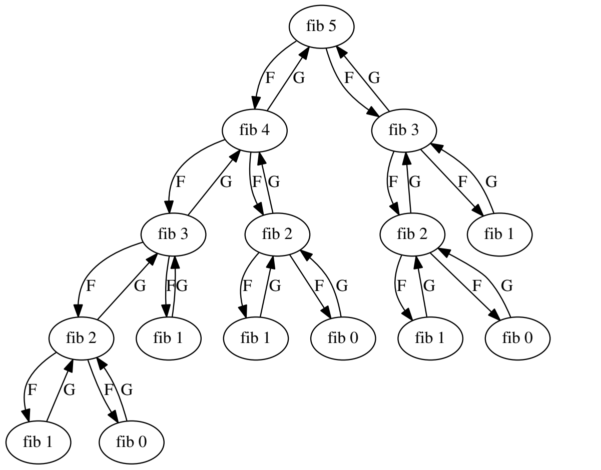


**Figure 2.** The simple dataflow graph for *fib* 5

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)
```

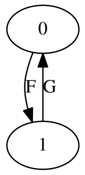
In the case where $n$ is greater than 1 we spawn two threads which will compute *fib* $(n-1)$ and *fib* $(n-2)$ in parallel and put their results in the *lv* and *rv IVar*s. We then obtain the results from *lv* and *rv* respectively. Finally we return the sum of the result of *fib* $(n-1)$ and *fib*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

## 3 Visualisation

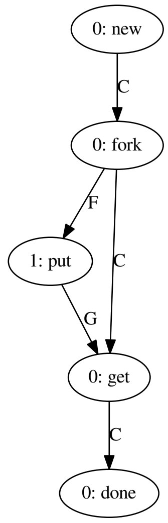
In order to provide informative visualisation we extend the *Par* monad interface with a function *forkNamed* :: *String* → *Par* () → *Par* (). This makes it possible to give forked threads names, as can be seen in the graph for the *fib* function in Figures 1 and 2. The default name is a thread ID (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*.

```
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)

The graphs in this paper are one way of visualising parallel computations. Each node represents an *event* in the computation and is labeled with its thread identifier and event type. The edges between nodes denote the relationship between events:

- An **F** means the "parent" forked the "child"
- A **C** means that the "child" is the continuation of the "parent", the "child" event occur after the "parent" event.
- A **G** means that the "child" depends on the result of a *get* from an *IVar* filled by the "parent"

Here the "parent" thread corresponds to the label on the source node and the "child" thread corresponds to the label on the target node.

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 which only contain information about what thread forked and got from what other thread.

## 4 Visualising Merge Sort

The following implementation of *mergeSort* implements the parallel 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 = case xs of
  [ ]  →  return [ ]
  [x] →  return [x]
  xs  →  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)
```

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 [7]. 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 in two halves, *ls* and *rs*, and sorted in parallel by spawning two recursive calls to *mergeSort* using *spawn*, note the decreasing $d$ argument. Finally, the results of the two recursive calls, *lr* and *rr*, are obtained from the *IVar*s *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 large list *xs*.

While the simple graph hides a lot of the information about the implementation of *mergeSort* present in the extended graph, 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.

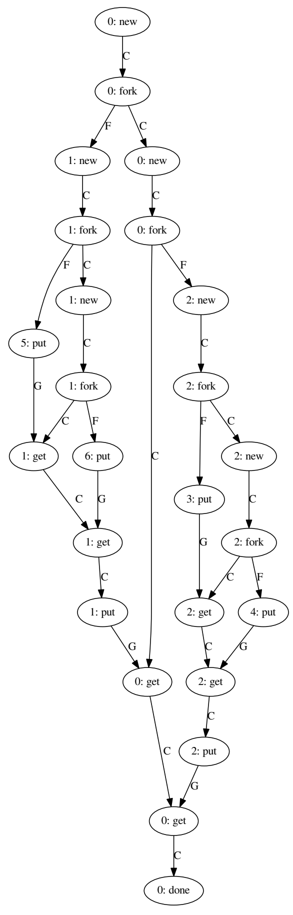


**Figure 4.** The extended dataflow graph for *mergeSort* 2 *xs*

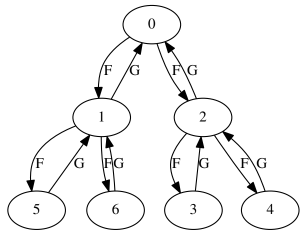


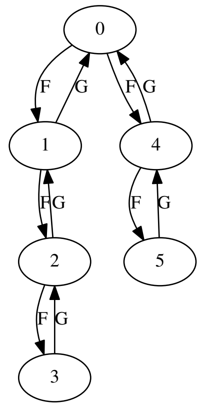
**Figure 5.** Merge sort dataflow graph for depth 2

## 5 Debugging *reduce*

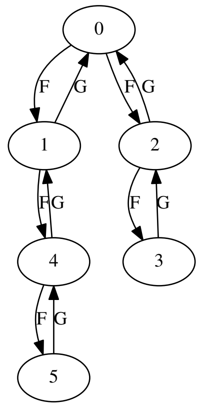
In this section we show how our visualisations can help debug in understanding the behaviour of parallel programs. It briefly covers the use of our visualisation tool to tackle a recent exam question in a course on parallel functional programming at Chalmers University of Technology [15]. The original problem featured an underperforming version of the parallel *reduce* function written in Erlang [1] and has been translated to the following *Par* monad 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 a binary function. 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. The problem is that the line *r* ← *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 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. More interestingly it also gives rise to the extended dataflow graph in Figure 7b which clearly shows that useful parallel work can be done after the code has been fixed.



**(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

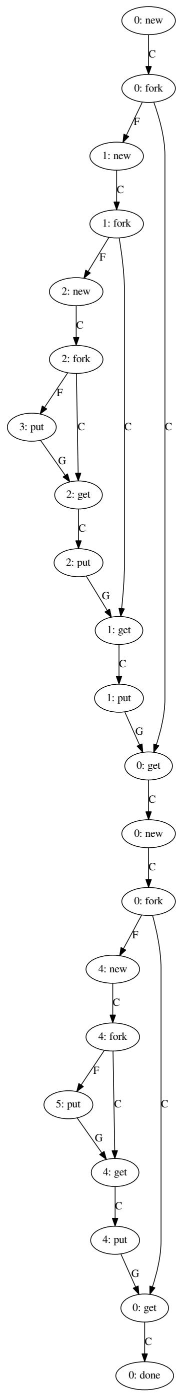
## 6 The implementation of `VisPar`

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.)

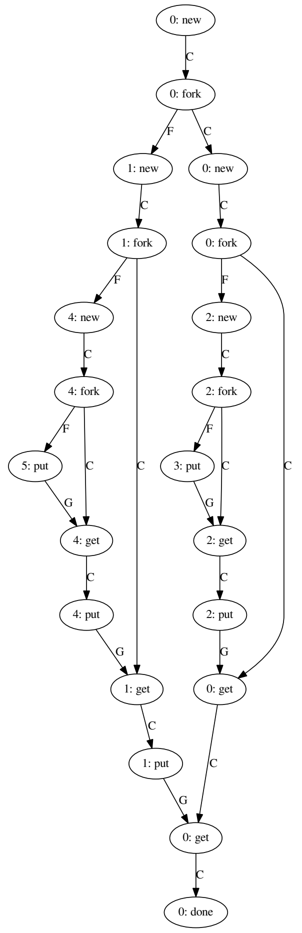
We begin with recap of the implementation of the *Par* monad by Marlow et al. It is implemented in continuation passing style by the type **newtype** *Par a* = *Par* ((*a* → *Trace*) → *Trace*) where the *Trace* type is shown below. The function *runPar* is by default implemented using a work-stealing 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



**(a)** Slow *reduce*



**(b)** Correct *reduce*

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

*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* → *Trace* → *Trace* and *GetName* :: (*String* → *Trace*) → *Trace* which are used to implement the primitives *setName* :: *String* → *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 brake determinism and referential transparency. 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 [4] to construct the dataflow graph as the computation is run. Finally we use `graphviz` [5] to render the graph (currently as a PDF).

## 7 Related Work

The `ThreadScope` tool [8] provides an interface for visualising the resource consumption of parallel programs in Haskell. VisualStudio 2010 provides similar functionality to `ThreadScope` [14]. In the functional programming domain the `percept` tool [9] 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.

The `gotracer` and `gothree.js` tools [3] allow Go [6] programmers to visualise concurrent and parallel computations as interactive three dimensional animations. 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 [13] and Repa [10], 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. 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.

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

## 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] 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/.

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

[5] 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.

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

[7] 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.

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

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

[15] 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/.