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

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:  
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 IVars 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).

2 The Par Monad

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

\[
\begin{align*}
\text{fork} & : \text{Par} () \rightarrow \text{Par} () \\
\text{new} & : \text{Par} (\text{IVar} a) \\
\text{put} & : \text{NFData} a \Rightarrow \text{IVar} a \rightarrow a \rightarrow \text{Par} a \\
\text{get} & : \text{IVar} a \rightarrow \text{Par} a \\
\text{spawn} & : \text{NFData} a \Rightarrow \text{Par} a \rightarrow \text{Par} (\text{IVar} a)
\end{align*}
\]

- 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 IVars, 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 puts 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.  

\[
\text{spawn} :: \text{NFData} a \Rightarrow \text{Par} a \Rightarrow \text{Par} (\text{IVar} a)
\]

\[
\text{spawn} p \equiv \text{do}
\]

\[
\begin{align*}
    & \quad v \leftarrow \text{new} \\
    & \quad \text{fork} (p \Rightarrow \text{put} v) \\
    & \quad \text{return} v
\end{align*}
\]

The run function for Par monad computations is a scheduler.

\[
\text{runPar} :: \text{NFData} a \Rightarrow \text{Par} a \Rightarrow a
\]

\[
\begin{align*}
    & \quad \text{return} v \\
    & \quad \text{put lv} \\
    & \quad \text{lv} \leftarrow \text{ivar} \\
    & \quad \text{fork} (\text{put'} \text{ivar} 5) \\
    & \quad \text{get ivar}
\end{align*}
\]

\[\text{(a) Source code of tiny.}\]

\[\text{(b) The simple dataflow graph}\]

\[\text{(c) The extended dataflow graph.}\]

\[\text{Figure 3. A tiny program (a) with its dataflow graphs (b), (c)}\]

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 \(\rightarrow\) Par a \(\rightarrow\) 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 \(\rightarrow\) Par a \(\rightarrow\) 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:

\[
\begin{align*}
    \text{fib} :: \text{Int} \rightarrow \text{Par Int} \\
    \text{fib} n | n < 2 &= \text{return} 1 \\
    | \quad \text{otherwise} & = \text{do} \\
    & \quad lv \leftarrow \text{spawnNamed} (\text{"fib}\ (n - 1)) \$ \text{fib} (n - 1) \\
    & \quad rv \leftarrow \text{spawnNamed} (\text{"fib}\ (n - 2)) \$ \text{fib} (n - 2) \\
    & \quad l \leftarrow \text{get lv} \\
    & \quad r \leftarrow \text{get rv} \\
    & \quad \text{return} (l + r)
\end{align*}
\]

In the case where \(n \geq 2\) we spawn two threads which will compute \(\text{fib} (n - 1)\) and \(\text{fib} (n - 2)\) in parallel and put their results in the IVars \(lv\) 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 \(\text{fib} (n - 1)\) and \(\text{fib} (n - 2)\).

\section{Visualisation}

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

\[
\begin{align*}
    \text{tiny} :: \text{Par Int} \\
    \text{tiny} &= \text{do} \\
    \text{ivar} &\leftarrow \text{new} \\
    \text{fork} (\text{put'} \text{ivar} 5) \\
    \text{get} \text{ivar}
\end{align*}
\]

\[\text{(a) Source code of tiny.}\]

\[\text{(b) The simple dataflow graph}\]

\[\text{(c) The extended dataflow graph.}\]

\[\text{Figure 3. A tiny program (a) with its dataflow graphs (b), (c)}\]

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:

\begin{itemize}
    \item An F means the source thread forked the target thread.
    \item A C means that the target is the continuation of the source; the target event occurs after the source event in the same thread.
    \item A G means that the target event depends on the result of a get from an IVar filled by the source event.
\end{itemize}
Using our extended scheduler to run the code for \textit{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 \textit{fib} 3 quickly become large. We therefore also provide a way to obtain simpler dataflow graphs like the one for \textit{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:

\[
\max \left( \frac{W}{P}, D \right) \leq T < \frac{W}{P} + D
\]  

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

\[
S \leq \frac{W}{D}
\]  

Using the graphs produced by VisPar we can compute lower bounds for both work and depth for an execution of a \textit{Par} program. The work for a particular run of a \textit{Par} program is bounded below by the number of nodes (\( \omega \)) 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 \textit{mergeSort} implements a parallel version of the merge sort algorithm using the \textit{Par} monad:

\begin{verbatim}
mergeSort :: (Ord a) => Int -> [a] -> Par [a]
mergeSort 0 xs = return (sort xs)
mergeSort d xs
  | length xs \leq 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)
\end{verbatim}

The first argument controls the "depth of parallelism", giving control over granularity, \textit{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) \textit{mergeSort} resorts to the sequential \textit{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 \textit{mergeSort} using \textit{spawn}. (Note the decreasing argument \( d \)). Finally, the results of the two recursive calls, \( lr \) and \( rr \), are obtained from the \textit{IVars} \( lv \) and \( rv \) and combined using \textit{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 \textit{mergeSort} \( 2 \) \( xs \) with a long list \( xs \).

While the simple graph hides a lot of the information about the implementation of \textit{mergeSort} present in the extended graph (as it omits the ordering of operations like \textit{new} and \textit{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.
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:

\[
\text{reduce} :: \text{NFData} a \Rightarrow (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow \text{Par} a \\
\text{reduce} f [x] = \text{return} x \\
\text{reduce} f \text{ xs} \equiv \text{do} \\
\text{let} (ls, rs) = \text{splitAt} (\text{length} \text{ xs}) \text{’div’} 2 \text{ xs} \\
rv \leftarrow \text{spawn} (\text{reduce} f \text{ rs}) \\
r \leftarrow \text{get} rv \\
l \leftarrow \text{reduce} f \text{ ls} \\
\text{return} (f \text{ l r})
\]

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, \( S \approx \frac{W}{D} = 1 \), making our intuition from looking at the graph precise.

The problem is that the line \( r \leftarrow \text{get} rv \) comes directly after the line \( l \leftarrow \text{reduce} f \text{ ls} \). 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 \( S \approx 1.75 \). 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.

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 \( \text{newtype} \text{Par} a = \text{Par} (a \rightarrow \text{Trace}) \rightarrow \text{Trace} \) where the Trace type is shown below. The function \( \text{runPar} \) 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 \rightarrow (a \rightarrow \text{Trace}) \rightarrow \text{Trace}
Put :: IVar a \rightarrow a \rightarrow \text{Trace}
New :: (IVar a \rightarrow \text{Trace}) \rightarrow \text{Trace}
Fork :: \text{Trace} \rightarrow \text{Trace} \rightarrow \text{Trace}
Done :: \text{Trace}
\quad \:\leftarrow \text{Our added primitives}
SetName :: \text{String} \rightarrow \text{Trace} \rightarrow \text{Trace}
GetName :: (\text{String} \rightarrow \text{Trace}) \rightarrow \text{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
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 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.

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