A Monad for Deterministic Parallelism

Simon Marlow
Simon Peyton Jones
Microsoft Research

Ryan Newton
Indiana University, Bloomington

Sep 22, 2011
The motivation for this work is the hope that one day parallel programming will be easier, and will become the norm rather than the exception. This work focuses on task parallelism rather than fine grained vector parallelism. Obviously easy parallelism is a big goal, but maybe we can take some small steps toward it.
The motivation for this work is the hope that one day parallel programming will be easier, and will become the norm rather than the exception. This work focuses on task parallelism rather than fine grained vector parallelism. Obviously easy parallelism is a big goal, but maybe we can take some small steps toward it.
The motivation for this work is the hope that one day parallel programming will be easier, and will become the norm rather than the exception. This work focuses on task parallelism rather than fine grained vector parallelism. Obviously easy parallelism is a big goal, but maybe we can take some small steps toward it.
The motivation for this work is the hope that one day parallel programming will be easier, and will become the norm rather than the exception. This work focuses on task parallelism rather than fine grained vector parallelism. Obviously easy parallelism is a big goal, but maybe we can take some small steps toward it.
First, we would argue that for parallel programming to be easy, it has to be predictable. The programmer needs to have some idea what will happen. Having a deterministic parallel programming model is one part of this (but only part).
First, we would argue that for parallel programming to be easy, it has to be predictable. The programmer needs to have some idea what will happen. Having a deterministic parallel programming model is one part of this (but only part).
First, we would argue that for parallel programming to be easy, it has to be predictable. The programmer needs to have some idea what will happen. Having a deterministic parallel programming model is one part of this (but only part).
First, we would argue that for parallel programming to be easy, it has to be predictable. The programmer needs to have some idea what will happen. Having a deterministic parallel programming model is one part of this (but only part).
Purely functional programming is a great way to achieve determinism. And, fortunately, it also exposes opportunities for parallelism. Let’s take a look at that.
The power of purity for parallelism is that expressions can be evaluated in any order without affecting one another. Wherever you see two expressions, you’re looking at potential parallelism. For example, take this function $f$ (let’s say it’s strict) and its two operands. Worker thread 1 can evaluate $x$ while thread 2 evaluates $y$. Fantastic.
The power of purity for parallelism is that expressions can be evaluated in any order without affecting one another. Wherever you see two expressions, you’re looking at potential parallelism. For example, take this function $f$ (let’s say it’s strict) and its two operands. Worker thread 1 can evaluate $x$ while thread 2 evaluates $y$. Fantastic.
The power of purity for parallelism is that expressions can be evaluated in any order without affecting one another. Wherever you see two expressions, you’re looking at potential parallelism. For example, take this function $f$ (let’s say it’s strict) and its two operands. Worker thread 1 can evaluate $x$ while thread 2 evaluates $y$. Fantastic.
Ah, but of course programs are large. Thus we might even say that purity provides a plethora of parallelism possibilities! But not all of them are profitable... Actually, this picture grows worse. This is just the static picture. Really, ever dynamic evaluation of an expression could potentially be parallel (or not).
...has proven too plentiful to profit (easily)

\[
\begin{align*}
\text{let } e &= s t r \text{ in} \\
\text{let } d &= q z \text{ in} \\
\text{let } c &= f z \ w \text{ in} \\
\text{let } b &= h z \ w \text{ in} \\
\text{let } a &= g x \text{ in} \\
&f x y
\end{align*}
\]

Ah, but of course programs are large. Thus we might even say that purity provides a plethora of parallelism possibilities! But not all of them are profitable... Actually, this picture grows worse. This is just the static picture. Really, ever dynamic evaluation of an expression could potentially be parallel (or not).
...has proven too plentiful to profit \textit{(easily)}

\begin{verbatim}
let e = s \ t \ r \ in
let d = q \ z \ in
let c = f \ z \ w \ in
let b = h \ z \ w \ in
let a = g \ x \ in
f x y
\end{verbatim}

And for highly data-dependent computations, evaluations of the same expression may vary greatly in execution cost. Big ones and little ones. To find a good target for parallelism compiler and runtime system would have to be able to answer several questions that constitute universal parallelism concerns.
...has proven too plentiful to profit (easily)

```haskell
let e = s t r in
let d = q z in
let c = f z w in
let b = h z w in
let a = g x in

f x y f x y f x y f x y
f x y f x y f x y
f x y f x y f x y
f x y f x y f x y
f x y f x y f x y
```

And for highly data-dependent computations, evaluations of the same expression may vary greatly in execution cost. Big ones and little ones. To find a good target for parallelism compiler and runtime system would have to be able to answer several questions that constitute universal parallelism concerns.
...has proven too plentiful to profit (easily)

\[
\begin{align*}
& \text{let } e = s \text{ t r in} \\
& \text{let } d = q z \text{ w in} \\
& \text{let } c = f z w \text{ in} \\
& \text{let } b = h z w \text{ in} \\
& \text{let } a = g x \text{ in} \\
\end{align*}
\]

Is the granularity big enough. How much communication cost is there in moving this computation to another processor -- how much data needs to move? Are there resource conflicts? Even a pure program may be doing lazy IO in the background, opening up pitfalls like false sharing. Or programs may be contending for access to runtime system resources -- allocation can even be viewed in that light. Finally synchronization cost, which can be amortized by large granularity. What are our poor worker threads to do! Faced with a potpourri of potential pitfalls...


...has proven too plentiful to profit (easily)

```
let e = s t r in
let d = q z in
let c = f z w in
let b = h z w in
let a = g x in

```

Granularity

Is the granularity big enough. How much communication cost is there in moving this computation to another processor -- how much data needs to move? Are there resource conflicts? Even a pure program may be doing lazy IO in the background, opening up pitfalls like false sharing. Or programs may be contending for access to runtime system resources -- allocation can even be viewed in that light. Finally synchronization cost, which can be amortized by large granularity. What are our poor worker threads to do! Faced with a potpourri of potential pitfalls...
...has proven too plentiful to profit (easily)

```
let e = s t r in
let d = q z in
let c = f z w in
let b = h z w in
let a = g x in
```

Granularity

Comm. cost?

Is the granularity big enough. How much communication cost is there in moving this computation to another processor -- how much data needs to move? Are there resource conflicts? Even a pure program may be doing lazy IO in the background, opening up pitfalls like false sharing. Or programs may be contending for access to runtime system resources -- allocation can even be viewed in that light. Finally synchronization cost, which can be amortized by large granularity. What are our poor worker threads to do! Faced with a potpourri of potential pitfalls...
...has proven too plentiful to profit (easily)

Is the granularity big enough. How much communication cost is there in moving this computation to another processor -- how much data needs to move? Are there resource conflicts? Even a pure program may be doing lazy IO in the background, opening up pitfalls like false sharing. Or programs may be contending for access to runtime system resources -- allocation can even be viewed in that light. Finally synchronization cost, which can be amortized by large granularity. What are our poor worker threads to do! Faced with a potpourri of potential pitfalls...
Is the granularity big enough. How much communication cost is there in moving this computation to another processor -- how much data needs to move? Are there resource conflicts? Even a pure program may be doing lazy IO in the background, opening up pitfalls like false sharing. Or programs may be contending for access to runtime system resources -- allocation can even be viewed in that light. Finally synchronization cost, which can be amortized by large granularity. What are our poor worker threads to do! Faced with a potpourri of potential pitfalls...
Has proven too plentiful to profit (easily)

let e = s t r i n
let d = q z i n
let c = f z w i n
let b = h z w i n
let a = g x i n

A potpourri of potential pitfalls!

Is the granularity big enough. How much communication cost is there in moving this computation to another processor -- how much data needs to move? Are there resource conflicts? Even a pure program may be doing lazy IO in the background, opening up pitfalls like false sharing. Or programs may be contending for access to runtime system resources -- allocation can even be viewed in that light. Finally synchronization cost, which can be amortized by large granularity. What are our poor worker threads to do! Faced with a potpourri of potential pitfalls...
...has proven too plentiful to profit (easily)

Is the granularity big enough. How much communication cost is there in moving this computation to another processor -- how much data needs to move? Are there resource conflicts? Even a pure program may be doing lazy IO in the background, opening up pitfalls like false sharing. Or programs may be contending for access to runtime system resources -- allocation can even be viewed in that light. Finally synchronization cost, which can be amortized by large granularity. What are our poor worker threads to do! Faced with a potpourri of potential pitfalls...
Perhaps our intrepid programmer can point the way! This brings us to the dominant existing Haskell paradigm for parallelism -- par and pseq. You have probably seen par before. Par is a hint from the programmer to the compiler and runtime system. It evaluates to the value of y, but asserts that x can be evaluated in parallel from the point where the value of y -- of the whole par expression -- is requested. The programmer must then answer these four questions, which is all well and good.
Perhaps our intrepid programmer can point the way! This brings us to the dominant existing Haskell paradigm for parallelism -- par and pseq. You have probably seen par before. Par is a hint from the programmer to the compiler and runtime system. It evaluates to the value of y, but asserts that x can be evaluated in parallel from the point where the value of y -- of the whole par expression -- is requested. The programmer must then answer these four questions, which is all well and good.
Programmer, please point the way!

Granularity

par x y

Perhaps our intrepid programmer can point the way! This brings us to the dominant existing Haskell paradigm for parallelism -- par and pseq. You have probably seen par before. Par is a hint from the programmer to the compiler and runtime system. It evaluates to the value of y, but asserts that x can be evaluated in parallel from the point where the value of y -- of the whole par expression -- is requested. The programmer must then answer these four questions, which is all well and good.
Perhaps our intrepid programmer can point the way! This brings us to the dominant existing Haskell paradigm for parallelism -- par and pseq. You have probably seen par before. Par is a hint from the programmer to the compiler and runtime system. It evaluates to the value of y, but asserts that x can be evaluated in parallel from the point where the value of y -- of the whole par expression -- is requested. The programmer must then answer these four questions, which is all well and good.
Perhaps our intrepid programmer can point the way! This brings us to the dominant existing Haskell paradigm for parallelism -- `par` and `pseq`. You have probably seen `par` before. Par is a hint from the programmer to the compiler and runtime system. It evaluates to the value of y, but asserts that x can be evaluated in parallel from the point where the value of y -- of the whole `par` expression -- is requested. The programmer must then answer these four questions, which is all well and good.
Perhaps our intrepid programmer can point the way! This brings us to the dominant existing Haskell paradigm for parallelism -- par and pseq. You have probably seen par before. Par is a hint from the programmer to the compiler and runtime system. It evaluates to the value of y, but asserts that x can be evaluated in parallel from the point where the value of y -- of the whole par expression -- is requested. The programmer must then answer these four questions, which is all well and good.
Ah, but there’s more trouble. Before even getting to wrestle with those more fundamental questions the programmer must achieve a zen state where they are perfectly in tune with the operational properties of their Haskell implementation. They must first know how the program will be evaluated. And with GHC this means following three additional rules.

First, X must remain unevaluated, otherwise there’s no work left to do there. Second, the evaluation mechanics of Y must be understood and it must be that Y does not itself demand the value of X too soon. Otherwise the window for parallel evaluation is too small.
Ah, but there’s more trouble. Before even getting to wrestle with those more fundamental questions the programmer must achieve a zen state where they are perfectly in tune with the operational properties of their Haskell implementation. They must first know how the program will be evaluated. And with GHC this means following three additional rules.

First, X must remain unevaluated, otherwise there’s no work left to do there. Second, the evaluation mechanics of Y must be understood and it must be that Y does not itself demand the value of X too soon. Otherwise the window for parallel evaluation is too small.
Ah, but there’s more trouble. Before even getting to wrestle with those more fundamental questions the programmer must achieve a zen state where they are perfectly in tune with the operational properties of their Haskell implementation. They must first know how the program will be evaluated. And with GHC this means following three additional rules.

First, X must remain unevaluated, otherwise there’s no work left to do there. Second, the evaluation mechanics of Y must be understood and it must be that Y does not itself demand the value of X too soon. Otherwise the window for parallel evaluation is too small.
Here’s an example of a common mistake that violates rule (b). (Also, I’ve switched to an infix notation for `par`.) This program looks awful good. Fork off the evaluation of `x` in parallel, then evaluate `y` on our current thread and add its result to `x`. But wait! Haskell does not have a defined evaluation order for function arguments. Plus could evaluate `y` first or it could evaluate `X` first. This program may or may not yield parallelism, and it’s essentially undefined which is the case.

The correct program looks like this. It also includes another annotation, `pseq`, that tells the compiler to evaluate `y` before evaluating `y+x`.

Finally our third rule is to make sure that the program forces the value of the thunk containing the `par` expressions itself at the right time. This is when the window for parallelism begins. And of course if the thunk is never used, the parallelism is lost.
Here’s an example of a common mistake that violates rule (b). (Also, I’ve switched to an infix notation for \texttt{par}.) This program looks awful good. Fork off the evaluation of \(x\) in parallel, then evaluate \(y\) on our current thread and add its result to \(x\). But wait! Haskell does not have a defined evaluation order for function arguments. Plus could evaluate \(y\) first or it could evaluate \(x\) first. This program may or may not yield parallelism, and it’s essentially undefined which is the case.

The correct program looks like this. It also includes another annotation, \texttt{pseq}, that tells the compiler to evaluate \(y\) before evaluating \(y+x\).

Finally our third rule is to make sure that the program forces the value of the thunk containing the \texttt{par} expressions itself at the right time. This is when the window for parallelism begins. And of course if the thunk is never used, the parallelism is lost.
rule (a): X is unevaluated

rule (b): don’t use X too soon in y

\[
x \ `\text{par}\` (y `\text{pseq}\`(y+x))
\]

Here’s an example of a common mistake that violates rule (b). (Also, I’ve switched to an infix notation for par.) This program looks awful good. Fork off the evaluation of x in parallel, then evaluate y on our current thread and add its result to x. But wait! Haskell does not have a defined evaluation order for function arguments. Plus could evaluate y first or it could evaluate X first. This program may or may not yield parallelism, and it’s essentially undefined which is the case.

The correct program looks like this. It also includes another annotation, pseq, that tells the compiler to evaluate y before evaluating y+x.

Finally our third rule is to make sure that the program forces the value of the thunk containing the par expressions itself at the right time. This is when the window for parallelism begins. And of course if the thunk is never used, the parallelism is lost.
Here’s an example of a common mistake that violates rule (b). (Also, I’ve switched to an infix notation for par.) This program looks awful good. Fork off the evaluation of x in parallel, then evaluate y on our current thread and add its result to x. But wait! Haskell does not have a defined evaluation order for function arguments. Plus could evaluate y first or it could evaluate X first. This program may or may not yield parallelism, and it’s essentially undefined which is the case.

The correct program looks like this. It also includes another annotation, pseq, that tells the compiler to evaluate y before evaluating y+x.

Finally our third rule is to make sure that the program forces the value of the thunk containing the par expressions itself at the right time. This is when the window for parallelism begins. And of course if the thunk is never used, the parallelism is lost.
Here’s an example of a common mistake that violates rule (b). (Also, I’ve switched to an infix notation for `par`.) This program looks awful good. Fork off the evaluation of `x` in parallel, then evaluate `y` on our current thread and add its result to `x`. But wait! Haskell does not have a defined evaluation order for function arguments. Plus could evaluate `y` first or it could evaluate `X` first. This program may or may not yield parallelism, and it’s essentially undefined which is the case.

The correct program looks like this. It also includes another annotation, `pseq`, that tells the compiler to evaluate `y` before evaluating `y+x`.

Finally our third rule is to make sure that the program forces the value of the thunk containing the `par` expressions itself at the right time. This is when the window for parallelism begins. And of course if the thunk is never used, the parallelism is lost.
Here’s an example of a common mistake that violates rule (b). (Also, I’ve switched to an infix notation for par.) This program looks awful good. Fork off the evaluation of \( x \) in parallel, then evaluate \( y \) on our current thread and add its result to \( x \). But wait! Haskell does not have a defined evaluation order for function arguments. Plus could evaluate \( y \) first or it could evaluate \( x \) first. This program may or may not yield parallelism, and it’s essentially undefined which is the case.

The correct program looks like this. It also includes another annotation, pseq, that tells the compiler to evaluate \( y \) before evaluating \( y+x \).

Finally our third rule is to make sure that the program forces the value of the thunk containing the par expressions itself at the right time. This is when the window for parallelism begins. And of course if the thunk is never used, the parallelism is lost.
So we’ve looked at the possibility of completely implicit parallelism. There’s basically too much of it.

We looked at using `par` for parallelism annotations, but it’s still not explicit enough. Much remains implicit in the Haskell implementation’s operational characteristics. The results are not easily predictable without an extensive mental model of the implementation.

So how about more explicit approaches?
So we’ve looked at the possibility of completely implicit parallelism. There’s basically too much of it.

We looked at using `par` for parallelism annotations, but it’s still not explicit enough. Much remains implicit in the Haskell implementation’s operational characteristics. The results are not easily predictable without an extensive mental model of the implementation.

So how about more explicit approaches?
Well, being the world’s greatest imperative language, Haskell of course has IO threads and nondeterministic communication channels. We can write a program like this. Create a mutable storage cell -- an MVar. Fork a computation to compute \((f \, x)\) and write the result to that cell, and finally have the calling thread keep itself busy with some other work and then block on the result of the computation.

So this is good because it makes the communication more explicit and and it’s more clear what is evaluated where. Or is it? There’s actually a bug here if you can spot it. That’s right -- there’s nothing here forcing \(f \, x\) to actually be evaluated!! This program will go through the whole song and dance of creating an extra thread, and then will anticlimactically put an unevaluated “\(f \, x\)” thunk in the MVar! We’ve opened ourselves up to a different form of the forgetting-pseq bug.
Well, being the world’s greatest imperative language, Haskell of course has IO threads and nondeterministic communication channels. We can write a program like this. Create a mutable storage cell -- an MVar. Fork a computation to compute \( f \times \) and write the result to that cell, and finally have the calling thread keep itself busy with some other work and then block on the result of the computation.

So this is good because it makes the communication more explicit and and it’s more clear what is evaluated where. Or is it? There’s actually a bug here if you can spot it. That’s right -- there’s nothing here forcing \( f \times \) to actually be evaluated!! This program will go through the whole song and dance of creating an extra thread, and then will anticlimactically put an unevaluated “\( f \times \)” thunk in the MVar! We’ve opened ourselves up to a different form of the forgetting–pseq bug.
Well, being the world’s greatest imperative language, Haskell of course has IO threads and nondeterministic communication channels. We can write a program like this. Create a mutable storage cell -- an MVar. Fork a computation to compute \( f \times \) and write the result to that cell, and finally have the calling thread keep itself busy with some other work and then block on the result of the computation.

So this is good because it makes the communication more explicit and and it’s more clear what is evaluated where. Or is it? There’s actually a bug here if you can spot it. That’s right -- there’s nothing here forcing \( f \times \) to actually be evaluated!! This program will go through the whole song and dance of creating an extra thread, and then will anticlimactically put an unevaluated “\( f \times \)” thunk in the MVar! We’ve opened ourselves up to a different form of the forgetting-pseq bug.
Well, being the world’s greatest imperative language, Haskell of course has IO threads and nondeterministic communication channels. We can write a program like this. Create a mutable storage cell -- an MVar. Fork a computation to compute \( f \times x \) and write the result to that cell, and finally have the calling thread keep itself busy with some other work and then block on the result of the computation.

So this is good because it makes the communication more explicit and and it’s more clear what is evaluated where. Or is it? There’s actually a bug here if you can spot it. That’s right -- there’s nothing here forcing \( f \times x \) to actually be evaluated!! This program will go through the whole song and dance of creating an extra thread, and then will anticlimactically put an unevaluated “\( f \times x \)” thunk in the MVar! We’ve opened ourselves up to a different form of the forgetting-pseq bug.
Well, being the world’s greatest imperative language, Haskell of course has IO threads and nondeterministic communication channels. We can write a program like this. Create a mutable storage cell -- an MVar. Fork a computation to compute \((f \ x)\) and write the result to that cell, and finally have the calling thread keep itself busy with some other work and then block on the result of the computation.

So this is good because it makes the communication more explicit and and it’s more clear what is evaluated where. Or is it? There’s actually a bug here if you can spot it. That’s right -- there’s nothing here forcing \(f \ x\) to actually be evaluated!! This program will go through the whole song and dance of creating an extra thread, and then will anticlimactically put an unevaluated “\(f \ x\)” thunk in the MVar! We’ve opened ourselves up to a different form of the forgetting-pseq bug.
Corrected version

Here's the correct program. We explicitly reduce to either weak–head or full normal form with the \texttt{rnf} function. And then we force the resulting thunk before the \texttt{put} happens with \texttt{Control.Exception.evaluate}.

Ok, so then we’re in pretty good shape... but some significant problems remain.
First, there’s the problem of overhead. forkIO threads are lighter-weight than OS threads, but they still allocate a new stack and are preemptable. Functionality that we don’t really need or want just to parallelize the work in our pure computation. In our experience parallel applications and schedulers that use forkIO for every unit of parallel work, every task, consistently lose rather badly as problem size scales up.
First, there’s the problem of overhead. forkIO threads are lighter-weight than OS threads, but they still allocate a new stack and are preemptable. Functionality that we don’t really need or want just to parallelize the work in our pure computation. In our experience parallel applications and schedulers that use forkIO for every unit of parallel work, every task, consistently lose rather badly as problem size scales up.
First, there’s the problem of overhead. forkIO threads are lighter-weight than OS threads, but they still allocate a new stack and are preemptable. Functionality that we don’t really need or want just to parallelize the work in our pure computation. In our experience parallel applications and schedulers that use forkIO for every unit of parallel work, every task, consistently lose rather badly as problem size scales up.
But wait, there’s an even bigger drawback than that. An obvious one and one you know well. We’ve entered into the IO monad just to get our parallelism! And of course you can’t get out of it... Further, it pollutes the previously pure computation by destroying its determinism guarantee.
Big drawback

But wait, there’s an even bigger drawback than that. An obvious one and one you know well. We’ve entered into the IO monad just to get our parallelism! And of course you can’t get out of it... Further, it pollutes the previously pure computation by destroying its determinism guarantee.
But wait, there’s an even bigger drawback than that. An obvious one and one you know well. We’ve entered into the IO monad just to get our parallelism! And of course you can’t get out of it... Further, it pollutes the previously pure computation by destroying its determinism guarantee.
But wait, there’s an even bigger drawback than that. An obvious one and one you know well. We’ve entered into the IO monad just to get our parallelism! And of course you can’t get out of it... Further, it pollutes the previously pure computation by destroying its determinism guarantee.
So here we are. A plate full of less-than-ideal options. That brings us to the Par monad, which sits somewhere inbetween the IO threads and par/pseq approaches.

<table>
<thead>
<tr>
<th>Implicit</th>
<th>Too much!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-Implicit</td>
<td>Works, not predictable</td>
</tr>
<tr>
<td></td>
<td>Say no to IO</td>
</tr>
</tbody>
</table>
So here we are. A plate full of less-than-ideal options. That brings us to the Par monad, which sits somewhere inbetween the IO threads and par/pseq approaches.
Haskell IO threads are provided with a lot of functionality. They’re preemptable. There are libraries providing concurrent data structures such as MVars and Channels, as well as semaphores.

What we want to do is essentially take a small subset of this menu. We want fork and we want MVars, but to restrict them to be single assignment, which turns them into something called IVars (or I-structures).
Haskell IO threads are provide a lot of functionality. They’re preemptable. There are libraries providing concurrent data structures such as MVars and Channels, as well as semaphores.

What we want to do is essentially take a small subset of this menu. We want fork and we want MVars, but to restrict them to be single assignment, which turns them into something called IVars (or I-structures).
Haskell IO threads are provided with a lot of functionality. They’re preemptable. There are libraries providing concurrent data structures such as MVars and Channels, as well as semaphores.

What we want to do is essentially take a small subset of this menu. We want fork and we want MVars, but to restrict them to be single assignment, which turns them into something called IVars (or I-structures).
And with that the sky clears. IVars single assignment property gives us back determinism. You refer to the paper for a sketch of the proof.

An analogy here is to doing State in IO rather than separating it into a state monad. But we’re not purely subtracting functionality, we also want to add extra strictness to prevent the bug that we looked at earlier.
And with that the sky clears. IVars single assignment property gives us back determinism. You refer to the paper for a sketch of the proof.

An analogy here is to doing State in IO rather than separating it into a state monad. But we’re not purely subtracting functionality, we also want to add extra strictness to prevent the bug that we looked at earlier.
And with that the sky clears. IVars single assignment property gives us back determinism. You refer to the paper for a sketch of the proof.

An analogy here is to doing State in IO rather than separating it into a state monad. But we’re not purely subtracting functionality, we also want to add extra strictness to prevent the bug that we looked at earlier.
The Par Monad

Here’s what the Par monad looks like. Par a is a new monad. Because nothing expressible in Par is ultimately impure, runPar enables elimination the Par type. The monad provides explicit sequencing of computation vis-a-vis parallel operations. Par provides fork which exposes a child computation for parallel execution -- we call the rest of the computation, the continuation of fork, the parent. New, put, and get create and operate on Ivars.
Here’s what the Par monad looks like. Par a is a new monad. Because nothing expressible in Par is ultimately impure, runPar enables elimination the Par type. The monad provides explicit sequencing of computation vis-à-vis parallel operations. Par provides fork which exposes a child computation for parallel execution -- we call the rest of the computation, the continuation of fork, the parent. New, put, and get create and operate on Ivars.
The Par Monad

newtype Par a
runPar :: Par a → a

Here’s what the Par monad looks like. Par a is a new monad. Because nothing expressible in Par is ultimately impure, runPar enables elimination the Par type. The monad provides explicit sequencing of computation vis-à-vis parallel operations. Par provides fork which exposes a child computation for parallel execution -- we call the rest of the computation, the continuation of fork, the parent. New, put, and get create and operate on Ivars.
Here’s what the Par monad looks like. Par a is a new monad. Because nothing expressible in Par is ultimately impure, runPar enables elimination the Par type. The monad provides explicit sequencing of computation vis-à-vis parallel operations. Par provides fork which exposes a child computation for parallel execution -- we call the rest of the computation, the continuation of fork, the parent. New, put, and get create and operate on lvars.
Here's what the Par monad looks like. Par a is a new monad. Because nothing expressible in Par is ultimately impure, runPar enables elimination the Par type. The monad provides explicit sequencing of computation vis-à-vis parallel operations. Par provides fork which exposes a child computation for parallel execution — we call the rest of the computation, the continuation of fork, the parent. New, put, and get create and operate on Ivars.
The Par Monad

newtype Par a
runPar :: Par a → a

do v ← new
   fork (put v (f x)
   y ← get v

Here’s what the Par monad looks like. Par a is a new monad. Because nothing expressible in Par is ultimately impure, runPar enables elimination the Par type. The monad provides explicit sequencing of computation vis-a-vis parallel operations. Par provides fork which exposes a child computation for parallel execution -- we call the rest of the computation, the continuation of fork, the parent. New, put, and get create and operate on !vars.
Here’s the full IVar interface. The only thing to note here is the extra strictness I mentioned before. By default put requires that the data being put into the IVar be fully evaluated and for this reason it has an type NFData class constraint.

This isn’t always desirable of course, it would, for example, prevent sending arrow types with put. For that reason there’s a put_ that only requires weak head normal form. But the point is that strictness is opt–out in this framework, not opt–in.
Here’s the full IVar interface. The only thing to note here is the extra strictness I mentioned before. By default put requires that the data being put into the IVar be fully evaluated and for this reason it has an type NFData class constraint.

This isn’t always desirable of course, it would, for example, prevent sending arrow types with put. For that reason there’s a put_ that only requires weak head normal form. But the point is that strictness is opt–out in this framework, not opt–in.
Here’s the full IVar interface. The only thing to note here is the extra strictness I mentioned before. By default put requires that the data being put into the IVar be fully evaluated and for this reason it has an type NFData class constraint.

This isn’t always desirable of course, it would, for example, prevent sending arrow types with put. For that reason there’s a put_ that only requires weak head normal form. But the point is that strictness is opt-out in this framework, not opt-in.
The Par Monad

```haskell
newtype Par a
runPar :: Par a → a

do  v ← new
    fork (put v (f x))
y ← get v
```

If we return for a moment to our previous simple example usage of fork and one IVar... This is a pretty common pattern. Forked computations in this model don’t have return values, but we are using the IVar here to that purpose. It’s convenient to give this a name....
The Par Monad – spawn shorthand

```haskell
newtype Par a
runPar :: Par a → a

do
   y ← spawn (return (f x))
```

we call it spawn rather than fork. This creates a future not dissimilar from the ‘par/pseq’ approach. There’s no consistent terminology for futures, but “spawn” has some precedent in Cilk.
And of course spawn is simply defined in terms of fork exactly how you would expect.

```haskell
define spawn p = do
  i ← new
  fork (do x ← p; put i x)
  return i
```

**Spawn Definition**

```
fork :: Par () → Par ()
```
fib n | n < 2 = return 1  
fib n =  
    do xf ← spawn$ fib (n-1)  
       y ← fib (n-2)  
       x ← get xf  
    return (x+y)

Here’s a simple program written with spawn. This is the classic parfib microbenchmark that tests scheduler overhead.  
In this example we don’t achieve the complete separation of concerns of the Strategies approach based on par/pseq. But the program isn’t too badly mangled either. Our nonparallel program is still there. We just have to cover up some bits.
Here’s a simple program written with spawn. This is the classic parfib microbenchmark that tests scheduler overhead.

In this example we don’t achieve the complete separation of concerns of the Strategies approach based on par/pseq. But the program isn’t too badly mangled either. Our nonparallel program is still there. We just have to cover up some bits.
Here’s a simple program written with spawn. This is the classic parfib microbenchmark that tests scheduler overhead.

In this example we don’t achieve the complete separation of concerns of the Strategies approach based on par/pseq. But the program isn’t too badly mangled either. Our nonparallel program is still there. We just have to cover up some bits.
It’s worth taking a moment to reflect on the difference between futures and Ivars. Futures have a two-part API, the spawn and the get or force. The spawn does double duty. It creates a placeholder for the result, and it identifies the computation to spawn. Thus there can be no placeholder without knowing exactly what code will produce the value for it.
It’s worth taking a moment to reflect on the difference between futures and Ivars. Futures have a two-part API, the spawn and the get or force. The spawn does double duty. It creates a placeholder for the result, and it identifies the computation to spawn. Thus there can be no placeholder without knowing exactly what code will produce the value for it.
It’s worth taking a moment to reflect on the difference between futures and Ivars. Futures have a two-part API, the spawn and the get or force. The spawn does double duty. It creates a placeholder for the result, and it identifies the computation to spawn. Thus there can be no placeholder without knowing exactly what code will produce the value for it.
It’s worth taking a moment to reflect on the difference between futures and Ivars. Futures have a two-part API, the spawn and the get or force. The spawn does double duty. It creates a placeholder for the result, and it identifies the computation to spawn. Thus there can be no placeholder without knowing exactly what code will produce the value for it.
IVars vs. Futures

\[
\begin{align*}
  x & \leftarrow \text{new} \\
  \quad \ldots
  \quad \text{put } x & \ 3 \\
  \quad \ldots
  \quad \text{get } x
\end{align*}
\]

IVars decompose this further into a three-step API. Placeholders can be created with no knowledge of what computation will eventually fill them -- that is, without knowing the producer.
IVars decompose this further into a three-step API. Placeholders can be created with no knowledge of what computation will eventually fill them -- that is, without knowing the producer.
IVars decompose this further into a three-step API. Placeholders can be created with no knowledge of what computation will eventually fill them -- that is, without knowing the producer.
IVars decompose this further into a three-step API. Placeholders can be created with no knowledge of what computation will eventually fill them -- that is, without knowing the producer.
IVars decompose this further into a three-step API. Placeholders can be created with no knowledge of what computation will eventually fill them -- that is, without knowing the producer.
Ivars vs. Futures

New wrinkle:
get on unavailable data → BLOCK

get \ x \quad \text{Consumer}

IVars decompose this further into a three-step API. Placeholders can be created with no knowledge of what computation will eventually fill them -- that is, without knowing the producer.
IVars decompose this further into a three-step API. Placeholders can be created with no knowledge of what computation will eventually fill them — that is, without knowing the producer.
One benefit of the three-part IVar interface is modularity between producers and consumers. Here’s a simple picture with foo talking to bar. If foo is easily encapsulated there doesn’t seem to be much of a modularity argument. Foo itself could be placed bundled up as a future.
One benefit of the three-part IVar interface is modularity between producers and consumers. Here’s a simple picture with foo talking to bar. If foo is easily encapsulated there doesn’t seem to be much of a modularity argument. Foo itself could be placed bundled up as a future.
One benefit of the three-part IVar interface is modularity between producers and consumers. Here’s a simple picture with foo talking to bar. If foo is easily encapsulated there doesn’t seem to be much of a modularity argument. Foo itself could be placed bundled up as a future.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With Ivars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With Ivars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With Ivars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With Ivars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
But the picture can grow more complex. Futures mix up control and dataflow, whereas IVars allow us to create whatever sort of data structures we want. Here’s a matrix of IVars. Foo may have its own method of traversing this matrix of IVars – its own control strategy. Likewise, bar may differ in the shape and granularity of slices it needs to do work. With IVars and the Par monad this is fine; foo and bar can both be recursive, continuously running computations with their own traversal strategies. The benefit of this is even more evident if foo or bar maintains state through its traversal, which it updates as it moves through the data space.

An analogy here is to threaded programming vs. programming concurrent systems with event handlers where events need to explicitly manage their state and continuations before returning control to the scheduler.
Futures, operationally:

consuming $\Rightarrow$ computing

So that was a point about how programs are structured. But there are also some operational advantages to IVars that translate to performance in some situations. Let’s replace the matrix of a the previous example and use an ordered stream for communicating between foo and bar. The problem is that futures make every consumer a potential producer -- that’s what happens when an unavailable future is touched. This makes it very hard to achieve a pipeline arrangement at runtime where foo stays on core1 and feeds data to bar on core2.
Futures, operationally:
consuming $\Rightarrow$ computing

So that was a point about how programs are structured. But there are also some operational advantages to IVars that translate to performance in some situations. Let’s replace the matrix of a the previous example and use an ordered stream for communicating between foo and bar. The problem is that futures make every consumer a potential producer -- that’s what happens when an unavailable future is touched. This makes it very hard to achieve a pipeline arrangement at runtime where foo stays on core1 and feeds data to bar on core2.
So that was a point about how programs are structured. But there are also some operational advantages to IVars that translate to performance in some situations. Let’s replace the matrix of a the previous example and use an ordered stream for communicating between foo and bar. The problem is that futures make every consumer a potential producer -- that’s what happens when an unavailable future is touched. This makes it very hard to achieve a pipeline arrangement at runtime where foo stays on core1 and feeds data to bar on core2.
Futures, operationally:

consuming ⇒ computing

So that was a point about how programs are structured. But there are also some operational advantages to IVars that translate to performance in some situations. Let’s replace the matrix of a the previous example and use an ordered stream for communicating between foo and bar. The problem is that futures make every consumer a potential producer -- that’s what happens when an unavailable future is touched. This makes it very hard to achieve a pipeline arrangement at runtime where foo stays on core1 and feeds data to bar on core2.
Futures, operationally:
consuming \Rightarrow computing

So that was a point about how programs are structured. But there are also some operational advantages to IVars that translate to performance in some situations. Let’s replace the matrix of a the previous example and use an ordered stream for communicating between foo and bar. The problem is that futures make every consumer a potential producer -- that’s what happens when an unavailable future is touched. This makes it very hard to achieve a pipeline arrangement at runtime where foo stays on core1 and feeds data to bar on core2.
Again, what we would like to write is something like this. Each stream transducer is written as a tight loop that fetches input data, computes, and produces output. If there is state to the transducer it is passed and modified functionally as part of the recursion.

```
loop inS outS = do
   x ← head inS
   y ← compute x
   put outS y
   inS' ← tail inS
   loop inS'
```
Again, what we would like to write is something like this. Each stream transducer is written as a tight loop that fetches input data, computes, and produces output. If there is state to the transducer it is passed and modified functionally as part of the recursion.
IVars let you do it!

```haskell
data IList a = Null | Cons a (IVar (IList a))
```

Ivars will let you write stream transducers in this way. One way to accomplish it is to represent Streams using lists in which tail, the cdr field, is an IVar. That is, the IVar analogy of lazy lists. This isn’t the only representation. (And in fact it doesn’t do anything to bound space usage if the producer gets ahead of the consumer.) But it’s the simplest way.
Implementation
Scheduler

• Standard work-stealing.
  ▸ Many possible implementations

• Easiest: write a Haskell level scheduler with existing constructs (MVars, forkIO)
An earlier tech-report on a similar model to monad Par explored implementation options in greater detail. It compared schedulers, including one based on IO threads... Another one coopted the sparks mechanism itself already present in GHC. And several others implemented work-sharing or work-stealing directly in Haskell, testing several different datatypes for representing queues and atomic variables.
An earlier tech-report on a similar model to monad Par explored implementation options in greater detail. It compared schedulers, including one based on IO threads... Another one coopted the sparks mechanism itself already present in GHC. And several others implemented work-sharing or work-stealing directly in Haskell, testing several different datatypes for representing queues and atomic variables.
An earlier tech-report on a similar model to monad Par explored implementation options in greater detail. It compared schedulers, including one based on IO threads... Another one coopted the sparks mechanism itself already present in GHC. And several others implemented work-sharing or work-stealing directly in Haskell, testing several different datatypes for representing queues and atomic variables.
An earlier tech-report on a similar model to monad Par explored implementation options in greater detail. It compared schedulers, including one based on IO threads... Another one coopted the sparks mechanism itself already present in GHC. And several others implemented work-sharing or work-stealing directly in Haskell, testing several different datatypes for representing queues and atomic variables.
An earlier tech-report on a similar model to monad Par explored implementation options in greater detail. It compared schedulers, including one based on IO threads... Another one coopted the sparks mechanism itself already present in GHC. And several others implemented work-sharing or work-stealing directly in Haskell, testing several different datatypes for representing queues and atomic variables.
An earlier tech-report on a similar model to monad Par explored implementation options in greater detail. It compared schedulers, including one based on IO threads... Another one coopted the sparks mechanism itself already present in GHC. And several others implemented work-sharing or work-stealing directly in Haskell, testing several different datatypes for representing queues and atomic variables.
An earlier tech-report on a similar model to monad Par explored implementation options in greater detail. It compared schedulers, including one based on IO threads... Another one coopted the sparks mechanism itself already present in GHC. And several others implemented work-sharing or work-stealing directly in Haskell, testing several different datatypes for representing queues and atomic variables.
A clean approach

Claessen, 1999

data Trace
= Fork Trace Trace
  | Done
  | forall a . Get (IVar a) (a -> Trace)
  | forall a . Put (IVar a) a Trace
  | forall a . New (IVar a -> Trace)

One way elegant way to construct a scheduler is to have computations in a Par monad produce a Trace as a lazy data structure. This is the approach we’ve taken in this paper and the complete scheduler appears in the text of the paper.

This means that Par can is not directly implemented with IO. The scheduler itself is a separate component that processes a trace. We can have a trivial pure, serial scheduler for reference, and implement other schedulers in the IO monad, invoking them inside runPar with unsafePerformIO.
Figure 5. Speedup results on 24 cores (Y axis is speedup, X axis is number of OS threads)
puting) is quite different. Indeed, with coarser grains of computa-
guage [15], although the context (distributed-memory cloud com-
monad and its
computations.
on statically known networks rather than dynamic (forking) com-
more complex model than
of
is the Intel Concurrent Collections (CnC) model [4]. CnC provides
there are no practical parallel programming systems currently avail-
tions and communication channels. Kahn process networks [12]
streamitive than the
StreamIt [10], retain determinism but are significantly less expres-
constructs including explicit futures.
paralellised [17]. Manticore [9] is a parallel variant of ML that sup-
recttions as primitives, structuring the parallel scheduler around them.

Figure 5. Speedup results on 24 cores (Y axis is speedup, X axis is number of OS threads)

Fib perf not so great yet. (Trace deforestation)
One question that comes up about the Par monad is whether it’s less Haskelly than par/pseq. After all it includes awful things like strictness and sequencing.
One question that comes up about the Par monad is whether it’s less Haskelly than par/pseq. After all it includes awful things like strictness and sequencing.
Not really.
Not really.

Haskell is heterogenous, includes the right tool for the job (IO, etc)
Not really.

Haskell is heterogeneous, includes the right tool for the job (IO, etc)

And... Par is eliminatable.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure Haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure Haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure Haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure Haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
The programmer’s mental model while using par is that they compose sizable computations, pure lazy computations, in parallel. Everything between actions in the par API is pure Haskell of course. And finally, after runPar, we’re back to pure Haskell again.

Thus the goal of the Par monad is to provide a tool for describing the parallel control in the application. Usually this can be an isolated piece of the application sandwiched between complex, possibly non-parallel functions at the bottom, and the larger application using the result of runPar at the top.
Conclusions & Future

• Similar perf., fewer pitfalls than par/pseq.
• Additional expressiveness
  ‣ enables modularity, helps for some apps

• Composes w/ other (splitable) effects
  ‣ random generation

• Parallelism as a eliminatable effect?
End.