A Monad for Deterministic Parallelism

Simon Marlow
Microsoft Research, Cambridge, U.K.
simonmar@microsoft.com

Ryan Newton
Intel, Hudson, MA, U.S.A
ryan.r.newton@intel.com

Simon Peyton Jones
Microsoft Research, Cambridge, U.K.
simonpj@microsoft.com

Abstract

We present a new programming model for deterministic parallel computation in a pure functional language. The model is monadic and has explicit granularity, but allows dynamic construction of dataflow networks that are scheduled at runtime, while remaining deterministic and pure. The implementation is based on monadic concurrency, which has until now only been used to simulate concurrency in functional languages, rather than to provide parallelism. We present the API with its semantics, and argue that parallel execution is deterministic. Furthermore, we present a complete work-stealing scheduler implemented as a Haskell library, and we show that it performs at least as well as the existing parallel programming models in Haskell.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming (Parallel programming)

General Terms Languages, Performance

1. Introduction

The prospect of being able to express parallel algorithms in a pure functional language and thus obtain a guarantee of determinism is tantalising. Haskell, being a language in which effects are explicitly controlled by the type system, should be an ideal environment for deterministic parallel programming.

For many years we have advocated the use of the `par` and `pseq`[1] operations as the basis for general-purpose deterministic parallelism in Haskell, and there is an elaborate parallel programming framework, Evaluation Strategies, built in terms of them [14, 20]. However, a combination of practical experience and investigation has lead us to conclude that this approach is not without drawbacks. In a nutshell, the problem is this: achieving parallelism with `par` requires that the programmer understand *operational* properties of the language that are at best implementation-defined (and at worst undefined). This makes `par` difficult to use, and pitfalls abound — new users have a high failure rate unless they restrict themselves to the pre-defined abstractions provided by the Strategies library. Section 2 elaborates.

In this paper we propose a new programming model for deterministic parallel programming in Haskell. It is based on a monad, has explicit granularity, and uses I-structures [1] for communication. The monadic interface, with its explicit `fork` and communication, resembles a non-deterministic concurrency API; however by carefully restricting the operations available to the programmer we are able to retain determinism and hence present a pure interface, while allowing a parallel implementation. We give a formal operational semantics for the new interface.

Our programming model is closely related to a number of others; a detailed comparison can be found in Section 8. Probably the closest relative is pH [16], a variant of Haskell that also has I-structures; the principal difference with our model is that the monad allows us to retain referential transparency, which was lost in pH with the introduction of I-structures. The target domain of our programming model is large-grained irregular parallelism, rather than fine-grained regular data parallelism (for the latter Data Parallel Haskell [6] is more appropriate).

Our implementation is based on *monadic concurrency* [18], a technique that has previously been used to good effect to simulate concurrency in a sequential functional language [7], and to unify threads with event-driven programming for scalable I/O [13]. In this paper, we put it to a new use: implementing deterministic parallelism.

We make the following contributions:

- We propose a new programming model for deterministic parallel programming, based on a monad, and using I-structures to exchange information between parallel tasks (Section 3).
- We give a semantics (Section 5) for the language and a (sketch) proof of determinism (Section 5.2).
- Our programming model is implemented entirely in a Haskell library, using techniques developed for implementing concurrency as a monad. This paper contains the complete implementation of the core library (Section 6), including a work-stealing scheduler. Being a Haskell library, the implementation can be readily modified, for example to implement alternative scheduling policies. This is not a possibility with existing parallel programming models for Haskell.
- We present results demonstrating good performance on a range of parallel benchmarks, comparing `Par` with Strategies (Section 7).

2. The challenge

To recap, the basic operations provided for parallel Haskell programming are `par` and `pseq`:

```
par  :: a -> b -> b
pseq :: a -> b -> b
```

Informally, `par` annotates an expression (its first argument) as being potentially profitable to evaluate in parallel, and evaluates to the value of its second argument. The `pseq` operator expresses sequen-

[1] formerly `seq`

Haskell'11, September 22, 2011, Tokyo, Japan.

tial evaluation ordering: its first argument is evaluated, followed by its second.

The `par` operator is an attractive language design because it capitalises on the overlap between lazy evaluation and *futures*. To implement lazy evaluation we must have a representation for expressions which are not yet evaluated but whose value may later be demanded; and similarly a future is a computation whose value is being evaluated in parallel and which we may wait for. Hence, `par` was conceived as a mechanism for annotating a lazy computation as being potentially profitable to evaluate in parallel, in effect turning a lazy computation into a future.

Evaluation Strategies [14, 20] further capitalise on lazy-evaluation-for-parallelism by building composable abstractions that express parallel evaluation over lazy data structures.

However, difficulties arise when we want to be able to program parallel algorithms with these mechanisms. To use `par` effectively, the programmer must

(a) pass an *unevaluated computation* to `par`,

(b) ensure that its value will not be required by the enclosing computation for a while, and

(c) ensure that the result is *shared by the rest of the program*.

If either (a) or (b) are violated, then little or no parallelism is achieved. If (c) is violated then the garbage collector may (or may not) garbage-collect the parallelism before it can be used. We often observe both expert and non-expert users alike falling foul of one or more of these requirements.

These preconditions on `par` are *operational* properties, and so to use `par` the programmer must have an operational understanding of the execution — and that is where the problem lies. Even experts find it difficult to reason about the evaluation behaviour, and in general the operational semantics of Haskell is undefined.

For example, one easy mistake is to omit `pseq`, leading to a program with undefined parallelism. For example, in

```
y `par` (x + y)
```

it is unspecified whether the arguments of `(+)` are evaluated left-to-right or right-to-left. The first choice will allow `y` to be evaluated in parallel, while the second will not. Compiling the program with different options may yield different amounts of parallelism.

A closely-related pitfall is to reason incorrectly about strictness. Parallelism can be lost either by the program being unexpectedly strict, or by being unexpectedly lazy. As an example of the former, consider

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

Here the programmer presumably intended to evaluate `x` in parallel with the call to `f`. However, if `f` is strict, the compiler may decide to use call-by-value for `f`, which will lose all parallelism. As an example of the latter, consider this attempt to evaluate all the elements of a list in parallel:

```
parList :: [a] -> [a]
parList []     = []
parList (x:xs) = x `par` (x : parList xs)
```

The problem is that this is probably too lazy: the head is evaluated in parallel, but the tail of the list is lazy, and so further parallelism is not created until the tail of the list is demanded.

There is an operational semantics for `par` in Baker-Finch et al. [2], and indeed it can be used to reason about some aspects of parallel execution. However, the host language for that semantics is Core, not Haskell, and there is no direct operational relationship between the two. A typical compiler will perform a great deal of optimisation and transformation between Haskell and Core (for example, strictness analysis). Hence this semantics has limited usefulness for reasoning about programs written in Haskell with `par`.

In Marlow et al. [14] we attempted to improve matters with the introduction of the `Eval` monad; a monad for "evaluation order". The purpose of the `Eval` monad is to allow the programmer to express an ordering between instances of `par` and `pseq`, something which is difficult when using them in their raw infix form. In this it is somewhat successful: `Eval` would guide the programmer away from the `parList` mistake above, although it would not help with the other two examples. In general, `Eval` does not go far enough — it partially helps with requirements (a) and (b), and does not help with (c) at all.

In practice programmers can often avoid the pitfalls by using the higher-level abstractions provided by Evaluation Strategies. However, similar problems emerge at this higher level too: Strategies consume lazy data structures, so the programmer must still understand where the laziness is (and not accidentally introduce strictness). Common patterns such as `parMap` work, but achieving parallelism with larger or more complex examples can be something of an art.

In the next section we describe our new programming model that avoids, or mitigates, the problems described above. We will return to evaluate the extent to which our new model is successful in Section 8.1.

3. The `Par` Monad

Our goal with this work is to find a parallel programming model that is expressive enough to subsume Strategies, robust enough to reliably express parallelism, and accessible enough that non-expert programmers can achieve parallelism with little effort.

Our parallel programming interface[2] is structured around a monad, `Par`:

```
newtype Par a
instance Functor Par
instance Applicative Par
instance Monad Par
```

Computations in the `Par` monad can be extracted using `runPar`:

```
runPar :: Par a -> a
```

Note that the type of `runPar` indicates that the result has no side effects and does no I/O; hence, we are guaranteed that `runPar` produces a deterministic result for any given computation in the `Par` monad.

The purpose of `Par` is to introduce parallelism, so we need a way to create parallel tasks:

```
fork :: Par () -> Par ()
```

The semantics of `fork` are entirely conventional: the computation passed as the argument to `fork` (the "child") is executed concurrently with the current computation (the "parent"). In general, `fork` allows a tree of computations to be expressed; for the purposes of the rest of this paper we will call the nodes of this tree "threads".

Of course, `fork` on its own isn't very useful; we need a way to communicate results from the child of `fork` to the parent. For our communication abtraction we use `IVars` (also called I-structures):

```
data IVar a  -- instance Eq

new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
```

[2] The current version is available at `http://hackage.haskell.org/package/monad-par`

An IVar is a write-once mutable reference cell, supporting two operations: put and get. The put operation assigns a value to the IVar, and may only be executed once per IVar (subsequent puts are an error). The get operation waits until the IVar has been assigned a value, and then returns the value.

One unusual aspect of our interface is the NFData ("normal-form data") context on put: our put operation is fully-strict in the value it places in the IVar, and the NFData context is a prerequisite for full-strictness. This aspect of the design is not forced; indeed our library also includes another version of put, put_, that is only head-strict. However, making the fully-strict version the default eliminates a common pitfall, namely putting a lazy computation into an IVar, and thereby deferring the work until the expression is extracted with get and its value subsequently demanded. By forcing values communicated via IVars to be fully evaluated, the programmer gains a clear picture of which work happens on which thread.

3.1 Derived combinators

A common pattern is for a thread to fork several children and then collect their results; indeed, in many parallel programs this is the only parallel pattern required. We can implement this pattern straightforwardly using the primitives. First, we construct an abstraction for a single child computation that returns a result:

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork (do x <- p; put i x)
  return i
```

The IVar in this context is commonly called a *future*, because it represents the value of a computation that will be completed at some later point in time.

Generalising spawn to a list is trivial: the monadic combinator mapM does the trick. However, a more useful pattern is to combine a spawn with a map: for each element of a list, create a child process to compute the application of a given function to that element. Furthermore, we usually want to wait for all the children to complete and return a list of the results. This pattern is embodied in the combinator parMapM, defined as follows:

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as
  mapM get ibs
```

The parMapM given above works on lists, but it can be trivially extended to work on any Traversable structure.

In principle, spawn subsumes fork, new, and put. Furthermore, spawn ensures that each IVar it creates is only ever put into once, so using spawn instead of separate fork and put gives a static guarantee that a multiple-put exception cannot occur at runtime. Static guarantees are good, so one may reasonably ask whether spawn should replace fork, new and put in the API. In practice we have found it to be convenient to have access to fork, new and put as these sometimes allow a computation to be expressed more straightforwardly; we shall see an example of this in Section 4. Programmers who wish to statically rule out a multiple-put exception can elect to restrict themselves to spawn.

3.2 Dataflow

The programming interface we described above yields a dataflow model, in which each fork creates a new computation node in the dataflow graph, and each IVar gives rise to edges from the producer to each of the consumers. Figure 1 shows pictorially a particular example of a network.

3.3 Safety

The interface given above does not prevent the programmer from returning an IVar from runPar and then passing it to other instances of runPar; the behaviour of the programming model is undefined under these circumstances. The semantics we give later will rule out such cases, but the API given above and our implementation of it do not. There is a well-known solution to this problem using parametricity [11], wherein a type variable parameter s is added both to the monad type and the variables (IVars, in our case):

```
newtype Par s a

runPar :: (forall s . Par s a) -> a

data IVar s a
new :: Par s (IVar s a)
get :: IVar s a -> Par s a
put :: IVar s a -> a -> Par s ()
```

The type of this alternate version of runPar statically prevents any IVars from being returned by the Par computation. If the programmer were to try to do this, then the type variable a would be unified with a type involving s, and thus s would escape the forall quantifier, so the typechecker would complain. However, there is a tradeoff with this design: the extra type parameter pervades client code and can be somewhat inconvenient. We are currently investigating the tradeoff and expect at the least to provide this version of the Par monad as an alternative.

4. Examples

In the following sections we illustrate some realistic uses for the Par monad.

4.1 A parallel type inferencer

An example that naturally fits the dataflow model is program analysis, in which information is typically propagated from definition sites to usage sites in a program. For the sake of concreteness, we pick a particular example: inferring types for a set of non-recursive bindings. Type inference gives rise to a dataflow graph; each binding is a node in the graph with inputs corresponding to the free variables of the binding, and a single output represents the derived type for the binding. For example, the following set of bindings

```
f = ...
g = ... f ...
h = ... f ...
j = ... g ... h ...
```

can be represented by the dataflow graph in Figure 1.
We assume the following definitions:

```
type Env = Map Var (IVar Type)
infer :: Env -> (Var,Expr) -> Par ()
```

where Expr is the type of expressions, Type is the type of types, and Map is a type constructor for finite maps (such as that provided by the Data.Map module in Haskell). The function infer infers the type for a binding, and calls put to augment the environment mapping for the bound variable with its inferred type.

We can then define a parallel type inferencer as follows:

```
parInfer :: [(Var,Expr)] -> [(Var,Type)]
parInfer bindings = runPar $ do
  let binders = map fst bindings
  ivars <- replicateM (length binders) new
  let env = Map.fromList (zip binders ivars)
  mapM_ (fork . infer env) bindings
  types <- mapM_ get ivars
  return (zip binders types)
```

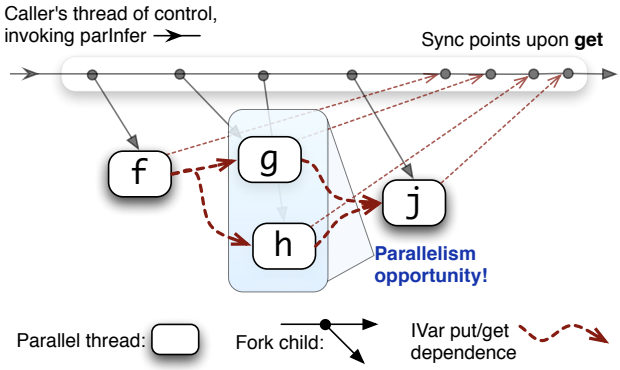

Figure 1. A dataflow graph, dynamically constructed by an invocation of the `parInfer` function. Threads synchronize upon requests for an `IVar`'s value. `IVars` are not shown explicitly above, but a red arrow indicates that a thread `gets` an `IVar` which was `put` by another thread.

The first three lines of the `do` expression build up the environment of type `Env` for passing to `infer`, in which each variable is mapped to a newly created `IVar`. The fourth line forks a call to `infer` to run the type inferencer for each binding. The next line waits for all the results using `mapM_ get`, and the last line returns the results.

This implementation extracts the maximum parallelism inherent in the dependency structure of the given set of bindings, with very little effort on the part of the programmer. There is no explicit dependency analysis; the `Par` monad is doing all the work to resolve the dependencies. The same trick can be pulled using lazy evaluation of course, and indeed we could achieve the same parallel structure using lazy evaluation together with Strategies, but the advantage of the `Par` monad version is that the structure is programmed explicitly and the runtime behaviour is not tied to one particular evaluation strategy (or compiler).

This example illustrates the motivation for several aspects of the design:

- The dataflow graph generated is entirely *dynamic*; there is no way that we could statically predict the structure of the graph, because it depends only on the input program. (We highlight this point because some dataflow programming models only allow *static* dataflow graphs to be constructed.)
- We considered earlier whether it would make sense to replace `new` and `put` by `spawn` (Section 3.1). This example demonstrates two reasons why that would be inconvenient:
  - Consider extending the type inferencer to support Haskell's *pattern bindings* in which each equation binds multiple variables; then it is extremely convenient for `infer` to simply call `put` once for each variable bound by a particular equation.
  - If we had `spawn` only and no `new`, then then consider how to write `parInfer`. The output of `spawn` would be required in order to construct the `Env` which is passed back into the call to `spawn` itself. Hence we would need recursion, either in the form of a `fixPar` operator or recursive `do` notation.

4.2 A divide and conquer skeleton

Algorithmic skeletons are a technique for achieving modularity in parallel programming [8]. The idea is that the parallel algorithm is expressed as the composition of a parallel *skeleton* with the sequential algorithm (suitably factorised into pieces that can be executed in parallel). In the `Par` monad, as with other functional parallel programming models, skeletons are readily expressed as higher-order functions. We saw an example of a simple skeleton above: `parMap`, otherwise known as the master-worker skeleton. A skeleton corresponding to divide-and-conquer algorithms can be defined as follows:

```
divConq :: NFData sol
        => (prob -> Bool)   -- indivisible?
        -> (prob -> [prob]) -- split into subproblems
        -> ([sol] -> sol)   -- join solutions
        -> (prob -> sol)    -- solve a subproblem
        -> (prob -> sol)
divConq indiv split join f prob
 = runPar $ go prob
 where
   go prob
     | indiv prob = return (f prob)
     | otherwise = do
         sols <- parMapM go (split prob)
         return (join sols)
```

this is a general divide-and-conquer of arbitrary (even variable) degree. The caller supplies functions that respectively determine whether the current problem is indivisible, split the problem into a list of sub-problems, join the results from sub-problems into a single result, and solve a particular problem instance.

4.3 Stream Processing Pipelines

`IVars` can be used as communication channels between threads, enabling producer-consumer parallelism. This is possible because while an `IVar` carries only a single value during its lifetime, that value may in turn contain other `IVars`. Thus, a linked list using `IVars` as tail pointers can serve as a stream datatype:

```
data IList a = Null | Cons a (IVar (IList a))
type Stream a = IVar (IList a)
```

Actors, or *kernels*, that process streams are nothing more than `Par` threads that perform `gets` and `puts`. Our current `Par` distribution includes a library of higher-order stream operators built in this way. As an example, the following function applies a stateful kernel to the elements of a stream, updating the state after each element processed and writing results to an output stream:

```
kernel :: NFData b => (s -> a -> (s,b)) -> s
       -> Stream a -> Stream b -> Par ()
kernel fn state inS outS =
   do ilst <- get inS
      case ilst of
          Null -> put outS Null -- End of stream.
          Cons h t -> do
            newtl <- new
            let (newstate, outp) = fn state h
            put outS (Cons outp newtl)
            kernel fn newstate t newtl
```

Notice that the above kernel will execute continuously as long as data is available, keeping that kernel on a single processor, where its working set is in-cache. Our experiments show this is effective: kernels typically execute on a single core for long periods.

The authors tried to make a similar construction using Strategies, and found it to be surprisingly difficult. Due to space constraints we are unable to describe the problem in detail, but in summary it is this: previously published techniques for pipeline Strategies [20] no longer work because they fall victim to requirement (c) from Section 2, the constraint that was added to avoid space leaks with the original formulation of Strategies [14]. It is possible to use element-wise parallelism instead, but that does not take advantage of the locality within a kernel. Ultimately we were not

$$
\begin{array}{rrcll}
 & x, y & \in & \textit{Variable} & \\
 & i & \in & \texttt{IVar} & \\
\textit{Values} & V & ::= & x \mid i \mid \texttt{\textbackslash}x \texttt{->} M & \\
 & & \mid & \texttt{return}\ M \mid M \texttt{>>=} N & \\
 & & \mid & \texttt{runPar}\ M & \\
 & & \mid & \texttt{fork}\ M & \\
 & & \mid & \texttt{new} & \\
 & & \mid & \texttt{put}\ i\ M & \\
 & & \mid & \texttt{get}\ i & \\
 & & \mid & \texttt{done}\ M & \\
\textit{Terms} & M, N & ::= & V \mid M\ N \mid \cdots & \\
\textit{States} & P, Q & ::= & M & \text{thread of computation} \\
 & & \mid & \langle\rangle_i & \text{empty } \texttt{IVar} \text{ named } i \\
 & & \mid & \langle M \rangle_i & \text{full } \texttt{IVar} \text{ named } i, \text{ holding } M \\
 & & \mid & \nu i.P & \text{restriction} \\
 & & \mid & P \mid Q & \text{parallel composition}
\end{array}
$$

Figure 2. The syntax of values and terms

$$
\begin{array}{c}
P \mid Q \equiv Q \mid P \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
\nu x.\nu y.P \equiv \nu y.\nu x.P \\
\nu x.(P \mid Q) \equiv (\nu x.P) \mid Q, \qquad x \notin \mathit{fn}\,(Q)
\end{array}
$$

$$
\frac{P \ \rightarrow\ Q}{P \mid R \ \rightarrow\ Q \mid R} \qquad \frac{P \ \rightarrow\ Q}{\nu x.P \ \rightarrow\ \nu x.Q}
$$

$$
\frac{P \equiv P' \quad P' \ \rightarrow\ Q' \quad Q' \equiv Q}{P \ \rightarrow\ Q}
$$

Figure 3. Structural congruence, and structural transitions.

able to achieve as much speedup with Strategies as we were with the `Par` monad on this example[3].

5. Operational Semantics

In this section we add precision to our informal description of the programming model with a formal operational semantics. The operational semantics will allow non-determinism in the reduction order, modeling a truly parallel implementation, but we will argue (in Section 5.2) that the result of any given application of `runPar` is deterministic.

5.1 The semantics

Figure 2 gives the syntax of values and terms in our language. The only unusual form here is `done` M, which is an internal tool we shall use in the semantics for `runPar`; it is not a term form available to the programmer.

The main semantics for the language is a big-step operational semantics written

$$M \Downarrow V$$

[3] The reader is invited to try; sample code can be found at: `https://github.com/simonmar/monad-par/blob/master/examples/stream/disjoint_working_sets_pipeline.hs`.

$$
\frac{M \not\equiv V \quad M \Downarrow V}{\mathcal{E}[M] \ \rightarrow\ \mathcal{E}[V]} \qquad (\textit{Eval})
$$

$$
\begin{array}{rcll}
\mathcal{E}[\texttt{return}\ N \texttt{>>=} M] & \rightarrow & \mathcal{E}[M\ N] & (\textit{Bind}) \\
\mathcal{E}[\texttt{fork}\ M] & \rightarrow & \mathcal{E}[\texttt{return ()}] \mid M & (\textit{Fork}) \\
\mathcal{E}[\texttt{new}] & \rightarrow & \nu i.(\langle\rangle_i \mid \mathcal{E}[\texttt{return}\ i]), \quad i \notin \mathit{fn}\,(\mathcal{E}) & (\textit{New}) \\
\langle M \rangle_i \mid \mathcal{E}[\texttt{get}\ i] & \rightarrow & \langle M \rangle_i \mid \mathcal{E}[\texttt{return}\ M] & (\textit{Get}) \\
\langle\rangle_i \mid \mathcal{E}[\texttt{put}\ i\ M] & \rightarrow & \langle M \rangle_i \mid \mathcal{E}[\texttt{return ()}] & (\textit{PutEmpty}) \\
\texttt{return}\ M & \rightarrow & & (\textit{GCReturn}) \\
\nu i.\langle M \rangle_i & \rightarrow & & (\textit{GCFull}) \\
\nu i.(\langle\rangle_i \mid \mathcal{E}[\texttt{get}\ i]^*) & \rightarrow & & (\textit{GCDeadlock})
\end{array}
$$

$$
\frac{(M \texttt{>>=} \texttt{\textbackslash}x.\texttt{done}\ x) \ \rightarrow^*\ \texttt{done}\ N, \quad N \Downarrow V}{\texttt{runPar}\ M \Downarrow V} \qquad (\textit{RunPar})
$$

Figure 4. Transition Rules

meaning that term M reduces to value V in zero or more steps. It is entirely conventional, so we omit all its rules except one, namely $(\textit{RunPar})$ in Figure 4. We will discuss $(\textit{RunPar})$ shortly, but the important point for now is that it in turn depends on a small-step operational semantics for the `Par` monad, written

$$P \ \rightarrow\ Q$$

Here P and Q are *states*, whose syntax is given in Figure 2. A state is a bag of terms M (its active "threads"), and `IVars` i that are either full, $\langle M \rangle_i$, or empty, $\langle\rangle_i$. In a state, the $\nu i.P$ serves (as is conventional) to restrict the scope of i in P. The notation $P_0 \rightarrow^* P_i$ is shorthand for the sequence $P_0 \rightarrow ... \rightarrow P_i$ where $i \geq 0$.

States obey a structural equivalence relation $\equiv$ given by Figure 3, which specifies that parallel composition is associative and commutative, and scope restriction may be widened or narrowed provided no names fall out of scope. The three rules at the bottom of Figure 3 declare that transitions may take place on any sub-state, and on states modulo equivalence. So the $\rightarrow$ relation is inherently non-deterministic.

The transitions of $\rightarrow$ are given in in Figure 4 using an *evaluation context* $\mathcal{E}$:

$$\mathcal{E} ::= [\cdot] \mid \mathcal{E} \texttt{>>=} M$$

Hence the term that determines a transition will be found by looking to the left of `>>=`. Rule $(\textit{Eval})$ allows the big-step reduction semantics $M \Downarrow V$ to reduce the term in an evaluation context if it is not already a value.

Rule $(\textit{Bind})$ is the standard monadic bind semantics.

Rule $(\textit{Fork})$ creates a new thread.

Rules $(\textit{New})$, $(\textit{Get})$, and $(\textit{PutEmpty})$ give the semantics for operations on `IVars`, and are straightforward: `new` creates a new empty `IVar` whose name does not clash with another `IVar` in scope, `get` returns the value of a full `IVar`, and `put` creates a full `IVar` from an empty `IVar`. Note that there is no transition for `put` when the `IVar` is already full: in the implementation we would signal an error to the programmer, but in the semantics we model the error condition by having no transition. Also note that the semantics here does not model the hyperstrict behaviour of the programmer-level `put`; indeed there are various options for the strictness of `put`, and the choice has no effect on the rest of the semantics or its determinism property, so we ignore it here.

Several rules that allow parts of the state to be *garbage collected* when they are no longer relevant to the execution. Rule ($GCReturn$) allows a completed thread to be garbage collected. Rule ($GCFull$) allows a full `IVar` to be garbage collected provided the `IVar` is not referenced anywhere else in the state. The equivalences for ν in Figure 3 allow us to push the ν down until it encloses only the dead `IVar`.

Rule ($GCDeadlock$) allows an empty `IVar` and a set of zero or more deadlocked threads to be garbage collected: the syntax $\mathcal{E}[\texttt{get}\ i]^*$ means zero or more threads of the given form. Since there can be no other threads that refer to i, none of the `get`s can ever make progress. Hence the entire set of deadlocked threads together with the empty `IVar` can be removed from the state.

The final rule, ($RunPar$), gives the semantics of `runPar` and connects the `Par` reduction semantics $\rightarrow$ with the functional reduction semantics $\Downarrow$. Informally it can be stated thus: if the argument M to `runPar` runs in the `Par` semantics yielding a result N, and N reduces to V, then `runPar` M is said to reduce to V. In order to express this, we need a distinguished term form to indicate that the "main thread" has completed: this is the reason for the form `done` M. The programmer is never expected to write `done` M directly, it is only used as a tool in the semantics. We will use a similar trick in the implementation of `runPar` itself (Section 6.2).

5.2 Determinism

An informal argument for determinism follows. We require that if `runPar` $M \Downarrow N$ and `runPar` $M \Downarrow N'$, then $N = N'$, and furthermore that if `runPar` $M \Downarrow N$ then there is no sequence of transitions starting from `runPar` M that reaches a state in which no reduction is possible. Informally, `runPar` M should either produce the same value consistently, or produce no value consistently (which is semantically equivalent to $\bot$).

First, observe that the system contains no transitions that take a full `IVar` to an empty `IVar`, or that change the contents of a full `IVar`. A non-deterministic result can only arise due to a *race condition*: two different orderings of reductions that lead to a different result. To observe non-determinism, there must be multiple reduction sequences leading to applications of the rule ($PutEmpty$) for the same `IVar` with different values. There is nothing in the semantics that prevents this from happening, but our determinism argument rests on the ($RunPar$) rule, which requires that the state at the end of the reduction consists of only `done` M for some M. That is, the rest of the state has completed or deadlocked and been garbage collected by rules ($GCReturn$), ($GCFull$), and ($GCDeadlock$). In the case where there is a choice between multiple `put`s, one of them will not be able to complete and will remain in the state, and thus the `runPar` transition will never be applicable.

The reader interested in a full proof of determinism should refer to Budimlic et al. [4], which contains a proof of determinism for Featherweight CnC, a language which is essentially equivalent to `Par`.

6. Implementation

As in Claessen [7], the implementation is in two parts: computations in the monad produce a lazy stream of operations, or *trace*, and a *scheduler* consumes the traces of the set of runnable threads, switching between them according to a scheduling policy.

In this section we first describe the implementation of the `Par` monad, and then give two scheduler implementations: first a sequential scheduler (Section 6.2), and then a true parallel scheduler using work-stealing (Section 6.3).

The separation between the `Par` monad and the scheduler is not a fundamental requirement of the technique, indeed we could combine the `Par` monad with the scheduler for a more efficient implementation. However, this modular presentation is much easier to understand, and separating the details of the scheduler from the `Par` monad allows us to describe two independent scheduler implementations without repeating the implementation of `Par`. Indeed, in our real implementation we retain this separation so as to facilitate the provision of multiple scheduling policies, or user-defined schedulers. Specialising the `Par` monad with respect to a particular scheduler is entirely possible and mechanical; we describe how to do this in Section 6.4.1.

6.1 Implementation of the `Par` monad

Computations in the `Par` monad produce a `Trace`:

```
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)
```

Our `fork` and `get` operations require that we be able to suspend a computation and resume it later. The standard technique for implementing suspension and resumption is to use continuation-passing, which is exactly what we do in the form of a continuation-passing monad. The continuation result is fixed to be of type `Trace`, but otherwise the monad is completely standard:

```
newtype Par a = Par {
     unPar :: (a -> Trace) -> Trace
  }
instance Monad Par where
    return a = Par $ \c -> c a
    m >>= k  = Par $
       \c -> unPar m (\a -> unPar (k a) c)
```

The basic operations in `Par` simply return the appropriate `Trace` values. Firstly, `fork`:

```
fork :: Par () -> Par ()
fork p = Par $
   \c -> Fork (unPar p (\_ -> Done)) (c ())
```

recall that `Fork` has two arguments of type `Trace`, these represent the traces for child and the parent respectively. The child is the `Trace` constructed by applying the argument `p` to the continuation `(\_ -> Done)`, while the parent `Trace` results from applying the continuation `c` to `()` – the unit value because `fork` returns `Par ()`.

The `IVar` operations all return `Trace` values in a straightforward way:

```
new :: Par (IVar a)
new  = Par $ New

get :: IVar a -> Par a
get v = Par $ \c -> Get v c

put :: NFData a => IVar a -> a -> Par ()
put v a = deepseq a (Par $ \c -> Put v a (c ()))
```

Note that `put` fully evaluates the argument `a` using `deepseq` before returning the `Put` trace.

6.2 A sequential scheduler

This sequential scheduler is implemented in the `IO` monad, but using only deterministic operations (`runPar` uses `unsafePerformIO`). By using `IO` here we will be able to smoothly extend the implementation to support true parallelism in Section 6.3.

Here are some of the main types in the scheduler:

```
sched :: SchedState -> Trace -> IO ()
type SchedState = [Trace]
newtype IVar a = IVar (IORef (IVarContents a))
data IVarContents a = Full a | Blocked [a -> Trace]
```

The scheduler is an `IO` function that takes some state of type `SchedState`, and the current thread of type `Trace`: The state of the scheduler, `SchedState`, is its work pool, represented by a list of threads ready to run. A `IVar` is represented by an `IORef`, which contains either a value (`Full a`), or a list of blocked threads waiting for the value (`Blocked [a -> Trace]`).

We need a small auxiliary function, `reschedule`, to choose the next thread to run from the pool and invoke the scheduler proper:

```
reschedule :: SchedState -> IO ()
reschedule []     = return ()
reschedule (t:ts) = sched ts t
```

We also use the following auxiliary function to modify the contents of an `IORef`:

```
modifyIORef :: IORef a -> (a -> (a,b)) -> IO b
```

Next, we describe each case of `sched` separately. First, the case for `Fork`:

```
sched state (Fork child parent) =
  sched (child:state) parent
```

We simply push the child thread on the stack of runnable threads, and continue with the parent. Note that we treat the runnable threads like a stack rather than a queue. If we were implementing concurrency, then fairness would be a consideration and a different data structure might be more appropriate. However, here the scheduling policy is relevant only for performance, and so in the sequential scheduler we use a stack for simplicity. We return to the question of scheduling policies in Section 6.4.

When a thread has completed, its trace ends with `Done`; in that case, we look for more work to do with `reschedule`:

```
sched state Done = reschedule state
```

Next we deal with creating new `IVar`s:

```
sched state (New f) = do
  r <- newIORef (Blocked [])
  sched state (f (IVar r))
```

The case for `Get` checks the current contents of the `IVar`. If it is `Full`, then we continue by applying the continuation in the `Get` constructor to the value in the `IVar`. If the `IVar` is empty, we block the current thread by adding its continuation to the list already stored in the `IVar` and pick the next thread to run with `reschedule`:

```
sched state (Get (IVar v) c) = do
  e <- readIORef v
  case e of
    Full a -> sched state (c a)
    Blocked cs -> do
      writeIORef v (Blocked (c:cs))
      reschedule state
```

The case for `Put` also checks the current state of the `IVar`. If it is `Full`, then this is a repeated put, so the result is an error. Otherwise, we store the value `a` in the `IVar` as `Full a`, and unblock any blocked threads by applying them to `a`, and putting them in the work pool.

```
sched state (Put (IVar v) a t) = do
  cs <- modifyIORef v $ \e -> case e of
          Full _     -> error "multiple put"
          Blocked cs -> (Full a, cs)
  let state' = map ($ a) cs ++ state
  sched state' t
```

Finally, the implementation of `runPar` is below. Arranging for the return value of the main thread to be communicated out is a little tricky: the scheduler has no return value itself, and to give it one would require parameterising all our types with the return type (including `Par` and `IVar`). Instead, we create an `IVar` for the purpose of collecting the return value, and compose the main `Par` action, `x`, with a `put` operation to store the return value. (This is similar to the technique we used to define the semantics of `runPar` earlier.) When the scheduler returns, we extract the return value by reading the `IVar`'s `IORef`.

```
runPar :: Par a -> a
runPar x = unsafePerformIO $ do
   rref <- newIORef (Blocked [])
   sched [] $ unPar (x >>= put_ (IVar rref))
                    (const Done)
   r <- readIORef rref
   case r of
     Full a -> return a
     _      -> error "no result"
```

6.3 A parallel work-stealing scheduler

We now modify the sequential scheduler described in the previous section to support true parallel execution, with a work-stealing scheduling policy. Any scheduling policy can be implemented, of course; the work-stealing algorithm happens to perform relatively well on a wide range of parallel programs, but the optimal scheduling policy may well depend on the algorithm to be parallelised [19]. Since our scheduler implementation is entirely in Haskell and is relatively simple, different schedulers can be readily implemented by the programmer, and over time we expect to build up a library of scheduling algorithms.

As described earlier, the `Par` monad is independent of the choice of scheduler. Specifically the definition of `Trace`, the definition of `Par` and its `Monad` instance, and the implementations of `fork`, `get`, `put` and `new` all remain the same as in Section 6.1.

Our parallel scheduler works as follows:

- One Haskell thread is created per processor core. We call these the *worker threads*.
- Each thread runs its own instance of the scheduler, and each has a local work pool of runnable `Trace`s. When the local work pool runs dry, a scheduler attempts to steal items from the work pools of other worker threads.
- When a worker thread cannot find any work to do it becomes *idle*. The set of idle processors is represented as a list of `MVar`s, such that calling `putMVar` on one of these `MVar`s will wake up an idle worker thread. When a worker thread creates a new work item, it wakes up one idle worker thread.
- When all work pools are empty, the computation is complete and `runPar` can return.

For efficiency we use an `IORef` to represent shared state, and `atomicModifyIORef` to perform atomic operations on it. The `atomicModifyIORef` operation has the same signature as `modifyIORef` that we used earlier:

```
atomicModifyIORef :: IORef a -> (a -> (a,b)) -> IO b
```

the difference is that the update is performed atomically; the contents of the `IORef` are replaced by a lazy thunk representing the result.

When we require not only atomicity but also *blocking*, we use `MVar`s. Alternatively, we could have used STM for all of the shared state, although that would add some overhead to the implementation.

The scheduler state `SchedState` is no longer just a work pool, but a record with four fields:

```
data SchedState = SchedState
    { no       :: Int,
      workpool :: IORef [Trace],
      idle     :: IORef [MVar Bool],
      scheds   :: [SchedState] }
```

containing, in order:

- `no`: the thread number
- `workpool`: the local work pool, stored in an `IORef` so that other worker threads may steal from it
- `idle`: an `IORef` shared by all threads, containing the list of currently idle worker threads, each represented by an `MVar Bool` (The `Bool` indicates whether the computation is complete or not: if a thread is woken up with `putMVar m False` then it should continue to look for more work, otherwise it should stop.)
- `scheds`: the list of `SchedState`s corresponding to all schedulers (this is used by the current scheduler to find other workpools to steal from).

The `reschedule` function has the same signature as before, but is different in two ways: first, we must use `atomicModifyIORef` to remove an item from the work pool, and second if the work pool is empty the scheduler attempts to *steal* from other work pools. Stealing is implemented by a function `steal` that we shall describe shortly.

```
reschedule :: SchedState -> IO ()
reschedule state@SchedState{ workpool } = do
  e <- atomicModifyIORef workpool $ \ts ->
         case ts of
           []      -> ([], Nothing)
           (t:ts') -> (ts', Just t)
  case e of
    Nothing -> steal state
    Just t  -> sched state t
```

We add a new auxilliary function `pushWork` that is used for adding a new item to the local work pool. It also checks the list of idle worker threads; if there are any, then one is woken up.[4] Note that here the value returned from `atomicModifyIORef` has type `IO ()`; we bind this to `r` and then execute it in the `IO` monad.

```
pushWork :: SchedState -> Trace -> IO ()
pushWork SchedState { workpool, idle } t = do
  atomicModifyIORef workpool $ \ts -> (t:ts, ())
  idles <- readIORef idle
  when (not (null idles)) $ do
    r <- atomicModifyIORef idle $ \is ->
      case is of
        []     -> ([], return ())
        (i:is) -> (is, putMVar i False))
    r  -- r has type IO ()
```

The scheduler itself has the same signature as before:

```
sched :: SchedState -> Trace -> IO ()
```

`Fork` and `Done` are straightforward:

```
sched state (Fork child parent) = do
  pushWork state child
  sched state parent

sched state Done = reschedule state
```

The implementation of `New` is unchanged. `Get` is different only in that an `atomicModifyIORef` is needed to operate on the contents of the `IVar`:

```
sched state (Get (IVar v) c) = do
  e <- readIORef v
  case e of
    Full a -> sched state (c a)
    _other -> do
      r <- atomicModifyIORef v $ \e -> case e of
             Full a     ->
               (Full a, sched state (c a))
             Blocked cs ->
               (Blocked (c:cs), reschedule state)
      r  -- r has type IO ()
```

Put requires `atomicModifyIORef` to operate on the `IVar`, and calls `pushWork` to wake up any blocked `get`s, but otherwise is straightforward:

```
sched state (Put (IVar v) a t) = do
  cs <- atomicModifyIORef v $ \e -> case e of
          Full _   -> error "multiple put"
          Blocked cs -> (Full a, cs)
  mapM_ (pushWork state . ($ a)) cs
  sched state t
```

The `steal` function implements work-stealing. It loops through the list of `SchedState`s (omitting the current one, which is known to have an empty work pool). For each sibling worker, we attempt to remove an item from its work pool using `atomicModifyIORef`. If successful, we can call `sched` to continue executing the stolen work item. If we get to the end of the list without stealing anything, we create a new `MVar`, add it to the `idle` list, and block in `takeMVar`, waiting to be woken up. The result of the `takeMVar` is a `Bool` value indicating whether we should stop, or continue to look for more work to do. If the list of idle workers already contains `numCapabilities-1` elements (where `numCapabilities` is the number of processors), then we know that all the work queues are empty, and we wake up all the idle workers passing `True` to indicate that they should exit.

Note that there is a race condition in `steal`. We do not atomically look at *all* the scheduler queues simultaneously, so it is possible that the search reaches the end of the list without finding any work and the worker becomes idle, meanwhile another worker has added items to its work pool. The result is lost parallelism (but not deadlock), since there would be an idle worker and a non-empty work queue. However, typically work items are being created regularly, so the next work item to be pushed will result in the idle worker being woken up again, hence we believe that there will be little lost parallelism in practice. So far we have not observed the problem in our benchmarks, but if it did become an issue then there are a number of ways to fix it, for example using STM.

```
steal :: SchedState -> IO ()
steal state@SchedState{ idle, scheds, no=my_no }
  = go scheds
  where
    go (x:xs)
      | no x == my_no = go xs
      | otherwise     = do
         r <- atomicModifyIORef (workpool x) $
                \ts -> case ts of
                    []     -> ([], Nothing)
                    (x:xs) -> (xs, Just x)
         case r of
```

[4] Note that we first read the list of idle threads using a non-atomic `readIORef` and only use `atomicModifyIORef` if the list is found to be non-empty. This avoids performing an atomic operation in the common case that all schedulers are working.

```
        Just t  -> sched state t
        Nothing -> go xs
  go [] = do
    m <- newEmptyMVar
    r <- atomicModifyIORef idle $
           \is -> (m:is, is)
    if length r == numCapabilities - 1
       then mapM_ (\m -> putMVar m True) r
       else do
         done <- takeMVar m
         if done then return ()
                 else go scheds
```

Finally, the `runPar` operation is a small elaboration of the previous sequential version. We create one `SchedState` for each core (the number of which is given by `numCapabilities`). Then, we create one thread per core with `forkOnIO`; this GHC primitive creates a thread which is tied to a particular core and may not be migrated by the runtime system's automatic load-balancing mechanisms. One of these threads runs the "main thread" (that is, the `Par` computation passed to `runPar`); this should normally be chosen to be the current CPU, since it is likely to be cache-hot. In the implementation below, we simply refer to it as `main_cpu` and omit the details. The non-main threads simply call `reschedule`, and hence will immediately start looking for work to do. An `MVar` is used to pass the final result from the main thread back to `runPar`, which blocks in `takeMVar` until the result is available before returning it.

```
runPar :: Par a -> a
runPar x = unsafePerformIO $ do
   let n = numCapabilities
   workpools <- replicateM n $ newIORef []
   is <- newIORef []
   let scheds =
          [ SchedState { no = x,
                         workpool = wp,
                         idle = is,
                         scheds=scheds }
          | (x,wp) <- zip [0..] workpools ]

   m <- newEmptyMVar
   forM_ (zip [0..] states) $ \(cpu,state) ->
      forkOnIO cpu $
         if (cpu /= main_cpu)
            then reschedule state
            else do
                 rref <- newIORef Empty
                 sched state $
                    runPar (x >>= put_ (IVar rref))
                           (const Done)
                 r <- readIORef rref
                 putMVar m r

   r <- takeMVar m
   case r of
     Full a -> return a
     _ -> error "no result"
```

6.3.1 Correctness

If some subterm evaluates to $\bot$ during execution of `runPar`, then the value of the whole `runPar` should be $\bot$. The sequential scheduler implements this, but the parallel scheduler given above does not. In practice, however, it will make little difference: if one of the workers encounters a $\bot$, then the likely result is a deadlock, because the thread that the worker was executing will probably have outstanding `put`s. Since deadlock is semantically equivalent to $\bot$, the result in this case is equivalent. Nevertheless, modifying the implementation to respect the semantics is not difficult, and we plan to modify our real implementation to do this in due course.

6.4 Optimising the scheduler

The implementation given above was used in generating the benchmark measurements presented in the next section, and gives entirely acceptable results for many benchmarks. However, when pushed to the limit, particularly with very fine-grained parallelism, there are overheads in this implementation that become apparent (measurements of the overhead are given in Section 7.2).

There are several ways in which we can reduce the overhead, as outlined in the following sections.

6.4.1 Deforesting `Trace`

As we remarked earlier, the `Trace` datatype is a useful device for separating the scheduler from the `Par` monad implementation, but it is not a fundamental requirement of the implementation. We can eliminate the intermediate `Trace` data structure entirely mechanically, and thereby specialise the `Par` monad to a particular scheduler, as follows. First we define

```
type Trace = SchedState -> IO ()
```

and then replace each constructor in `Trace` with a function of the same type, whose implementation is the appropriate case in `sched`. For example, instead of the `Fork` constructor, we define

```
traceFork :: Trace -> Trace -> Trace
traceFork child parent = ...
```

and then replace instances of `Fork` with `traceFork`. Finally, we replace any calls of the form `sched state t` with `t state`.

We have experimented with this optimisation but at present it fails to achieve any significant benefit due to known deficiencies in GHC's optimiser. When the issues with the optimiser are resolved, we anticipate that eliminating the `Trace` data structure in this way will improve the performance of `Par` significantly.

6.4.2 Alternative scheduling policies

Choice of scheduling policy may have an asymptotic effect on space usage [19], and so ideally we would like to be able to select the scheduling policy per-algorithm and at runtime. Some of the scheduling options we have in mind are:

- scheduling the child rather than the parent of `fork` first. This is as simple as switching the arguments to `Fork`.
- oldest-first or random work-stealing. This requires changing the data structure used to represent the work queue from a list to something else, such as `Data.Sequence`. Another option is to use lock-free work-stealing deques, which are implemented in the GHC runtime system for `par`, but not currently available at the Haskell level. We hope to expose this functionality to Haskell and hence make it possible to use lock-free work-stealing deques in our `Par` monad scheduler.

While our primary implementation supports scheduling the parent in `fork` and newest-first work-stealing only, we have an experimental version that also supports child-first scheduling of `fork` and random work stealing, selectable at compile-time[5] and we hope to be able to allow runtime-selectable schedulers in the future.

6.4.3 Optimising nested `runPar`s

Each call to `runPar` creates a new gang of Haskell threads and schedulers. This is a rather large overhead, and so the current implementation of `runPar` is only suitable for large-scale parallelism. If multiple `runPar`s are active simultaneously, their threads will compete with each other for the processors, and pathological scheduling may occur. We believe it should be possible to have multiple

[5] Indeed, toggling those options can reduce the running time of `parfib` by 40% from what is reported in Section 7.2.

runPar instances share a single scheduler instance via some global shared state, and we plan to investigate this in future work.

7. Performance

In this section we analyse the performance of the Par monad in two ways: firstly we measure the raw overhead imposed by the Par monad, and secondly we consider the scaling properties of a selection of existing parallel benchmarks when using both the Par monad and the Strategies abstraction.

7.1 Experimental setup

We benchmarked on a machine containing 4 Intel Xeon E7450 processors running at 2.4GHz. Each processor has 6 cores, for a total of 24 cores. The OS was Windows Server 2008, and we used a development version of GHC (7.1.20110301).[6]

GHC by default uses a 2-generation stop-the-world parallel garbage collector. We configured the runtime to use a fixed nursery size of 1MB (which we have measured as close to optimal for this hardware). We also used a fixed overall heap size chosen for each benchmark to be about three times the maximum heap residency required by the program. The purpose here is to level the playing field: by using a fixed heap size we eliminate the possibility that one version is gaining an advantage by using a larger memory footprint in order to reduce the overhead of GC. A larger heap size also helps avoid artifacts due to the timing of old-generation GCs.

The Par implementation used is exactly the work-stealing implementation as described in Section 6.

7.2 Overhead

To assess the overhead of the Par monad relative to the low-level par/pseq operations used in the Strategies library, we used the parfib microbenchmark. The implementations are as follows, with par/pseq on the left and the Par monad version on the right:

```
parfib :: Int -> Int
parfib n | n < 2 = 1
parfib n =
  x `par` y `pseq` (x+y)
  where
     x = parfib (n-1)
     y = parfib (n-2)
```

```
parfib :: Int -> Par Int
parfib n | n < 2 = return 1
parfib n = do
   xf <- spawn_ $ parfib (n-1)
   y  <-          parfib (n-2)
   x  <- get xf
   return (x+y)
```

We measured the time taken to evaluate parfib 34 on a single processor with each version, and the total memory allocated by each program. The results were as follows:

version	time (slowdown)	memory allocated
par/pseq	0.29s	0.17GB
Par monad	6.15s (21×)	6.19GB

So the overhead of the Par monad is at least a factor of 21 over par/pseq. This is unarguably high, but there is ample opportunity for improvement: we examined the code generated by GHC for the Par monad version and observed several opportunities for optimisation that are not yet being performed by GHC's optimiser. Hence there is reason to believe that by paying attention to GHC's optimiser and by optimising the Par monad itself (Section 6.4) we should be able to reduce the overhead considerably.

Neither version of the parfib benchmark above parallelises well in practice, because the granularity is much too fine. As with many divide-and-conquer algorithms, it is necessary to define a cut-off point below which the normal sequential version of the algorithm is used. Given a suitable cut-off point for parfib, the overhead of the Par monad becomes irrelevant (as indeed it is in most of our actual benchmarks). Running parfib 43 with a cut-off at 24, both implementations achieve a speedup of 21× on 24 cores and almost identical running times.

7.3 Scaling

To measure scaling performance we used a selection of benchmarks from two sources. The following benchmarks were taken from the Intel CnC Haskell[7] distribution, and converted into two versions, one using Strategies and one using the Par monad:

- blackscholes: An implementation of the Black-Scholes algorithm for modeling financial contracts. This program is a straightforward use of parMap in both Strategies and Par versions.
- nbody: calculate the forces due to gravity between a collection of bodies in 3-dimensional space.

The following benchmarks were obtained from the nofib benchmark suite and converted to use the Par monad:

- mandel: a mandelbrot set renderer.
- matmult: matrix multiplication (unoptimised; using a list-of-lists representation for the matrices).
- minimax: a program to find the best move in a game of 4×4 noughts-and-crosses, using alpha-beta searching of the game tree to a depth of 6 moves. The game tree is evaluated in parallel.
- queens: calculate the number of solutions to the N-queens problem for 14 queens on a 14x14 board. Parallelism is via divide-and-conquer.
- sumeuler: compute the sum of the value of Euler's function applied to each integer up to a given bound.

We measure the wall-clock running time of the whole program, which in some cases includes a sequential component where the program constructs its input data (matmult in particular has to construct the input matrices); the sequential component obviously limits the possible speedup.

Figure 5 shows the scaling results up to 24 cores for each benchmark, comparing the Strategies version with the Par monad version. To make a direct comparison, we normalised both sets of results to the same baseline, namely the Strategies program compiled for sequential execution. We have omitted the results below 12 cores, which are largely uninteresting. Error bars are shown at one standard deviation; in most case the deviation is too small to be visible, except in one case — the 24-core result for minimax shows high variability. We believe this is an artifact of the high rate of GC performed by this benchmark together with the requirement that each GC perform a full synchronisation across all cores.

The results are surprisingly similar, and show a high degree of parallelism being achieved by both Strategies and Par. At 24 cores, two programs perform slightly better with Par: mandel and minimax, whereas one program is slightly worse: queens.

While the graphs only measure scaling, the absolute performance is also almost identical: the Strategies versions are on average 1% faster at 1 core, 2% slower at 12 cores and 2% slower at 24 cores. Hence we conclude that for typical parallel programs, the choice between Par and Strategies makes little difference to scaling or overall performance.

[6] the Par library works on older versions too, but we added an extra primitive in GHC 7.1 that allows a thread to determine which processor it is running on, which allows runPar to make a better choice about how to place the main thread, which in turn improves performance slightly.

[7] http://hackage.haskell.org/package/haskell-cnc

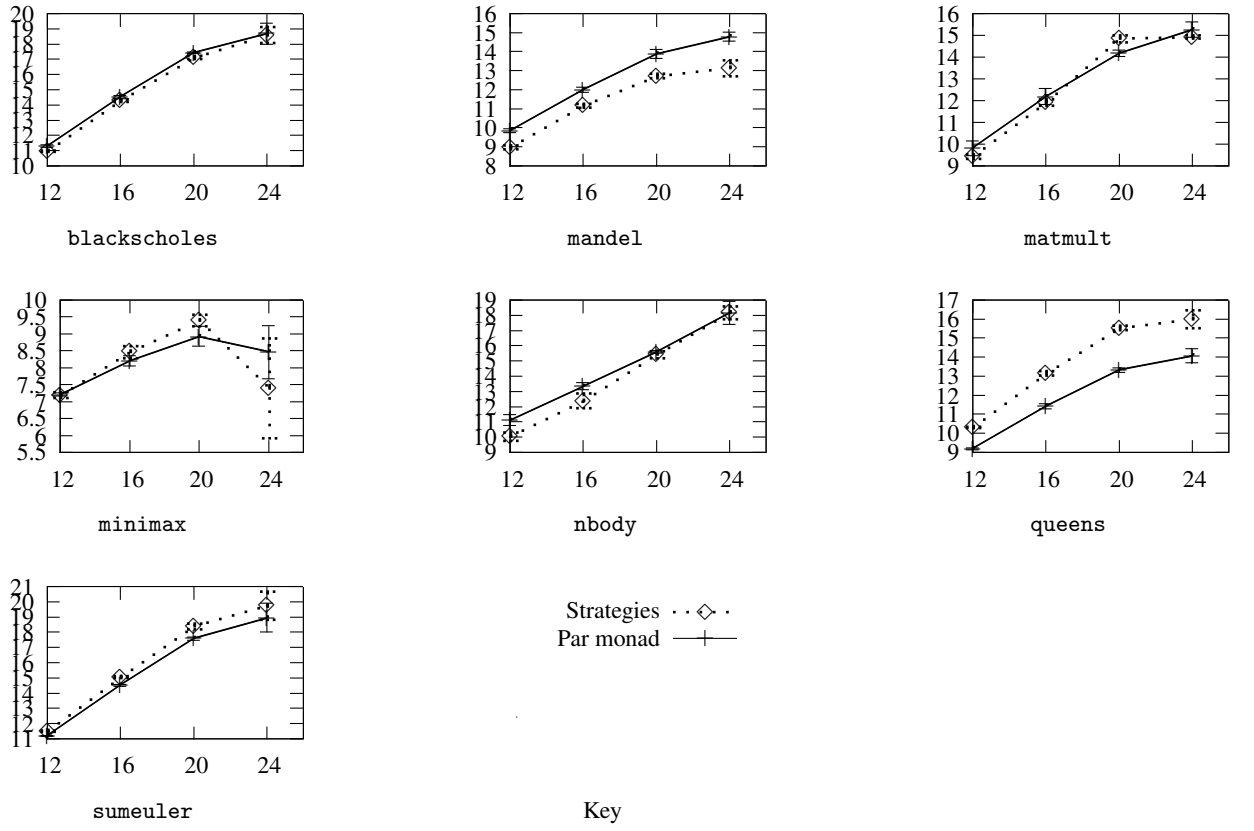

Figure 5. Speedup results on 24 cores (Y axis is speedup, X axis is number of OS threads)

8. Related Work

Many parallel programming systems include a fork/join construct for dividing control flow, for example: Habanero Java [5] and Cilk [3]. These work well for divide-and-conquer algorithms and other problems where dataflow closely matches control flow.

Some of these models also integrate synchronous data structures as primitives, structuring the parallel scheduler around them. pH [16], described in Section 1, is one example, including both `IVars` and `MVars`. Further, the Concurrent ML model provides synchronous point-to-point communication and has recently been paralellised [17]. Manticore [9] is a parallel variant of ML that supports both the CML programming model and a selection of parallel constructs including explicit futures.

Some systems based on Synchronous Data Flow [12] such as StreamIt [10], retain determinism but are significantly less expressive than the `Par` monad, being limited to first-order kernel functions and communication channels. Kahn process networks [12] are more general that Synchronous Dataflow, but to our knowledge there are no practical parallel programming systems currently available based on this abstraction. One that is closely related, however, is the Intel Concurrent Collections (CnC) model [4]. CnC provides a superset of the `Par` monad's functionality, including an extension of `IVars` that store a collection of key-value pairs. But CnC is a more complex model than `Par` and its implementations are focused on statically known networks rather than dynamic (forking) computations.

Finally, the dataflow model of computation induced by the `Par` monad and its `IVars` is very similar to that of the Skywriting language [15], although the context (distributed-memory cloud computing) is quite different. Indeed, with coarser grains of computation we conjecture that our monadic formalism constitutes an excellent domain-specific language for large-scale parallel programming, something we hope to try in the future.

A sceptical reader might reasonably ask whether the `Par` monad merely solves a problem that is unique to Haskell, namely laziness. After all, many of the difficulties identified in Section 2 relate to reasoning about strictness, a problem that simply does not arise in a call-by-value language. True enough – but the strict-vs-lazy debate is a larger question than we address here; what we show in this paper is how to integrate selective control over evaluation order into a lazy language, without throwing the baby out with the bathwater. Moreover the `Par` monad lifts the implementation of the parallel fabric to the library level where it is vastly more malleable; a direct benefit of this is the ability to easily change the scheduling policy. This would be useful in any language, strict or lazy.

8.1 Comparison with `par` and Strategies

In Section 2 we outlined some difficulties with using the `par` operator and Strategies, and claimed that our new interface would avoid the problems; here we return to assess that claim, and also offer some comments on the other differences between the `Par` monad and Strategies.

Is the `Par` monad any easier to use than `par`?

- Each `fork` creates exactly one parallel task, and the dependencies between the tasks are explicitly represented by `IVars`. Parallel structure programmed in the `Par` monad is well-defined: we gave an operational semantics in Section 5 in terms of the `Par` monad operations themselves.

- The programmer does not need to reason about laziness. In fact, we have deliberately made the `put` operation hyperstrict by default, to head off any confusion that might result from communicating lazy values through an `IVar`. Hence, if the program is written such that each parallel task only reads inputs from `IVar`s and producing outputs to `IVars`, the programmer can reason about the cost of each parallel task and the overall parallelism that can be achieved.

Nevertheless, the `Par` monad does not prevent lazy computations from being shared between threads, and the reckless programmer might even be able to capture some (unreliable) parallelism using only `fork` and laziness. This is not likely to work well: the `Par` monad scheduler cannot detect a thread blocked on a lazy computation and schedule another thread instead. Sharing lazy computations between threads in the `Par` monad is therefore to be avoided, but unfortunately we lack a way to statically prevent it.

A key benefit of the Strategies approach is that it allows the algorithm to be separated from the parallel coordination, by having the algorithm produce a lazy data structure that is consumed by the Strategy. The `Par` monad does not provide this form of modularity. However, many algorithms do not lend themselves to being decomposed in this way even with Strategies, because it is often inconvenient to produce a lazy data structure as output. We believe that higher-order skeletons (Section 4.2) are a more generally effective way to provide modular parallelism.

Strategies supports *speculation* in a strictly stronger sense than `Par`. In Strategies, speculative parallelism can be eliminated by the garbage collector when it is found to be unreferenced, whereas in `Par`, all speculative parallelism must be eventually executed.

The scheduler for Strategies is built-in to the runtime system, whereas the `Par` monad scheduler is written in Haskell, which enables us to easily use different scheduling policies.

Strategies includes a `parBuffer` combinator that can be used to evaluate elements of a lazy list in parallel. There is nothing directly equivalent in `Par`: all the parallelism must be complete when `runPar` returns, so we cannot return a lazy list from `runPar` with elements evaluated in parallel[8]. It is possible to program something equivalent to `parBuffer` inside the `Par` monad, however.

Finally, `Par` and Strategies can be safely combined. The GHC runtime system will prioritise `Par` computations over Strategies, because sparks are only evaluated when there is no other work.

9. Conclusion

We have presented our new parallel programming model, and demonstrated that it improves on the existing parallel programming models for Haskell in certain key ways, while not sacrificing performance on existing parallel benchmarks. In due course we hope that it is possible to go further and show that the extra control offered by the `Par` monad allows parallel algorithms to be tuned more effectively; we encountered one case of this in the pipeline example of Section 4.3.

It would be premature to claim that `par` and Strategies are redundant; indeed Strategies has advantages outlined in the previous section. Still, the difficulties with `par` indicate that it may be more appropriate as a mechanism for *automatic* parallelisation, than as a programmer-level tool.

References

[1] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11:598–632, October 1989.

[2] C. Baker-Finch, D. J. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 162–173, 2000.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, 1995.

[4] Z. Budimlic, M. Burke, V. Cave, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. The CnC programming model. *Journal of Scientific Programming*, 2010.

[5] V. Cavé, Z. Budimlić, and V. Sarkar. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 9:1–9:6. ACM, 2010.

[6] M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *DAMP'07 — Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2007.

[7] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9:313–323, May 1999.

[8] M. Cole. *Algorithmic Skeletons: structured management of parallel computation*. MIT Press, 1989.

[9] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: a heterogeneous parallel language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 37–44, 2007.

[10] M. I. Gordon et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, 2006.

[11] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 24–35, 1994.

[12] E. A. Lee and T. M. Parks. Readings in hardware/software co-design. chapter Dataflow process networks, pages 59–85. Kluwer Academic Publishers, 2002.

[13] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 189–199, 2007.

[14] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: better strategies for parallel Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 91–102, 2010.

[15] D. G. Murray and S. Hand. Scripting the cloud with Skywriting. In *HotCloud '10: Proceedings of the Second Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, USA, 2010. USENIX.

[16] R. A. Nikhil. *Implicit parallel programming in pH*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-644-0.

[17] J. Reppy, C. V. Russo, and Y. Xiao. Parallel concurrent ML. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 257–268, 2009.

[18] E. Scholz. A concurrency monad based on constructor primitives, or, being first-class is not enough. Technical report, Universitt Berlin, 1995.

[19] D. J. Spoonhower. *Scheduling deterministric parallel programs*. PhD thesis, Carnegie Mellon University, 2009.

[20] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + strategy = parallelism. 8:23–60, Jan. 1998.

[8] although we believe this may be possible with modifications to `runPar`