

# A Meta-Scheduler for the Par-Monad

## Composable Scheduling for the Heterogeneous Cloud

Adam Foltzer      Abhishek Kulkarni      Rebecca Swords  
Sajith Sasidharan      Eric Jiang      Ryan R. Newton  
Indiana University  
{afoltzer, adkulkar, raingram, sasasidh, erjiang, rrnewton} @indiana.edu

### Abstract

Modern parallel computing hardware demands increasingly specialized attention to the details of scheduling and load balancing across heterogeneous execution resources that may include GPU and cloud environments, in addition to traditional CPUs. Many existing solutions address the challenges of particular resources, but do so in isolation, and in general do not compose within larger systems. We propose a general, composable abstraction for execution resources, along with a continuation-based meta-scheduler that harnesses those resources in the context of a deterministic parallel programming library for Haskell. We demonstrate performance benefits of combined CPU/GPU scheduling over either alone, and of combined multithreaded/distributed scheduling over existing distributed programming approaches for Haskell.

**Categories and Subject Descriptors** D.3.2 [Concurrent, Distributed, and Parallel Languages]

**General Terms** Design, Languages, Performance

**Keywords** Work-stealing, Composability, Haskell, GPU

### 1. Introduction

Ideally, we seek parallel code that not only performs well, but for that performance to be *preserved under composition*. Alas, this is not always the case even in serial code: implementations of functions  $f$  and  $g$  may be well-optimized individually, but if  $f \circ g$  is run inside a recursive loop, the composition may, for example, exceed the machine’s instruction cache. Nevertheless, sequential composition is far easier to reason about than *parallel composition*, which is the topic of this paper.

Historically, there have been many reasons for parallel codes not to compose. First, many parallel programming models are *flat* rather than nested—i.e. a parallel computation may not contain another parallel computation [9, 28]. Moreover, many parallel codes take direct control of hardware or operating system resources, for example by using Pthreads directly. These programs, when

composed, result in *oversubscription*, as has famously troubled OpenMP<sup>1</sup> [3].

Yet the rising popularity of work-stealing schedulers (Section 3) is a step forward for composability, at least on symmetric multiprocessors (SMPs). By abstracting away explicit thread management these schedulers enable mutually ignorant parallel subprograms to coexist peacefully without oversubscription, and are now available for a wide range of different languages, including Haskell [26], C++ [2, 19, 32], Java [18], and Manticore [15], as well as many others. New problems arise, however, namely:

1. Multiple schedulers for the same language are difficult to coordinate effectively and in a principled manner (e.g. TBB / Cilk / TPL [2, 19, 32], or even Haskell’s sparks [26] and IO threads).
2. Non-CPU resources such as GPUs are competing for attention, and are not treated by existing schedulers.
3. Parallel work schedulers are themselves complex software artifacts (Section 3), typically non-modular [1], and difficult to extend.

The approach we take in this paper is to factor an existing work-stealing implementation into composable pieces. This addresses the complexity problem, but also leads the way to extensibility and interoperability—even beyond the CPU.

We describe a new system, Meta-Par<sup>2</sup>, which is an extensible implementation of the Par-monad library for Haskell [25]. The Par monad (Section 2) provides only basic parallel operations: *forking* control flow and communication through write-once synchronization variables called *IVars*. The extension mechanism we propose allows new variants of *fork* (e.g. to fork a computation on the GPU), but remains consistent with the semantics of the original Par monad, in particular retaining deterministic parallelism.

We present a set of these extensions, which we call *Resources*, that address challenges posed by current hardware: (1) dealing with larger and larger multi-socket (NUMA) SMPs, (2) programming GPUs, and (3) running on clusters of machines. Further, we observe that from the perspective of a CPU scheduler, these Resources have much in common; for example, handling asynchronous completion of work on a GPU or on another machine across the network presents largely the same problem. We argue that Resources provide a useful abstraction boundary for scheduler components, and show that they compose into more sophisticated schedulers using a simple associative binary operator.

Using a composed scheduler, a single program written for Meta-Par today can handle a variety of hardware that it might encounter

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP’12, September 9–15, 2012, Copenhagen, Denmark.  
Copyright © 2012 ACM 978-1-4503-1054-3/12/09...\$10.00

<sup>1</sup> OpenMP: A popular set of parallel extensions to the C language widely used in the high-performance computing community.

<sup>2</sup> <http://hackage.haskell.org/package/meta-par>

in the wild: for example, an ad-hoc collection of machines some of which have GPUs while others do not. Hence the *heterogeneous cloud*: mixed architectures within and between nodes.

The primary contributions of this paper are:

- A novel design for composable scheduler components (Section 4).
- A demonstration of how to cast certain aspects of scheduler design—aspects which go beyond multiplexing sources of work—using Resources. One example is adding *backoff* to a scheduler loop to prevent excessive busy-waiting (Section 4.4).
- An empirical evaluation of the Meta-Par scheduler(s), which includes evaluation of a number of recent pieces of common infrastructure in the Haskell ecosystem (network transports, CUDA libraries, and the like), as well as an in-depth case study of parallel comparison-based sorting implementations (Section 6.2).
- The first, to our knowledge, unified CPU/GPU work-stealing scheduler<sup>3</sup> (Section 4.5), along with an empirical demonstration that GPU-aware CPU-scheduling can outperform GPU-oblivious (Section 6.3). With further validation, this principle may generalize beyond our implementation and beyond Haskell.

These results are preliminary, but encouraging. Meta-Par can provide a foundation for future work applying functional programming to the heterogeneous hardware wilderness. The reader is encouraged to try the library, which is [hosted on github](#) and released via Haskell’s community package manager, Hackage: [here](#), [here](#), and [here](#).

## 2. The Par Monad(s)

Earlier work [25] introduced a Par monad with the following operations:

```
runPar :: Par a -> a
fork   :: Par () -> Par ()
new    :: Par (IVar a)
get    :: IVar a -> Par a
put_   :: IVar a -> a -> Par ()
```

A series of `fork` calls creates a **binary tree of threads**. We will call these *Par-threads*, to contrast them with Haskell’s IO threads (i.e. user-level threads) and OS threads. Par threads do not return values—hence the unit type in `Par ()`—instead they communicate only through IVars. IVars are first class, and an IVar can be read or written anywhere within the tree of Par-threads (albeit written only once).

By blocking to read an IVar, Par-threads can indeed be de-scheduled and resumed, thereby earning the moniker “thread”. Abstractly, IVars introduce synchronization constraints that transform the *tree* describing the structure of the parallel computation into a directed acyclic graph (DAG), as in Figure 1. DAGs are the standard abstraction for parallel computations used in most literature on scheduling [5, 7, 8, 35].

The simple primitives supported by Par can be used to build up combinators capturing common parallelism patterns, and one extremely simple and useful combinator is `spawn_`, which provides *futures*:

```
spawn_ :: Par a -> Par (IVar a)
spawn_ p = do i <- new
            fork (do x <- p; put_ i x)
            return i
```

<sup>3</sup> Though the idea has been discussed [17].

The original paper [25] has many more examples, and explains aspects of the design which we do not cover here, such as the distinction between `put_` and `put` (weak-head-normal-form strictness vs. full strictness), and the reasoning behind this design.

The `spawn_` abstraction is sufficient to define divide-and-conquer parallel algorithms by recursively creating a future for every sub-problem (a common idiom). We will use mergesort as a running example of this style. Below we define a mergesort on Vectors, a random-access, immutable array type commonly used in high-performance Haskell code.

```
parSort :: Vector Int -> Par (Vector Int)
parSort vec =
  if length vec <= seqThreshold
  then return (seqSort vec)
  else let n          = (length vec) `div` 2
         (left, right) = splitAt n vec
         in do leftIVar <- spawn_ (parSort left)
              right'  <- (parSort right)
              left'   <- get leftIVar
              parMerge left' right'
```

This function splits the vector to be sorted and uses `spawn_` on the left half, giving rise to a balanced binary tree of work to be run in parallel. A sequential sort is called once the length of the vector falls below a threshold.

### 2.1 Meta-Par Preliminary: Generalizing Par

In later sections we introduce variations on `fork` and `spawn_` that correspond to alternate flavors of child computations, such as those that might run on a GPU or over the network. Because a scheduler might have any combination of these capabilities, there are many possible schedulers. Therefore, each scheduler will have a distinct variant of the Par monad (a distinct type), so that a subcomputation that depends on, say, a GPU capability cannot encounter a runtime error because it is combined with a scheduler lacking the capability.

Thus we need to take a refactoring step that is common in Haskell library engineering<sup>4</sup>—introduce type classes to generalize over a collection of types that provide the same operations, in this case, multiple Par monads.

```
class Monad m => ParFuture future m
  | m -> future where
  spawn_ :: m a -> m (future a)
  get    :: future a -> m a

class ParFuture ivar m => ParIVar ivar m
  | m -> ivar where
  fork :: m () -> m ()
  new  :: m (ivar a)
  put_ :: ivar a -> a -> m ()
```

In the above classes we take an opportunity to separate levels of Par functionality. A given Par implementation may support *just futures*<sup>5</sup> (ParFuture class), or may support futures *and* IVars (ParIVar class). The distinction in levels of capability will become more important as we introduce capabilities such as `gpuSpawn` and `longSpawn` (and classes `ParGPU`, `ParDist`) in Sections 4.5 and 4.5.2.

In the above class definitions, the type variable ‘m’ represents the type of a specific Par monad that satisfies the interface (i.e. an *instance*). Some of the complexity above is specific to Haskell and may safely be ignored for the reader of this paper. Namely,

<sup>4</sup> A common example being the `PrimMonad` type class generalizing over IO and ST—true external side effects and localized, dischargeable ones.

<sup>5</sup> Indeed, we have a scheduler that uses *sparks* [26] and supports only futures. This allows us to compare the efficiency of our scheduling primitives to those built in to the GHC runtime, using the former if desired.

the `ParFuture` and `ParIVar` classes are *multi-parameter* type classes, both parameterized by a type variable `ivar` as well as `m`. This is necessary because two `Par` monads may require different representations for their synchronization variables. Finally, because the type for `ivar` is determined by the choice of `Par` monad, the above includes another advanced feature of GHC type classes: a *functional dependency*,  $m \rightarrow ivar$ . We do not simplify these classes for the purpose of presentation, because they correspond exactly to those used in [the released code](#).

The writer of a reusable library should always use the generic functions, and never commit to a concrete `Par` monad. The final application is then free to decide which concrete implementation—and therefore which heterogeneous execution capabilities—to use. For the remainder of the paper, let us assume that all `Par` implementations reside in their own distinct modules, `Control.Monad.Par.Foo`, each providing a *concrete* type constructor named `Par`, as well as instances for the appropriate generic operations. These are the *schedulers* (plural) in our system, whereas Meta-Par itself is a *meta-scheduler*—not touched directly by users, but instantiated to create concrete schedulers. For readability, we will informally write concrete type signatures, `Par a`, (referring to any valid concrete `Par` monad) rather than the more generic `ParFuture iv p => p a`.

### 3. Work-Stealing Schedulers

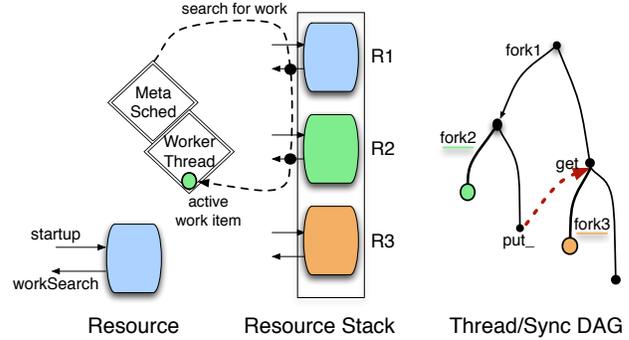
In work-stealing schedulers, each worker maintains a private work pool, synchronizing with other workers *only* when local work is exhausted (the *parsimony* property [34]). Thus the burden of synchronizing and load-balancing falls on idle nodes. Like any parallel scheduler, work-stealing schedulers map work items (e.g. forked `Par`-threads) onto  $P$  workers; workers are most often OS threads with a one-to-one correspondence to processor cores.

As a work-stealing algorithm, the original implementation of the `Par` monad [25] is rather standard and even simple. Yet schedulers that “grow up”—for example TBB, Cilk, or the GHC runtime—become very complex, dealing with concerns such as the following:

- Idling behavior to prevent wasted CPU cycles in tight work-stealing loops (“busy waiting”).
- Managing contention of shared data structures (backoff, etc.).
- Interacting with unpredictable user programs that can call into the scheduler (e.g. call `runPar`) from different hardware threads or in a nested manner.
- Multiplexing multiple sources of work.

Alas, in spite of this complexity, such schedulers typically have monolithic, non-modular implementations [1, 19, 32]. Regarding work-source multiplexing in particular: a typical work-stealing scheduler is described in pseudocode as an ordered series of checks against possible sources of work. For example, in the widely-used Threading Building Blocks (TBB) package, the reference manual [4], Section 12.1, includes the following description of the task-scheduling algorithm:

- After completing a task `t`, a thread chooses its next task according to the first applicable rule below:
1. The task returned by `t.execute()`
  2. The successor of `t` if `t` was its last completed predecessor.
  3. A task popped from the end of the thread’s own deque.
  4. A task with affinity for the thread.
  5. A task popped from approximately the beginning of the shared queue.
  6. A task popped from the beginning of another randomly chosen thread’s deque.



**Figure 1.** [Left] Meta-scheduling: scan a stack of work sources, always starting at the top. Work sources are heterogeneous, but all work is retrieved as unit computations in the `Par` monad (i.e. `Par ()`). [Right] Work DAGs formed by `forks` and `gets`; the circles at the leaves represent tasks bound for the resource with matching color.

Six possible sources of work! And that is only for CPU scheduling. Rather than *committing* to a list like the above and hardcoding it into the scheduler (the state of the art today), we construct schedulers that are composed of reusable components. For example, a rough description of a distributed CPU/GPU scheduler may look like the following:

1. Steal from CPU-local deque (try  $N$  times) *else*
2. Steal-back from GPU *else*
3. Steal from network *else*
4. Goto step 1

This resembles a *stack* of resources. In fact, the purpose of this paper is to demonstrate that scheduler composition *need only be a simple associative binary operator*. The familiar `mappend` operation from Haskell’s `Monoid` type class then suffices to combine Resources into compound [stacks of] Resources.

### 4. Meta-Scheduling: The Resource Stack

The scheduler for Meta-Par is parameterized by a *stack of heterogeneous execution resources*, each of which may serve as a source of work. All workers participating in a Meta-Par execution (on all threads and all machines) run a scheduling loop that interacts with the resource stack. Resource stacks are built using `mappend`, where (`a ‘mappend’ b`) is a stack with `a` on *top* and `b` on the *bottom*. Below, the type `Resource` is used for both singular and composed Resources. We will use “resource stack” informally to refer to complete, composed Resources.

The division of labor in our design is between *schedulers*, *Resources*, and the Meta-Par infrastructure (meta-scheduler).

#### First, the meta-scheduler itself:

- Creates worker threads, each with a work-stealing deque.
- Detects nested invocations of `runPar` and avoids re-initialization of the Resource (i.e. oversubscription)<sup>6</sup>.
- Provides concrete `Par` and `IVar` types that all Meta-Par-based schedulers use and repackage.

<sup>6</sup>This ultimately requires global mutable state via the well-known `unsafePerformIO` with `NOINLINE` pragma hack, both in Meta-Par and for some supplementary Resource data structures. See Figure 2

- This `Par` provides blocking `get` operations via a continuation monad, using continuations to suspend `Par`-threads in the style of Haynes, Friedman, and Wand [16].

Further, each `Resource` may introduce:

- Additional (internal) data structures for storing work, above and beyond the per-worker thread dequeues. These might contain work for an external device of a different type than `Par ()`.
- One or more *fork*-like operations appropriate to the resource. These push work into the per-resource data structures.

Finally, each scheduler contains:

- A new `Par` type (a *newtype* as described in Section 2.1),
- a corresponding `runPar`, and
- a composed `Resource` [stack]

Thus a scheduler is a mere mashup of `Resources`, re-exporting components of `Meta-Par` and of constituent `Resources`. In fact, schedulers can be created on demand with a few lines of code<sup>7</sup>. Typically, each scheduler and each `Resource` reside their own module. An example module implementing a `Resource` is shown in Figure 2, and an example module implementing a scheduler is shown in Figure 3.

Because each `Resource` manages its own data structures, `Meta-Par` is not strictly *just for work-stealing*. For example, a `Resource` could choose to ignore `Meta-Par`'s `spawn` in favor of its own operator with work-sharing semantics. Indeed, even the built-in work-stealing behavior can be cast as a stand-alone `Resource`; however we choose to include it in the core of the system in order to keep the `Meta-Par` interface simpler.

#### 4.1 Resource Internals

A `Resource` presents an interface composed of two callbacks: a startup callback, and a work-searching callback.

```
type Startup    = Resource → Vector WorkerState → IO ()
type WorkSearch = Int → Vector WorkerState →
  IO (Maybe (Par ()))
```

```
data Resource = Resource {
  startup    :: Startup,
  workSearch :: WorkSearch
}
```

The `startup` callback is responsible for performing any work necessary to prepare a `Resource`, such as spawning worker threads for SMP scheduling, or opening network connections for distributed coordination. A global barrier ensures that no work commences until each `Resource` in the stack has completed initialization. The `Resource` argument to `startup` ties the knot to make the final composed `Resource` available when initializing any of its component `Resources`. `WorkerState` structures store each worker's work-stealing deque, along with certain shared information such as the random number generator used for randomized work stealing.

Each worker may have a single *active Par-thread* currently executing. When that `Par-thread` is finished or blocks on an `IVar`, the worker first tries to pop from the top of its work-stealing deque,

<sup>7</sup> However, there is one error prone aspect of scheduler composition. The *newtype* `Par` may use *newtype-deriving* to derive capabilities such as `ParGPU` corresponding to *only* the resources actually composed. A mismatch here could result in a runtime error when a computation is run on an incompatible scheduler. An alternative would be constructing resource stacks explicitly at the type level (like a monad transformer stack), but this comes with significant complications, including our reluctance to introduce *lift* operations.

and if no work is found, invokes `workSearch`. The arguments to `workSearch` provide the searcher's ID (just an `Int`) along with the global `WorkerState` vector. The former can be used to look up the local `WorkerState` structure in the latter. The worker expects the `workSearch` to respond either with a unit of work (`Just work`), or with `Nothing`.

`Resources`, combined with `mappend`, form a *non-commutative monoid* so we can compose them using the `Monoid` type class:<sup>8</sup>

```
instance Monoid Startup
instance Monoid WorkSearch
instance Monoid Resource
```

The `Startup` instance is straightforward, where the empty action does nothing, and composing two startups means to run them in sequence with the same arguments. The interesting instance is for `WorkSearch`, which must be composed so that the work-finding attempt runs the second `workSearch` only when the first `workSearch` returns `Nothing`.

```
instance Monoid WorkSearch where
  mempty = λ_ _ → return Nothing
  mappend ws1 ws2 =
    λwid stateVec → do
      mwork ← ws1 wid stateVec
      case mwork of
        Nothing → ws2 wid stateVec
        _       → return mwork
```

In order to satisfy the axioms of a monoid, the empty `Resource` `mempty` does nothing—no `Meta-Par` workers are ever spawned by its `Startup`, no work is ever found by its `WorkSearch`, and so no work can be computed if it is the only `Resource`. `Meta-Par` leaves it to non-empty implementations of `Resources` to decide how many and on which CPUs to spawn worker threads. The `Meta-Par` module itself, absent any `Resources`, provides very little. It simply exposes primitives for spawning workers (handling exceptions, waiting for the `startup` barrier, logging debugging info) and running entire `Par` computations with a particular `Resource` configuration:

```
spawnWorkerOnCPU :: Resource → Int → IO ()
runMetaParIO    :: Resource → Par a → IO a
```

`Meta-Par` commits to a specific concrete `Par` type (`Control.Monad.Par.Meta.Par`) for its internal implementation and for the construction of new `Resources` and composed schedulers. This `Par` type allows arbitrary computation via a `MonadIO` instance, which would put the `Par`-monad determinism guarantee at risk if exposed to the end user. Instead, the privileged `Meta.Par` is wrapped by the schedulers in *newtype* `Par` types that provide only appropriate instances. For example, the "SMP+GPU" scheduler exports a `Par` monad that is an instance of `ParFuture`, `ParIVar`, and `ParGPU`, but *not* an instance of unsafe classes like `MonadIO`, or even classes for other `Meta-Par` `Resources` (e.g., `ParDist`) not included in that particular scheduler.

#### 4.2 CPU Scheduling: Single-threaded and SMP

To show that `Meta-Par` subsumes the previous implementation of `Par`-monad, we implement `Resources` for serial execution and SMP. In section 6.1, we compare the performance against previous results.

The single-threaded `Resource` is the minimal `Resource` required for the meta scheduler to execute work. Its `startup` creates a single worker on the current CPU, and its `workSearch` always returns `Nothing`, as the `Resource` has nowhere to look for more work.

<sup>8</sup> For clarity, we use `type` here to present `Startup` and `WorkSearch`, but our implementation uses *newtype* to avoid type synonym instances.

```

module Control.Monad.Par.Meta.Resources.GPU where
...
{-# NOINLINE gpuQueue #-}
gpuQueue :: ConcurrentQueue (Par (), IO ())
gpuQueue = unsafePerformIO newConcurrentQueue

{-# NOINLINE resultQueue #-}
resultQueue :: ConcurrentQueue (Par ())
resultQueue = unsafePerformIO newConcurrentQueue

class ParFuture ivar m => ParGPU ivar m
  | m -> ivar where
  gpuSpawn :: (Arrays a) => Acc a -> m (ivar a)

instance ParGPU IVar Par where
  gpuSpawn :: (Arrays a) => Acc a -> Par (IVar a)
  gpuSpawn comp = do
    iv <- new
    let wrapCPU = put_ iv (AccCPU.run comp)
        wrapGPU = do
            ans <- evaluate (AccGPU.run comp)
            push resultQueue (put_ iv ans)
        liftIO (push gpuQueue (wrapCPU, wrapGPU))
    return iv

gpuProxy :: IO ()
gpuProxy = do
  -- block until work is available
  (_, work) <- pop gpuQueue
  -- run the work and loop
  work >> gpuProxy

mkResource :: Resource
mkResource = Resource {
  startup = \ _ _ -> forkOS gpuProxy
  workSearch = \ _ _ -> do
    mfinished <- tryPop resultQueue
    case mfinished of
      Just finished -> return (Just finished)
      Nothing -> do
        mwork <- tryPop gpuQueue
        fst 'fmap' mwork
}

```

**Figure 2.** An Accelerate-based GPU Resource implementation module.

```

{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Control.Monad.Par.Meta.SMPGPU (Par, runPar) where
...

resource = SMP.mkResource 'mappend' GPU.mkResource

newtype Par a = Par (Meta.Par a)
  deriving (Monad, ParFuture Meta.IVar,
           ParIVar Meta.IVar, ParGPU Meta.IVar, ...)

runPar :: Par a -> a
runPar (Par work) = Meta.runMetaPar resource work

```

**Figure 3.** A scheduler implementation module combining two Resources.

```

singleThreadStartup resource _ = do
  cpu <- currentCPU
  spawnWorkerOnCPU resource cpu

```

```

singleThreadSearch _ _ = return Nothing

```

The SMP Resource offers the same capability as the original implementation of the work-stealing Par-monad scheduler. Its `startup` spawns a Meta-Par worker for each CPU available to the Haskell runtime system. Its `workSearch` selects a stealee worker at random and attempts to pop from the stealee’s work queue, looping a fixed number of times if the stealee has no work to steal.

```

smpSearch myid stateVec =
  let WorkerState {rng} = stateVec ! myid
      getNext :: IO Int
      getNext = randomRange (0, maxCPU) rng
      loop :: Int -> Int -> IO (Maybe (Par ()))
      loop 0 _ = return Nothing
      loop n i | i == myid =
          loop (n-1) =<< getNext
      loop n i =
          let WorkerState {workpool} = stateVec ! i
              in do mtask <- tryPopBottom workpool
                  case mtask of
                    Nothing -> loop (n-1) =<< getNext
                    _ -> return mtask
          in loop maxTries =<< getNext

```

### 4.3 CPU Scheduling: NUMA

Modern multi-socket, multi-core machines employ a shared memory abstraction, but exhibit Non-Uniform Memory Access (NUMA) costs. This means that it is significantly cheaper to access some memory addresses than others from a given socket, or *NUMA node*. Unfortunately, even if the memory allocation subsystem correctly allocates into node-local memory, work-stealing can disrupt locality by moving work which depends on that memory to a different node. Thus NUMA provides an incentive for work-stealing algorithms to prefer stealing work from cores within the same NUMA node. Although topology-aware schedulers have been proposed [6], most of the widely deployed work-stealing schedulers [2, 18, 19, 32] are oblivious to such topology issues.

In the Meta-Par SMP setting, workers first try to pop work from their own queues before making more costly attempts to steal work from other CPUs. In the NUMA case, we support analogous behavior: a worker first attempts to steal work from CPUs in its own NUMA node, and only moves on to attempt more costly inter-node steals when no local work is available.

Our NUMA-aware Meta-Par implementation notably is a *resource transformer*, rather than a regular Resource, and demonstrates the *first-class* nature of Resources. Instead of duplicating the work-stealing functionality of the SMP Resource, the NUMA Resource is composed of a subordinate SMP Resource for each NUMA node in the machine. Unlike the SMPs in Section 4.2, which may randomly steal from all CPUs, these subordinate SMP `workSearches` are restricted to steal *only* from CPUs in their respective node. With these subordinate Resources in place, the NUMA `workSearch` first delegates to the SMP `workSearch` of the calling worker’s local node. If no work is found locally, it then enters a loop analogous to the SMP loop that calls all nodes’ SMP `workSearches` at random.

### 4.4 Another Resource Transformer: Adding Backoff

An essential and pervasive aspect of practical schedulers is the ability to detect when little work is available for computation and back off from busy-waiting in the scheduler loop. Detecting a lack of available work may seem like a primitive capability that must be

built into the core implementation of the scheduler loop, but we can in fact implement backoff for arbitrary Meta-Par Resource stacks as a Resource transformer.

The backoff `workSearch` does not alter the semantics of the `workSearch` it transforms. Instead, it calls the inner `workSearch`, leaving both the arguments and the return value unchanged. It does, however, observe the number of consecutive times that a `workSearch` call returns `Nothing` for each Meta-Par thread (a counter kept in the `WorkerState` structure). When little work is available across the scheduler, these counts increase, and the back-off Resource responds by calling a thread sleep primitive with a duration that increases exponentially with the count. When a `workSearch` again returns work for a thread, the count is reset, and the scheduling loop resumes without interruption.

#### 4.5 Heterogeneous Resources - Blocking on foreign work

A key motivation for composable scheduling is to handle different mixes of heterogeneous resources outside of the CPU(s). Working with a non-CPU resource requires launching foreign tasks and scheduling around **blocking operations** that wait on foreign results (or arranging to poll for completion). Existing CPU work-stealing schedulers have varying degrees of awareness of blocking operations. Common schedulers for C++ (e.g. Cilk or TBB) are *completely oblivious* to all blocking operations ranging from blocking in-memory data structures (e.g. with locks) to IO system calls. Obliviousness means that while the scheduler attempts to maintain  $P$  worker threads for  $P$  processors, fewer than  $P$  may be active at a given time.

It is often suggested to use Haskell’s IO threads directly to implement Par-threads, as there is widespread satisfaction with how *lightweight* they are. This would appear attractive, as the Glasgow Haskell Compiler (GHC) implements blocking operations at the Haskell thread layer (IO threads) using non-blocking system calls via the GHC event manager [30]. IO threads are even appropriately preempted when blocking on in-memory data structures, namely *MVars*. Unfortunately, GHC’s IO threads are not lightweight *enough* for fine-grained parallelism. They still require allocating large contiguous stacks and Par schedulers based on them cannot compete [25, 29].

Ultimately, the lightest-weight approaches for pausing and resuming computations are based on *continuation passing style* (CPS). The relationship between CPS and coroutines or threading is old and well known [16], but has increasingly been applied for concurrency and parallelism [11, 20, 21, 33, 37]. In the Par-monad, CPS is already a necessity for efficient blocking on IVars (Meta-Par uses the continuation-monad-transformer, `ContT`). Using continuations, we gain the ability to schedule around foreign work—e.g. to keep the CPU occupied while the GPU computes—for free.

##### 4.5.1 Heterogeneous Resource 1: GPU

Several embedded domain-specific languages (EDSLs) have been proposed to enable GPU programming from within Haskell [9, 24, 36]. In addition, raw bindings to the CUDA and OpenCL are available [13, 27]. Accelerate and other EDSLs typically introduce new types (e.g. `Acc`) for GPU computations as well as a `run` function—much like `Par`, in fact.

In Meta-Par, we provide built-in support for launching Accelerate computations from Par computations:

```
gpuSpawn :: Arrays a => Acc a -> Par (IVar a)

-- Asynchronous Acc computation, filling IVar when done:
do gpuSpawn (Acc.fold (+) 0 (Acc.zipWith ...
```

You might well ask why `gpuSpawn` is needed, given that both `runPar` and Accelerate’s `run` are *pure* and should therefore be freely

composable. Indeed, they are, semantically, but as discussed in the previous section, we do not want CPU threads to remain idle while waiting on GPU computations. Nor can this be delegated to Haskell’s foreign function interface itself, which quite reasonably assumes that a foreign call does actual work on the CPU from which it is invoked!

To avoid worker idleness, we follow the approach of Li & Zdancewic [21], making blocking resource calls only on proxy threads which stand in as an abstraction of the blocking resource. Par-monad workers communicate with these proxies via channels; when a worker would otherwise make a blocking call, it instead places the corresponding IO callback in the appropriate channel, and returns a new `IVar` which will be filled only when the operation is complete. (As usual, reading the `IVar` prematurely will save the current continuation and free the worker to execute other Par work.) The proxy runs in a loop, popping callbacks from its channel and executing them. It writes the results to a channel read by the Par-monad workers, who call `put` to fill the `IVar`, waking its waiting continuations with the result value.

As shown in Figure 2, the Accelerate Resource’s `workSearch` first checks the queue of results returned by the proxy, and if none are found, attempts to steal unexecuted Accelerate work for execution on a CPU backend in case the GPU is saturated.

##### 4.5.2 Heterogeneous Resource 2: Distributed Execution

We expose remote execution through another variant of `spawn`, called `longSpawn`, and follow CloudHaskell’s conventions [14] for remote procedure calls and serialization:

```
longSpawn :: Serializable a
           => Closure (Par a) -> Par (IVar a)
```

In the type of `longSpawn`, the `Serializable` constraint and `Closure` type constructor denote that a given unit of Par work and its return type can be transported over the network. A `Serializable` value must have a runtime type representation via the `Data.Typeable` class as well as serialization methods, both of which can be generically derived by GHC [22]<sup>9</sup>.

To employ `longSpawn`, the programmer uses a Template Haskell shorthand, making distributed calls only slightly more verbose than their parallel counterparts:

```
parVer = spawn_ (bar baz)
distVer = longSpawn $(mkClosure bar) baz
```

In our implementation the `Closure` values contain both a local version (a plain closure in memory) and a serializable remote version of the computation. As with other resources in our work-stealing environment, `longSpawned` work is not *guaranteed* to happen remotely, it merely exposes that possibility.

One complication is that the above `longSpawn` requires that the user must further *register* `bar` with the remote execution environment<sup>10</sup>:

```
bar :: Int -> Par Int
bar x = ...
remotable ['bar]
```

A bigger limitation is that functions like `bar` above are currently restricted to be *monomorphic*, which makes it very difficult to

<sup>9</sup>Generic serialization routines, however, are frequently much slower than routines specialized for a type, so we provide efficient serialization routines for commonly-used types like `Data.Vector`

<sup>10</sup>Specifically, `remotable` is a macro that creates additional top level bindings with mangled names. The `mkClosure` and `mkClosureRec` macros turn an ordinary identifier into its `Closure` equivalent. `Remotable` functions must be monomorphic, have `Serializable` arguments, return either a pure or a Par value, and only have free variables defined at the top level.

define higher-order combinators like *parMap* or *parFold* which are the bread and butter of the original Par-monad library. We share the hope of CloudHaskell’s authors that native support from the GHC compiler will improve this situation, and, if other volunteers are not forthcoming, plan to implement such native support ourselves in the future.

Returning to our running example, a parallel merge sort could be augmented with *both* distributed and GPU execution with the following two-line changes (assuming that `gpuSort` is a separate sorting procedure in the Accelerate EDSL):

```
parSort :: Vector Int → Par (Vector Int)
parSort vec =
  if length vec ≤ gpuThreshold
  then gpuSpawn (gpuSort vec) >>= get
  else let n = (length vec) `div` 2
        (l, r) = splitAt n vec
        in do lf ← longSpawn ($mkClosureRec 'parSort) l)
            r' ← parSort r
            l' ← get lf
            parMerge l' r'
```

It may appear as though work never reaches the CPU. Recall, however, that `gpuSort` is effectively a *hint*; the work may end up on either the GPU or CPU. Ultimately, it will be possible for a `gpuSort` based on Accelerate default to an efficient (e.g. OpenCL) implementation when the computation ends up on the CPU<sup>11</sup>.

## 5. Semantics

The operational semantics of Par [25] remain nearly unchanged by Meta-Par. The amended rules appear in Appendix A. The only minor extension is that there is more than one *fork* (e.g. *fork*<sub>1</sub>, *fork*<sub>2</sub>, ...) in the grammar, corresponding to the resources *R*<sub>1</sub>, *R*<sub>2</sub>, ... Fortunately, this changes nothing important. The semantics do not need to model the Meta-Par scheduling algorithm (or any other Par scheduling algorithm). Rather, execution proceeds inside a parallel evaluation structure in which any valid redex can be reduced at any time. Because these loose semantics are sufficient to guarantee both determinism and deadlock/livelock freedom it is therefore safe for Meta-Par to use an *arbitrary* strategy for selecting between work-sources.

### Semantics of Scheduler Composition

To ensure correctness, at minimum we need a single guarantee from Resources: they must be *lossless*—everything pushed by a fork eventually is produced by a `searchWork`. Subsequently the following properties will hold:

- monoid laws: scheduler composition is an associative operation with an identity
- commuting Resources in a stack will preserve correctness but may incur asymptotic differences in performance

These hold for any scheduler which is a purely monoidal (`mappend`) composition of Resources as in Figure 2.

Like other work-stealing schedulers, Meta-Par is designed for a scenario of finite work; infinite work introduces the possibility of starvation (i.e. because other workers are busy, given piece of work may never execute). Because `runPar` is used to schedule *pure* computation, fairness of scheduling Par-threads is semantically unimportant—there are no observable effects other than the final value. (And the entire `runPar` completes before the value returned is in weak head normal form.)

<sup>11</sup> Unfortunately, at the time of writing the `accelerate` distribution includes only an interpreter for CPU evaluation of `Acc` expressions.

## Time and Space Usage

A precise analysis of scheduler time and space usage is desirable, but is confounded both by (1) Meta-Par being parameterized by arbitrary Resources and (2) by intrinsic difficulty with the powerful class of programming models that include user directed synchronizations (i.e. reading IVars) [5, 7, 34].

Nevertheless, while Meta-Par targets a general model, it can *preserve* good behavior of schedulers when certain conditions are met. For example, consider the class of programs that are *strictly phased*. That is, given a resource stack *R*<sub>1</sub> ‘`mappend`’ *R*<sub>2</sub> ‘`mappend`’ *R*<sub>3</sub>, *fork*<sub>2</sub> computations may call *fork*<sub>2</sub>, or *fork*<sub>1</sub>, but not *fork*<sub>3</sub>. In a strictly phased program, all paths down the binary *fork*-tree proceed monotonically from deeper to shallower Resources (i.e. *fork*<sub>3</sub> to *fork*<sub>1</sub>). In general, this represents good practice; in a divide-and-conquer algorithm, the programmer should call `longSpawn` before `spawn`.

In such a scenario, as long as the resources themselves manage work in a LIFO manner (a second requirement) the composed Resource stack behaves the same way *as a single, extended stack*. We conjecture that existing analyses would apply in this scenario [34], but do not treat the topic further here.

## 6. Evaluation

In this section, we analyze the performance of Meta-Par schedulers in several heterogeneous execution environments. First, we compare the performance of the Meta-Par SMP scheduler to the previously published Par-monad scheduler in both a multicore desktop and many-core server environment. Then, we examine the performance of parallel merge sort on a multicore workstation with a GPU. Finally, we compare Meta-Par with a distributed Resource to other distributed Haskell implementations [14, 23].

### 6.1 Traditional Par-Monad CPU Benchmarks

Our goal in this section is twofold:

- to compare the Meta-Par scheduler to the previously published scheduler [25], which we will call *Trace* (being based on the lazy trace techniques of [11, 21]), and
- to analyze the extent to which our results are contingent upon GHC versions and runtime system parameters, mainly those affecting garbage collection.

The original work [25] studied a set of benchmarks on a 24-core Intel E7450. The benchmarks are all standard algorithms, so we refer the reader to the abbreviated description in that paper, rather than describing their purpose here. In this section we give results for `blackscholes`, `nbody`, `mandel`, `sumeuler`, and `matmult`, while omitting `queens`, `coins`, and `minimax` (which have fallen into disrepair).

We show our latest scaling numbers in Figure 4. These compare the Meta-Par scheduler against the original Par-monad scheduler which does not suffer the overhead of indirection through Resource stacks. The scaling results in Figure 4 come from our largest server platform: four 8-core Intel Xeon E7-4830 (Westmere) processors running at 2.13GHz. Hyper-Threading was disabled via the machine’s BIOS for a total of 32 cores. The total memory was 64GB divided into a NUMA configuration of 16GB per processor. The operating system was the 64-bit version of Red Hat Enterprise Linux Server 6.2. SMP shows better scaling on `blackscholes`, but compromised performance on `MatMult` and `mandel`. Overall, we consider the performance of SMP close enough to Trace.

### Clusterbench

As with other garbage collected languages (e.g. Java) there are many runtime system parameters that affect GHC memory man-

agement and can have a large effect on performance<sup>12</sup>. These are the relevant runtime system options:

- `-A` size of allocation area (first generation)
- `-H` suggested heap size
- `-qa` affinity: pin Haskell OS threads to physical CPUs
- `-qg` enable parallel garbage collection (GC) for one or more generations
- `-qb` control which, if any, generations use a load-balancing algorithm in their parallel GC

Of course, varying all of these parameters results in a combinatorial explosion. Thus most performance evaluations for Haskell experiment by hand, find what seems like a reasonable compromise, and stick with that configuration.

To increase confidence in our results we wanted a more systematic approach. We decided to exhaustively explore a reasonable range of settings for the above parameters (e.g. `-A` between 256K and 2M), resulting in 360 configurations. But when running each benchmark for 5-9 trials (and varying number of threads and scheduler implementation) each configuration requires between 324 and 1000 individual program runs and takes between 10 minutes and two hours. Thus exploring 360 configurations can take up to thirty days on one machine. To address this problem we created a program we call *clusterbench* that can run either in a dedicated cluster or search among a set of identically configured workstations for idle machines to farm out benchmarking work.

We used a collection of twenty desktop workstations, each with an Intel Core i5-2400 (Westmere) running at 3.10 GHz with 4GB of memory under 64-bit Red Hat Enterprise Linux Workstation 6.2. All of these workstations together were able to complete the 360 benchmark configurations in a few days. The results are summarized in Table 5 and were pleasantly surprising. In spite of some past problems with excessive variance in performance in response to GC parameters (in the GHC 6.12 era [29]), under GHC 7.4.1 we see remarkably little impact relative to our default settings, except insofar as high settings of `-H` compromise performance significantly.

## 6.2 Case Study: Sorting

We analyzed parallel merge sort performance on our “GPU workstation” platform, which has one quad-core Intel W3530 (Nehalem) running at 2.80GHz with 12GB and an NVIDIA Quadro 5000 GPU under 64-bit Red Hat Enterprise Linux Workstation 6.2.

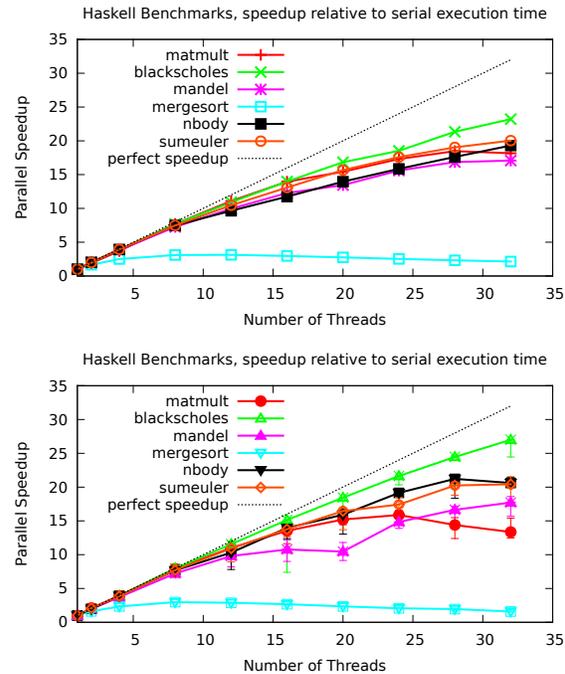
Our first comparison examines performance of three task-parallel (but not vectorized) CPU-only configurations, Figure 6. Each configuration sorts `Data.Vector.Storable` vectors of 32-bit integers, and uses a parallel Haskell merge sort until falling below a fixed threshold, when one of three routines is called:

- **Haskell:** A sequential Haskell merge sort [12].
- **Cilk:** A *parallel C* merge sort using the Cilk parallel runtime.
- **C:** A sequential C quick sort called through the FFI. This is the same sequential code as the Cilk algorithm employs at the leaves of its parallel computation.

The pure Haskell sort does quite well, considering its limitation that while it is in-place at the sequential leaves (using the `ST` monad), it must copy the arrays during the parallel phase of the algorithm. In general, `ST` and `Par` are effects that do not compose.<sup>13</sup>

<sup>12</sup> See Table 2 in [29].

<sup>13</sup> However, in the future we are interested in exploring mechanisms to guarantee that slices of mutable arrays are passed *linearly* to only one branch of a `fork` or the other, but not both.

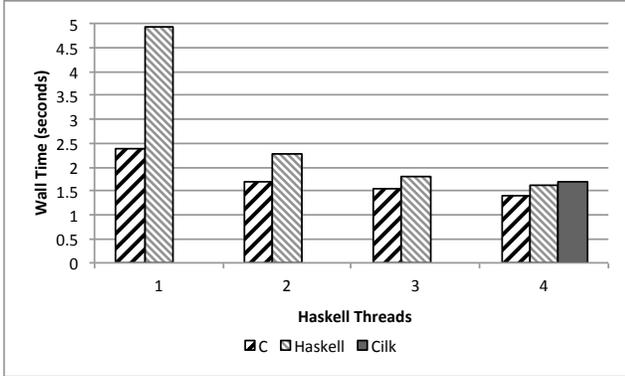


**Figure 4.** Scaling behavior of original `Par` scheduler (top) vs. Meta-`Par` SMP scheduler on 32-core server platform. Error bars represent minimum and maximum times over five trials. Mergesort is memory bound beyond four cores. (The pure Cilk version likewise has a maximum speedup less than five.)

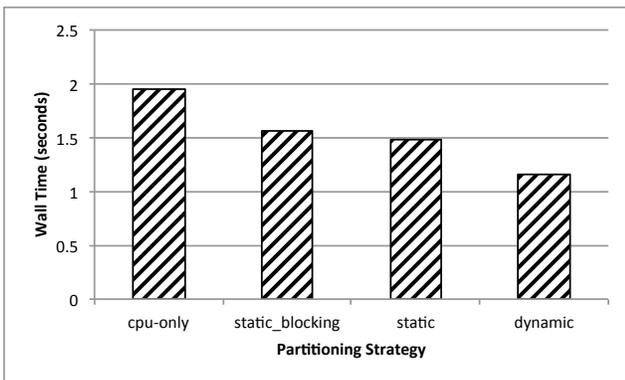
threads	1	2	3	4
-	0.00	0.00	0.00	0.00
<code>-qa</code>	-0.05	-0.43	-0.81	-0.97
<code>-qb</code>	0.00	0.00	0.00	0.00
<code>-qb0</code>	-0.16	-0.35	0.14	0.75
<code>-qb1</code>	-0.29	-0.52	-0.21	0.31
<code>-qg</code>	-0.06	-3.34	-7.09	-10.37
<code>-qg0</code>	0.00	0.00	0.00	0.00
<code>-qg1</code>	-0.09	-3.51	-6.96	-9.54
-	0.00	0.00	0.00	0.00
<code>-H128M</code>	5.08	-6.06	-4.91	-5.08
<code>-H256M</code>	6.49	2.22	-3.88	-7.51
<code>-H512M</code>	2.22	-5.85	-9.02	-14.42
<code>-H1G</code>	-8.39	-24.76	-34.51	-42.10
<code>-A256K</code>	-0.07	-0.11	-0.16	-0.17
<code>-A512K</code>	0.00	0.00	0.00	0.00
<code>-A1M</code>	-0.05	-0.08	-0.09	-0.09
<code>-A2M</code>	-0.07	-0.11	-0.14	-0.14

(Speedup/slowdown in percentages.)

**Figure 5.** Effect of runtime system parameters on performance variation. Each number represents the percentage *faster* or *slower* that the benchmark suite ran given the particular setting of that parameter, and relative to the *default* setting of the parameter (i.e. the 0.00 row). Each percentage represents a geometric mean over all parameter settings *other* than the selected one.



**Figure 6.** Median elapsed time (over 9 trials) to sort a random permutation of  $2^{24}$  32-bit integers on the CPU. Parallel phase of the algorithm in Haskell in all cases, below a threshold of 4096 the algorithm switches to either sequential C, sequential Haskell, or Cilk.



**Figure 7.** Median elapsed time (over 9 trials) to sort a random permutation of  $2^{24}$  32-bit integers with 4 threads. For all cases, the CPU threshold was 4096 elements. The GPU threshold was  $2^{22}$  elements for static partitioning, and between  $2^{14}$  and  $2^{22}$  for dynamic partitioning.

We include the parallel Cilk routine for two reasons. First, it provides an objective performance comparison: `cilksort.c`<sup>14</sup>. It is not a *world-class* comparison-based CPU sorting algorithm—that would require a much more sophisticated algorithm including vectorization and a multi-way merge sort at sizes larger than the cache [10]—but it is an extremely good sort, especially for its level of complexity. Second, by calling between Haskell and Cilk, two *unintegrated* schedulers, we explore a concrete example of oversubscription. When running on a 4-core machine, the Cilk runtime dutifully spawns 4 worker threads, oblivious to the 4 Meta-Par workers already contending for CPU resources. The overhead introduced by this contention makes overall performance worse than both pure Haskell and hybrid Haskell/C merge sorts.

### 6.3 GPU Sorting

Next, to demonstrate the support of Meta-Par for heterogeneous resources, we examine merge sort configurations that add a GPU Resource in addition to the parallel CPU sort of the last section. (We select the version that bottoms out to a sequential C sort routine.)

<sup>14</sup><http://bit.ly/xrgc8P>

Initially, we selected a sort routine based on Accelerate (and coupled with an Accelerate-supporting Meta-Par Resource). However, due to temporary stability and performance problems<sup>15</sup> we are instead making direct use of the CUDA SDK through the `cuda` package. Using this we call a GPU routine that is an adaptation of the NVIDIA CUDA SDK `mergesort` example that can sort vectors up to length  $2^{22}$ . Since our benchmark input size of  $2^{24}$  is larger than this limit, we always use a divide and conquer strategy to allow subproblems to be computed by the GPU.

We examine three strategies for distributing the work between the CPU and GPU:

- **static\_blocking:** Static partitioning of work (50% CPU, 50% GPU) where GPU calls block a Meta-Par worker. This work division was selected based on the fact that our 4-core Cilk (CPU) and CUDA (GPU) sorts perform at very nearly the same level.
- **static:** Static partitioning of work (50% CPU, 50% GPU) with non-blocking GPU calls as described in section 4.5.
- **dynamic:** Dynamic partitioning of work with CPU-GPU work stealing.

In all cases, adding GPU computation improves performance over CPU-only computation. Performance improves further by adding non-blocking GPU calls. The dynamic strategy, representative of the Accelerate Resource implementation described in section 4.5, pairs each spawn of a GPU computation with a CPU version of that same computation, allowing the utilization of both resources to benefit from work stealing. Figure 7 shows that this yields better performance than a priori static partitioning of the work.

### 6.4 Comparison against other Distributed Haskell

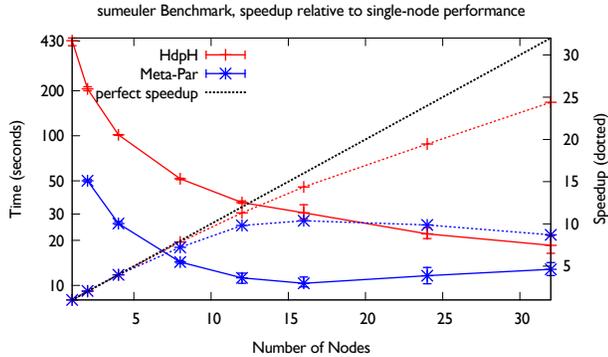
Meta-Par offers Resources for execution on distributed memory architectures. We have prototyped Resources based on different communication backends: (1) `haskell-mpi`, Haskell bindings for the Message Passing Interface (MPI), and (2) `network-transport`, an abstract transport layer for communication that itself has multiple backends (e.g. TCP, linux pipes). All results below are from our `network-transport/TCP` Resource. This results in lower performance than using MPI, but our MPI runs are currently unstable due to a number of bugs.

We evaluated the distributed performance on a cluster of 128 nodes, each with two dual-core AMD Opteron 270 processors running at 2GHz and 4GB of memory. The nodes were connected by a 10GB/s Infiniband network and Gigabit Ethernet. Figure 8 shows a comparison between HdpH [23] and Meta-Par for the `sumEuler` benchmark which involved computing the sum of Euler’s totient function between 1 and 65536 by dividing up the work in 512 chunks. The tests were run up to 32 nodes since significant speedup was not observed beyond that for the above workload. The HdpH tests were run with 1 core per node<sup>16</sup>, whereas the Meta-Par tests used all 4 cores on the node. Both tests measured the overall execution time including the time required for startup and shutdown of the distributed instances.

As seen in Figure 8, Meta-Par performed nearly 4 times better than HdpH in this test, but HdpH continued scaling to a larger num-

<sup>15</sup>We are working with the Accelerate authors to resolve these issues.

<sup>16</sup>In the original HdpH paper [23] the authors report not seeing further increases in performance when attempting multi-process (rather than multi-threaded) parallelism within each node. Recent versions of HdpH, after the submission of this paper, have added better support for SMP parallelism. But we have not yet evaluated these; our attempt at using HdpH (r0661a3) with multiple threads per node led to a regression involving extremely variable runtimes.



**Figure 8.** Performance comparison and scaling behavior of Meta-Par (distributed) vs. HdpH.

ber of nodes: 30 rather than 16. The advantage in performance in this case was due entirely to the *composed scheduler*, which mixed fine-grained SMP work-stealing with distributed work-stealing. For our chosen workload, the lower bound in performance was limited by the Meta-Par bootstrap time and the time to process a single chunk sequentially. As a result, no significant speedup was achieved beyond 16 nodes with a best execution time of 9 seconds.

HdpH ran using MPI in this example, and it also uses a more sophisticated system than Meta-Par for global work-stealing in which nodes “prefetch” work when their own work-pools run low, not waiting for them to run completely dry. This helps hide the latency of distributed steals and may contribute to the better scaling seen in Figure 8. There is clearly much work left to be done. Fortunately, HdpH and Meta-Par expose very similar interfaces, and we hope by standardizing on interfaces (type-classes) that it will be possible for the community to incrementally develop and optimize a number of (compatible) distributed execution backends for Haskell.

### Distributed KMeans

To compare directly against CloudHaskell we ported the KMeans benchmark. We have not yet run this benchmark on our cluster infrastructure, but we have run a small comparison with two workers (on the 3.10 GHz Westmere configuration). Under this configuration CloudHaskell takes 2540.9 seconds to process 600K 100-dimensional points in 4 clusters for 50 iterations. Meta-Par, on the other hand took 175.8 seconds to accomplish the same.

In this case we believe the difference comes from (1) Meta-Par having a more efficient dissemination of the (>1GB) input data, and (2) a high level of messaging overhead in CloudHaskell. In our microbenchmarks, CloudHaskell showed a 10ms latency for sending small messages.

## 7. Related Work

In the introduction we mentioned the triple problem of scheduler’s non-compositionality, complexity, and restriction to CPUs-only. Several systems have attempted to solve one or more of these problems. For example, the Lithe [31] system addresses the first—the *scheduler* composition problem—by instrumenting a number of different schedulers to support the dynamic addition and subtraction of worker threads. Thereafter *hardware resources themselves* can become first class objects that are passed along in subroutine calls: that is, a subroutine inside one library may receive a certain resource allocation that it may divvy among its own callees. Lithe is focused on composing a-priori unrelated schedulers, whereas

Meta-Par focuses on creating an architecture for composable heterogeneity.

In the functional programming context, Manticore [15] also aims at scheduler deconstruction, by providing a set of primitives used to construct many different schedulers. But, like Lithe, Manticore only targets CPU computation. Further, to our knowledge Manticore has not been used to demonstrate advantages of simultaneous use of different scheduling algorithms in the same application (rather than between different applications).

Li, Marlow, Peyton-Jones, and Tolmach’s work on lightweight concurrency primitives [20] is a wonderfully clear presentation of an architecture very similar to Meta-Par. The core of their system exposes a spartan set of primitives used by client libraries to implement callbacks, an arrangement much like our Resources. Their work focuses on the lower-level details of implementing Haskell concurrency primitives for CPUs, while Meta-Par extends the higher-level, deterministic Par-monad framework to heterogeneous environments, and it is encouraging to see similar architectures yield good results toward both goals.

CloudHaskell [14] is a library providing Erlang-like functionality for Haskell. CloudHaskell offers a relatively large API in one package (messaging, monitors, serialization, task farming), and in our experiments was high-overhead. We found small messages incurring a 10ms latency on a gigabit Ethernet LAN. For these reasons we ended up basing our own Meta-Par library on lower level communication libraries rather than CloudHaskell.

## 8. Future Work and Conclusions

While we have achieved some initial results that show strong CPU/GPU and CPU/distributed integration, there remain many areas where we need to improve our infrastructure and apply it to more applications. In the process, we plan to continue to contributing to low-level libraries for high-performance Haskell. (This work has resulted in both `GHC`<sup>17</sup> and `haskell-mpi` bug fixes!) We also will work on Accelerate development until Accelerate CPU/GPU programs can be written and run efficiently in Meta-Par.

We want to ensure that Meta-Par is usable by the community, and ultimately we regard it as a relatively thin layer in an ecosystem of software including GPU and networking drivers, EDSLs, concurrent data structures, and so on. But by integrating disparate capabilities in one framework, Meta-Par opens up interesting possibilities, such as automatically generating code for separate phases of a recursive algorithm (e.g. distributed, parallel, sequential).

<sup>17</sup> Atomic compare-and-swap operations were missing a GC barrier.

## References

- [1] Code for cilk runtime system. <https://github.com/mirrors/gcc/tree/cilkplus/libcilkrts>.
- [2] Intel Cilk Plus. <http://software.intel.com/en-us/articles/intel-cilk-plus/>.
- [3] Openmp article. <http://intel.ly/9h7c7B>.
- [4] Threading Building Blocks Reference Manual, 2011. <http://threadingbuildingblocks.org/documentation.php>.
- [5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.
- [6] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.
- [7] G. Blelloch, P. Gibbons, Y. Matias, and G. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 12–23, Newport, RI, jun 1997.
- [8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.
- [9] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.
- [10] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB*, 1(2):1313–1324, 2008.
- [11] K. Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9:313–323, May 1999.
- [12] D. Doel. The vector-algorithms package. <http://hackage.haskell.org/package/vector-algorithms>. Efficient algorithms for vector arrays.
- [13] M. Dybdal. The hopencl package. <http://hackage.haskell.org/package/hopencl>. Haskell bindings for OpenCL.
- [14] J. Epstein, A. P. Black, and S. Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.
- [15] 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*, DAMP '07, pages 37–44, New York, NY, USA, 2007. ACM.
- [16] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3.4):143 – 153, 1986.
- [17] C. Lauterback, Q. Mo, and D. Manocha. Work distribution methods on GPUs. University of North Carolina Technical Report TR009-16.
- [18] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.
- [19] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *SIGPLAN Not.*, 44:227–242, Oct. 2009.
- [20] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach. Lightweight concurrency primitives for ghc. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 107–118, New York, NY, USA, 2007. ACM.
- [21] 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, New York, NY, USA, 2007. ACM.
- [22] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löb. A generic deriving mechanism for haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.
- [23] P. Maier, P. Trinder, and H.-W. Loidl. Implementing a High-Level Distributed-Memory parallel Haskell in Haskell, 2011. Submitted to IFL 2011.
- [24] G. Mainland and G. Morrisett. Nikola: embedding compiled gpu functions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM.
- [25] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM.
- [26] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.
- [27] T. L. McDonell. cuda. <http://hackage.haskell.org/package/cuda>. FFI binding to the CUDA interface for programming NVIDIA GPUs.
- [28] C. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 224–235, april 2011.
- [29] R. Newton, C.-P. Chen, and S. Marlow. Intel Concurrent Collections for Haskell, March, 2011. MIT CSAIL Technical Report, MIT-CSAIL-TR-2011-015.
- [30] B. O'Sullivan and J. Tibell. Scalable i/o event handling for ghc. *SIGPLAN Not.*, 45(11):103–108, Sept. 2010.
- [31] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. *SIGPLAN Not.*, 45:376–387, June 2010.
- [32] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, July 2007.
- [33] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *SIGPLAN Not.*, 44:317–328, Aug. 2009.
- [34] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 91–100, New York, NY, USA, 2009. ACM.
- [35] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 253–264, New York, NY, USA, 2008. ACM.
- [36] J. Svensson, M. Sheeran, and K. Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In S.-B. Scholz and O. Chitil, editors, *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer Science*, pages 156–173. Springer Berlin / Heidelberg, 2011.
- [37] D. Syme, T. Petricek, and D. Lomov. The # asynchronous programming model. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag.

## A. Appendix: Operational Semantics

[Reproduced for convenience in largely identical form to [25]]

Figure 9 gives the syntax of values and terms in our language. The only unusual form here is `done M`, which is an internal tool for the semantics of `runPar`. The main semantics for the language is a big-step operational semantics written  $M \Downarrow V$  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 (*RunPar*) in Figure 11. We will discuss (*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 9. A state is a bag of terms  $M$  (its active “threads”), and `IVar`s  $i$  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  $P_0 \rightarrow \dots \rightarrow P_i$  where  $i \geq 0$ .

States obey a structural equivalence relation  $\equiv$  given by Figure 10, 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 10 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 11 using an *evaluation context*  $\mathcal{E}$ :

$$\mathcal{E} ::= [\ ] \mid \mathcal{E} \gg = M$$

Hence the term that determines a transition will be found by looking to the left of  $\gg =$ . Rule (*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 (*Bind*) is the standard monadic bind semantics.

Rule (*Fork*) creates a new thread.

Rules (*New*), (*Get*), and (*PutEmpty*) give the semantics for operations on `IVar`s, 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.

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. Rules (*GCEmpty*) and (*GCFull*) allow an empty or full `IVar` respectively to be garbage collected provided the `IVar` is not referenced anywhere else in the state. The equivalences for  $\nu$  in Figure 10 allow us to push the  $\nu$  down until it encloses only the dead `IVar`.

Rule (*GCDeadlock*) allows a set of deadlocked threads to be garbage collected: the syntax  $\mathcal{E}[\text{get } i]^*$  means one or more threads of the given form. Since there can be no other threads that refer to  $i$ , none of the `gets` 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.

	$x, y$	$\in$	Variable
	$i$	$\in$	<code>IVar</code>
Values	$V$	$::=$	$x \mid i \mid \backslash x \mid - \mid > \mid M$ $\text{return } M \mid M \gg = N$ $\text{runPar } M$ $\text{fork}_n M$ $\text{new}$ $\text{put } i M$ $\text{get } i$ $\text{done } M$
Terms	$M, N$	$::=$	$V \mid M N \mid \dots$
States	$P, Q$	$::=$	$M$ thread of computation $\langle \rangle_i$ empty <code>IVar</code> named $i$ $\langle M \rangle_i$ full <code>IVar</code> named $i$ , holding $M$ $\nu i.P$ restriction $P \mid Q$ parallel composition

Figure 9. The syntax of values and terms

$$\begin{aligned}
P \mid Q &\equiv Q \mid P \\
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\
\nu x.\nu y.P &\equiv \nu y.\nu x.P \\
\nu x.(P \mid Q) &\equiv (\nu x.P) \mid Q, \quad x \notin \text{fn}(Q) \\
\frac{P \rightarrow Q}{P \mid R \rightarrow Q \mid R} & \quad \frac{P \rightarrow Q}{\nu x.P \rightarrow \nu x.Q} \\
\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} &
\end{aligned}$$

Figure 10. Structural congruence, and structural transitions.

$$\begin{aligned}
\frac{M \not\equiv V \quad M \Downarrow V}{\mathcal{E}[M] \rightarrow \mathcal{E}[V]} & \quad (\text{Eval}) \\
\mathcal{E}[\text{return } N \mid \gg = \mid M] \rightarrow \mathcal{E}[M N] & \quad (\text{Bind}) \\
\mathcal{E}[\text{fork}_n M] \rightarrow \mathcal{E}[\text{return } () \mid M] & \quad (\text{Fork}) \\
\mathcal{E}[\text{new}] \rightarrow \nu i.(\langle \rangle_i \mid \mathcal{E}[\text{return } i]), \quad i \notin \text{fn}(\mathcal{E}) & \quad (\text{New}) \\
\langle M \rangle_i \mid \mathcal{E}[\text{get } i] \rightarrow \langle M \rangle_i \mid \mathcal{E}[\text{return } M] & \quad (\text{Get}) \\
\langle \rangle_i \mid \mathcal{E}[\text{put } i M] \rightarrow \langle M \rangle_i \mid \mathcal{E}[\text{return } ()] & \quad (\text{PutEmpty}) \\
\text{return } M \rightarrow & \quad (\text{GCReturn}) \\
\nu i.\langle \rangle_i \rightarrow & \quad (\text{GCEmpty}) \\
\nu i.\langle M \rangle_i \rightarrow & \quad (\text{GCFull}) \\
\nu i.(\langle \rangle_i \mid \mathcal{E}[\text{get } i]^*) \rightarrow & \quad (\text{GCDeadlock}) \\
\frac{(M \gg = \backslash x.\text{done } x) \rightarrow^* \text{done } N, \quad N \Downarrow V}{\text{runPar } M \Downarrow V} & \quad (\text{RunPar})
\end{aligned}$$

Figure 11. Transition Rules