# The Monad.Reader Issue 19: Parallelism and Concurrency

by Kazu Yamamoto ⟨kazu@iij.ad.jp⟩
and Bernie Pope ⟨florbitous@gmail.com⟩
and Dmitry Astapov ⟨dastapov@gmail.com⟩
and Mario Blažević ⟨blamario@acanac.net⟩

October 26, 2011

Brent Yorgey and Edward Z. Yang, editors.

# Contents

# Editorial

by Brent Yorgey ⟨byorgey@cis.upenn.edu⟩

```
RWWJQXUQJQXHWDFQGZRTWNWFJATNBJUSYYMQYNTNMDTJWTWITNUJUXUSUTJNPQYJNUQRTW
XWJDDTSJLKSGJUHTWTIWFFQZFUHWRGAFWGNWSJXUJXLXSFXXTTHXFJFSJYYTPFYHSMTHXW
FQGSUIDUXRYQFQJFQNRPZSTJQXQXFXYFIXKBUNXWQSHSTNSFRTZYUWJJXWJDNJRFTPUTXW
JJWFHJKNLZUNYJYDYTIJZTXXFZFNIZHUHJEFHWMJRBWYYYGJFNXAJNWYYYGNEKBTQHYNST
FTNTU
```

# Mighttpd – a High Performance Web Server in Haskell

by Kazu Yamamoto ⟨kazu@iij.ad.jp⟩

*In 2002, Simon Marlow implemented a web server in Haskell, whose performance is acceptable for most web sites [1]. Since then, the Haskell community has been developing high performance tools such as `ByteString`, the new IO manager in GHC 7, `attoparsec`, and `Enumerator/Iteratee`. Also, good web application frameworks have appeared one after another. It is a good time to revisit a web server to show that we can implement a high performance web server in Haskell which is comparable to highly tuned web servers written in C.*

## The Evolution of Mighttpd

IIJ Innovation Institute Inc., to which the author belongs, is a designated research company under Internet Initiative Japan Inc. (IIJ), one of the biggest and oldest ISPs in Japan. In the autumn of 2009, we needed a web server whose back-end storage could be freely changed for our cloud research. Instead of modifying existing web servers, we started implementing a web server from scratch with GHC 6. For this implementation, we created three Haskell packages which are all in HackageDB:

- `c10k`: a network library which handles more than 1,024 connections with the *pre-fork* technique. Since the IO manager of GHC 6 is based on the *select* system call, it cannot handle more than 1,024 connections at the same time. Classical network programming issues the *fork* system call after a socket is *accept*ed. If we use *fork* before accepting connections, the listening port is shared among the processes and the OS kernel dispatches a new connection randomly to one of processes when accepting. After that, the OS kernel ensures that each connection is delivered to the appropriate process.

- `webserver`: an HTTP server library including an HTTP engine (parser/-formatter, session management), the HTTP logic (caching, redirection), and a CGI framework. The library makes no assumptions about any particular back-end storage.
- `mighttpd`: a file-based web server with configuration and a logging system. Mighttpd [2] should be pronounced "mighty".

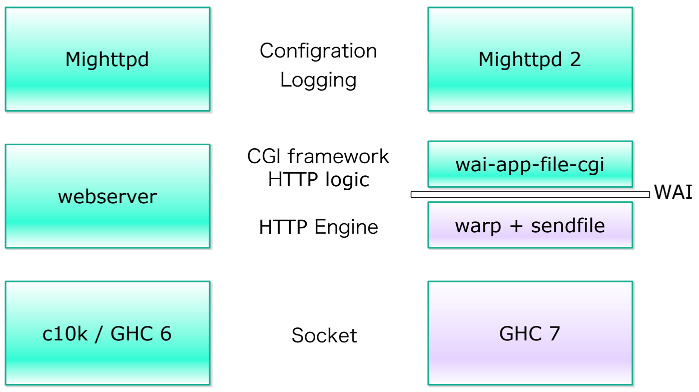

**Figure 1:** The left side is Mighttpd and the right side is Mighttpd 2 which uses WAI.

This architecture is schematically shown on the left side of Figure 1. Mighttpd is stable and has been running on the author's private web site, Mew.org, for about one year, providing static file contents, mailing list management through CGI, and page search services through CGI.

Next autumn, we applied to the Parallel GHC Project [3], hoping that we could get advice on how best to use Haskell's concurrency features. This project is organized by Well-Typed and sponsored by GHC HQ. Fortunately, we were elected as one of four initial members.

Around this time, GHC 7 was released. GHC 7.0.1 provided a new IO manager based on *kqueue* and *epoll* [4]. GHC HQ asked us to test it. So, we tried to run Mighttpd with it but it soon appeared that the new IO manager was unstable. Together with Well-Typed, we discovered six bugs, which were quickly fixed by both GHC HQ and Well-Typed. GHC 7.0.2 included all these fixes, and the IO manager became stable in the sense that it could run Mighttpd.

A lot of great work has been done by others on web programming in Haskell, and several web application frameworks such as Happstack [5], Snap [6], and Yesod [7] have appeared over the years. We started to take a closer look, and it turned out that Yesod is particularly interesting to us because it is built on the concept of the Web Application Interface (WAI). Using WAI, a CGI script, FastCGI server, or web application can all be generated from the same source code.

The HTTP engine of WAI, called Warp [8], is extremely fast, and we decided to adopt it. In addition, since GHC 7 no longer has the *select*-based limit of 1024 connections, there was no need to continue using the `c10k` package. Since we now relied on Warp, we could strip the HTTP engine from the `webserver` package. The rest was implemented as a WAI-based application as illustrated on the right side of Figure 1. The corresponding HackageDB packages are called `wai-app-file-cgi` and `mighttpd2`.

## Benchmark

The Yesod team has made an effort to benchmark Warp using `httperf` with the following options[9]:

```
httperf --hog --num-conns 1000 --rate 1000 \
        --burst-length 20 --num-calls 1000 \
        --server localhost --port 3000 --uri=/
```

This means that `httperf` repeats “20 burst requests” 50 times for one connection, tries to make 1,000 connections per second, and stops the measurement when 1,000 connections are completed. Their complete results are given in a blog post [10]. In particular, the performance of Warp in Amazon EC2 is 81,701 queries per second. Note that in this benchmark Warp just sends an HTTP response against an HTTP request without handling the HTTP logic and touching files. We decided to benchmark Mighttpd similarly. Our own benchmark environment is as follows:

- Guest OS: Ubuntu 10.10, four cores, 1G memory.
- Host OS: Ubuntu 10.04, KVM 0.12.3, Intel Xeon CPU L5520 @ 2.27GHz x 8, four cores/CPU (32 cores total), 24G memory.

In this environment, the performance of Warp is 23,928 queries/s and that of Mighttpd 2 is 4,229 queries/s. We were disappointed that Mighttpd 2 was so much slower than plain Warp, and started tuning our web server.

## Performance tuning

It may be surprising, but the main bottleneck was `Data.Time`. This library had been used to parse and generate date-related HTTP fields such as `Last-Modified:`.

For our purpose, this library is simply too generic and unacceptably slow. About 30–40% of CPU time was spent in `Data.Time` functions. To alleviate the problem, we implemented the `http-date` package which directly uses `ByteString` and is about 20 times faster than `Data.Time`.

The next bottlenecks were the functions that manipulate files. A typical Haskell idiom to get the status of a file is the following:

```
exists <- doesFileExist file
when exists $ do
    stat <- getFileStatus file
    ...
```

Both `doesFileExist` and `getFileStatus` issue the *stat* system call. Since they are slow, we decided to use `getFileStatus` without `doesFileExist` and catch errors for better performance. Moreover, because the status of files does not change often, we modified Mighttpd 2 to cache the file status in an `IORef (Map ByteString FileStatus)`.

We first believed that the reason these functions were slow was that they manipulate files. But it appeared that the real reason was that they issued a system call. *System calls are evil for network programming in Haskell.* When a user thread issues a system call, a context switch occurs. This means that all Haskell user threads stop, and instead the kernel is given the CPU time. If we are striving for highly concurrent Haskell programs, we must avoid issuing system calls as much as possible.

Warp uses the *recv* system call to receive an HTTP request and the *writev* system call to send an HTTP header. When sending an HTTP body based on a file, it uses the `sendfile` package which unnecessarily issues the *lseek* and *stat* system calls in addition to *sendfile*. While one could believe that the *sendfile* system call is fast thanks to its zero-copying approach, the package is actually much slower than we expected. We implemented the `simple-sendfile` package which does not use *lseek* and *stat*. The system calls that the package uses are only *open*, *sendfile*, and *close*. Since sockets are marked non-blocking, *sendfile* returns *EAGAIN* if the size of a file is large. In this case, the `simple-sendfile` package issues *sendfile* again without *lseek* to send the rest of the file.

At this point, the performance of Mighttpd 2 (without logging) became 21,601 queries per second. We also measured the performance of `nginx`, which is written in C. The performance of `nginx` (without logging) is 22,713 queries per second. Finally, we could say that a web server written in Haskell was comparable to one written in C.

## Scaling on a multi-core processor

Since the new IO manager is single-threaded, Haskell network programs cannot realize the potential of a multi-core processor even if the `+RTS -Nx` command line option is specified. For our benchmark, the performance of Mighttpd 2 with the `-N3` option is 15,082 queries per second, which is slower than that with `-N1` (21,601). Moreover, `forkProcess` and `+RTS -Nx` cannot be used together. This means that we cannot daemonize the network programs if `+RTS -Nx` is specified.

To get around these limitations, we again introduced the *pre-fork* technique. Since Mighttpd 2 does not share essential information among user threads, we don't have to share any information among pre-forked processes. We compared Mighttpd 2 and `nginx` with three worker processes. The performance of Mighttpd 2 (without logging) is 61,309 queries per second, while that of `nginx` (without logging) is 30,471. Note that we are no experts in the use of `nginx`, so we might have been able to obtain better results by tweaking its configuration. We can at least say that the *pre-fork* technique is very effective for these kinds of servers.

## Logging

Needless to say, logging is an important function of web servers. Since a logging system has to manipulate files, it could be a big bottleneck. Note also that we needed to ensure that log messages written concurrently were not mangled. We implemented several logging systems and compared their performance. The techniques which we tested were:

- Using `hPut` with a `Handle` in `AppendMode`
- Using `fdWriteBuf` in non-blocking and append mode
- Using the *ftruncate* and *mmap* system calls to append log messages to a log file
- Buffering log messages with `MVar` or `Chan`
- Separating a dedicated logging process from worker processes

Let us describe the most complex of the approaches: we separated the server processes into several worker processes and one dedicated logging process. Each worker process shares memory with the logging process, and communicates using UNIX pipes. In a worker process, user threads put log messages to a `Chan` for serialization. They are written to the shared memory by a dedicated user thread. When the shared memory is about to overflow, the user thread asks the logging process to write $N$ bytes to a log file. The logging process copies the memory to another area of memory mapped to a log file with the *mmap* system call. Then the logging process sends an acknowledgement to the user thread.

Fortunately or unfortunately, our conclusion was that *simple is best*. That is, the fastest way was to have each user thread append a log message through a `Handle`. Since the buffer of the `Handle` is protected with an `MVar`, user threads can safely use it concurrently. If we use the `hPut` family to write a log message and the buffer mode for the `Handle` is `LineBuffering`, we can ensure that the entire line is written to its log file atomically. But a non-blocking *write* system call is issued for each log event.

We noticed that `BlockBuffering` can be used as "multi" `LineBuffering`. Suppose that we use the `hPut` family to write a log message to a `Handle` whose buffer mode is `BlockBuffering`. If the buffer has enough space, the log message is simply stored. If the buffer is about to overflow, it is first flushed, then the log message is stored in the empty buffer. In other words, the `hPut` family never splits lines. We chose 4,096 bytes as the size of the buffer since it is the typical page size.

Also, we re-implemented `hPut` so that no unnecessary intermediate data is produced. A log message consists of a list of `ByteString`s and/or `String`s. Our `hPut` directly copies `ByteString`s and directly packs `String`s into the buffer.

The typical log format of Apache contains an IP address and a zoned time. To generate a numeric string of an IP address including IPv4 and IPv6, `getNameInfo` should be used. However, it appeared that `getNameInfo` was slow. We thus implemented an IP address pretty printer ourselves. As we mentioned earlier, `Data.Time` was unacceptably slow for our purpose, but re-implementation of `Data.Time` was a tough job due to time zones. So, Mighttpd 2 uses `Data.Time` once every second and caches a `ByteString` containing the zoned time. For this cache, we use an `IORef` instead of an `MVar`, taking a lesson from the experience of the Yesod team [8].

The performance of `nginx` (3 workers with logging) is 25,035 queries per second, while that of Mighttpd 2 (3 workers with logging) is 31,101 queries per second. Though Mighttpd 2 is still faster, `nginx` loses only 18% of its performance through logging whereas Mighttpd 2 loses 49%. This indicates that more efficient logging systems might be possible. However, we have currently run out of ideas for further optimizations. Implementing a better logging system remains an open problem. Feedback would be appreciated.

## Space leak

We observed a space leak in Mighttpd 2. If no connection requests arrive for a long time, Mighttpd 2's processes get fatter. The cause was `atomicModifyIORef`.

As described earlier, Mighttpd 2 caches a `ByteString` of the time adapted to the current time zone every second. The `ByteString` is stored in `IORef` with `atomicModifyIORef` as follows:

```
atomicModifyIORef ref (\_ -> (tmstr, ()))
```

We realized that the reason for the space leak is that the result of `atomicModifyIORef` is not used. If we used the `ByteString` with `readIORef` for logging, the space leak quickly disappeared. To prevent the space leak, we adopted the following idiom.

```
x <- atomicModifyIORef ref (\_ -> (tmstr, ()))
x ‘seq‘ return ()
```

## Using Mighttpd 2

As we described earlier, Mighttpd 2 is registered in HackageDB. You can install it with the `cabal` command as follows:

```
$ cabal install mighttpd2
```

To run Mighttpd 2, you need to specify a configuration file and a path routing file. Please consult the home page of Mighttpd 2 [2] to learn the syntax. Mighttpd 2 has been running on Mew.org for several months providing static file content, mailing list management through CGI, and page search services through CGI.

## Conclusion

With a good Haskell compiler such as GHC 7 and high performance tools such as `ByteString` and `simple-sendfile`, it is possible to implement a high performance web server in Haskell comparable to highly tuned web servers written in C. Though event-driven network programming is still popular in other programming languages, Haskell provides user thread network programming which makes the code simple and concise. To fully take advantage of user threads, we should avoid issuing system calls as much as possible.

## Acknowledgment

We would like to acknowledge Michael Snoyman, Matt Brown, and Greg Weber for their efforts on Yesod/Warp. We thank the members of the Parallel GHC project (GHC HQ at Microsoft Research and Well-Typed) for their help in developing Mighttpd. We are deeply grateful to Duncan Coutts, Andres Löh, and Randy Bush for their review of a draft version of this article.

## References

[1] Simon Marlow. Developing a high-performance web server in Concurrent Haskell. **Journal of Functional Programming**, 12:pages 359–374 (July 2002).

[2] `http://mew.org/~kazu/proj/mighttpd/`.

[3] Dmitry Astapov. Parallel Haskell project underway (Nov 2010). `http://www.well-typed.com/blog/48`.

[4] Bryan O'Sullivan and Johan Tibell. Scalable I/O Event Handling for GHC. In **Proceedings of the third ACM Haskell symposium**, pages 103–108. Haskell '10, ACM, New York, NY, USA (2010).

[5] `http://happstack.com/`.

[6] `http://snapframework.com/`.

[7] `http://www.yesodweb.com/`.

[8] Steve Vinoski. Warp: A Haskell Web Server (May 2011). `http://steve.vinoski.net/blog/2011/05/01/warp-a-haskell-web-server/`.

[9] `https://github.com/yesodweb/benchmarks`.

[10] Michael Snoyman. Preliminary Warp Cross-Language Benchmarks. `http://www.yesodweb.com/blog/2011/03/preliminary-warp-cross-language-benchmarks` (Mar 2011).

# High Performance Haskell with MPI

by Bernie Pope ⟨bjpope@unimelb.edu.au⟩
and Dmitry Astapov ⟨dastapov@gmail.com⟩

*In this article, we give a brief overview of the Haskell-MPI library and show how it can be used to write distributed parallel programs. We use the trapezoid method for approximating definite integrals as a motivating example and compare the performance of an implementation using Haskell-MPI to three variations of the same algorithm: a sequential Haskell program, a multi-threaded Haskell program, and a C program also using MPI.*

## Distributed-memory parallelism and MPI

We are fast approaching the era of mega-core supercomputers. For example, the Lawrence Livermore National Laboratory is currently preparing to install Sequoia, a 1.6 million core IBM BlueGene/Q [1]. There are many technical challenges to building such behemoths, not the least of which is the CPU-to-memory bottleneck. In broad architectural terms, there are two basic ways to divvy up the RAM amongst the cores: you can share it, or you can distribute it. In the shared model, all processors participate in a single unified memory address space even if the underlying interconnects are non-uniform. In the distributed model, each processor (or small group of processors) has its own private memory address space and access to non-local memory is performed by explicit copying. The advent of multi-core CPUs has made shared-memory systems widely available, but it is difficult to scale this abstraction in a cost-effective way beyond a few thousand cores. A quick glance at the Top 500 list of supercomputers reveals that distributed-memory systems dominate the top-end of high-performance computing [2].

Distributed-memory parallelism does not really solve the CPU-to-memory bottleneck (over the whole machine); after all, copying data between computers over a network is a relatively costly operation. Instead it forces programmers to address the non-uniformity head on, which typically means adopting an explicitly distributed style of parallel programming and devising new algorithms.

Haskell already has excellent support for shared-memory parallelism via the multi-threaded runtime system of GHC. However, if we want to use Haskell on the latest supercomputers we need to go beyond threads and embrace the distributed model. Haskell-MPI attempts to do that in a pragmatic way by providing a Haskell wrapper to the Message Passing Interface (MPI). MPI is a "message-passing library interface specification" for writing distributed-parallel programs [3]. Various realizations of the specification are provided by software libraries, some open source, such as Open MPI [4] and MPICH [5], and some proprietary. As the name suggests, MPI is firmly rooted in the paradigm of message passing. An MPI application consists of numerous independent computing processes which collaborate by sending messages amongst themselves. The underlying communication protocols are programming language agnostic, but standard APIs are defined for Fortran, C and C++. Bindings in other languages, such as Haskell-MPI, are typically based on foreign interfaces to the C API.

Haskell-MPI provides a fairly modest wrapping of MPI, and is guided by two objectives:

1. Convenience: for a small cost, it should be easy to send arbitrary (serializable) data structures as messages.
2. Performance: low overhead communications should be possible, particularly for array-like data structures.

It is difficult to satisfy both objectives in one implementation, so Haskell-MPI provides two interfaces. The first is simple to use (more automated) but potentially slower (more data copying), the second is more cumbersome to use (less automated) but potentially faster (less data copying).

This article aims give you a taste of distributed parallel programming with Haskell-MPI, enough to whet your appetite, without getting too bogged down in details. Those who find themselves hungry for more can consult the haddock pages and check out examples in the package sources.

We begin by introducing the technique of computing definite integrals by the trapezoid method. This lends itself to an easy-to-parallelize algorithm which will serve as the basis of programming examples in the following sections. We take a simple sequential implementation of the algorithm, and convert it into two different parallel implementations. The first uses shared-memory and threads, and the second uses distributed-memory and Haskell-MPI. To see how well we fare against the conventional school, we also provide an MPI implementation in C. We then evaluate the performance of each version on a non-trivial problem instance and compare the results.

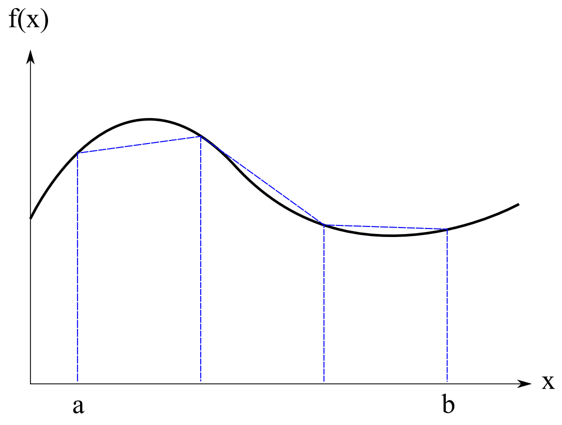

**Figure 1:** Approximating $\int_{x=a}^{b} f(x)$ by summing the area of three trapezoids.

## Computing definite integrals with trapezoids

We now consider the problem of computing definite integrals using the trapezoid method. The algorithm is naturally data parallel, and is a common introductory example in parallel programming tutorials. Our presentation is inspired by Pacheco's textbook on MPI [6].

We can approximate integrals by summing the area of a consecutive trapezoids lying under a function within a given interval, as illustrated in Figure 1. A single trapezoid spanning the interval $[x_0, x_1]$ has area:

$$\frac{(x_1 - x_0)(f(x_0) + f(x_1))}{2}$$

Extending this to $n$ equally spaced sub-intervals $[x_0, x_1, \ldots, x_n]$ we arrive at the formula:

$$\begin{aligned}
\int_{x=a}^{b} f(x) &\approx \frac{h}{2} \sum_{i=1}^{n} f(x_{i-1}) + f(x_i) \\
&= h \left( \frac{f(x_0) + f(x_n)}{2} + f(x_1) + \ldots + f(x_{n-1}) \right) \\
& \quad x_0 = a,\ x_n = b,\ h = (b - a)/n \\
& \quad \forall i \in \{0 \ldots n-1\},\ x_{i+1} - x_i = h
\end{aligned}$$

Listing 1 provides a sequential implementation of the trapezoid method in Haskell, with the integrated function defined to be $f(x) = sin(x)$.

```haskell
module Trapezoid (trapezoid, f) where

trapezoid :: (Double -> Double)  -- Function to integrate
             -> Double -> Double -- integration bounds
             -> Int              -- number of trapezoids
             -> Double           -- width of a trapezoid
             -> Double
trapezoid f a b n h =
   h * (endPoints + internals)
   where
   endPoints = (f a + f b) / 2.0
   internals = worker 0 (a + h) 0.0
   worker :: Int -> Double -> Double -> Double
   worker count x acc
      | count >= n - 1 = acc
      | otherwise = worker (count + 1) (x + h) (acc + f x)

f :: Double -> Double
f = sin
```

**Listing 1:** Calculating definite integrals using the trapezoid method.

There are undoubtedly more elegant ways to sum the internal points of a trapezoid than our `worker` function, but we found that GHC produces better compiled code in this case when given an explicit loop.

Listing 2 provides a `main` function which takes `a`, `b` and `n` from the command line, calls `trapezoid` to compute the integral, and prints the result.[1] Later on we will provide alternative `main` functions which will parallelize the program without requiring any changes to `trapezoid` or `f`.

```
module Main where

import System (getArgs)
import Trapezoid

main :: IO ()
main = do
  aStr:bStr:nStr:_ <- getArgs
  let [a,b] = map read [aStr,bStr]
      n = read nStr
      h = (b - a) / fromIntegral n
      integral = trapezoid f a b n h
  print integral
```

**Listing 2:** Sequential program for calculating definite integrals.

## Parallelization of the trapezoid method on a single machine

The trapezoid method is a classic data-parallel problem because the computations on each sub-interval can be computed independently. For an interval $[a, b]$, $n$ sub-intervals, and $p$ processors, we can parallelize the algorithm using a simple chunking scheme like so:

1. The master processor splits the sub-intervals into chunks of size $s = n/p$ (assuming $n \geq p$).
2. In parallel, each processor $p_i$ computes the definite integral on the sub-interval $[a_i, b_i]$, where $h = (b - a)/n$, $a_i = a + i \times s \times h$ and $b_i = a_i + s \times h$.
3. The master processor collects the results for each chunk and sums them up.

[1]Normally, we would check the program inputs for correctness, but in the interests of saving space, we have elided such checks in the program examples in this article.

```
module Main where

import GHC.Conc
import Control.Parallel.Strategies

import System (getArgs)
import Trapezoid

main :: IO ()
main = do
  let numThreads = numCapabilities
  aStr:bStr:nStr:_ <- getArgs
  let [a,b] = map read [aStr,bStr]
      n = read nStr
      h = (b - a) / fromIntegral n
      localN = n `div` fromIntegral numThreads
      chunks = parMap rseq (\threadNo ->
         let localA = a + fromIntegral (threadNo * localN) * h
             localB = localA + fromIntegral localN * h
             in trapezoid f localA localB localN h) [0..numThreads-1]
  print (sum chunks)
```

**Listing 3:** Multi-threaded parallel program for calculating definite integrals using the trapezoid method.

Listing 3 shows that it is quite easy to use threads to parallelize the trapezoid program, thanks to the convenient `parMap` combinator provided by the parallel strategies library [7].

## Parallelization on multiple machines using MPI

### Point-to-point communications

We can use the same chunking approach to parallelize the program using MPI, whereby the workload is divided over many independent processes, each with its own private address space. One of the processes is designated to be the master, which, in addition to computing its own chunk of the problem, is also responsible for collecting the results from the other processes and combining them into the final answer. On a distributed-memory system we can spread the MPI processes over multiple networked computers (normally one process per core), and thus scale the parallelization well beyond the number of cores on a single machine.

In multi-threaded programs each parallel task is identified by a simple numeric index, whereas MPI uses a two-level numbering scheme. The first level indicates a group of processes called a communicator; the second level is the rank of an individual process within such a group. Each process can participate in an arbitrary number of communicators, which can be created and destroyed at run time. By default, all processes are members of the single pre-defined communicator called `commWorld`.

Listing 4 shows how to parallelize our program using two point-to-point communication functions:

```
send :: Serialize msg => Comm -> Rank -> Tag -> msg -> IO ()
recv :: Serialize msg => Comm -> Rank -> Tag -> IO (msg, Status)
```

The first three arguments of both functions are of the same type. The first argument is an abstract data type representing a communicator. In our example program we use the default `commWorld` for all messaging. The second argument specifies a process rank. In the case of `send`, it indicates the identity of receiver, whereas conversely, in the case of `recv`, it indicates the identity of the sender. The third argument is a tag which is useful for distinguishing different messages sent between the same processes. We do not need this feature in our program, so we have chosen to make it the dummy value `unitTag`, which is a synonym for `()`. However, in general, tags can be any enumerable type. The fourth argument of `send`, and the first component in the result of `recv`, is the message itself, which, in the simple interface of Haskell-MPI, can be any data type which is an instance of the `Serialize` type class from the cereal library [8]. Both functions return an

`IO` action as their result; the `send` action yields unit, and the `recv` action yields a tuple containing the received message and a status indicator. By default any errors in the MPI functions will cause the program to abort, but, by setting an appropriate error handler, you can change this behavior so that exceptions are raised instead.

It should be noted that `send` and `recv` are synchronous, which means that the calling process will block until the message has been successfully delivered. Non-blocking variants are also available which return immediately, allowing the processes to do other work while the message is in transit. A non-blocking receiver must poll for completion of the message before using its value.

Besides point-to-point messaging, MPI provides one-to-many, many-to-one and many-to-many communication primitives, capturing the majority of the typical real-world communication scenarios.

In addition to `send` and `recv` the program also calls three other MPI functions:

```
mpi :: IO () -> IO ()
commSize :: Comm -> IO Int
commRank :: Comm -> IO Rank
```

The first function takes an `IO` action as input (something which presumably uses other MPI features) and runs that action within an initialized MPI environment, before finalizing the environment at the end. The other functions allow a process to query the total number of processes in a communicator and the identity of the process within a communicator.

It might seem strange at first that the processes do not exchange any messages to determine how to divide up the work. This is unnecessary in our example because all of the important local information for each process can be deduced from its rank and the command line arguments, so no additional communication is required. In other more complex programs it is common to begin the program with an initialization phase, in which the master sends the configuration values of the computation to the other processes.

Another surprising aspect of our example is that every MPI process runs the same executable program, following the so-called Single Program Multiple Data (SPMD) style. MPI also allows the Multiple Program Multiple Data (MPMD) style, where different executables can be run by different processes — you can even mix programs written in different languages. However, the SPMD style is more common because it reduces development and deployment efforts. In the SPMD style you typically see blocks of code which are conditionally executed depending on the value of a the process rank. In our example, all processes call the `trapezoid` function, but only the master process calls `recv` and `print`, while all processes except for the master call `send`. If you use MPI in an eager language, such as C, and forget to make some of the computations or memory allocations

```
module Main where

import Control.Parallel.MPI.Simple
import System (getArgs)
import Trapezoid

main :: IO ()
main = mpi $ do
  numRanks <- commSize commWorld
  rank <- commRank commWorld
  let master = 0 :: Rank

  aStr:bStr:nStr:_ <- getArgs
  let [a,b] = map read [aStr,bStr]
      n = read nStr
      h = (b - a) / fromIntegral n
      localN = n `div` fromIntegral numRanks
      localA = a + fromIntegral rank * fromIntegral localN * h
      localB = localA + fromIntegral localN * h
      integral = trapezoid f localA localB localN h
  if rank == master then do
    rest <- sequence [ recv' commWorld (toRank proc) unitTag
                     | proc <- [1..numRanks-1] ]
    print (integral + sum rest)
    else send commWorld master unitTag integral
  where
    recv' comm rank tag = do
      (msg, status) <- recv comm rank tag
      return msg
```

**Listing 4:** Multi-node parallel program for calculating definite integrals, using point-to-point communication.

conditional, processes would spend time computing values that would not actually be used. Haskell, with its lazy evaluation and automatic memory management, makes it much easier to avoid such problems. Pure computations are simply not executed in processes that do not use their value, without requiring any explicit housekeeping code.

### Many-to-one communications

The use of the point-to-point communications in the previous section is workable but clumsy. Thankfully, this pattern of many-to-one communication is sufficiently common that MPI provides a convenient way to do it collectively:

```
gatherSend :: Serialize msg => Comm -> Rank -> msg -> IO ()
gatherRecv :: Serialize msg => Comm -> Rank -> msg -> IO [msg]
```

In this scenario the master process performs a `gatherRecv` while the others perform a `gatherSend`. The result of `gatherRecv` is an `IO` action that yields all the messages from each process in rank order. Note that the receiver also sends a message to itself, which appears in the first position of the output list.[2] The collective communications cannot be overlapping, so there is no need for a tag argument to distinguish between them.[3]

As you can see in Listing 5, the use of collective communications leads to a much more succinct implementation of the program. However, this is not their only virtue. An MPI implementation can take advantage of the underlying network hardware to optimize the collective operations, providing significant performance gains over point-to-point versions.

## Performance results

We now consider a small benchmark test to get a feeling for how well Haskell-MPI performs in practice. For comparison we ran the same test on three alternative implementations of the same program: the baseline sequential program, the multi-threaded Haskell version, and a C version which also uses MPI.

All performance tests were executed on an IBM iDataplex cluster, featuring 2.66GHz Intel Nehalem processors, with 8 cores and 24GB of RAM per node, running Red Hat Enterprise Linux 5, connected to a 40 Gb/s InfiniBand network. We used the following software versions to build the programs:

[2]It might seem strange that the receiver mentions its own rank and passes a message to itself. These oddities are required by a feature of MPI called intercommunicators. We do not discuss them in this article, but an interested reader can consult the MPI report for more details [3].

[3]See Chapter 5 of the MPI report [3].

```
module Main where

import Control.Parallel.MPI.Simple
import System (getArgs)
import Trapezoid

main :: IO ()
main = mpi $ do
  numRanks <- commSize commWorld
  rank <- commRank commWorld
  let master = 0 :: Rank

  aStr:bStr:nStr:_ <- getArgs
  let [a,b] = map read [aStr,bStr]
      n = read nStr
      h = (b - a) / fromIntegral n
      localN = n `div` fromIntegral numRanks
      localA = a + fromIntegral rank * fromIntegral localN * h
      localB = localA + fromIntegral localN * h
      integral = trapezoid f localA localB localN h
  if rank == 0
     then print . sum =<< gatherRecv commWorld master integral
     else gatherSend commWorld master integral
```

**Listing 5:** Multi-node parallel program for calculating definite integrals, using many-to-one communication.

1. GHC 7.0.3, with optimization flags `-O2`.
2. GCC 4.4.4, with optimization flags `-O2`.
3. Open MPI 1.4.2.

The benchmark test case computes the definite integral of Sine on the interval $[0, 1000\pi]$, using $10^9$ trapezoids. We had to choose a very large number of trapezoids to make it worth parallelizing this toy example at all.

| Haskell | | | | C | | | |
|---|---|---|---|---|---|---|---|
| method | cores | time(s) | scaling | method | cores | time(s) | scaling |
| MPI | 1 | 54.364 | – | MPI | 1 | 53.570 | – |
| MPI | 2 | 26.821 | 2.0 | MPI | 2 | 23.664 | 2.3 |
| MPI | 4 | 12.022 | 2.2 | MPI | 4 | 12.500 | 1.9 |
| MPI | 8 | 6.142 | 2.0 | MPI | 8 | 6.656 | 1.9 |
| MPI | 16 | 4.975 | 1.2 | MPI | 16 | 4.142 | 1.6 |
| MPI | 32 | 3.952 | 1.3 | MPI | 32 | 3.360 | 1.2 |
| MPI | 64 | 3.291 | 1.2 | MPI | 64 | 3.037 | 1.1 |
| MPI | 128 | 3.141 | 1.0 | MPI | 128 | 2.861 | 1.1 |
| MPI | 256 | 3.674 | 0.9 | MPI | 256 | 2.934 | 1.0 |
| sequential | 1 | 54.301 | – | | | | |
| threads | 1 | 48.866 | – | | | | |
| threads | 2 | 24.118 | 2.0 | | | | |
| threads | 4 | 12.080 | 2.0 | | | | |
| threads | 8 | 6.014 | 2.0 | | | | |

**Figure 2:** Performance figures for sequential, threaded and MPI versions of the trapezoid program, when integrating Sine on the interval $[0, 1000\pi]$ using $10^9$ trapezoids.

Figure 2 shows the raw benchmark results taken by averaging three simultaneous runs of the same computation. Figure 3 plots the results on a graph. The tests illustrate that we get strong scaling for all parallel implementations up to 8 cores. The performance improvement of the MPI implementations is fairly similar with a slight edge to the C version. Scaling tends to decline around the 16 core mark, although small improvements in overall performance are made up to 128 cores, but both programs begin to slow down at 256 cores. The threaded implementation stops at 8 cores because that is the maximum available on a single node in our test machine. Given more cores in a single node we might expect the threaded version to show similar improvements to the MPI versions. Having said that, the limitation of the threaded version to a single shared-memory instance will ultimately be a

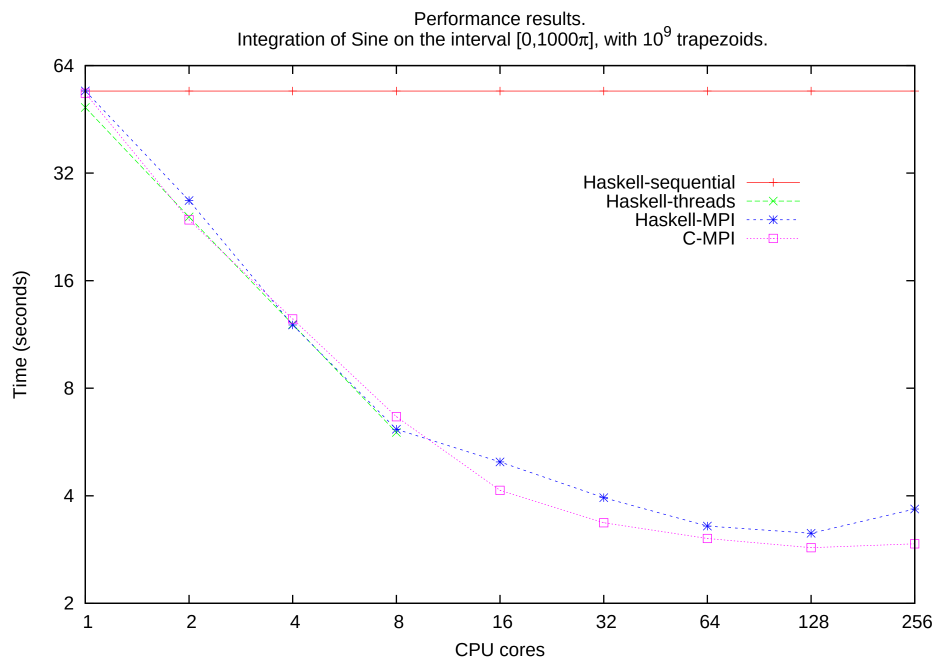

**Figure 3:** Performance comparison of sequential, threaded and MPI versions of the trapezoid program.

barrier to very large-scale parallelism on current hardware. However, there is nothing to stop you from using threads within an MPI application.

Obviously, we should not to pay too much heed to one toy benchmark. We would need a much bigger problem to show strong scaling beyond a dozen or so cores, and a truly gigantic problem to scale to the size of a machine such as LLNL's upcoming Sequoia!

### The Simple and Fast interfaces of Haskell-MPI

One of the biggest limitations of our test case is that we are only sending trivially small messages between processes (individual double precision floating point numbers). For larger messages the simple interface to Haskell-MPI imposes additional performance costs due to the need to make a copy of the data for serialization. Furthermore, in many cases, each message sent is preceded implicitly by another message carrying size information about the serialized data stream. For this reason Haskell-MPI provides an alternative "fast" interface which works on data types that have a contiguous in-memory representation, thus avoiding the need for serialization. `ByteString`s, unboxed arrays, and instances of the `Storable` type class can all be handled this way. The fast interface is more cumbersome to use, but it is a necessary evil for programs with large message sizes and tight performance constraints.

## Conclusion

Haskell-MPI provides a pragmatic way for Haskell programmers to write high performance programs in their favorite language today. The current version of the library covers the most commonly used parts of the MPI standard, although there are still several missing features, the most significant of which is parallel I/O. We plan to include more parts of the standard over time, with emphasis on those which are requested by users.

As you can see from the examples presented in this article, Haskell-MPI does not provide a particularly functional interface to users – most of the provided functions return `IO` results, and there is no satisfying way to send functions as messages. This reflects the modest ambitions of the project, but we may investigate more idiomatic APIs in future work.

## Acknowledgments

Dmitry's work on Haskell-MPI was funded by Microsoft Research and the Well-Typed LLP partnership as a part of the Parallel Haskell project.

All the performance tests for this article were executed on one of the x86 clusters at the Victorian Life Sciences Computation Initiative (VLSCI), in Melbourne, Australia [9]. We would like to thank the VLSCI for generously donating time on their computers for our work.

Michael Weber and Hal Daumé III provided an earlier Haskell binding to MPI called hMPI. Their work provided much inspiration for Haskell-MPI, especially in the early stages of development.

We would also like to thank Duncan Coutts, Ben Horsfall, Chris Samuel, Lee Naish, John Wagner, Edward Yang and Brent Yorgey for providing comments on earlier drafts of this article.

The full text of this article and all accompanying code is available on GitHub under the BSD3 license (see [10]).

## Appendix A: Installation and configuration

In order to use Haskell-MPI you need to have one of the MPI libraries installed on your computer. If you don't already have one installed, then Open MPI [4] is a good choice. We've tested Haskell-MPI with Open MPI 1.4.2 and 1.4.3 and MPICH 1.2.1 and 1.4, and there is a good chance it will work with other versions out of the box.

If the MPI libraries and header files are in the search paths of your C compiler, then Haskell-MPI can be built and installed with the command:[4]

```
cabal install haskell-mpi
```

Otherwise, the paths to the include files and libraries need to be specified manually:

```
cabal install haskell-mpi \
  --extra-include-dirs=/usr/include/mpi \
  --extra-lib-dirs=/usr/lib/mpi
```

If you are building the package with an MPI implementation other than Open MPI or MPICH, it is recommended to pass `-ftest` to `cabal install`, running the test-suite to verify that the bindings perform as expected. If you have problems configuring MPI, take a look at the useful hints in the Haddock documentation for the module `Testsuite`.

## References

[1] Sequoia. `https://asc.llnl.gov/computing_resources/sequoia/`.

[4]MPICH v1.4 requires extra argument `-fmpich14`

[2] Top 500 website. `http://www.top500.org/`.

[3] Official MPI report and other documents. `http://www.mpi-forum.org/docs/`.

[4] Open MPI. `http://www.open-mpi.org/`.

[5] MPICH. `http://www.mcs.anl.gov/research/projects/mpich2/`.

[6] Peter S. Pacheco. **Parallel Programming with MPI**. Morgan Kaufman Publishers, Inc., San Francisco, California, USA (1997).

[7] Parallel library on Hackage. `http://hackage.haskell.org/package/parallel`.

[8] Cereal library on Hackage. `http://hackage.haskell.org/package/cereal`.

[9] VLSCI. `http://www.vlsci.org.au/`.

[10] Article text and code. `https://github.com/bjpop/mpi-article-monad-reader`.

# Coroutine Pipelines

by Mario Blažević

*The basic idea of trampoline-style execution is well known and has already been explored multiple times, every time leading in a different direction. The recent popularity of iteratees leads me to believe that the time has come for yet another expedition. If you're not inclined to explore this territory on your own, the* **`monad-coroutine`** *and* **SCC** *packages [1, 2] provide a trodden path.*

## Trampolining a monad computation

I won't dwell on the basics of trampoline-style execution, because it has been well covered elsewhere [3, 4]. Let's jump in with a simple monad transformer that allows the base monad to pause its computation at any time, shown in Listing 6.

Once lifted on a Trampoline, in a manner of speaking, a computation from the base monad becomes a series of alternating bounces and pauses. The bounces are the uninterruptible steps in the base monad, and the during the pauses the trampoline turns control over to us. Function *run* can be used to eliminate all the pauses and restore the original, un-lifted computation. Here's a little example session in GHCi (this example, and all following, are shown with newlines for readability, but must actually be entered into GHCi all one one line):

```
*Main> let hello = do { lift (putStr "Hello, ")
                      ; pause
                      ; lift (putStrLn "World!") }
*Main> run hello
Hello, World!
```

Alternatively, we can just *bounce* the computation once, use the pause to perform some other work, and then continue the trampoline:

```
{-# LANGUAGE FlexibleContexts, Rank2Types, ScopedTypeVariables #-}
import Control.Monad (liftM)
import Control.Monad.Trans (MonadTrans (..))
newtype Trampoline m r = Trampoline {
  bounce :: m (Either (Trampoline m r) r)
}
instance Monad m => Monad (Trampoline m) where
  return = Trampoline . return . Right
  t >>= f = Trampoline (bounce t
                         >>= either
                           (return . Left . (>>=f))
                           (bounce . f))
instance MonadTrans Trampoline where
  lift = Trampoline . liftM Right
pause :: Monad m => Trampoline m ()
pause = Trampoline (return $ Left $ return ())
run :: Monad m => Trampoline m r -> m r
run t = bounce t >>= either run return
```

**Listing 6:** The Trampoline monad transformer

```
*Main> do { Left continuation <- bounce hello
          ; putStr "Wonderful "
          ; run continuation }
Hello, Wonderful World!
```

Though all examples in this article will be using the IO monad for brevity, keep in mind that this is a monad transformer which can be applied to any monad whatsoever.

The most interesting thing the trampoline transformer gives us is the ability to run multiple interleaved computations. The function *mzipWith* defined below – I find it more practical than plain *mzip* [5] – interleaves two trampolines, and then we use it to interleave an arbitrary number of them.

```
mzipWith :: Monad m
         ⇒ (a → b → c)
         → Trampoline m a → Trampoline m b → Trampoline m c
mzipWith f t1 t2 = Trampoline (liftM2 bind (bounce t1) (bounce t2))
  where
    bind (Left  a) (Left  b) = Left  (mzipWith f a b)
    bind (Left  a) (Right b) = Left  (mzipWith f a (return b))
    bind (Right a) (Left  b) = Left  (mzipWith f (return a) b)
    bind (Right a) (Right b) = Right (f a b)
interleave :: Monad m ⇒ [Trampoline m r] → Trampoline m [r]
interleave = foldr (mzipWith (:)) (return [])
```

When we apply *interleave* to a list of trampolines, it combines them all into a single trampoline that bounces all the computations together. I wish I had an appropriate video to insert here, but the best I can offer is this GHCi output:

```
*Main> run $ interleave [hello, hello, hello]
Hello, Hello, Hello, World!
World!
World!
[(),(),()]
```

If the base monad happens to support parallel execution, we have the option of bouncing all the trampolines in parallel instead of interleaving them. All we need to do is import the *liftM2* function from the `monad-parallel` package [6] instead of using the default one from `base`.

The amount of parallelism we can gain this way depends on how close the different trampolines' bounces are to each other in duration. The function *interleave* will wait for all trampolines to complete their first bounces before it initiates their second bounces. This is cooperative, rather than preemptive multi-tasking.

Because of their ability of interleaved execution, trampoline computations are an effective way to implement **coroutines**, and that is what we'll call them from now on. Note, however, that the *hello* coroutines we have run are completely independent. This is comparable to an operating system running many sandboxed processes completely unaware of each other. Though the processes are concurrent, they cannot cooperate. Before we remedy this failing, let's take a closer look at what a coroutine can accomplish during the pause.

## Suspension functors

### Generators

During the 1970s, coroutines were actively researched and experimented with [7, 8, 9], but support for them disappeared from later mainstream general-purpose languages for a long time. Rejoice now, because the Dark Ages of the coroutines are coming to an end.

This renaissance has started with a limited form of coroutine called a **generator**, which has become a part of JavaScript, Python, and Ruby, among other recent programming languages. A generator (Listing 7) is just like a regular trampoline except it **yields** a value whenever it suspends.

The difference between the old function *run*, that we've used for running a Trampoline, and the new function *runGenerator* is that the latter collects all values that its generator argument yields:

```
*Main> let gen = do { lift (putStr "Yielding one, ")
                    ; yield 1
                    ; lift (putStr "then two, ")
                    ; yield 2
                    ; lift (putStr "returning three: ")
                    ; return 3 }
*Main> runGenerator gen
Yielding one, then two, returning three: ([1,2],3)
```

### Iteratees

A generator is thus a coroutine whose every suspension provides not only the coroutine resumption but also a single value. We can easily define a monad transformer dual to generators, whose suspension demands a value instead of yielding it (Listing 8). The terminology for this kind of coroutine is not as well established

```
newtype Generator a m x = Generator {
  bounceGen :: m (Either (a, Generator a m x) x)
}
instance Monad m ⇒ Monad (Generator a m) where
  return = Generator ∘ return ∘ Right
  t >>= f = Generator (bounceGen t
                          >>= either
                            (λ(a, cont) →
                               return $ Left (a, cont >>= f))
                            (bounceGen ∘ f))
instance MonadTrans (Generator a) where
  lift = Generator ∘ liftM Right
yield :: Monad m ⇒ a → Generator a m ()
yield a = Generator (return $ Left (a, return ()))
runGenerator :: Monad m ⇒ Generator a m x → m ([a], x)
runGenerator = run′ id where
  run′ f g = bounceGen g
              >>= either
                (λ(a, cont) → run′ (f ∘ (a:)) cont)
                (λx → return (f [], x))
```

**Listing 7:** Generators

as for generators, but in the Haskell community the name **iteratee** appears to be the most popular.

```
newtype Iteratee a m x = Iteratee {
  bounceIter :: m (Either (a → Iteratee a m x) x)
}
instance Monad m ⇒ Monad (Iteratee a m) where
  return  = Iteratee ∘ return ∘ Right
  t ≫= f = Iteratee (bounceIter t
                       ≫= either
                          (λcont → return $ Left ((≫=f) ∘ cont))
                          (bounceIter ∘ f))
instance MonadTrans (Iteratee a) where
  lift = Iteratee ∘ liftM Right
await :: Monad m ⇒ Iteratee a m a
await = Iteratee (return $ Left return)
runIteratee :: Monad m ⇒ [a] → Iteratee a m x → m x
runIteratee (a : rest) i = bounceIter i
                             ≫= either
                                (λcont → runIteratee rest (cont a))
                                return
runIteratee [] i = bounceIter i
                     ≫= either
                          (λcont → runIteratee []
                             (cont $ error "No more values to feed."))
                          return
```

**Listing 8:** Iteratees

To run a monad thus transformed, we have to supply it with values:

```
*Main> let iter = do { lift (putStr "Enter two numbers: ")
                     ; a <- await
                     ; b <- await
                     ; lift (putStrLn ("sum is " ++ show (a + b))) }
*Main> runIteratee [4, 5] iter
Enter two numbers: sum is 9
```

## Generalizing the suspension type

Let's take another look at the three kinds of coroutines we have defined so far. Their definitions are very similar; the only differences stem from the first argument of the Either constructor, which always contains the coroutine resumption but wrapped in different ways: plain resumption in the case of Trampoline; $(x, \textit{resumption})$ for Generator; and $x \rightarrow \textit{resumption}$ in the case of Iteratee. All three wrappings, not coincidentally, happen to be functors. It turns out that just knowing that the suspension is a functor is sufficient to let us define the trampolining monad transformer. The time has come to introduce the generic Coroutine data type (Listing 9).

```
newtype Coroutine s m r = Coroutine {
  resume :: m (Either (s (Coroutine s m r)) r)
}
instance (Functor s, Monad m) ⇒ Monad (Coroutine s m) where
  return x = Coroutine (return (Right x))
  t >>= f  = Coroutine (resume t
                         >>= either
                           (return ∘ Left ∘ fmap (>>=f))
                           (resume ∘ f))
instance Functor s ⇒ MonadTrans (Coroutine s) where
  lift = Coroutine ∘ liftM Right
suspend :: (Monad m, Functor s) ⇒
  s (Coroutine s m x) → Coroutine s m x
suspend s = Coroutine (return (Left s))
```

**Listing 9:** The generic Coroutine transformer

The Coroutine type constructor has three parameters: the functor type for wrapping the resumption, the base monad type, and the monad's return type. We can now redefine our three previous coroutine types as mere type aliases, differing only in the type of functor in which they wrap their resumption (Listing 10). All these definitions and more can also be found in the `monad-coroutine` package [1].

## Other possible suspension types

Any functor can serve as a new coroutine suspension type, though some would be more useful than others. Of the three Functor instances from the Haskell 98

```
import Data.Functor.Identity (Identity (..))
type Trampoline  m x = Coroutine Identity  m x
type Generator a m x = Coroutine ((,) a)   m x
type Iteratee    a m x = Coroutine ((→) a) m x
pause :: Monad m ⇒ Trampoline m ()
pause = suspend (Identity $ return ())
yield :: (Monad m, Functor ((,) x)) ⇒ x → Generator x m ()
yield x = suspend (x, return ())
await :: (Monad m, Functor ((,) x)) ⇒ Iteratee x m x
await = suspend return
run :: Monad m ⇒ Trampoline m x → m x
run t = resume t ≫= either (run ∘ runIdentity) return
runGenerator :: Monad m ⇒ Generator x m r → m ([x], r)
runGenerator = run′ id where
  run′ f g = resume g
     ≫= either
       (λ(x, cont) → run′ (f ∘ (x:)) cont)
       (λr → return (f [], r))
runIteratee :: Monad m ⇒ [x] → Iteratee x m r → m r
runIteratee (x : rest) i =
  resume i ≫= either (λcont → runIteratee rest (cont x)) return
runIteratee [] i =
  resume i
   ≫= either
    (λcont → runIteratee [] (cont $ error "No more values to feed."))
    return
```

**Listing 10:** Redefining examples in terms of Coroutine

Prelude, IO, Maybe and [], only the list functor would qualify as obviously useful. A list of resumptions could be treated as offering a choice of resumptions or as a collection of resumptions that all need to be executed.

We can get another practical example of a suspension functor if we combine the generator's functor which yields a value and iteratee's which awaits one. The result can be seen as a **request** suspension: the coroutine supplies a request and requires a response before it can proceed.

```
data Request request response x = Request request (response → x)
instance Functor (Request x f) where
  fmap f (Request x g) = Request x (f ∘ g)
request :: Monad m ⇒ x → Coroutine (Request x y) m y
request x = suspend (Request x return)
```

As noted above, the Request functor is just a composition of the two functors used for generators and iteratees. More generally, any two functors can be combined into their composition or sum:

```
-- From the transformers package
newtype Compose f g a = Compose {getCompose :: f (g a)}
instance (Functor f, Functor g) ⇒ Functor (Compose f g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)
data EitherFunctor l r x = LeftF (l x) ∨ RightF (r x)
instance (Functor l, Functor r) ⇒ Functor (EitherFunctor l r) where
  fmap f (LeftF l)  = LeftF (fmap f l)
  fmap f (RightF r) = RightF (fmap f r)
```

If we use these type constructors, we can redefine Request as

```
type Request a b = Compose ((,) a) ((→) b).
```

We can also define a sum of functors like

```
type InOrOut a b = EitherFunctor ((,) a) ((→) b)
```

to get a coroutine which can either demand or supply a value every time it suspends, but not both at the same time as Request does.

## Relating the suspension types

Having different types of coroutines share the same Coroutine data type parameterized by different resumption functors isn't only useful for sharing a part of their implementation. We can also easily map a computation of one coroutine type into another:

```
mapSuspension :: (Functor s, Monad m) ⇒
  (∀ a. s a → s′ a) → Coroutine s m x → Coroutine s′ m x
mapSuspension f cort = Coroutine {resume = liftM map′ (resume cort)}
  where map′ (Right r) = Right r
    map′ (Left s) = Left (f $ fmap (mapSuspension f) s)
```

This functionality will come in handy soon.

## Communicating coroutines

We should now revisit the problem of running multiple coroutines while letting them communicate. After all, if they cannot cooperate they hardly deserve the name of coroutines.

The simplest solution would be to delegate the problem to the base monad. Coroutines built on top of the monad IO or State can use the monad-specific features like MVars to exchange information, like threads normally do. Apart from tying our code to the specific monad, this solution would introduce an unfortunate mismatch between the moment of communication and the moment of coroutine switching.

Another way to let our coroutines communicate is to explicitly add a communication request to their suspension functor. We could, for example, extend the suspension functor with the ability to request a switch to another coroutine. Together with a scheduler at the top level, this approach gives us classic symmetric coroutines [4], or non-preemptive green threads.

This time, I want to explore some different ways to let coroutines communicate. Explicit transfers of control between coroutines, though they're easier to handle, share many of the problems that plague thread programming.

### Producer-consumer

One of the most popular and useful examples of communicating coroutines is a producer-consumer coroutine pair. The producer coroutine **yields** values which the consumer coroutine **awaits** to process. This pair also happens to be an example of two coroutines that are easy to run together, because their two suspension functors are dual to each other.

The first function shown in Listing 11, *pipe1*, assumes that the producer always yields at least as many values as the consumer awaits. If that is not the case and the producer may end its execution before the consumer does, *pipe2* should be used

```
-- a helper function that really belongs in Control.Monad
bindM2 :: Monad m ⇒ (a → b → m c) → m a → m b → m c
bindM2 f ma mb = do {a ← ma; b ← mb; f a b}
pipe1 :: Monad m ⇒ Generator a m x → Iteratee a m y → m (x, y)
pipe1 g i = bindM2 proceed (resume g) (resume i) where
  proceed (Left (a, c)) (Left f)  = pipe1 c (f a)
  proceed (Left (a, c)) (Right y) = pipe1 c (return y)
  proceed (Right x)     (Left f)  = error "The producer ended too soon."
  proceed (Right x)     (Right y) = return (x, y)
pipe2 :: Monad m ⇒ Generator a m x → Iteratee (Maybe a) m y → m (x, y)
pipe2 g i = bindM2 proceed (resume g) (resume i) where
  proceed (Left (a, c)) (Left f)  = pipe2 c (f $ Just a)
  proceed (Left (a, c)) (Right y) = pipe2 c (return y)
  proceed (Right x)     (Left f)  = pipe2 (return x) (f Nothing)
  proceed (Right x)     (Right y) = return (x, y)
```

**Listing 11:** A producer-consumer pair

instead. This function informs the consumer of the producer's death by supplying Nothing to the consumer's resumption. The following GHCi session shows the difference:

```
*Main> pipe1 gen iter
Yielding one, Enter two numbers: then two, returning three: sum is 3
(3,())
*Main> pipe1 (return ()) iter
Enter two numbers: *** Exception: The producer ended too soon.
*Main> let iter2 s = lift (putStr "Enter a number: ") >> await
                     >>= maybe (lift (putStrLn ("sum is " ++ show s)))
                               (\n -> iter2 (s + n))
*Main> pipe2 (return ()) (iter2 0)
Enter a number: sum is 0
((),())
*Main> pipe2 (gen >> gen) (iter2 0)
Yielding one, Enter a number: then two, Enter a number:
returning three: Yielding one, Enter a number: then two,
Enter a number: returning three: Enter a number: sum is 6
(3,())
```

The first clause of the helper function *proceed* handles the most important case, where both coroutines suspend and can be resumed. The remaining three cover the cases where one or both coroutines return.

Our definition of *bindM2* happens to first resume the producer, waits until its step is finished, and only then resume the consumer; however, we could replace it by the same-named function from the `monad-parallel` package and step the two coroutines in parallel.

When we compare *pipe* to our earlier *interleave* function or any other symmetrical coroutine scheduler, the first thing to note is that the two coroutines now have different types corresponding to their different roles. We are beginning to use Haskell's type system to our advantage.

Another thing to note is that producer-consumer pair synchronizes and exchanges information whenever the next *yield*/*await* suspension pair is ready. There can be no race conditions nor deadlocks. That in turn means we need no locking, mutexes, semaphores, transactions, nor the rest of the bestiary.

## Transducers in the middle

The producer-consumer pair is rather limited, but it's only the minimal example of a coroutine pipeline. We could insert more coroutines in the middle of the pipeline, **transducer** coroutines [10], also known as **enumeratees** lately. A transducer has two operations available to it, *awaitT* for receiving a value from upstream and *yieldT* to pass a value downstream. The input and output values may have different types. The *awaitT* function returns Nothing if there are no more upstream values to receive.

```
type Transducer a b m x
   = Coroutine (EitherFunctor ((→) (Maybe a)) ((, ) b)) m x
awaitT :: Monad m ⇒ Transducer a b m (Maybe a)
awaitT = suspend (LeftF return)
yieldT :: Monad m ⇒ b → Transducer a b m ()
yieldT x = suspend (RightF (x, return ()))
```

Although transducers can lift arbitrary operations from the base monad, few of them need any side-effects in practice. Any pure function can be lifted into a transducer. Even a stateful transducer can be pure, in the sense that it doesn't need to rely on any base monad operation.

```
lift121 :: Monad m ⇒ (a → b) → Transducer a b m ()
lift121 f = awaitT
              >>= maybe (return ()) (λa → yieldT (f a) >> lift121 f)
```

```
liftStateless :: Monad m ⇒ (a → [b]) → Transducer a b m ()
liftStateless f = awaitT
                    ≫= maybe (return ()) (λa → mapM_ yieldT (f a)
                                                   ≫ liftStateless f)
liftStateful :: Monad m ⇒ (state → a → (state, [b]))
           → (state → [b]) → state → Transducer a b m ()
liftStateful f eof s = awaitT
                          ≫= maybe
                             (mapM_ yieldT (eof s))
                             (λa → let (s′, bs) = f s a
                                   in  mapM_ yieldT bs
                                         ≫ liftStateful f eof s′)
```

## Transducer piping

Now we have the means to create lots of transducer coroutines, but how can we use them? One possibility would be to define a new variant of the *pipe* function that would run the entire pipeline and reduce it to the base monad:

```
pipe3 :: Monad m ⇒ Generator a m x → Transducer a b m y
   → Iteratee (Maybe b) m z → m (x, y, z)
```

This solution would be very limited, as it could not handle more than the one transducer in the pipeline. As usual, big problems require modular solutions. In this case, the solution is a function that combines two transducers into one, piping the output of the first into the input of the second transducer (Listing 12).

We could also define similar operators for combining a generator/transducer or a transducer/iteratee pair into another generator or iteratee, respectively. Instead of doing that, however, we can provide functions for casting generators, iteratees, and plain trampolines into transducers and back (Listing 13).

The empty data type Naught should not be exposed to the end-user. Its purpose is to ensure we don't get a run-time error by attempting to cast a non-trivial transducer into a simpler coroutine it cannot fit.

Using the piping combinator ⋙=, we can easily construct a coroutine pipeline of unlimited length and run it as a simple generator, iteratee, or plain trampoline, all the while retaining the type safety and synchronization guarantees.

```
(=>=) :: ∀ a b c m x y. Monad m
      ⇒ Transducer a b m x → Transducer b c m y
      → Transducer a c m (x, y)
t1 =>= t2 = Coroutine (bindM2 proceed (resume t1) (resume t2)) where
  proceed (Left (LeftF s)) c =
    return (Left $ LeftF $ fmap (=>= Coroutine (return c)) s)
  proceed c (Left (RightF s)) =
    return (Left $ RightF $ fmap (Coroutine (return c) =>=) s)
  proceed (Left (RightF (b, c1))) (Left (LeftF f)) =
    resume (c1 =>= f (Just b))
  proceed (Left (RightF (b, c1))) (Right y) =
    resume (c1 =>= (return y :: Transducer b c m y))
  proceed (Right x) (Left (LeftF f)) =
    resume ((return x :: Transducer a b m x) =>= f Nothing)
  proceed (Right x) (Right y) =
    return $ Right (x, y)
```

**Listing 12:** Composing transducers

```
data Naught
fromGenerator :: Monad m ⇒
   Generator a m x → Transducer Naught a m x
fromGenerator = mapSuspension RightF
fromIteratee :: Monad m ⇒
   Iteratee (Maybe a) m x → Transducer a Naught m x
fromIteratee = mapSuspension LeftF
toGenerator :: Monad m ⇒ Transducer Naught a m x → Generator a m x
toGenerator = mapSuspension (λ(RightF a) → a)
toIteratee :: Monad m ⇒
   Transducer a Naught m x → Iteratee (Maybe a) m x
toIteratee = mapSuspension (λ(LeftF a) → a)
toTrampoline :: Monad m ⇒
   Transducer Naught Naught m x → Trampoline m x
toTrampoline = mapSuspension ⊥
```

**Listing 13:** Converting to and from transducers

```
*Main> let double = liftStateless (\a -> [a, a])
*Main> runGenerator (toGenerator $ fromGenerator gen =>= double)
Yielding one, then two, returning three: ([1,1,2,2],(3,()))
*Main> runIteratee [Just 3, Nothing]
                 (toIteratee $ double =>= fromIteratee (iter2 0))
Enter a number: Enter a number: Enter a number: sum is 6
((),())
*Main> run (toTrampoline $
             fromGenerator (yield 3) =>=
               double =>= fromIteratee (iter2 0))
Enter a number: Enter a number: Enter a number: sum is 6
(((),()),())
*Main> run (toTrampoline $
             fromGenerator (yield 3) =>=
               double =>= double =>= fromIteratee (iter2 0))
Enter a number: Enter a number: Enter a number: Enter a number:
Enter a number: sum is 12
((((),()),()),())
```

## Parting and joining the stream

The next extension would be to generalize the coroutine pipeline by allowing it to sprout branches and form a non-linear data-flow network. Every node of the network is a coroutine that communicates only with its neighbours. For example, we could have a stream-branching coroutine with a single *await* suspension and two different *yield* suspensions that feed two different downstream coroutines. Or dually, we can imagine a join coroutine that can *await* value from two different input coroutines, combine the two streams and *yield* to a single downstream coroutine (Listing 14).

The main reason that the input and output of the Splitter and Join coroutines are declared to be the same is to enforce the division of labour. If there is any data conversion to be done, the task should be given to a Transducer. The Splitter and Join coroutines are to be used for splitting and merging the streams, without modifying any individual item. This helps keep the number of possible combinators under control.

The stream-branching coroutine type is especially interesting because it can be used to implement conditionals as coroutines which stream all their upstream data unmodified to their two outputs. All data output to one downstream branch would be treated as satisfying the coroutine's condition, and vice versa. Once we have these coroutine components, we can combine them with combinators based on Boolean logic, if... then... else combinator, and others [11].

```
type Splitter a m x =
  Coroutine (EitherFunctor ((→) (Maybe a))
    (EitherFunctor ((, ) a) ((, ) a))) m x
type Join a m x =
  Coroutine (EitherFunctor (EitherFunctor ((→) (Maybe a))
    ((→) (Maybe a)))
    ((, ) a)) m x
yieldLeft :: Monad m ⇒ a → Splitter a m ()
yieldLeft a = suspend (RightF $ LeftF (a, return ()))
yieldRight :: Monad m ⇒ a → Splitter a m ()
yieldRight a = suspend (RightF $ RightF (a, return ()))
awaitLeft :: Monad m ⇒ Join a m (Maybe a)
awaitLeft = suspend (LeftF $ LeftF return)
awaitRight :: Monad m ⇒ Join a m (Maybe a)
awaitRight = suspend (LeftF $ RightF return)
```

**Listing 14:** Splitter and Join

```
ifThenElse :: Monad m ⇒ Splitter a m x → Transducer a b m y
           → Transducer a b m z → Transducer a b m (x, y, z)
not        :: Monad m ⇒ Splitter a m x → Splitter a m x
and        :: Monad m ⇒ Splitter a m x → Splitter a m x → Splitter a m x
or         :: Monad m ⇒ Splitter a m x → Splitter a m x → Splitter a m x
groupBy    :: Monad m ⇒ Splitter a m x → Transducer a [a] m x
any        :: Monad m ⇒ Splitter a m x → Splitter [a] m x
all        :: Monad m ⇒ Splitter a m x → Splitter [a] m x
```

The *ifThenElse* combinator sends all true output of the argument splitter to one transducer, all false output to the other, and merges together the transducers' outputs in the order they appear. The result behaves as a transducer. The *groupBy* combinator takes every contiguous section of the stream that its argument splitter considers true and packs it into a list, discarding all parts of the input that the splitter sends to its false output. The *any* combinator considers each input list true iff its argument splitter deems any of it true. The *all* combinator is equivalent, except it considers the entire list false iff the its argument considers any of it false. Many other splitter combinators beside these can be found in the `scc` package [2].

While the number of primitive splitters is practically unlimited, corresponding to the number of possible yes/no questions that can be asked of any section of the input, there are relatively few interesting primitive joins. The two input streams can be fully concatenated one after the other, they can be interleaved in some way, or their individual items can be combined together. The set of possible combinators for the Join coroutines is to some extent dual to the set of the Splitter combinators, but without the helpful analogy with the Boolean logic.

```
joinTransduced :: Monad m ⇒ Transducer a b m x → Transducer a b m y
                     → Join b m z → Transducer a b m (x, y, z)
flipJoin :: Join a m x → Join a m x
```

## Pipeline examples

The following pipeline generates all lines of the file `input.txt` that contain the string `FIND ME`:

```
toGenerator (fromGenerator (readFile "input.txt")
   >-> groupBy line
   >-> ifThenElse (any $ substring "FIND ME")
                  (lift121 id)
                  (fromIteratee suppress)
   >-> concatenate)
```

To get the effect of `grep -n`, with each line prefixed by its ordinal number, we can use the following generator instead:

```
toGenerator (joinTransduced
               (joinTransduced (fromGenerator naturals >-> toString)
                               (fromGenerator $ repeat ": ")
                               zipMonoids)
               (fromGenerator (readFile "input.txt")
                  >-> groupBy line)
               zipMonoids)
             >-> ifThenElse (any $ substring "FIND ME")
                            (lift121 id)
                            (fromIteratee suppress)
             >-> concatenate)
```

Here are the type signatures of all the primitive coroutines used above. Their names are mostly evocative enough to explain what each coroutine does. The *line* splitter sends the line-ends to its false output, and all line characters to its true output. The *zipMonoids* join outputs the result of *mappend* of each pair of items it reads from its two inputs. It returns as soon as either of its input streams ends.

```
line        :: Monad m ⇒ Splitter Char m ()
readFile    :: FilePath → Generator Char IO ()
naturals    :: Monad m ⇒ Generator Int m ()
toString    :: (Show a, Monad m) ⇒ Transducer a String m ()
concatenate :: Monad m ⇒ Transducer [a] a m ()
zipMonoids  :: (Monad m, Monoid a) ⇒ Join a m ()
repeat      :: Monad m ⇒ a → Generator a m ()
suppress    :: Monad m ⇒ Iteratee a m ()
substring   :: Monad m ⇒ [a] → Splitter a m ()
```

# Final overview

We have now seen what the trampoline-style stream processing looks like. It feels like a paradigm of its own while programming, subjectively speaking, but that shouldn't prevent us from comparing it to alternatives and finding its place in the vast landscape of programming techniques.

## Coroutines

First, this is not a domain-specific programming language that can stand on its own. Every single coroutine listed above relies on plain Haskell code to accomplish its task. That remains true even for coroutines that are largely composed of smaller coroutines, though the proportion of the host-language code gets smaller and smaller as the pipeline complexity grows.

The concept of coroutines in general makes sense only in the presence of state. If your Haskell code never uses any monads, you've no use for coroutines.

Coroutines are an example of cooperative multitasking, as opposed to threads where the multitasking is preemptive. If a single coroutine enters an infinite loop, the entire program is stuck in the infinite loop.

## Data-driven coroutines

The specific coroutine design presented above has some additional characteristics which are not typical of other coroutine designs. A coroutine can resume only its neighbours, not just any coroutine. Furthermore, to resume a coroutine one must satisfy the interface of its suspension: an iteratee, for example, cannot be resumed without feeding it a value. This requirement is checked statically, so no run-time error can happen like, for example, in Lua [12].

Some coroutines, like Transducer for example, can suspend in more than one way. The interface for resuming such a coroutine depends on how it last suspended – whether it was *awaitT* or *yieldT*. The correct resumption is still statically ensured, however, because the two types of suspension always switch to two different coroutines, and each of them has only the one correct way of resuming the suspended coroutine.

The price of this safety is the inability to implement any pipeline that forms a cyclic graph. Consider for example a pipeline, unfortunately impossible in this system, that performs a streaming merge sort of the input. This would be a transducer consisting of the following four coroutines arranged in a diamond configuration:

- a stream splitter that directs odd input items to one output and even items to the other,
- two transducer coroutines, **odd** and **even**, recursively sorting the two halves of the stream, and
- a join coroutine with a single output and two inputs fed from **odd** and **even**.

If this configuration was possible, the following sequence of events would become possible as well: **splitter** → **odd** → **join** → **even** → **splitter** → **even** → **join** → **even** → **splitter** → **odd**. The transducer **odd** suspends by yielding a value to **join**, but the next time it gets resumed, it's not **join** waking it up to ask for another value, it's **splitter** handing it a value instead! The resumption created by *yield* does not expect and cannot handle any value, so it would have to raise a run-time error.

### Iteratee libraries

Oleg Kiselyov's Iteratee design [13] and various Haskell libraries based on it [14, 15, 16] are in essence asymmetrical coroutines, where only the Iteratee data type can suspend. In comparison, the coroutine design presented here is more generic but, being expository, not as optimized. The existence on Hackage of three different implementations of the same basic design indicates that the sweet spot has not been found yet. I hope that the present paper helps clarify the design space.

## References

[1] Mario Blažević. The `monad-coroutine` package. `http://hackage.haskell.org/package/monad-coroutine`.

[2] Mario Blažević. The `SCC` package. `http://hackage.haskell.org/package/scc`.

[3] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In **ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming**, pages 18–27. ACM, New York, NY, USA (1999).

[4] William L. Harrison. The essence of multitasking. In **Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology, volume 4019 of Lecture Notes in Computer Science**, pages 158–172. Springer (2006).

[5] Tomas Petricek. Fun with parallel monad comprehensions. **The Monad Reader**, pages 17–41 (2011). `http://themonadreader.files.wordpress.com/2011/07/issue18.pdf`.

[6] Mario Blažević. The `monad-parallel` package. `http://hackage.haskell.org/package/monad-parallel`.

[7] Leonard I. Vanek and Rudolf Marty. Hierarchical coroutines, a mechanism for improved program structure. In **ICSE '79: Proceedings of the 4th international conference on Software engineering**, pages 274–285. IEEE Press, Piscataway, NJ, USA (1979).

[8] Pal Jacob. A short presentation of the SIMULA programming language. **SIGSIM Simul. Dig.**, 5:pages 19–19 (July 1974). `http://doi.acm.org/10.1145/1102704.1102707`.

[9] Christopher D. Marlin. **Coroutines**. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1980).

[10] Olin Shivers and Matthew Might. Continuations and transducer composition. In **Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation**, pages 295–307. PLDI '06, ACM, New York, NY, USA (2006). `http://matt.might.net/papers/might2006transducers.pdf`.

[11] Mario Blažević. Streaming component combinators (2006). `http://conferences.idealliance.org/extreme/html/2006/Blazevic01/EML2006Blazevic01.html`. Extreme Markup Languages 2006.

[12] Ana Lúcia De Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. **Journal of Universal Computer Science**, 10:page 925 (2004).

[13] Oleg Kiselyov. Incremental multi-level input processing and collection enumeration (2011). `http://okmij.org/ftp/Streams.html`.

[14] Oleg Kiselyov and John W. Lato. The `iteratee` package. `http://hackage.haskell.org/package/iteratee`.

[15] David Mazieres. The `iterIO` package. `http://hackage.haskell.org/package/iterIO`.

[16] John Millikin. The `enumerator` package. `http://hackage.haskell.org/package/enumerator`.