Parallel Functional Programming Lab C
Data Parallel Haskell: A Tutorial For the Curious Haskeller

Jens Katelaan, Dmytro Lypai & Baldur Blöndal

May 7, 2013

1 Introduction

You have decided to read a tutorial on Data Parallel Haskell, so chances are that you are interested in writing parallel Haskell programs. Great idea! Writing parallel programs is *so* easy in Haskell![1]

So what is this Data Parallel Haskell (DPH)? It's an extension to the Glasgow Haskell Compiler as well as a set of libraries to support the programming model of *nested data parallelism*. Now everything's crystal clear, right? Well, maybe we should take a step back. As you might already be aware, there are two basic approaches to parallel computing: *control parallelism*, also known as task parallelism, and *data parallelism*, which is suspiciously present in the title of this tutorial. It is kind of important to understand these distinctions to figure out if DPH is the right tool for you, so let's take a minute or two to get these things straight.

The idea behind control parallelism is that you split your program into independent parts that can then be executed in parallel. There are several approaches to control parallelism in Haskell, including par/pseq spark parallelism, Parallel Strategies, and the Par monad [9, 6, 7].

Control parallelism is a very flexible approach to parallel programming, as you (and the compiler) have full control over what should be parallelized. Sounds great, right? So it's a silver bullet and reading a tutorial on data parallelism is a waste of time, right? Well, no.[2]

The problem with control parallelism is that it is terribly complicated. You need multi-threaded execution models, need to figure out what parts of your computation you can reasonably parallelize, keep track of the dependencies between the execution threads etc.

Depending on the application you're writing, this might be the only sensible approach (think: Web server). In many other cases, however, we want parallelism just because we're dealing with large amounts of data or lots of expensive numeric computations. In such cases it might turn out to be very difficult to make control parallelism scale well, or obtain decent parallelism at all, because there are a lot of details to get right: How do we get sufficiently fine-grained

[1]As long as you don't expect them to actually run faster than your sequential programs.

[2]We've put so much effort in it, you better read it! You can still go back to using the Par monad after finishing this tutorial.

parallelism? How can we avoid communication that introduces dependencies between threads so that they are blocked? Haskell's lazy evaluation often does not exactly help when reasoning about these things.

The alternative is data parallelism: an approach to parallel computations where the same computation is performed on different pieces of data. Mapping a function over an array, for example. This might seem like a rather annoying restriction – and it sometimes is – but it also results in much more predictable and possibly more fine-grained parallelism, as we don't have to reason as much about the dynamic control flow of the program execution.

But wait, what exactly are we allowed to map over? Only arrays? Well, that depends on the implementation. Maybe you've come across the Repa library.[3] Repa is indeed restricted to this kind of *flat data parallelism*, where all the parallelism comes from parallelizing operations on regular[4] arrays. You may say that computation on each data element must be sequential. So, it's only one level of parallelism. This is good for a language implementation and for the runtime system, but not expressive enough for a lot of programming tasks.

Data Parallel Haskell takes a different approach and instead offers you *nested data parallelism*, a programming model that was popularized by Guy E. Blelloch in his work on the NESL programming language. [2, 3]. Nested data parallelism allows arbitrarily nesting of parallel computations, for example, of tree shape, or really of any shape at all – it can be applied to very irregular data, if that's what you need.

But don't we lose a lot of performance when we're working on irregular structures compared to working on regular arrays? Not necessarily, because GHC performs clever transformations called *flattening* or *vectorization*. It transforms the program expressed in terms of nested data parallelism in a way that it manipulates only flat arrays. So we win a lot of flexibility, while mostly retaining the simple and efficient execution model of the more restrictive flat data parallelism. Isn't that cool?

If you are already familiar with ordinary Haskell, then it's mostly straightforward to write Data Parallel Haskell. From the programmer's perspective it contains the following additional features:

- Parallel arrays, denoted by the type `[:a:]`.
- Parallel operations that operate on these arrays. They are mostly counterparts of the well-known list functions, such as `map`, `filter` etc.
- Parallel array comprehensions. A syntactic sugar, which is similar to list comprehensions, but for parallel arrays.

So it's just the same as programming with lists! One of the design principles for DPH was that DPH programs should resemble ordinary list programming as much as possible. There is one important difference to keep in mind though: Lists and parallel arrays are different from the strictness point of view. If one element of the parallel array is demanded, the whole array gets evaluated. [8, 4].

Okay, I guess that's enough of a general introduction. Let's have some real parallel fun now! In the rest of this tutorial, we will show you how you can

[3]Repa (REgular PArallel arrays)

[4]That is: Possibly multi-dimensional, but not with differently structured dimensions. I.e., dense $n \cdot m$ matrices are okay, because each of the n rows has exactly m columns. On the other hand, an array of arrays of different lengths is not allowed in Repa.

work with DPH. We will guide you through the (sadly non-trivial) installation process, show you examples how to program in DPH and how to benchmark your DPH programs.

Be warned, however, that DPH is rather experimental. There has been a lot of progress over recent years, but it is still not guaranteed that you will get a huge speedup if you switch from your sequential vector implementation over to DPH.

2 Installation & Troubleshooting

So, let's get started. First, we need to install DPH. Because DPH is a work in progress, it does not have the most well-tested and well-documented installation process and there are potentially a lot of problems you might encounter when trying to make it work. We are trying to help with this by including "Possible problems" notes along the process of describing how to work with DPH (so it's usually a good idea to read those before trying things).

In order to install Data Parallel Haskell, we need to install the DPH libraries as follows:

```
$ cabal update
$ cabal install dph-examples
```

This will install all DPH packages and a set of examples. Note that the `dph-examples` package depends on OpenGL and Gloss that are used as a visualization tool for some examples. Also, a very important thing to be aware of is that DPH uses an LLVM backend and you need a full LLVM installation in order to install DPH successfully. You can get LLVM by using some package manager on your system (for example, aptitude, homebrew) or build it from source. This process is rather painless and it is well-documented.

Possible problems:

One of the biggest problems is that the installation of DPH succeeded only with the 64-bit version of GHC. If you already have one, you're lucky and can skip some text below. Otherwise, install the 64-bit version of GHC for your platform. This should be pretty straight-forward. After this process you probably will have two versions of GHC on your system. To install DPH with the right version of GHC (assume 7.6.3), execute the following on Unix-like systems (on Windows, most likely, you need to supply paths to `ghc` and `ghc-pkg` in some other way):

```
$ cabal install --with-compiler=‘which ghc-7.6.3‘
                --with-ghc-pkg=‘which ghc-pkg-7.6.3‘ dph-examples
```

Note, that in this case you need to specify `ghc-7.6.3` and `ghc-pkg-7.6.3` explicitly in all of the following occurences of `ghc` and `ghc-pkg`.

If, while building one of the packages that `dph-examples` depends on, in particular `bmp`, you get an error message:

```
Codec\BMP.hs:208:11:  Not in scope:  ‘BSL.fromStrict’,
```

you should try to install an older version of `bmp`. In order to do this, execute:

```
$ cabal install "bmp < 1.2.3"
```

Now that you have hopefully managed to install DPH, let's try to compile some DPH code! You will now need the files `Main.hs`, `Randomish.hs` [5], `DotP.hs` and `DotPSeq.hs` that were bundled with this tutorial (in the directory `dotp`). They contain a parallel dot product, which might be considered a "Hello World" of DPH. (It also contains some benchmarking with a sequential version.[6].) We will explain the code below in the examples section, but for now our goal is just to compile.

You need to compile and link provided files by typing:

```
$ ghc -o dotp -threaded -rtsopts -eventlog Main.hs
```

In older documentation on DPH, you might come across the `-fdph-par` flag. You can just ignore that (it doesn't exist anymore) and compile as above. Now we can run the program. For example, to run it on two cores, type:

```
$ ./dotp +RTS -N2
```

Yay! Congratulations, hopefully you have just run your first Data Parallel Haskell program. We are very proud of you, dear reader.

Possible problems:

If while compiling `DotP.hs` you get the following error message:

```
Failed to load interface for ‘Data.Array.Parallel’.  It is a member
of the hidden package ‘dph-lifted-vseg-0.7.0.1’.
```

you may try to run the following command:

```
$ ghc-pkg expose dph-lifted-vseg-0.7.0.1
```

or compile with an additional flag `-package dph-lifted-vseg`.

You also may get scary and verbose errors that begin with the message of the following form:

```
Undefined symbols for architecture x86_64
...
ld:  symbol(s) not found for architecture x86_64
collect2:  ld returned 1 exit status.
```

In this case make sure you are linking all of the object files. This problem should not happen if you are following the instructions from the tutorial and don't link object files manually.

3 Examples

3.1 Parallel Dot Product

Now that you finally have a working Data Parallel Haskell installation, it's time to write some DPH code! Let's begin with the program that we already compiled in the previous section: A data parallel program for computing the dot product of two vectors.[7]

In ordinary (list-based) Haskell, we would probably write something like:

```
dotp :: Num a => [a] -> [a] -> [a]
dotp xs ys = sum (zipWith (*) xs ys)
```

[5]Borrowed from `dph-examples`

[6]Sequential versions sometimes are not the most obvious ones and not the most efficient. This was done intentionally, for example, to avoid problems with lazy evaluation and stack overflow and to compare the same approaches in both DPH and sequential versions.

[7]This section is largely based on the (outdated) tutorial at `http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`.

Prelude Lists	DPH	Remarks
`[]`	`[::]`	Empty list literal.
`[a]`, `[Int]`	`[:a:]`, `[:Int:]`	Types with parametric polymorphism and monomorphism.
`[ a | a <- as ]`	`[: a | a <- as :]`	List comprehensions.
`length`, `map`, `!`	`lengthP`, `mapP`, `!:`	Functions making use of parametric polymorphism are also available in DPH.
`sum`, `product`	`sumP`, `productP`	Functions that normally make use of ad-hoc polymorphism are presented with various version in DPH for each supported type, so `Data.Array.Parallel.Prelude.Int.sumP` has type: sum_P :: [:Int:] $\rightarrow$ Int while `Data.Array.Parallel.Prelude.Word8.sumP` has type: sum_P :: $[\mathrm{:Word:}]_8 \rightarrow \mathrm{Word}_8$.

Table 1: Comparison of Prelude List API and DPH API.

In the introduction we mentioned that the philosophy underlying DPH's API is to mimic Haskell's list API as much as possible. The syntax `[:a:]` is obviously reminiscent of lists, and there are also parallel array comprehensions that follow the same syntax as ordinary list comprehensions. Most of the usual functions are there as well, with a suffix `P`. For example, there is a function `sumP` that replaces the `sum` function from the Prelude. An overview of the API is shown in table 1.

Thanks to this correspondence between list and array, we should be able to transform the above program into a DPH program without thinking, right? It just becomes:

```
dotp :: Num a => [:a:] -> [:a:] -> [:a:]
dotp xs ys = sumP (zipWithP (*) xs ys)
```

Well, that's actually only half the truth: The DPH API is not defined in terms of type classes, so some of the library functions in DPH are necessarily monomorphic. This includes `sumP`: There are different implementations for `Double`, `Float`, `Int`, and `Word8`, defined in modules `Data.Array.Parallel.Prelude.Double` etc. We thus have to implement the dot product separately for each data type we're interested in, for example:

```
dotp_double :: [:Double:] -> [:Double:] -> Double
dotp_double xs ys = sumP (zipWithP (*) xs ys)
```

We can also express this in terms of parallel array comprehensions. They are almost the same as familiar list comprehensions:

```
dotp_double :: [:Double:] -> [:Double:] -> Double
dotp_double xs ys = sumP [: x * y | x <- xs | y <- ys :]
```

What should be noticed, is the vertical bar between generators. In list comprehensions you might expect comma there, producing cartesian product of elements drawn from lists. This notation here produces the same result as `zipWith`.

Now that we have this data parallel function, we of course want to call it! This is not quite as easy as you might hope. Every DPH application you write contains three kinds of code:

- Your parallel algorithm implemented in terms of DPH's parallel arrays `[::]`, like `dotp_double`. This is the code that will be vectorized to produce an executable, flat data parallel program. This code has to be in a module that is compiled with the `-fvectorise` flag.
- An interface function that allows you to call the algorithm from other (non-vectorized) code. (It is not possible to call functions using the `[:a:]` types directly from non-vectroized code.) Such an interface function will usually be in the same module as the data parallel algorithm and will be the only function exported by the module. It is usually just a wrapper for the parallel algorithm defined in terms of `PArray` rather than `[::]`.
- Your main program, which is not to be vectorized, and therefore in a separate file. This program calls the interface / wrapper function to execute the parallel program.

Note that it is mandatory to have the algorithm and the main code in different files, as vectorization can only be activated on a per-file basis.

So let's first complete the DPH implementation and then turn to the main program.

You start the DPH code file with

```
{-# LANGUAGE ParallelArrays #-}
{-# OPTIONS_GHC -fvectorise #-}
```

to turn on the DPH language extension and the vectorization. Then you implement the algorithm, for example the `dotp_double` implementation from above. Once you are done implementing the actual algorithm, you write the wrapper function. For the dot product, this function might look as follows:

```
dotp_wrapper :: PArray Double -> PArray Double -> Double
{-# NOINLINE dotp_wrapper #-}
dotp_wrapper v w = dotp_double (fromPArrayP v) (fromPArrayP w)
```

We convert the function's arguments from `PArray`s to `[:Double:]` via `fromPArrayP` and pass them to the algorithm. There also is a `toPArrayP` function that we would have to apply to the result if the algorithm returned an array instead of a single value. The `NOINLINE` pragma makes sure that the compiler does not inline this function into non-vectorized code.

The full source code for the dot product thus becomes:

```
{-# LANGUAGE ParallelArrays #-}
{-# OPTIONS_GHC -fvectorise #-}

module DotP (dotp_wrapper) where

import qualified Prelude
```

```
  import Data.Array.Parallel (PArray, fromPArrayP)
  import Data.Array.Parallel.Prelude.Double (Double, sumP, (*))

  dotp_double :: [:Double:] -> [:Double:] -> Double
  dotp_double xs ys = sumP [: x * y | x <- xs | y <- ys :]

  dotp_wrapper :: PArray Double -> PArray Double -> Double
  {-# NOINLINE dotp_wrapper #-}
  dotp_wrapper v w = dotp_double (fromPArrayP v) (fromPArrayP w)
```

Now we have the implementation and the interface for `dot_product`, so it's time to test it! We have to generate some input data. We can use ordinary lists, vectors from `Data.Vector` and also unlifted parallel arrays (`UArray`, which are the target of the vectorization process, but can also be used in user code[8]). `Data.Array.Parallel.PArray` exports functions `fromList` etc. to transform those data types into the `PArray` representation that the `dotp_wrapper` function expects.

As we don't actually have any data we want to run the algorithm on, we could, for example, use a pseudo-random number generator to come up with input data for us. Our main file might thus look like [9]:

```
import Data.Array.Parallel.PArray (fromList)
import System.Random

import DotP (dotp_wrapper)  -- import vectorized code

main :: IO ()
main = do
  let v = fromList $ (take n (randoms (mkStdGen 1)) :: [Double])
  let w = fromList $ (take n (randoms (mkStdGen 1)) :: [Double])
  print $ dotp_wrapper v, w
    where n = 1000000
```

Once you have saved the above to `Main.hs` and the algorithm to `DotP.hs`, you should be able to compile the program as follows:

```
$ ghc -o dotp -threaded -rtsopts -Odph -eventlog Main.hs
```

We tell the compiler to compile the program for a multi-threaded environment (`-threaded`), enable passing of options to the runtime system (`-rtsopts`), to optimize for DPH (`-Odph`), and to turn on the Eventlog, which we only do because we want to benchmark our code. The `-Odph` flag is very important, we have seen up to a factor of 20 in performance due to the optimizations performed.

Note that depending on your environment, you might get compilation errors telling you that you are trying to import modules from hidden packages. Should you have this problem, you can try adding `-package dph-lifted-vseg -package dph-prim-par` to the flags above.

Now we can run the program with `./dotp +RTS -N -ls`

[8]See `http://hackage.haskell.org/packages/archive/dph-prim-par/0.7.0.1/doc/html/Data-Array-Parallel-Unlifted.html`

[9]In the provided source code you will find some additional things, like more sophisticated generation and benchmarking

We tell the Haskell runtime system to use all cores it can get and to create an eventlog.

The obvious question now is: Is the code we've written actually fast and parallel? There's only one way to find out: Benchmarking!

We are using a library called `criterion` which is probably the most popular tool for benchmarking Haskell programs. You can see how it's used in the provided source code. We will compare a straight-forward sequential version using ordinary Haskell lists, the DPH version you have seen above with data generation using `System.Random` and, for the sake of curiosity, the same version of DPH dot product but with a different data generation approach (using code with vectors from `Rahdomish.hs`). The size of input list of `Doubles` is 1000000 elements. Benchmarks were performed on a 2-core machine. So, are you excited? Here are the first results (with 2 and 4 HECs):

```
$ ./dotp +RTS -N2
benchmarking dph/dot_product/seq
collecting 100 samples, 1 iterations each, in estimated 336.7726 s
mean: 27.78532 ms, lb 27.75886 ms, ub 27.81169 ms, ci 0.950
std dev: 134.6534 us, lb 118.0943 us, ub 156.0363 us, ci 0.950

benchmarking dph/dot_product/system.random
collecting 100 samples, 1 iterations each, in estimated 16.01110 s
mean: 1.057726 ms, lb 1.031741 ms, ub 1.105446 ms, ci 0.950
std dev: 175.1990 us, lb 109.5998 us, ub 264.3019 us, ci 0.950
found 10 outliers among 100 samples (10.0%)
  3 (3.0%) high mild
  7 (7.0%) high severe
variance introduced by outliers: 91.518%
variance is severely inflated by outliers

benchmarking dph/dot_product/randomish
collecting 100 samples, 1 iterations each, in estimated 6.080198 s
mean: 1.215686 ms, lb 1.163627 ms, ub 1.277236 ms, ci 0.950
std dev: 289.9775 us, lb 254.4380 us, ub 327.1625 us, ci 0.950
variance introduced by outliers: 95.747%
variance is severely inflated by outliers


$ ./dotp +RTS -N4
benchmarking dph/dot_product/seq
collecting 100 samples, 1 iterations each, in estimated 371.3929 s
mean: 33.47458 ms, lb 33.39802 ms, ub 33.56407 ms, ci 0.950
std dev: 422.9719 us, lb 360.9744 us, ub 532.0728 us, ci 0.950
found 2 outliers among 100 samples (2.0%)
  2 (2.0%) high mild
variance introduced by outliers: 3.083%
variance is slightly inflated by outliers

benchmarking dph/dot_product/system.random
collecting 100 samples, 1 iterations each, in estimated 13.22770 s
```

```
mean: 1.054220 ms, lb 1.029201 ms, ub 1.095328 ms, ci 0.950
std dev: 160.2724 us, lb 109.2828 us, ub 222.4338 us, ci 0.950
found 21 outliers among 100 samples (21.0%)
  7 (7.0%) high mild
  14 (14.0%) high severe
variance introduced by outliers: 90.440%
variance is severely inflated by outliers

benchmarking dph/dot_product/randomish
collecting 100 samples, 1 iterations each, in estimated 6.002903 s
mean: 1.037064 ms, lb 1.012435 ms, ub 1.078744 ms, ci 0.950
std dev: 161.0487 us, lb 105.7631 us, ub 224.9651 us, ci 0.950
found 10 outliers among 100 samples (10.0%)
  4 (4.0%) high mild
  6 (6.0%) high severe
variance introduced by outliers: 90.463%
variance is severely inflated by outliers
```

Oh, wow! That was fast! But was it parallel? Looks like it was, but to be sure, let's look at the visualization of eventlog that we generated [10]. We are using Threadscope for this:

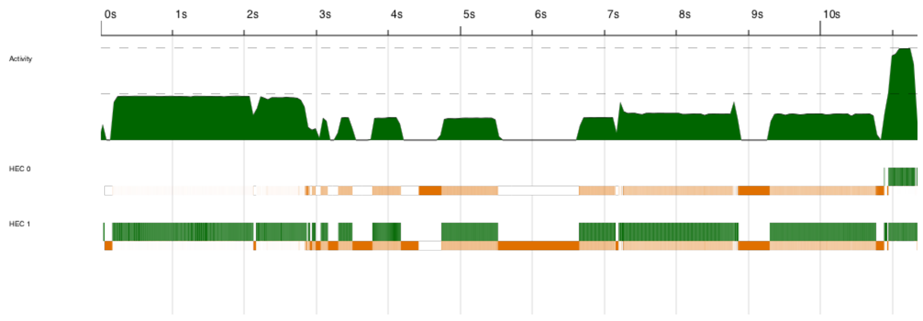

Figure 1: Threadscope displaying dot product with System.Random data generation on 2 HECs.

Well, not the nicest picture in the parallel world. There are mostly input data generation and garbage collection going on, but at the very end it utilized all available HECs (actually even 4 hyper-threads on the 2-core machine). This is kind of OK. But look at Figure 3 and Figure 4! There we are using the more efficient data generation from `dph-examples`.

Again, for such a simple example the bulk of time is used for data generation, but it looks really nice. The utilization at the end is very good. And it is also 30 times faster!

3.2 Sparse Matrix, Dense Vector Multiplication

So we got a decent speedup for the data parallel dot product, but we actually only made use of flat data parallelism, so we could just as well have used Repa,

[10]Actually we generated eventlogs separately for each DPH approach to avoid confusion.

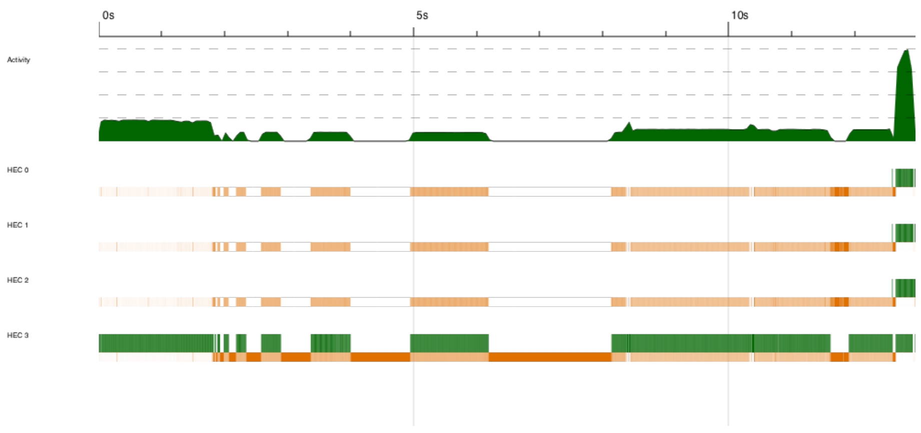

Figure 2: Threadscope displaying dot product with System.Random data generation on 4 HECs.

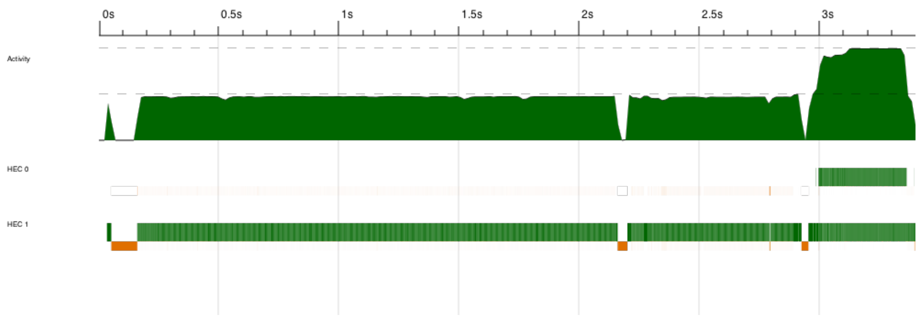

Figure 3: Threadscope displaying dot product with Randomish data generation on 2 HECs.

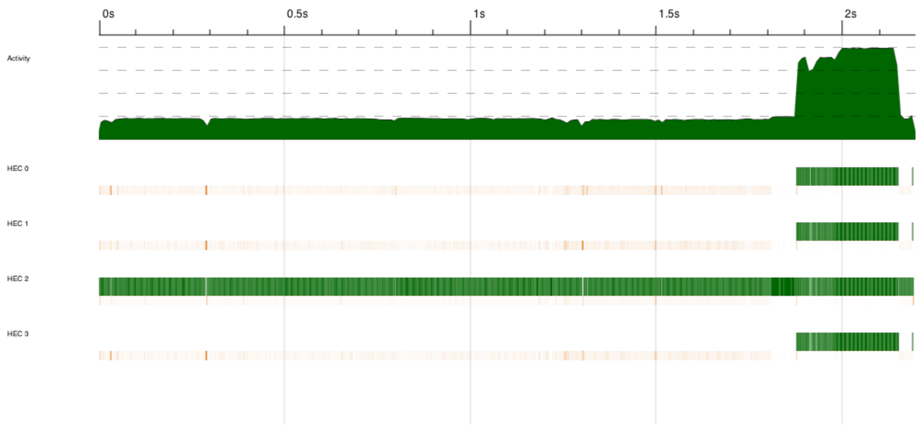

Figure 4: Threadscope displaying dot product with Randomish data generation on 4 HECs.

for example. As we told you in the introduction, we can also do nested data parallelism in DPH!

Pretty much the standard example to illustrate this is sparse matrix - dense vector multiplication. (This example is based on [5, 4].)

Let's first figure out how we would solve this in ordinary list-based Haskell. We could use the following types:

```
type Vector = [Double]            -- dense vectors
type SparseRow = [(Int, Double)]
type SparseMatrix = [SparseRow]
```

Each row of the matrix is represented by the indices that correspond to non-zero values and the corresponding values. A matrix then is a list of these row representations.

Now we need to derive how to compute `SparseMatrix * Vector`. Remember that the i-th entry of the resulting vector is the dot product of the i-th row of the matrix and the vector. We have already seen something very similar in the previous section, but this time one of the vectors involved is sparse. We thus need to work only with the entries specified in the sparse row:

`sum [x * (vec !! col) | (col, x) <- row]`

This is not exactly the fastest way to do this on lists as `!!` takes $O(n)$ time, but this is different for DPH arrays, so no need to optimize this.

We can then compute the matrix-vector product by repeating this for all rows, so we get:

`[sum [x * (vec !! col) | (col, x) <- row] | row <- sm]`

Yeah, one of the famous Haskell one-liners! So how do we translate this to DPH? Let's begin with the types:

```
type Vector = [:Double:]            -- Dense vector
type SparseRow = [:(Int, Double):]
type SparseMatrix = [:SparseRow:]
```

We just add a few `:`'s and are done! It's perfectly fine to just define a parallel array of parallel arrays of `(Int, Double)` pairs, even though the rows themselves can be (and ususally will be) of very different lengths. That's the great thing about nested data parallelism compared to flat data parallelism.

Thus, the matrix vector product can be expressed just as succinctly as in ordinary Haskell. We just need to replace `sum` with DPH's `sumP`, use the indexing function `!:` and add a few colons:

```
smvm :: SparseMatrix -> [:Double:] -> [:Double:]
smvm sm vec = [: sumP [: x * (vec !: col) | (col, x) <- row :]
              | row <- sm :]
```

This gives us three levels of parallelism: We're building a parallel array whose elements are determined by computing the parallel sum of a parallel array of products. We are allowed to use as many layers of parallelism as we need, and we're also not limited to just using plain arrays – we could for example also define a parallel tree type as

```
data Tree = Node [:Tree:]
```

But this is a bit beyond the scope of this tutorial; if you want to find out more about that, have a look at [8]. Back to the matrix multiplication: We now need to write the interface function for our algorithm. This is again just a wrapper, but this time we need to transform two levels of `PArray`s:

```
smvm_wrapper :: PArray (PArray (Int, Double))
             -> PArray Double -> PArray Double
{-# NOINLINE smvm_wrapper #-}
smvm_wrapper sm vec = toPArrayP (smvm psm pvec)
  where psm = fromNestedPArrayP sm
        pvec = fromPArrayP vec
```

Our first input is a nested `PArray` that represents the sparse matrix, and it is converted to a `[:[:Double:]:]` using `fromNestedPArrayP`. This time, we also have to use `toArrayP` to return the correct type. Otherwise, the wrapper code is just the same as in the previous example.

So let's test this code! You will find the source file together with a `Main.hs` file in the archive included with this tutorial (in the directory `matrixmult`). Let's compile:

`$ ghc -o smvm -threaded -rtsopts -Odph -eventlog Main.hs`[11]

Let's try to benchmark again. The approach is basically the same as before: We have some basic sequential version with lists and two approaches to input data generation (one uses just Haskell list ranges, the other uses `Randomish` for vector generation).

```
$ ./smvm +RTS -N2
benchmarking dph/smvm/seq
collecting 100 samples, 1 iterations each, in estimated 22.65658 s
mean: 360.1565 ms, lb 360.0510 ms, ub 360.2976 ms, ci 0.950
std dev: 624.3075 us, lb 495.4876 us, ub 787.1312 us, ci 0.950

benchmarking dph/smvm/non-random
collecting 100 samples, 1 iterations each, in estimated 433.7172 s
mean: 3.934112 ms, lb 3.884580 ms, ub 3.995562 ms, ci 0.950
std dev: 280.9008 us, lb 227.6404 us, ub 361.0336 us, ci 0.950
found 10 outliers among 100 samples (10.0%)
  5 (5.0%) high mild
  4 (4.0%) high severe
variance introduced by outliers: 65.624%
variance is severely inflated by outliers

benchmarking dph/smvm/randomish
mean: 3.853171 ms, lb 3.813830 ms, ub 3.891666 ms, ci 0.950
std dev: 200.4513 us, lb 180.5598 us, ub 222.0977 us, ci 0.950
variance introduced by outliers: 50.439%
variance is severely inflated by outliers


$ ./smvm +RTS -N4
```

[11] possibly plus `-package dph-lifted-vseg -package dph-prim-par`

```
benchmarking dph/smvm/seq
collecting 100 samples, 1 iterations each, in estimated 22.40732 s
mean: 361.4516 ms, lb 361.2417 ms, ub 361.9138 ms, ci 0.950
std dev: 1.527368 ms, lb 873.6000 us, ub 2.975064 ms, ci 0.950

benchmarking dph/smvm/non-random
collecting 100 samples, 1 iterations each, in estimated 515.8522 s
mean: 4.308898 ms, lb 4.230576 ms, ub 4.542644 ms, ci 0.950
std dev: 634.1632 us, lb 262.6555 us, ub 1.378819 ms, ci 0.950
found 5 outliers among 100 samples (5.0%)
  3 (3.0%) high mild
  2 (2.0%) high severe
variance introduced by outliers: 89.416%
variance is severely inflated by outliers

benchmarking dph/smvm/randomish
mean: 4.187424 ms, lb 4.150555 ms, ub 4.220781 ms, ci 0.950
std dev: 180.3436 us, lb 162.5623 us, ub 202.6099 us, ci 0.950
variance introduced by outliers: 40.514%
variance is moderately inflated by outliers
```

We got used to speed-ups from DPH, didn't we? This is really a great speed up (x90). Of course, those who actually looked at the sequential version may argue that we are using very inneficient implementation. Yes, we are, but the point is to try to compare two really straight-forward implementations in both approaches. And we are trying to learn Data Parallel Haskell and not optimizing sequential Haskell code, right? Let's look at eventlogs now (Figures 5-8).

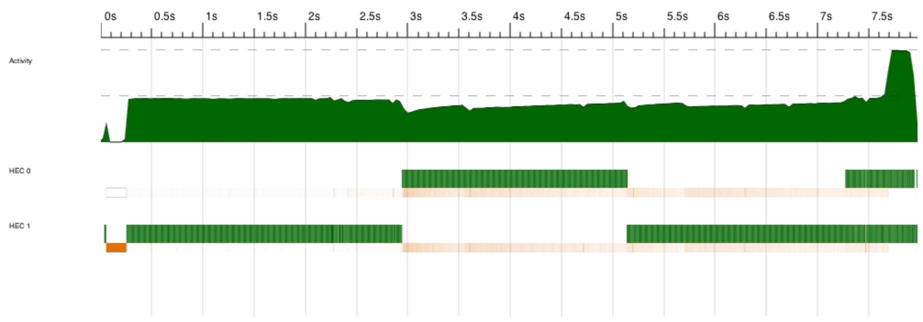

Figure 5: Threadscope displaying sparse matrix, dense vector multiplication with non-random data on 2 HECs.

We got used to long data generation process as well, but, again, there is a very decent parallelism at the the end.

3.3 Scan

Let's move to the last example. Scan is a fundamental building block in parallel programming and there are a lot of different algorithms which can be implemented in terms of scan [1]. Here is an implementation of so-called prescan (hugely inspired by NESL) in Data Parallel Haskell:

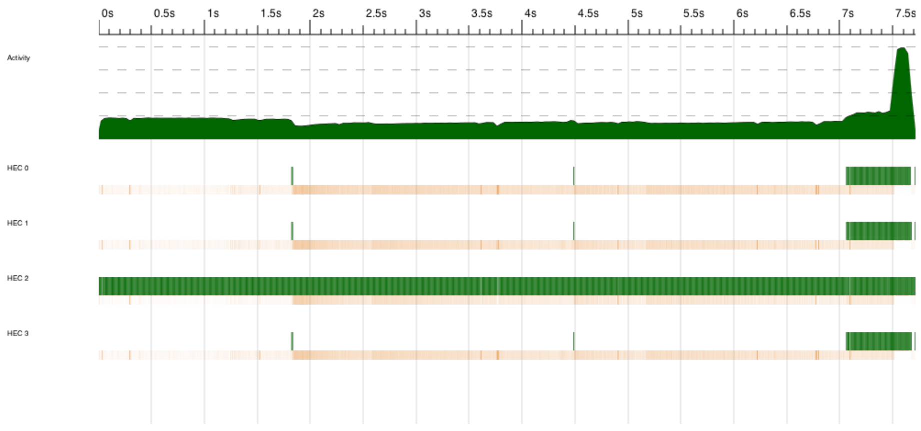

Figure 6: Threadscope displaying sparse matrix, dense vector multiplication with non-random data on 4 HECs.

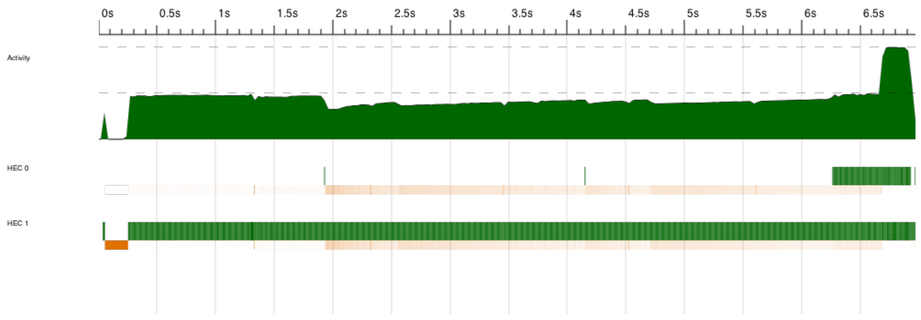

Figure 7: Threadscope displaying sparse matrix, dense vector multiplication with Randomish data generation on 2 HECs.

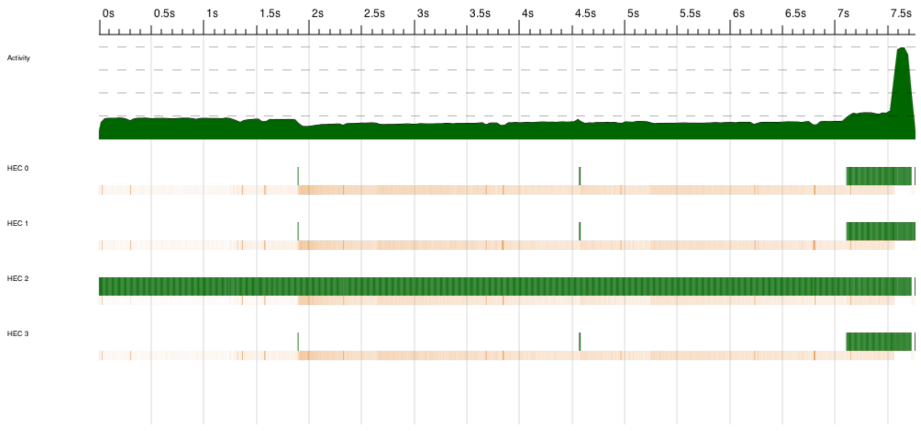

Figure 8: Threadscope displaying sparse matrix, dense vector multiplication with Randomish data generation on 4 HECs.

```
prescan :: (Int -> Int -> Int) -- Associative operation
        -> Int                 -- Identity element
        -> [:Int:]             -- Input length must be power of 2
        -> [:Int:]
prescan op id input
 | len == 1  = [: id :]
 | otherwise =
    -- 0 is even, 1 is odd
    let indices  = sliceP 0 len (concatP (replicateP len [: 0, 1 :]))
        evenElts = [: elt | (ind, elt) <- zipP indices input, ind == 0 :]
        oddElts  = [: elt | (ind, elt) <- zipP indices input, ind == 1 :]
        scanRslt = prescan op id (zipWithP op evenElts oddElts)
    in interleave scanRslt (zipWithP op scanRslt evenElts)
      where len = lengthP input

interleave :: [:Int:] -> [:Int:] -> [:Int:]
interleave first second =
  concatP (zipWithP (\fst snd -> [: fst, snd :]) first second)
```

We are basically dividing input into even and odd elements (by positions), zipping them with operator and call the `prescan` recursively on the result. Then we interleave the result of the recursive call with "fixing" (zipping the result of the recursive call with even elements). Notice that we are using a lot of functions from DPH to work on parallel arrays (like `sliceP`, `concatP`, `replicateP` etc.). From this example you can see that there are some things missing from usual Haskell code, like pattern matching on a singleton list, `take` etc. This is something you may come across writing DPH programs that are longer than just a few lines: Some features are not there yet, and sometimes you even get compiler panics.

Finally, we need the wrapper function `prescanSum` which will be the function that is exported and actually called, specifically not inlined as before:

```
prescanSum :: PArray Int -> PArray Int
{-# NOINLINE prescanSum #-}
prescanSum input = toPArrayP (prescan (+) 0 (fromPArrayP input))
```

The reason why we don't simply export the higher-order function `prescan` and pass the addition operator to it is because an unvectorized function cannot be passed as an argument to a higher-order vectorized function.

It's time to do a benchmarking again:

```
$ ./scan +RTS -N2
benchmarking dph/prescanSum_seq
mean: 4.119674 ms, lb 4.079000 ms, ub 4.174472 ms, ci 0.950
std dev: 239.3890 us, lb 191.4762 us, ub 342.8856 us, ci 0.950
found 6 outliers among 100 samples (6.0%)
  5 (5.0%) high mild
  1 (1.0%) high severe
variance introduced by outliers: 55.497%
variance is severely inflated by outliers
```

```
benchmarking dph/prescanSum/system.random
mean: 29.77641 ms, lb 29.65630 ms, ub 29.90320 ms, ci 0.950
std dev: 632.8736 us, lb 563.1852 us, ub 744.8489 us, ci 0.950
found 1 outliers among 100 samples (1.0%)
variance introduced by outliers: 14.222%
variance is moderately inflated by outliers

benchmarking dph/prescanSum/randomish
mean: 29.78969 ms, lb 29.68106 ms, ub 29.89597 ms, ci 0.950
std dev: 554.0416 us, lb 496.0738 us, ub 625.7622 us, ci 0.950
variance introduced by outliers: 11.350%
variance is moderately inflated by outliers


$ ./scan +RTS -N4
benchmarking dph/prescanSum_seq
mean: 5.333245 ms, lb 5.285440 ms, ub 5.398617 ms, ci 0.950
std dev: 285.7756 us, lb 220.3906 us, ub 377.8601 us, ci 0.950
found 5 outliers among 100 samples (5.0%)
  4 (4.0%) high severe
variance introduced by outliers: 51.471%
variance is severely inflated by outliers

benchmarking dph/prescanSum/system.random
mean: 35.39847 ms, lb 35.34041 ms, ub 35.45605 ms, ci 0.950
std dev: 294.6813 us, lb 264.8222 us, ub 332.4857 us, ci 0.950

benchmarking dph/prescanSum/randomish
mean: 35.32262 ms, lb 35.26251 ms, ub 35.38079 ms, ci 0.950
std dev: 301.0198 us, lb 267.3659 us, ub 357.7192 us, ci 0.950
```

Oh, these results are not very inspirational. We got a significant slow down comparing to the sequential version of the same algorithm with lists. At the same time, we see from the eventlogs (Figures 9-12) that work distribution is quite good at the end. You may also notice that there is a rather long span of time when all HECs seem to be busy, but actually the context switching is happening there all the time and we don't run anything in parallel, that's why the graph at the top of each picture shows a total utilization of around 1 HEC at a time. This might be a possible source of slowdown that we got.

4 Conclusions & Evaluation

After this exciting journey into the wonderful world of nested data parallelism we would like to conclude with some thoughts about the current status of Data Parallel Haskell. As almost all DPH-related articles mention, it is very much a work in progress. There are a number of severe limitations like the lack of support for type classes in vectorization, the inability to mix vectorized and non-vectorized code (as could be seen from examples), and then there are also some performance issues: As we saw in the previous section, the code generated by DPH is not always able to compete with ordinary Haskell code.

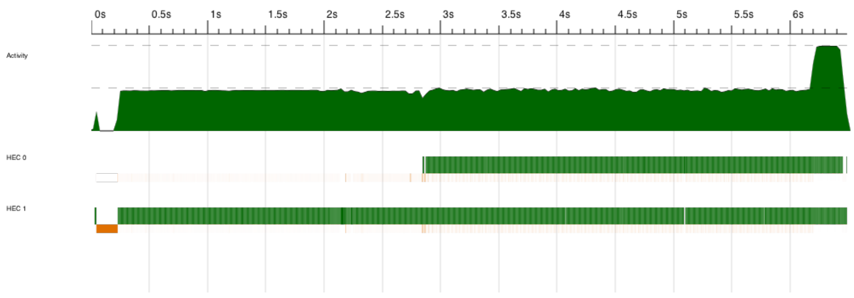

Figure 9: Scan with System.Random data generation on 2 HECs.

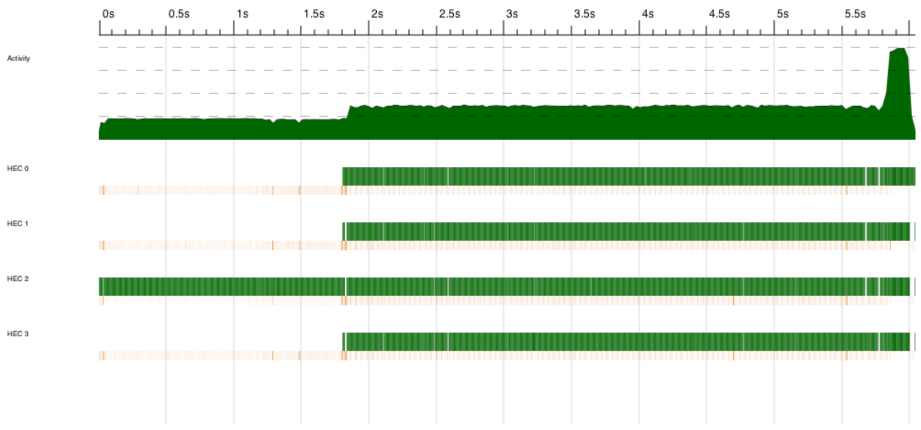

Figure 10: Scan with System.Random data generation on 4 HECs.

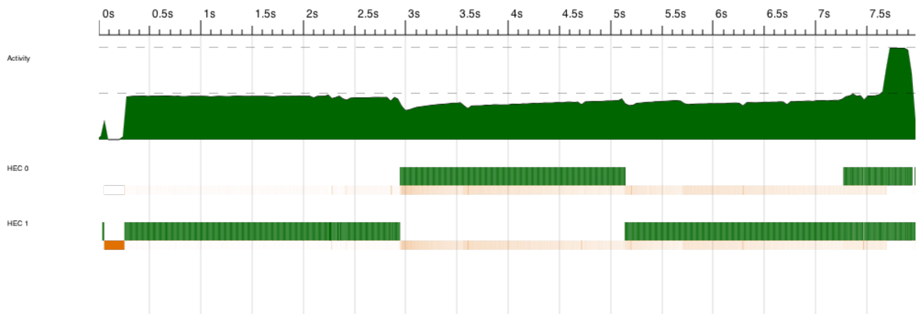

Figure 11: Scan with Randomish data generation on 2 HECs.

On the other hand, Data Parallel Haskell is a very complex and fundamental project, and we hope to see very good results in the future, as many of the concepts look really promising and suitable for the "many core era" that we are just about to enter at the time of writing.

Unfortunately, there currently is a very limited amount of sources of information on DPH and the official docs and tutorials are often slightly outdated. Hopefully, we have succeeded in improving this last point a little bit.

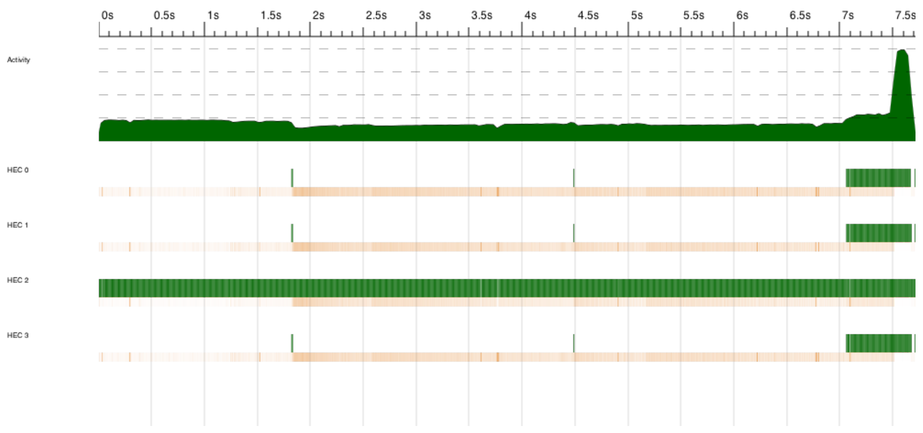

Figure 12: Scan with Randomish data generation on 4 HECs.

References

[1] Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[2] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, 1990.

[3] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996.

[4] Manuel M. T. Chakravarty, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon" Marlow. Data parallel haskell: a status report. 2007.

[5] Manuel M.T. Chakravarty, Manuel M.T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf" Pfannenstiel. Nepal – nested data-parallelism in haskell. *IN EURO-PAR '01*, 2150:524–534, 2001.

[6] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: better strategies for parallel haskell. *SIGPLAN Not.*, 45(11):91–102, September 2010.

[7] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. *SIGPLAN Not.*, 46(12):71–82, September 2011.

[8] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] P.W. Trinder, P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton" Jones. Algorithm + strategy = parallelism. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 8:23–60, 1998.