Deterministic parallel programming with Haskell

Duncan Coutts and Andres Löh
Well-Typed LLP

Abstract

Haskell is a modern functional programming language with an interesting story to tell about parallelism: rather than using concurrent threads and locks, Haskell offers a variety of libraries that enable concise high-level parallel programs with results that are guaranteed to be deterministic (independent of the number of cores and the scheduling being used).

We argue that Haskell and deterministic parallelism are a good match for many computing problems in science and engineering and demonstrate the effectiveness of this approach using the example of a naïve solver for a 1D Poisson equation[1].

[1]For context: a version of this article appears in the Nov–Dec 2012 issue of CiSE magazine – a special issue on parallelism and concurrency in modern programming languages. The guest editor of that issue set a challenge to parallelise a naïve solver for a Poisson equation.

1 Introduction

Before the current era of multi-core CPUs, most programmers didn't have to worry about writing parallel programs. These days, to make use of all the computing power that is available, we must write parallel programs that use multiple CPU cores. With more people writing – or trying to write – parallel programs, it has become obvious that the traditional approaches have numerous drawbacks and that we need other approaches.

The traditional model uses multiple concurrent threads, and the operating system achieves parallelism by scheduling these threads on multiple CPU cores. Haskell, however, has several approaches that don't use concurrent threads and locks; it offers a variety of libraries that enable concise, high-level, parallel programs with results that are guaranteed to be deterministic, independent of the number of cores and the scheduling being used.

1.1 Parallelism versus concurrency

We draw a distinction between parallelism and concurrency. By concurrency we mean techniques for structuring programs that let us model computations as hypothetical independent activities that can communicate and synchronise. Parallelism is about getting results in a shorter time by making use of multiple processing units such as CPU or GPU cores.

The distinction is important because we often want parallelism without needing or wanting concurrency. It is possible to have concurrency without parallelism, e.g. an OS running multiple processes on a single CPU, and it is also possible to have parallelism without concurrency. We will look at a couple approaches.

1.2 Why not concurrency?

Concurrency is great for structuring programs like web servers and databases that have to respond to multiple clients. For these applications concurrency makes the programs more modular and is simply a natural fit. Haskell has excellent support for concurrency, providing lightweight threads and libraries for styles of concurrency such as software transactional memory (STM) or message passing.

On the other hand, scientific and engineering computing problems typically involve pure calculation and have no need for concurrency as a way to organise programs. For calculation problems, such as the Poisson equation solver that we will look at shortly, we don't actually want to think about threads. We want to think about which parts of the computation are independent of each other and hence can be done in parallel. That is what your language, libraries and tools should offer you.

For calculation problems, changing a sequential program to use threads invariably makes it more complex. It also opens up the pitfalls of deadlock, livelock and non-determinism. The nondeterminism means that it is possible to accidentally write programs that give different results on different runs due to minor differences in timing. This is highly undesirable for pure calculation problems. And if you do go down this route, it is hard to be fully confident that you have written your program in such a way so that it is not sensitive to timing issues.

Our advice is simple: if your problem naturally involves concurrency then use concurrency in the solution, but if you have a typical calculation problem then it is usually preferable to avoid concurrency.

1.3 Deterministic parallelism

Let's reveal how we can talk about parallelism without resorting to concurrency.

With deterministic parallelism, the result of a calculation is determined only by the data inputs and not by timing, thread scheduler decisions or by how many CPU cores are used. Haskell sports a number of deterministic parallel programming models, suitable for different styles of problem.

A major category is the parallelism available within expressions. For example if $f\ x$ and $g\ y$ are both expensive calculations then in the expression $f\ x + g\ y$ we have an opportunity to evaluate the two parts in parallel. It is clear that the program fragment $f\ x + g\ y$ does not express any concurrency – it is a simple pure expression. The mechanism for evaluating the expression has the possibility to use parallelism to get the results sooner. Whatever evaluation mechanism is used however, the result must be the same. This approach extends well beyond simple expressions to full parallel data structures and algorithms.

1.4 Data parallelism

Closely related to parallelism within expressions is data parallelism. Data parallelism is all about doing the same operation to a large number of data items and because the operation on each item of data is independent, they can all be done in parallel. Typical examples involve bulk operations on large vectors and arrays.

The Poisson equation solver falls naturally into this category, so this is the style that we will focus on in this article. This is no coincidence; many problems in science and engineering turn out to be naturally data parallel.

The key idea for writing data parallel programs is to organise the data into collections (such as arrays) and express the algorithms in terms of bulk operations on those collections. This is as opposed to the conventional sequential style which uses explicit sequential loops and traversals over the data. As an additional benefit, data parallel programs are often shorter and clearer than their conventional counterparts.

1.5 Parallelism and Functional Programming

It should be noted that, even with the various nicer parallel programming models, parallelism is still hard. In particular getting real speedups is hard.

For example, we usually have to write the program differently to expose more parallelism and eliminate unnecessary sequential data dependencies. We have to indicate what chunks of a computation can profitably be evaluated in parallel and what should be kept sequential. The granularity of the parallelism has a significant effect on performance, though as we will see, data parallelism gives us a reasonable handle on granularity and lets the library automate some of the work.

So although parallel programming remains hard, pure functional languages are an easier place to start from than conventional imperative languages.

Imperative languages start from the notion of a linear sequence of instructions producing a linear sequence of state changes. In addition, the imperative style allows and encourages communication between different parts of a calculation by mutating shared variables. This makes it very hard to determine if different parts of a calculation can be executed independently or if there are dependencies between them.

By contrast, in a pure functional programming language such as Haskell, where calculations are expressed as pure mathematical expressions and functions, we can evaluate expressions in any order that respects the data dependencies. The data dependencies are explicit as function inputs and outputs rather than implicit in the order of state changes.

This is why there has been significant interest in using functional programming for parallelism in recent years: it makes a lot more sense to start from languages that are naturally insensitive to evaluation order than from languages that are naturally sequential where we have to try to reverse engineer some parallelism out of them.

Of course sometimes sequencing is necessary, such as ordering actions to read from or write to files. In Haskell such ‘side effects’ are explicit in the types of functions and actions so we can cleanly separate pure calculations (where evaluation order does not change the results) from side effecting I/O parts of the program (where strict ordering must be maintained).

2 Exploratory programming with Haskell

When solving a problem we would usually not jump straight in to writing a final fast program. Instead we would explore the problem by writing simple versions of the important parts of a solution. This helps us check we’re on the right track and getting the right results.

Haskell is particularly good for this kind of exploratory programming. Haskell lets us write short clear code. In particular, as a functional programming language, based on mathematical functions, the gap between a mathematical model and an runnable program can be quite small.

$$
\begin{aligned}
\phi(k,i) &= \begin{cases} 0 & \text{when } k = 0 \\ 0 & \text{when } i < 0 \text{ or } i \geqslant n \\ \frac{\phi(k-1,i-1)+\phi(k-1,i+1)}{2} + \frac{h}{2}\rho(i) & \text{otherwise} \end{cases} \\
\rho(i) &= \begin{cases} 1 & \text{when } i = \frac{n}{2} \\ 0 & \text{otherwise} \end{cases} \\
h &= 0.1 \qquad \text{(lattice spacing)} \\
n &= 10 \qquad \text{(number of sites)}
\end{aligned}
$$

Figure 1: Numerical solution to the 1D Poisson equation.

```
phi k i | k == 0              = 0
        | i < 0 ∨ i ⩾ nsites  = 0
        | otherwise           = (phi (k − 1) (i − 1) + phi (k − 1) (i + 1)) / 2
                                + h / 2 ∗ rho i
rho i   | i == nsites `div` 2 = 1
        | otherwise           = 0
h      = 0.1  -- lattice spacing
nsites = 10   -- number of sites
```

Figure 2: Haskell translation of the mathematical specification

We can also write fast code with Haskell. Once we are happy that we are getting the right results then we can incrementally rewrite key parts of our code to run faster, either serially or in parallel.

2.1 The 1D Poisson equation

As a very brief introduction to Haskell we will go through this exploratory process with our example problem before looking at how to parallelise it.

Our example problem is a solver for the 1D Poisson equation. We will solve it numerically by a relaxation method. Figure 1 shows the mathematical specification of our numerical approximation. The function $\phi(k,i)$ gives the result at site i (out of n) at the kth iteration. The function is defined recursively so that ϕ at iteration k is defined in terms of ϕ at iteration $k-1$. The base case at iteration 0 is all 0s. For i outside of the range $0..n-1$ we define $\phi(k,i)$ to be 0.

We can translate this mathematical function definition $\phi(k,i)$ into an equivalent Haskell function definition *phi k i* in Figure 2. You can see that the two versions are essentially identical with just a few differences in notation. Unlike in most programming languages, functions in Haskell are always mathematical functions in the usual sense. Given the same inputs they always yield the same outputs, there can be no hidden dependencies.

In Haskell we write function calls as $f\ x\ (y+1)$ rather than $f\ (x, y+1)$. For functions defined by cases we put the guard conditions to the left of the equality and separate it with a vertical bar. We distinguish equality for defining func-

```
phiA = array extents elements
  where
    extents  = ((0, −1), (niters − 1, nsites))
    elements = [((k, i), phi k i) | k ← [0 .. niters − 1], i ← [−1 .. nsites]]
phi k i | k == 0                = 0
        | i < 0 ∨ i ⩾ nsites    = 0
        | otherwise             = (phiA ! (k − 1, i − 1) + phiA ! (k − 1, i + 1)) / 2
                                    + h / 2 ∗ rho i
```

Figure 3: Lazy array version

tions and values (=), from equality for checking if two values are equal (==).

We can use the Haskell compiler's interactive environment to experiment with our new definition. We put the definition in a file and load it up in GHCi:

```
Main> phi 4 3
1.25e-2
```

We could use Haskell's list comprehension syntax to easily check our function on a range of values:

```
[phi 4 i | i ← [0 .. nsites − 1]]
```

2.2 Memoisation

Although this function definition gives us the right results, it is wildly inefficient for larger values of k. This is because each *phi* k depends on two recursive calls to *phi* $(k-1)$, and each of those depends on two more. There is also lots of duplicated work going on, for example *phi* k i will compute *phi* $(k-2)$ i twice, once via each of *phi* $(k-1)$ $(i-1)$ and *phi* $(k-1)$ $(i+1)$.

We could share the calculations so that each *phi* k i is calculated only once. We can do this very simply in Haskell using a lazy array. We define a two-dimensional array *phiA* of delayed or 'lazy' computations (see Figure 3). The entry at each index (k, i) is a delayed call to *phi* k i. We adjust our original *phi* definition so that instead of making recursive calls *phi* $(k-1)$ $(i-1)$ and *phi* $(k-1)$ $(i+1)$, we look up their values in the array *phiA*. Array indexing is written $a\ !\ i$, or $a\ !\ (i, j)$ for a 2-D array.

Lazy evaluation will now do the following: the first time we look up a value in the array, it will be computed, and the array will implicitly be updated with the result of the computation. If the same array entry is requested again, the already computed result can be returned immediately. This behaviour is an automatic consequence of Haskell's standard evaluation strategy, and its internal complexity is completely hidden from the programmer. This style of lazy array solution can be applied to a wide variety of dynamic programming problems. We do not need to know the pattern of dependencies to be able to use it. So long as there are no cycles it will work.

This implementation now runs in time directly proportional to $nsites * niters$. The performance would be quite adequate for moderate sized examples, such as generating graphical visualisations.

While the compiler does implement lazy arrays rather more efficiently than you might imagine, it is not suitable for high performance. To write a faster serial or parallel implementation we need to analyse and take advantage of the pattern of data dependencies.

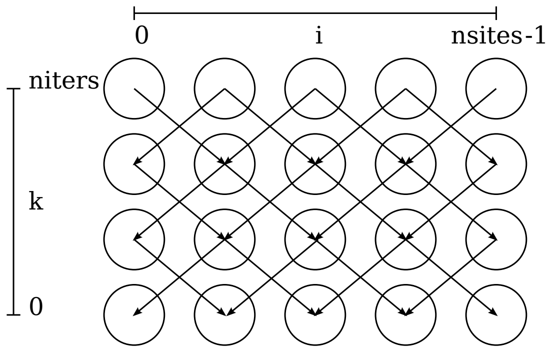

Figure 4: Data dependencies between values of *phi k i*

3 Going parallel

Let us now analyse the data dependencies of the Poisson solver and use them to come up with a more efficient, parallel program.

3.1 Data dependencies

Figure 4 shows the data dependencies between values of *phi k i*, or equivalently between entries in the lazy array version above. We can see that each row depends only on the previous row.

We could take advantage of this fact to write a faster serial version where we only keep one row at a time and calculate the next row recursively from the previous one.

Rather than writing the fast serial version we turn our attention to parallelising the algorithm. We start with looking at what parallelism there is available. If we look again at the data dependencies in Figure 4 we can see that within each row, every element depends only on a couple values from the previous row. So given the previous row, each element of the new row could be calculated independently and thus in parallel.

Notice that this operation to calculate the new row is a simple data parallel operation. This should be a strong hint to us to consider using a data parallel paradigm in our solution.

Because each element of a new row can be calculated in parallel then we have a parallel factor of *nsites*. The total work remains proportional to *nsites* ∗ *niters* and for large examples both *nsites* and *niters* will be large.

We might hope that as well as performing operations on rows in parallel that we could do something similar with columns, for example to get one CPU core to work on the left hand half and another to work on the right half. Unfortunately the data dependencies mean that we cannot do this, at least not without duplicating work.

3.2 Parallel granularity

Things are not quite so rosy as they may seem. Although we have a large number of independent calculations, individually each of them is a very small amount of work.

This leads us to the important issue of the granularity of parallelism. The issue of granularity invariably crops up with parallel programming, no matter what language or approach we take. In many cases where parallel programs fail to achieve expected speed improvements, the problem turns out to be due to bad parallel granularity. It is worth considering from the beginning.

The granularity refers to how large the independent chunks of work are. Too large and we may have fewer chunks than CPU cores, or have difficulty evenly balancing work between cores. The more usual problem however is having too fine a level of granularity. The problem with very fine granularity is that there is a certain overhead to running chunks of work on a CPU core so if each chunk is very small then the overheads can become quite significant or even dominant.

```
poissonIterate prevA = computeP (traverse prevA id next)
  where
    next prev (Z :. i) | i == 0 ∨ i == nsites + 1 = 0
                       | otherwise                 = (prev (Z :. i − 1) + prev (Z :. i + 1)) / 2
                                                     + h / 2 ∗ rho′ i
```

Figure 5: Data parallel version using Repa

In our example the amount of work for each element of each row is tiny, just a handful of arithmetic and array lookup operations. If we were to try to parallelise at the level of each element of each row then it is likely we would make the program dramatically slower than the serial version, even when using several CPU cores.

The usual solution is to make the granularity coarser by aggregating work into larger chunks. How this can be done depends on the program and what paradigm we are using to express the parallelism. In particular, with data parallelism there is the possibility to pick the granularity automatically.

3.3 A data parallel implementation

For our data parallel implementation we will use a Haskell library called Repa (short for *regular parallel arrays*). The central data structure in Repa is the parallel array. Every element of a parallel array can be calculated independently and the Repa library takes care of calculating them in parallel. In particular the Repa library helps with the granularity issue by partitioning the array into as many chunks as there are CPU cores. Our task is to express our algorithm in terms of parallel arrays.

We will focus on the operation to calculate the new row from the previous one. The parallelism is confined to this function. Once we have this function then the overall sequence of iterations is straightforward.

The code is in Figure 5. There are a number of points to notice.

Firstly a notational issue. Repa represents array indexes using the pattern $(Z :. i)$. The Z constructor is the zero-dimensional index type. The :. constructor adds an additional dimension, so a 2-D index would be $(Z :. i :. j)$. While somewhat clunky in appearance, the purpose is to make the dimension more flexible, for example to enable taking lower dimensional slices of multi-dimensional arrays.

The overall structure is a bulk operation to calculate a new array in terms of the old one. The *traverse* function takes three parameters:

- the input array,
- a function giving the extents of the new array in terms of the extents of the old, and
- a function that calculates the values of the new array in terms of values of the old array.

Since we are producing an array of the same size as the input one, we pass the identity function *id* as the second parameter. The interesting detail is all in third parameter: the *next* function. The *next* function defines the value for each index of the new array. As its first parameter it is supplied with a function *prev* to look up values from the old array. So while Repa does provide the array indexing operator !, in our *next* function we write *prev* $(Z :. i)$ to look up values from the input array.

As this example illustrates, it is not uncommon in Haskell to use functions that take other functions as parameters. It is a particularly common pattern when it comes to bulk operations. The bulk operation function defines the skeleton of the operation and a function passed in as a parameter fills in the detail.

In Repa, arrays come in two modes, *delayed* and *manifest*. Delayed arrays are represented as a function from index to value, while a manifest array is the traditional in-memory representation. The delayed representation is useful for defining arrays by composing several simpler array operations without generating lots of temporary intermediate arrays. This lets us write code in a clear concise style by composing multiple array operations and lets us do so without a significant performance penalty. The *computeP* function creates a manifest array – in parallel – from a delayed array.

A final minor detail is that in the previous array version we used array indexes from -1 to *nsites* which we can do because Haskell's ordinary arrays support arbitrary extents. Repa arrays are always indexed from 0 so we shift everything up by one and use indexes from 0 to $nsites + 1$.

3.4 Compiling and running

To compile and benchmark a standalone program we first need to define a *main* function that calculates an overall result. For brevity we omit the code but it is included in the code that accompanies this article online (see doi:10.1109/MCSE.2012.68).

We compile the executable using

```
ghc -O2 -threaded PoissonRepa.hs
```

The `-O2` tells GHC to apply optimisations. The `-threaded` flag tell GHC to link with the mutlicore runtime system (RTS).

We can run the resulting program by using

```
./PoissonRepa +RTS -N4
```

The `-N` flag tells the RTS how many cores to run on.

3.5 Improving performance

Of course the point of writing parallel programs is to improve performance. To see if we are getting good absolute performance we have written a fast implementation in C and parallelised it using the OpenMP extension.

Since we care about the performance we also ought to consider ordinary sequential optimisations, so long as they do not complicate the program too much. Our C version uses a few tricks.

1. The C version uses unsafe array indexing while our Haskell version in Figure 5 uses array bounds checking.
2. If we look at the worker function *next* in Figure 5 we see that it has to check the boundary condition $i == 0 \ \vee \ i == nsites{+}1$ for every single element. The C version avoids this check because it does a loop from 1 to *nsites* and handles the 0 and $nsites + 1$ cases separately outside of the loop.
3. Finally, the C version uses in-place update of arrays. It uses only a pair of arrays and recycles them between iterations. The Haskell version allocates a new array for each iteration and uses the garbage collector to clean up the old ones. The GC itself takes very little time because the heap only contains a small number of large objects. The in-place update also does not change the number of array reads and writes, but by recycling the same area of memory it is likely to be able to keep more of the arrays in the local CPU caches.

Figure 6 shows our optimised Repa version with two of the three tricks applied.

Haskell has a strong 'safety first' culture and unless you specifically use an unsafe feature

```
poissonIterate phi = computeP (single0 ⧺ middle ⧺ single0)
  where
    single0          = fromFunction (Z :. 1) (const 0)
    middle           = unsafeTraverse phi (const (Z :. nsites)) next
    next prev (Z :. i) = (prev (Z :. i) + prev (Z :. i + 2)) / 2
                         + h / 2 * rho i
```

Figure 6: Optimised Repa version

you should not be able to write a program that segfaults. The convention is that libraries that provide unsafe features label them very explicitly with the prefix 'unsafe'. For example with Repa we can use array indexing without bounds checking using *unsafeIndex*. To eliminate bounds checking for our *poissonIterate* function all we have to do is switch from using *traverse* to *unsafeTraverse*, no other changes are required.

The issue of the boundary condition $i == 0 \vee i == nsites + 1$ is rather interesting. If we define an array by a function from index to value it seems hard to handle special cases without having to test the index all the time. What we can do however is to build the array in parts. Repa provides an array append operator (⧺). We can append two singleton 0 arrays on either side of the main part which then no longer needs to test for the boundary indexes. In fact this provides an example of the advantage of Repa's delayed array representation. The ⧺ operator does not need to immediately copy both arrays, it simply defines a new function that reads from both arrays. We do not manifest a new array until we use *computeP*.

Finally, with Repa we cannot use the trick of using in-place array updates and recycling just a pair of arrays. While Haskell's ordinary non-parallel arrays come in both immutable and mutable variants, the Repa library provides only immutable arrays.

3.6 Benchmarks

As well as absolute performance, we are interested in how well the performance scales with the number of CPU cores we use. One of the first things we find when we try some sample runs is that the number of sample sites (the width of the rows) makes a big difference to how much improvement we get from using more cores. This comes back to granularity. Even though the whole row can be calculated in parallel, if this is only a few 1000 elements then the amount of work to do is still relatively small. All CPU cores need to synchronise after each row and this wastes some time. In particular, if we double the number of cores but keep the data size the same then we halve the amount of work per core, proportionately increasing the effects of synchronisation overheads.

With these caveats in mind we set the number of sample sites at $64,000$ and the number of iterations at $40,000$. We ran both versions on a 2-socket, 8-core, 3 GHz Xeon. We used `gcc` 4.6 with `-O2 -fopenmp` and `ghc` 7.4.1 with `-O2 -threaded`. The results are in Figure 7.

Cores	OpenMP		Repa	
	time	speedup	time	speedup
1	22.0s	1×	25.3s	1×
4	6.9s	3.2×	11.4s	2.2×
8	5.3s	4.2×	8.4s	3.0×

Figure 7: Benchmark comparison of C OpenMP and Haskell Repa versions on 1–8 cores.

The absolute timings are not themselves especially interesting, what is interesting is how well the performance scales with the number of CPU cores, and the comparison between the Haskell and C versions.

3.7 Assessment

Beating C on well-optimised array programs is always going to be difficult, and it is not what we are trying to do. Nevertheless, the performance in this example is fairly reasonable.

The point is to provide a way of writing parallel programs with reasonable performance that is less complex, that is quicker in development time and that provides greater confidence that the programs are right.

4 The future

Haskell sports a range of ways of approaching parallelism and concurrency, many of which are mature and some of which are still active research and development projects. An approach that has been in use for many years now is the parallelism within pure expressions that we mentioned previously.

Repa is actually a relatively new approach in Haskell. We chose to introduced you to Repa because the Poisson equation solver example falls so nicely into the data parallel style.

In fact Repa is a product of a much more ambitious ongoing project on Data Parallel Haskell (DPH). DPH takes the data parallel idea one step further by adding *nested* data parallelism. Usual 'flat' data parallelism is all well and good but you need to be able to express your algorithms using dense arrays. This includes vectors and higher-dimensional arrays but not sparse vectors/matrices or nested data structures such as trees. Nested data parallelism increases the range of parallel data structures and algorithms that can be expressed using parallel arrays by allowing parallel arrays to contain other arrays, or data structures containing arrays. Good performance is achieved by a compile-time 'flattening' transformation to turn nested arrays into a single flat array which can then be partitioned evenly across all processor cores.

Other recent and ongoing developments include libraries that provide elegant ways of programming GPUs (again using data parallelism) and libraries for doing distributed memory programming, e.g. clusters or grid/cloud systems. Finally, there is work under way to make the compiler GHC able to take advantage of CPU SIMD vector instructions, such as Intel's SSE and AVX instruction sets.

Put this all together and within a few years we have the potential to cover parallelism at all levels: SIMD vector, multi-core, GPU and clusters, and to do so in a coherent principled way, all within one language.

5 Further reading and related work

5.1 Haskell

There are a number of excellent resources for learning Haskell, both books and online.

- Online `http://haskell.org/` provides a central portal.

- For a short basic introduction to Haskell we recommend *Programming in Haskell* (Graham Hutton; Cambridge University Press; 184 pages).

- A fun and quirky alternative is *Learn you a Haskell*
(Miran Lipovača; No Starch Press; 400 pages).

- *Real World Haskell* is a huge book covering lots of practical material: FFI, DBs, GUI, web, performance, concurrency etc. (Bryan O'Sullivan, Don Stewart and John Goerzen; O'Reilly; 720 pages).

5.2 Parallelism in Haskell

Simon Marlow, one of the lead GHC developers, has a good tutorial *Parallel and Concurrent Programming in Haskell.* This covers the main parallel approaches other than Repa, plus concurrency.
`http://community.haskell.org/~simonmar/par-tutorial.pdf`

A similar introduction can be found in *A Tutorial on Parallel and Concurrent Programming in Haskell* by Simon Peyton Jones and Satnam Singh.
`http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/AFP08-notes.pdf`

Further details on Repa are available from its home page `http://repa.ouroborus.net/`. In particular if you were interested by the Poisson solver example then we would recommend the Repa paper *Efficient Parallel Stencil Convolution in Haskell* which covers similar examples and describes some techniques for extracting further performance gains.

Finally, if you're now wondering how you might do GPU or cluster programming in Haskell then we recommend the papers

- *Accelerating Haskell array codes with multi-core GPUs*
  `http://www.cse.unsw.edu.au/~chak/papers/CKLM+11.html`
- *Towards Haskell in the cloud*
  `http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/`

5.3 Background and related work

There is a range of academic papers on parallelism and concurrency in Haskell, both models and details of implementations.

- *Algorithm + Strategy = Parallelism* (Phil Trinder, Kevin Hammond, Hans-Wolfgang Loidl and Simon Peyton Jones. JFP 1998). This concerns the parallelism available within expressions. It introduces the parallel "strategies" approach. The attraction of this approach is that what to compute and the parallel evaluation strategy can be specified separately.
- *Seq no more* (Simon Marlow, Patrick Maier, Phil Trinder, Hans-Wolfgang Loidl, and Mustafa Aswad. Haskell Symposium 2010). This is a modern update on the "algorithms + strategies = parallelism" story.
- *A monad for deterministic parallelism* (Simon Marlow and Ryan Newton. Haskell Symposium 2011). Using par/seq and strategies to write parallel Haskell programs can be tricky; this paper explains why and offers a possible solution.
- *Concurrent Haskell* (Simon Peyton Jones, Andrew Gordon and Sigbjorn Finne. POPL 1996). This is the original description of the concurrency extension for Haskell. The long version of the paper also has implementation details.
- *Haskell on a Shared-Memory Multiprocessor* (Tim Harris, Simon Marlow and Simon Peyton Jones. Haskell Workshop 2005). This describes how to implement Haskell on multicore machines.

This list is just a small sample. For a more extensive list see `http://www.haskell.org/haskellwiki/Research_papers/Parallelism_and_concurrency`