Programming as if the Domain (and Performance) Mattered

Draft Version 1.0

(C) 2017 Carlo Pescio

Abstract

What if focusing on the problem domain, while still understanding the machine that will execute your code, could improve maintainability and collaterally speed up execution by a factor of over **100x** compared to popular hipster code?

Introduction

Every now and then, I see this problem coming up in talks, blogs, even tweets:

"You are given an input array whose each element represents the height of a line towers. The width of every tower is 1. It starts raining. How much water is collected between the towers?"

The question, quoted verbatim above, originally appeared on stackoverflow and most of the solutions I've seen over time are variations or replicas of what is being discussed there. As I'm writing this draft, there are 23 answers, but most of the proposed solutions (with the exception of a parallel one I'll mention later) are based on the same problem-solving strategy[1], which is explained in the accepted answer and that I'll copy verbatim again:

"Once the water's done falling, each position will fill to a level equal to the smaller of the highest tower to the left and the highest tower to the right. Find, by a rightward scan, the highest tower to the left of each position. Then find, by a leftward scan, the highest tower to the right of each position. Then take the minimum at each position and add them all up.

I'll let you ponder on that, and if you want, you can also take a look at some of the solutions to get a taste of how people are turning that idea into code. Many are in C-like languages, and involve explicit loops. Some make an attempt to adopt some domain terminology within the code, others don't. Of course, any procedural code looping over arrays is condemned to look like

[1] Even those which are not, are still based on the same problem solving strategy: focus on an array of integers, loop over it, keep track of heights, and so forth. So everything we learn about this specific strategy largely applies to the others as well.

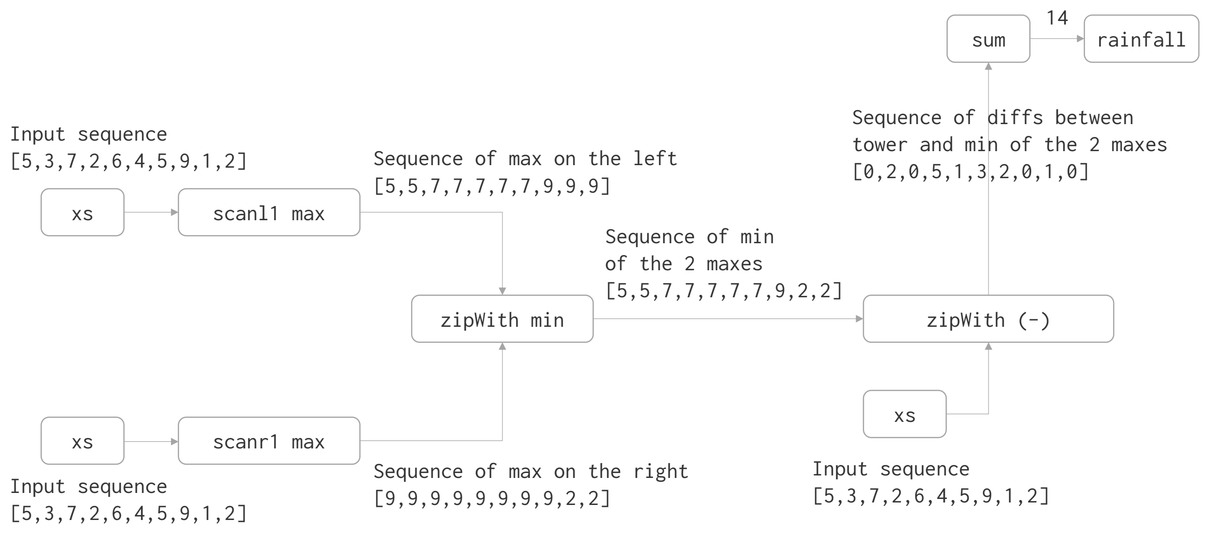
Fortran, so there is nothing spectacular there. Most people are concerned about the computational complexity of their code (which is a good thing). Many get to solve the problem in linear time (which follows immediately from a good implementation of the procedure above) and many are also trying to keep the number of passes over the entire array "small", usually two.

The second most upvoted solution, however, is in Haskell, and it looks much shorter and different, without all those pesky loops, and at last it's a breath of fresh air as it doesn't look like Fortran anymore:

```haskell
rainfall :: [Int] -> Int
rainfall xs = sum (zipWith (-) mins xs)
    where mins = zipWith min maxl maxr
          maxl = scanl1 max xs
          maxr = scanr1 max xs
```

You may want to spend a little time reading it, which should not be too hard as it's simply encoding the procedure above in a loop-free form. You just need to know what a left and right scans are (easy) and what zipWith does (look it up if you don't, but it basically merges 2 sequences using a function).

If you're unfamiliar with Haskell and too lazy to even try reading the code, the following picture might help (I added a sample input, and replicated the input sequence *xs* around for clarity)



That code has become quite popular and, to be honest, I would have loved it too, as a CS student toying around with FP back in 1987. After all:

- It's way shorter than all the imperative stuff. We are not bothered with implementation details like loops, and we let the compiler handle the small stuff.

- It looks a bit like math, not just because you don't have loops, but because of the way your brain is engaged when you read it.

- It's reusing many small domain-independent, higher-order functions. That's cool.

Still, in 2017 I can't look at that code and feel the same way. What is funny is that **most of the issues I see have nothing to do with Haskell or with FP per se**, but mostly with the way FP is being exemplified these days (which, to be honest, is the way academics always taught FP, because it's much easier for them this way):

- That code *actively removes domain knowledge*. The idea that we have towers and water is removed from the table and replaced with a list of integers. The clever removal of variable and function names, which is part of the strategy to get short / cool code, also removes any possibility to use domain terminology. To make things worse, even the input is called *xs*, neglecting our only chance to explain it was an array of tower heights. But anyway, *xs* is there just because the OP didn't write the entire thing point free, which would have been considered even better (and even less informative, but who cares).

- In fact, the only reminiscence of the domain is in the function name (*rainfall*). However, given that code alone, without an explanation of the problem to be solved, you would be able to understand the mechanical semantics (the way data is mechanically transformed) but not the domain semantics. There is nothing to guide you to an understanding of water collected between towers.

- While it may look like cool, "abstract" code, it's encoding a very procedural algorithm. **The real problem-solving happened in the wordy explanation** ("*Find, by a rightward scan, the highest tower to the left of each position...*"). That algorithm is very procedural, in the sense that it's a step-by-step recipe of calculations that have to be done to solve the *exact* problem that we were asked to solve. The fluffy functional stuff is merely removing loops from that procedure.

- The OP says "*Here's an efficient solution in Haskell [...] it uses the same two-pass scan algorithm mentioned in the other answers*". However it's unclear what "efficient" is supposed to mean, and that code is doing more than 2 passes over the data set anyway.

The dream that a compiler could turn that code into efficient, maybe parallel code was a dream in 1987 and it appears to be still a dream in 2017 (see later for numbers).

- As we'll see, that approach (solve the *exact* problem, then make it loop free and math-looking) tends to create extremely fragile code. This has always been known as a weakness of straightforward functional decomposition, but among many other things it is being ignored by the latest generation of functional programmers, who seem unaware of what we have learnt in the past 30-40 years.

This unawareness is causing a funny revisionism I've seen floating around a lot lately, like:

- It's ok if the code is hard to read: it's called CODE so you have to DECODE it[2].

- The idea that code should reflect the domain is a notion of the past.

- Since naming is hard, functional code that gets away without variable and function names helps the programmer being more productive (don't mind the fact that thinking about good names helps understanding the problem and that good names give hints to the reader about our solution; they are supposed to DECODE it anyway).

- It's ok if you have to throw away everything as soon as the problem changes a tiny bit. That prevents maintenance from crippling the beauty of your code.

- It's ok to model everything in terms of lists of lists of lists of integers, because then you can play with all those nifty library functions. One data structure with 1000 functions FTW! (It follows that domain types are for wimps, and we praise the type system because we can express algebraic concepts in it, not crappy stuff like worlds made of towers).

But what is worse is the persistent illusion that by not writing loops we are somehow "not describing an implementation" or even that we are "describing the problem to be solved and not the solution". Look at the code above: it's clearly a set of instructions, just for a different kind of machine (a functional / data flow machine). It is **not** describing the problem, actually is very far removed from the statement of the problem. It is merely encoding the algorithmic **solution** explained at the beginning of this paper.

[2] I can only stress this point so much, but short code got short by removing all kind of redundancy. Yt sm rdndnc hlps. Note that you got "sm" because of the context in which it appeared, which still had enough redundancy to help. Remove context, remove *meaning*, and understanding gets harder (requiring decoding).

But is there any real alternative to the entire thinking process that brought us to that code?

Use the Domain, Luke

Let me restate the main issue I have with the previous code: despite the cool look, it's a computational process devised to (rigidly) solve the stated problem, expressed in a way which neglects the problem domain and actively removes domain knowledge. As we'll see later, this is detrimental for maintenance and not particularly good for performance either. But first, I think it would be useful to look at the opposite way to get our answer – focusing on the domain and not on some computational solution. So, nothing like finding the minimum of 2 maxima.

We have a World made of Towers. Water (in the form of rain) is falling on that world, and some of it is collected between towers. We want to calculate the amount of that water. Ok, so what happens to the water that is **not** being collected? It *leaks*, of course. So in this hypothetical world water is basically leaking on the left and right side[3].

My first intuition was to build a model of that world, mostly like we do with Conway's game of life, using cells that may contain one of 3 substances / be in one of 3 states:

- Wall[4]
- Water
- Air

And then having a simple rule: *if a cell has Water but the cell to its left (or right) has Air, then that cell leaks, becoming Air as well.*

Note that I'm introducing an *explicit* notion of leaking: water leaks where possible, that is, where there is air. There is nothing about keeping track of towers height here. **Yet this rule is enough to model the entire behavior of our system.**

[3] Note how the concept of leaking is completely absent in the usual discussions of the problem, when people focus immediately on a computational solution.

[4] Took me some self-control not to call this Earth or Stone.

Unlike the game of life, this rule *immediately converges to a fixed point*[5] after you apply it once.

To avoid checking both the right and the left side of each cell, the rule can be applied left-to-right (looking only at the cell at left) and then right-to-left (looking only at the cell at right). This is just an optimization (we're now moving closer to the machine).
Also, in the second scan (say, right-to-left), if you find a cell that contains air, you don't need to go any further left, as the other cells have already leaked. That means you have to process each cell *at most once*.

Furthermore, if processing an entire vertical slice of the world does not cause any change in state, you don't need to move further in that direction either, because you have hit a "world-tall wall" and all water is trapped behind that wall in the current direction of movement. That means *in the average case you don't even have to process all the cells in the world.*[6]

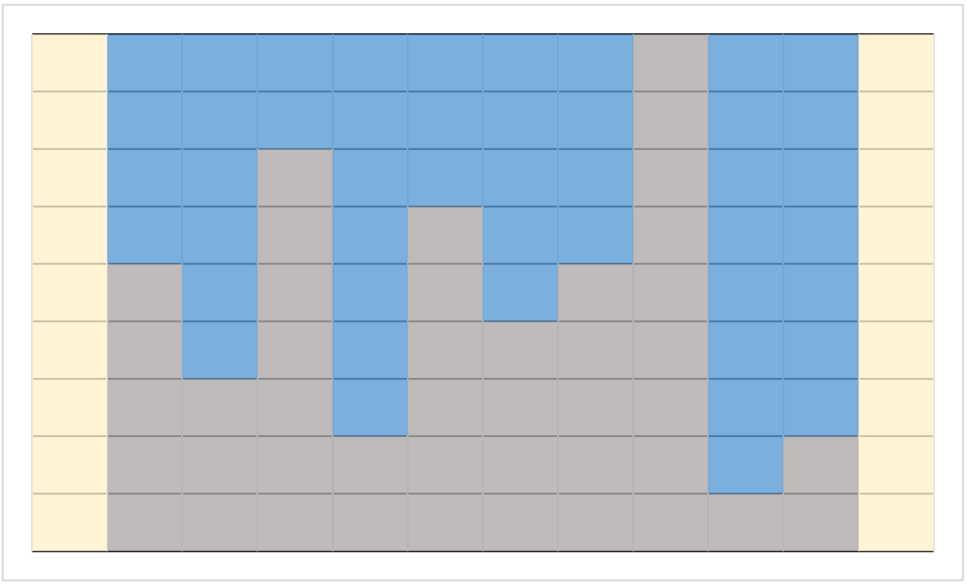
Note: all those statements would require a proof. Still, those proofs would be primarily about the **world** and the way water can leak in such a world, and the **result** of those proofs will grant us some optimizations. More on this later when I'll talk about the role of math.

The process is highly simplified if we enclose the world in two vertical slices[7] with an initial contents of Air, reifying the idea that in this world water leaks to the right and left sides. So, given the usual test case where towers heights are [5,3,7,2,6,4,5,9,1,2], the initial (flooded) state of the world would be:

[5] For those who were sleeping during math classes, **c** is a fixed point of a function **f** ⇔ **f**(**c**) = **c**.

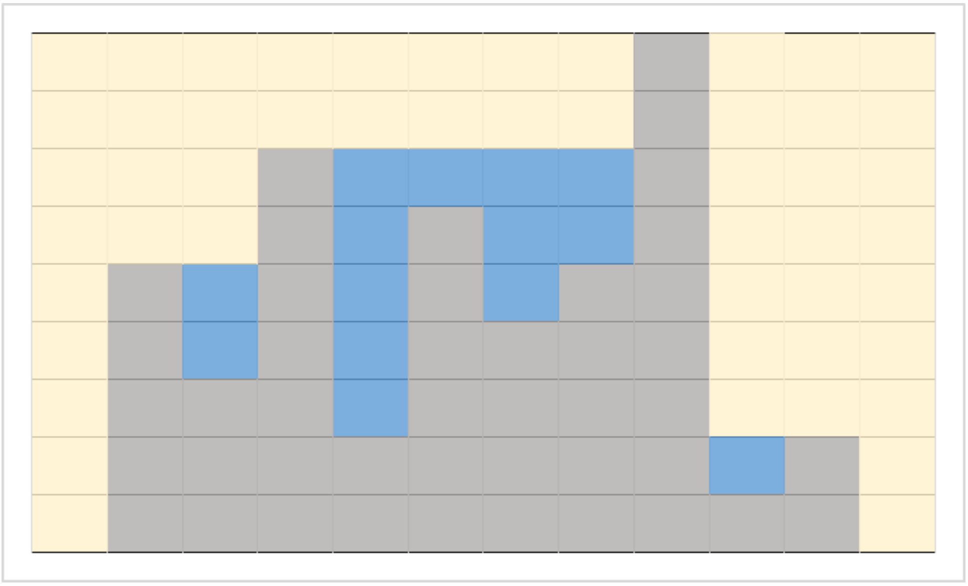
[6] I still have to go through every tower in the input once, to *build the world*. More on this when I'll discuss performance.

[7] Placing special values outside the boundaries of a world/collection/space to simplify and/or speed up an algorithm is a well-known problem-solving strategy and would deserve to be documented as a pattern.



Where yellow represents air, gray represents wall, and blue represents water.

After going through every cell and basically telling it to leak (if possible), first left-to-right and then right-to-left, we would be left with this (stable) world:



And of course the trapped water would amount to 14 cells, just as expected for this test case.

Note that in this approach I'm not trying to "solve the problem" by devising a procedural algorithm first, like "find the tallest tower to the left…" and then somehow "abstract" that solution by removing the dreaded control flow. I'm building a model of the world and its (simple) rules: water leaks into air. Then I just let the world evolve according to the domain rules. It just so happens that in this case a single pass will immediately converge to a fixed point, but it would be trivial to change the code to allow for multiple passes under more complex rules.

In other words, we have just moved from a more or less concrete **description of a recipe to solve the problem** to a **description of the rules governing the world**. There is no centralized / monolithic function calculating how much water is being trapped. Behavior is emerging through the interaction of simple cells. Nowhere I have to "keep track of the tallest tower" or anything like that – which despite the functional lipstick is so deeply procedural in nature.

While I liked the idea, and I've of course implemented it, I'm not going to show you the code because it has a major flaw: I have just moved from a solution which has a computational complexity $\mathcal{O}(n)$ to one which has, in the worst case, $\mathcal{O}(n*m)$ complexity, where n is the number of towers and m is the height of the tallest tower, because I have to process every cell once, and we have n*m cells[8].
This model also makes an assumption which was not present in the previous code, that is, it's assuming the world is discrete (towers heights are integers), and therefore it can be decomposed into cells. I don't particularly like this, although I haven't seen test cases with fractional heights around. I would like to remove this assumption as well.

Still, I'll get back to this model when I'll talk about evolving the problem and observing how the code has to adapt to those changes (if it can, that is).

Improving by exploiting constraints

The previous approach is inefficient because it's more general than required, that is, it can model a world way more complex than the worlds all the other solutions are capable of handling. It is therefore reasonable to accept the same constraints as the others and see if we can bring its worst-case complexity within $\mathcal{O}(n)$.

The main constraint is quite simple: in our world, given a vertical slice, there are at most 3 stacked layers: the wall layer, the water layer, the air layer. Each layer can have height 0, but the sum of the heights is always equal to the height of the world[9]. There are no "holes" inside a tower, which is a way to say that there cannot be multiple walls in a single vertical slice, interleaved with water or air.

[8] There is a chance of high parallelism in this model, as every horizontal slice of the world can be processed left-to-right (and then right-to-left) independently of the other horizontal slices. But that won't help much on commodity hardware with just a handful of cores. I'll discuss parallelism later.

[9] Which in turn is of course the height of the tallest tower.

While this constraint was somewhat implicit in the original formulation of the problem, and implicit in the published code as well, it is explicit now, and will be explicit in code too. Being explicit, and being expressed in the language of our problem domain, it would also be a perfect candidate for a conversation with the mythical "domain expert", who won't care less about you using *scanl1* instead of a loop, but will probably know about the correctness and stability of this assumption instead.

Within that assumption, we can model the World not as a matrix of cells, but as a sequence of Slices, where every slice is described by some amount of Wall, Water, and Air. Although I'll use integers to represent those amounts, this is no longer a constraint for correctness, and I could have used floats as well. We have one slice per tower, so processing all slices once will bring us back in $\mathcal{O}$(n).

How do you leak water between a source slice and a target slice (ignoring left and right)? It's a rather straightforward process:

- If the source contains air, it has leaked already, so there is nothing to do.
- Otherwise, by the laws of physics, wall + water in the source cannot be higher than wall + water in the target[10] (the excess of water will leak). The rest, of course, will be filled by air.
- Wall never changes, so this quantity is always preserved on transformation.

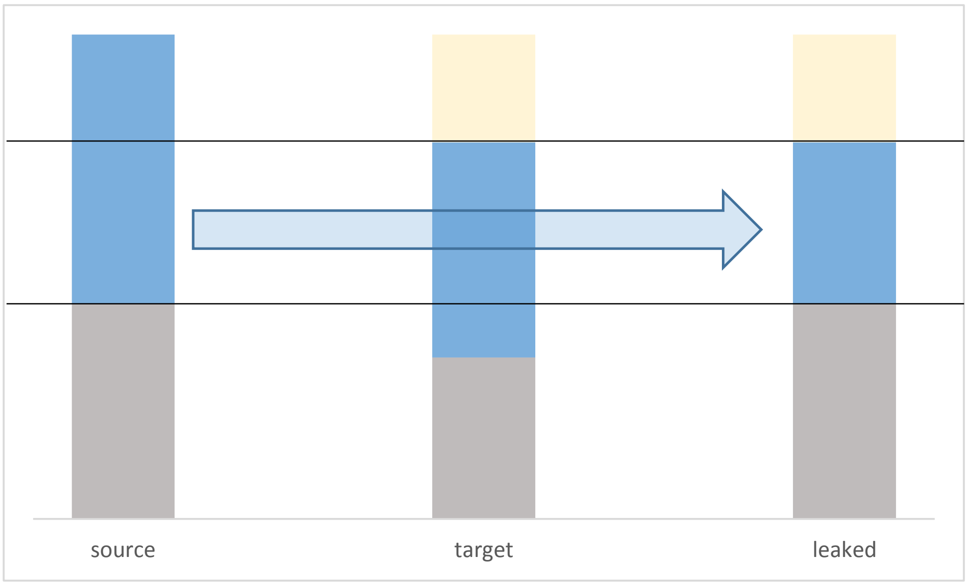
Therefore, if we represent our slices as a tuple[11] (wall,water,air), the process of leaking a source into a target is represented by:

Leak((srcWall,srcWater,0), (tgtWall,tgtWater,tgtAir)) = (srcWall, srcWater', srcAir') where:
srcWater' = max(tgtWall + tgtWater - srcWater,0)
srcAir' = srcWater - srcWater'

[10] We need to consider the case where the wall in the source is taller than wall + water in the target, that's why I said "cannot be higher" instead of "must be equal".

[11] Ignoring whatever the implementation will be.

Or visually (for the case when there is actually a leak)



Note, once again, how this is all about the physics of our world and not about a procedural algorithm to find the total collected water. In fact, this is all about leaking, not collecting.

Show me some code already!

For reasons that will be clear later, when I'll talk about performance, I have implemented this solution using C# and classes. However, nothing prevents from adopting the same approach in Haskell or in any functional programming language. I'm not using polymorphism, inheritance, or late binding, just strong typing and encapsulation, which any language with a notion of modules can give you as well. I'm mutating the world as I go, but that's a minor point – it would be trivial to adapt the code to immutability, by producing a new world upon transformation.

Of course, once you bring in domain types and give up expressing your logic in terms of lists of integers and scans and zips, your code will grow in size, so it won't impress hipsters anymore, but you can't please everybody.

Here is the first implementation I jot down; there are at least two important optimizations missing, so don't worry too much about performance yet.

A slice has a constructor with defaults because I'd like to use named parameters in the call (see World) and that's the only way to get named parameters in C#. Other than that, it has the leaking logic I explained before, and can return its amount of water.

```
class Slice
{
    private int wall;
    private int water;
    private int air;

    public Slice(int wall = 0, int water = 0, int air = 0 )
    {
        this.wall = wall;
        this.water = water;
        this.air = air;
    }

    public bool LeakInto(Slice target)
    {
        if(air != 0)
            return false;

        int newWater = Math.Max(target.wall + target.water - wall,0);
        air = water - newWater;
        water = newWater ;
        return air > 0;
    }

    public int Water()
    {
        return water;
    }
}
```

The world takes an array of integers (I've done so to preserve "compatibility" with all the previous code, and also, as we'll see, to somewhat penalize the performance of this code) and builds slices. Then it implements the logic of scanning for leaks and reports the total water:

```
class World
{
    private int width;
    private Slice[] slices;

    public World(params int[] towerHeights)
    {
        width = towerHeights.Length + 2;
        int worldHeight = towerHeights.Max();
        slices= new Slice[width];

        int x = 0;
        slices[x++] = new Slice(air:worldHeight);
        foreach( int h in towerHeights )
            slices[x++] = new Slice(wall:h, water:worldHeight -h);
        slices[x] = new Slice(air: worldHeight);
    }

    public void Drain()
    {
        for(int x = 0; x < width - 1; ++x)
            if( ! slices[x + 1].LeakInto(slices[x]) )
                break;

        for(int x = width - 2; x >= 0; --x)
            if( ! slices[x].LeakInto(slices[x+1]) )
                break;
    }

    public int Water()
    {
        return slices.Sum(s => s.Water());
    }
}
```

Note the *break* calls. I'll comment on those in a minute, just after the test case:

```
class Program
{
    static void Main(string[] args)
    {

        World w = new World(5, 3, 7, 2, 6, 4, 5, 9, 1, 2) ;
        w.Drain();
        Debug.Assert(w.Water() == 14);
    }
}
```

Now, World is a bit Fortran-like as it loops over an array, however it doesn't do that to play with integers but to call logic on slices. Of course, I could have avoided the *break* by using a *while* loop instead of a *for* loop. In a sense, I wanted the idea of early termination to be very explicit, because this is an important aspect of this solution.

Indeed, while the Haskell code doesn't seem to be really concerned with how many times we have to go through a collection with **n** items (either the input or a transformed collection) many other solutions in the stackoverflow page tried to minimize those passes. The code above is not yet optimized, but in the end will perform quite decently, also thanks to that early termination, so I wanted it to stand out.

By the way, Drain() is idempotent, which is nice.

Changes, changes

Real world problems tend to change, and the ability to change our code correspondingly, without having to rethink the entire strategy, is a significant advantage. I have suggested that "my" approach is better than the more common counterpart based on finding the tallest towers etc., but is that true?

Before we proceed, I have learnt from experience that I need to clarify a few key issues:

- People will bring up YAGNI. I think it's important to understand that I did **not** put anything in the C# code to speculatively allow any kind of extension. If you understand the notion of YAGNI, you'll see that it doesn't apply here.

- People will invariably claim that I have cherry picked changes that are easier for my "preferred" code. While I can say in full honesty that I didn't, you have any right to doubt it. I have tried, within the limitation of an artificial problem, to come up with reasonable changes. I welcome equivalently reasonable changes as opportunities to re-evaluate the strengths and weaknesses of the two approaches. The usual "trick" to favor functional code is to bring in some infinite input and rely on lazy evaluation, but anything doing a *scanr* or a *sum*[12] won't work on infinite input anyway so that opportunity is lost.

- You may consider ease of change irrelevant. As I have already mentioned, for some people is ok to say that if I change the problem, I have to reconsider the entire strategy. That might be ok within tiny programs in an academic settings. You're welcome to go work in the industry and apply that when working on any decent-size piece of software.

With that said, let's see how the Haskell and the C# code can adapt to changes.

[12] Or a number of other things, but this never gets mentioned.

Change 1 – easy
Given a density for air, water and wall, calculate the weight of the world. The Haskell code can be easily adapted to that. We can even do that by composition, without any change to the original code. We need the height of the tallest tower though, so if we use composition we cannot exploit the fact that the two scans already gave us the tallest tower (as first and last element respectively). But then, perhaps it would be considered a bad practice to rely on that.
Of course, the C# code is trivial to adapt as well. A striking difference, I would say, is that because of the way it has been written there is a clear center[13] attracting this new notion of density. While in my former code Air, Water and Wall are just quantities, they now can become quantities of some Material, with a specific Density. Our code stays close to the domain by introducing new domain concepts as our understanding evolves. Contrast that with what comes natural as an evolution of the Haskell code, which ignored the domain to begin with (take 3 densities as parameters, multiply and sum).
Similar changes, like calculating the weight of each slice, are equally easy on both sides. Of course you can't do that by composition anymore (because the final *sum* loses information), but it's an easy refactoring anyway.

Change 2 – harder than it should
Instead of giving back the total amount of water, show me what happens **while it's raining**. Even assuming a discrete time, by which at time 1 a unitary line of water is dropped into the world, some accumulates and other leaks, and then at time 2 another line of water is dropped and so on.
What is really bad here is not that we have to change the Haskell code. **It's that the entire problem-solving strategy falls down**. And yet I just asked to see "partial results" of a process. Seems innocent enough.
The C# code would require significant changes. If we want to show water falling, accumulating and then gradually leaking, we need a distinction between air and "nothing", because now water has to **move** into a surrounding cell if it contains air, and cannot just "become air". But the two boundaries must be made of something else (say, "nothing") so that water will disappear when moving there.
What is good is that the problem solving strategy stands up. The code structure stands up. I didn't speculatively made the code "general" in terms of states / substances and rules, so it's ok, I have to change it. But it doesn't fight / resist change. Actually, it accepts that change gracefully.

Change 4 – hard enough
Say that I have the notion of "hollow blocks". So in a tower we can have blocks that allow water to leak. Once again, **the entire problem-solving strategy based on the idea of finding the tallest tower etc. crumbles**. And yet it's just another innocent-looking change.
Note how the initial, inefficient idea (n * m discrete cells) would stand up easily. Just treat a hollow block as containing air, because yes, it does. So I would probably first solve it there, and

[13] As in Christopher Alexander's notion of Center.

then see if there is any real advantage in making the Slice more complicated to cater for the hollow block. Balancing efficiency and code complexity is not trivial here, and we would probably need a new notion (like a cluster of homogeneous cells in a slice), but again, it's an evolution over a well-founded model, not a "let's scrap everything and come up with something new" kind of thing.

I could go on, but the conclusion is rather clear: if you remove the domain as soon as possible so you can play with a list of integers, and concentrate on a recipe to solve that *exact* problem, and then merely abstract control flow, you get fragile code and even worse a fragile strategy, which won't accept many reasonable changes. Code that was based as much as possible on a model of the world behaves better, and the closer it was to the world[14], the better it fared. And, at the cost of repeating myself, it fared much better without any pre-cognition about the changes: the whole discussion is about **accepting changes easily**, not about pre-engineering code for extension (which I didn't).

Haskell: performance and "free" parallelism

The original question asked on stackoverflow was "*anyone can think of a time-optimized algorithm for this question*", so besides ease of change it makes sense to measure execution time for "large" number of towers. Of course creating hand-crafted skylines with millions of towers isn't practical, but we can just generate random ones and take an average or median measurement of execution time.

Haskell (and generally speaking all the lazy evaluated code) doesn't lend itself too well to measurement of execution time for purely computational parts (because they're not evaluated unless they're used in non-pure code). There is a profiling library, which however has different purposes and will slow down the code. There is a *CPUTime* module that you can use once you force the code to execute, and there is a *Control.DeepSeq* module that forces a full evaluation of a data structure, which to be honest has given me erratic results, so in the end I resorted to do the trivial thing and measure execution time *including* the side effect.

Here is the full source code, which generates a list of 10 million towers with random height between 0 and 200 and then measures the CPU time required to calculate the collected water (so the code generating the world is **excluded** from measurement).
The code was compiled with -O2 to enable optimizations; I tried to add other options that seemed potentially useful but I could not get any improvement (in most cases I got worse execution times).

[14] As in: all the optimizations were made to make it closer to the machine. But it's harder to evolve code based on Slices and easier when it's based on Cells.

```haskell
import System.Random
import System.CPUTime

rainfall :: [Int] -> Int
rainfall xs = sum (zipWith (-) mins xs)
    where mins = zipWith min maxl maxr
          maxl = scanl1 max xs
          maxr = scanr1 max xs


main = do
  g <- getStdGen
  let hs = take 10000000 (randomRs (0, 200) g :: [Int])
  startTime <- getCPUTime
  let n = rainfall hs
  putStrLn (show n)
  finishTime <- getCPUTime
  putStrLn (show (fromIntegral (finishTime - startTime) / 1000000000000))
```

I tested the code on the (unimpressive) notebook that I normally use for development: an Intel i7 5500U (2 HW cores / 4 HW threads) with a nominal frequency of 2.4 GHz and 16GB RAM. I took reasonable measures to make sure times were realistic:

- Processor was set to a fixed frequency, avoiding the usual frequency adjustment due to power savings that could indeed give erratic results. Reportedly, it was running at 121% the nominal frequency, or 2.9 GHz.

- I started the processes with a *realtime* priority, so to minimize interference from other background processes.

- Where necessary or useful, I imposed a core affinity on the process (more on this later) to force it to execute on specific cores.

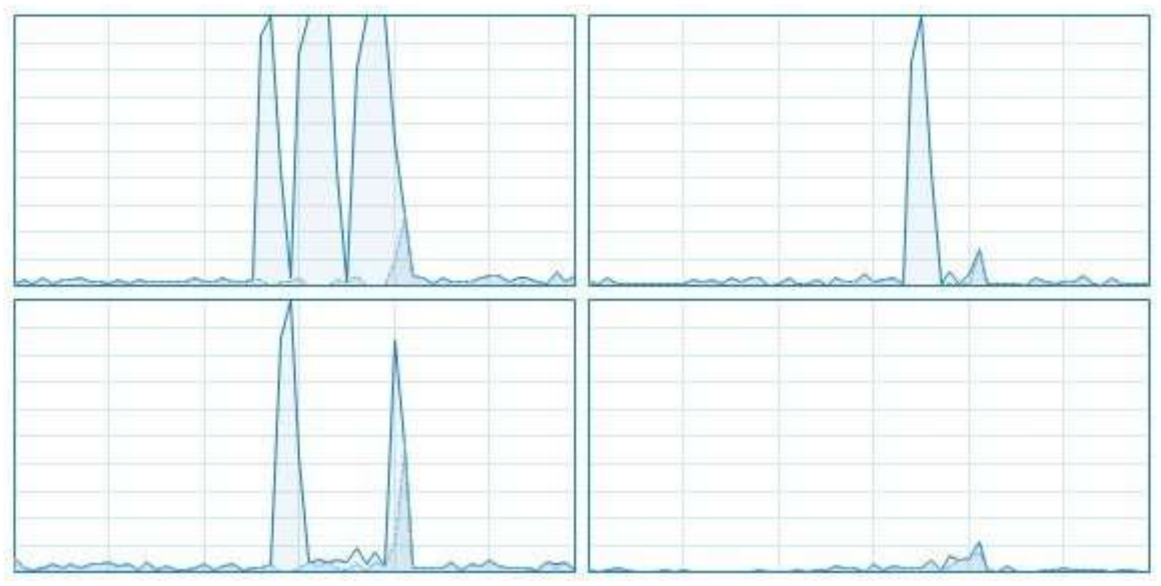
Running the code above a number of times, I got a rather stable execution time between 7.2 and 7.3 **seconds**. I'm sure you'll concur that a *putStrln* and a *show* call for a single integer number do not significantly contribute to that.

Was the code exploiting the 4 cores? After all, functional programming, purity, and relinquishing control from those pesky loops to semantically richer high-order functions are being sold as "the future" for their potential to give us parallel programming *for free*: just write pure functional code and let the compiler figure out how to make use of all those thousands of cores that we're always on the verge of having everywhere (or just the 4 I have now, thanks).

Unfortunately, that has been a promise of FP for ages but **a programming style based on scans and zips does not lend itself well to parallelization**. The only calls that could theoretically run completely in parallel in the code above are the scanl1 and scanr1. Even ignoring the problem of

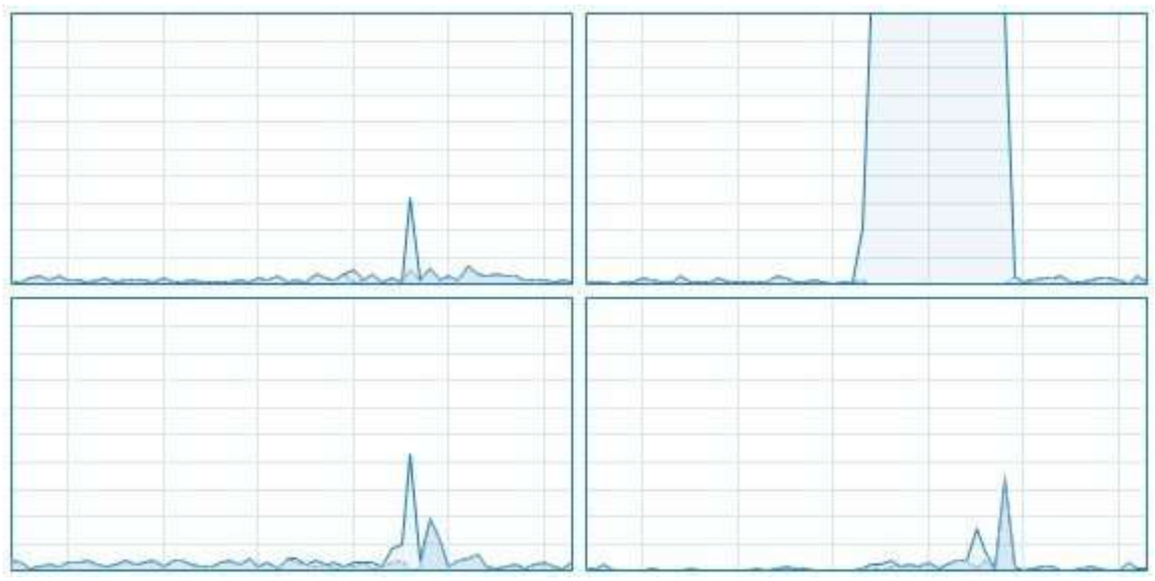
cache efficiency if you do so, **the Haskell compiler won't do that anyway**, because of her majesty Lazy Evaluation[15]. To exploit parallelism, the whole scanl1 and the whole scanr1 would have to be processed in parallel over the entire input, producing two sequences. That's just not how lazy evaluation works.
The only other chance in that code is to exploit some form of pipeline parallelism, which is not being done either. I don't know if the Haskell compiler is smart enough to figure out that the pipeline stages would be so simple that it won't work well in practice, or dumb enough that it doesn't even try. The net result, anyway, is that the code is using only one core, and without significant changes (see later) won't use your cores "for free" no matter what the FP propaganda keeps saying.

Note: a naïve look at the CPU utilization over those 7 seconds may fool some into thinking that the code is indeed using more than 1 core:



But it's simple enough to see that it's just the operating system migrating the process to different cores over time (7 seconds look like forever to a modern CPU), with only one core being used at a certain time. In fact, you can force the process to run on a single core by imposing a core affinity, so you get this picture:

[15] This is yet another concept that is being popularized as bordering on magic, while in fact it's the wrong default in a number of real-world applications.



And unsurprisingly enough, the total execution time **does not change**, because it was not using more than one core to begin with.

C#: performance and optimization

I used a similar strategy, building a world with 10 million towers, compiled with optimizations turned on and ran it a few times. Of course, the random sequences will be different than the ones I got in Haskell, but that's not significant anyway.

Here is the code for the main program (the rest being the same as in the previous C# listing).

```
class Program
{
    static void Main(string[] args)
    {
        Random r = new Random();
        int[] hs = new int[10000000];
        for(int i = 0; i < hs.Length; ++i)
            hs[i] = r.Next(200);

        double frequency = Stopwatch.Frequency;

        Stopwatch sw = Stopwatch.StartNew();
        long start = sw.ElapsedTicks;
        World sk = new World(hs);
        long lap = sw.ElapsedTicks;
        sk.Drain();
        int w = sk.Water();
        long end = sw.ElapsedTicks;
        long cre = lap - start;
        long sim = end - lap;
        long tot = end - start;
        Console.WriteLine("world creation: " + (cre / frequency) * 1000);
        Console.WriteLine("world draining: " + (sim / frequency) * 1000);
        Console.WriteLine("total: " + (tot / frequency) * 1000);
    }
}
```

Two important remarks:

- I left outside the random array generation (like in Haskell), but I have **included** the World construction time, because building the world is a necessary step when you do not build your program around a list of integers. It's the price to pay to have domain-specific types, and I wanted to consider that price.

- I took the two partial times (world construction and world draining) and the total time. In fact, I also added some code to count the average number of steps required in draining, but I'm not showing it here as it would slightly alter the timing.

I was optimist enough to move from seconds (the most convenient unit for Haskell) to milliseconds. That was almost unwarranted at first, because the average execution time on the same machine was 930 **milliseconds**. That's not too bad, considering the C# compiler is known not to optimize too well, but it's not too good either.

Having the two partial times and the counter proved useful though:

- On average, 840 ms were spent **creating the world**.

- On average, 90 ms were spent draining the world, yet only 100 to 1000 towers had to be processed, over a set of 10 million towers.

Those two numbers didn't sound right. This is where having some understanding of the virtual machine (and a programming language close enough to that virtual machine) actually helps.

The time spent on **building** the world was due to my poor choice of using a *class* for the Slice, while I should have used a *struct*. Due to the way the .NET virtual machine / type system works, an array of struct instances has a very different memory layout and performance figures than an array class instances[16].

The time spent **draining** the world couldn't possibly be 90ms for such a few towers; in fact, it was again my poor choice of returning the water by summing over the water of each tower. My algorithm managed to avoid processing all the towers to drain, and then I was processing all the towers just to get the water!

Thankfully, there is a simple alternative: calculate the amount of water for the flooded world while building the world (since I have to scan all the towers anyway) and then, while draining, simply subtract the air when a slice leaks. That will efficiently calculate the residual water.

The changes are small, but I'll show the full code anyway (except the main which wasn't impacted) with relevant changes highlighted:

```
struct Slice
{
    private int wall;
    private int water;
    private int air;

    public Slice(int wall = 0, int water = 0, int air = 0)
    {
        this.wall = wall;
        this.water = water;
        this.air = air;
    }

    public bool LeakInto(Slice target)
    {
        if(air != 0)
```

[16] An array of integers, like the one used in Haskell, would have the favorable layout afforded by value types like a struct. So no tricks here, mostly leveling the field.

```
            return false;

        int newWater = Math.Max(target.wall + target.water - wall, 0);
        air = water - newWater;
        water = newWater;
        return air > 0;
    }

    public int Air()
    {
        return air;
    }
}


class World
{
    private int width;
    private Slice[] slices;
    private int totalWater;

    public World(params int[] towerHeights)
    {
        totalWater = 0;

        width = towerHeights.Length + 2;
        int worldHeight = towerHeights.Max();
        slices = new Slice[width];

        int x = 0;
        slices[x++] = new Slice(air: worldHeight);
        foreach(int h in towerHeights)
        {
            int w = worldHeight - h;
            totalWater += w;
            slices[x++] = new Slice(wall: h, water:w );

        }
        slices[x] = new Slice(air: worldHeight);

    }

    public void Drain()
    {
        for(int x = 0; x < width - 1; ++x)
            if(!slices[x + 1].LeakInto(slices[x]))
                break;
            else
                totalWater -= slices[x + 1].Air();


        for(int x = width - 2; x >= 0; --x)
            if(!slices[x].LeakInto(slices[x + 1]))
                break;
            else
                totalWater -= slices[x].Air();
    }

    public int Water()
    {
        return totalWater;
    }
}
```

Lo and behold, I ran this code and the initial optimism paid off: the total execution time is now in the range of **130-140 ms**, with less than 1 ms spent draining the world. That's more like it, and it's already a **50x** improvement over Haskell. But of course, that's not the end of the story, because I promised you an "over **100x**" improvement.

A detour in a different language

As I mentioned, C# is not particularly good at optimizing code, and in this specific case the missing optimization was **inlining**. If you look for instance at the intermediate code (IL) for draining:

```
ldelem     SkyLine.Slice
call       instance bool SkyLine.Slice::LeakInto(valuetype SkyLine.Slice)
brfalse.s  IL_00a6
ldarg.0
ldarg.0
ldfld      int32 SkyLine.World::totalWater
ldarg.0
ldfld      valuetype SkyLine.Slice[] SkyLine.World::slices
ldloc.1
ldelema    SkyLine.Slice
call       instance int32 SkyLine.Slice::Air()
sub
stfld      int32 SkyLine.World::totalWater
```

It doesn't take a genius to see that both *LeakInto* and *Air* are being called, not inlined; being simple functions, the overhead of a call is significant enough.

Now, easy inlining is exactly one of the theoretical benefits of referential transparency, but is not an exclusive province of functional programming. In fact, old rusty languages like C++ can inline just fine. C# and C++ are syntactically close enough that a straightforward translation was certainly possible (using for instance std::vector as a counterpart for the managed array), but honestly, if we're going to rewrite that code in C++ to compare performance, it's better to think a bit about the costly phase (world construction) first.

Using a std::vector of Slices, or even a regular array of Slices, would require calling a default constructor on every instance, then overwrite each one with the appropriate values in a loop. That's really inefficient, and as I said, if we are moving from C# to C++, then we should be ready to exploit the language strengths. Now, **this is a memory-bound program**, and what we need is to avoid unnecessary memory access. A way to obtain that while still using objects is to obtain a chunk of uninitialized memory and then build each object in place using the *placement new*[17]. With that said, here is the full unglamorous code:

[17] There is also a collateral benefit doing so, that I'll discuss when talking about explicit parallelism.

```
#include <iostream>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace std::chrono;


class Slice
{
private:
	int wall;
	int water;
	int air;

public:
	Slice(int wall = 0, int water = 0, int air = 0)
	{
		Slice::wall = wall;
		Slice::water = water;
		Slice::air = air;
	}

	bool LeakInto(Slice target)
	{
		if (air != 0)
			return false;

		int newWater = max(target.wall + target.water - wall, 0);
		air = water - newWater;
		water = newWater;
		return air > 0;
	}

	int Air()
	{
		return air;
	}
};


class World
{
private:
	int width;
	int* raw;
	Slice* slices;
	int totalWater;

public:

	World(const int* heights, int count)
	{
		int worldHeight = heights[0];
		for (int i = 1; i < count; ++i)
		{
			if (heights[i] > worldHeight)
				worldHeight = heights[i];
		}

		width = count + 2;
		raw = new int[width * sizeof(Slice) / sizeof(int)];
		slices = reinterpret_cast<Slice*>(raw);
		totalWater = 0;

		Slice* cursor = slices;
		new(cursor) Slice(0, 0, worldHeight);
```

```
                for (int i = 0; i < count; ++i)
                {
                        ++cursor;
                        int w = worldHeight - heights[i];
                        totalWater += w;
                        new(cursor) Slice(heights[i], w);
                }

                ++cursor;
                new(cursor) Slice(0, 0, worldHeight);
        }

        ~World()
        {
                delete[] raw;
        }

        void Drain()
        {
                for (int x = 0; x < width - 1; ++x)
                        if (!slices[x + 1].LeakInto(slices[x]))
                                break;
                        else
                                totalWater -= slices[x + 1].Air();

                for (int x = width - 2; x >= 0; --x)
                        if (!slices[x].LeakInto(slices[x + 1]))
                                break;
                        else
                                totalWater -= slices[x].Air();
        }

        int Water()
        {
                return totalWater;
        }
};


int main()
{
        const int count = 10000000;
        int* hs = new int[count];
        for (int i = 0; i < count; ++i)
        {
                hs[i] = rand() % 200;
        }

        auto start = system_clock::now().time_since_epoch();
        World sk(hs, count);
        auto lap = system_clock::now().time_since_epoch();
        sk.Drain();
        auto end = system_clock::now().time_since_epoch();
        auto creation = duration_cast<std::chrono::milliseconds>(lap - start);
        auto draining = duration_cast<std::chrono::milliseconds>(end - lap);
        auto total = duration_cast<std::chrono::milliseconds>(end - start);
        cout << sk.Water();
        cout << "creation: " << creation.count() <<
                " draining: " << draining.count() <<
                " tot: " << total.count();

        delete[] hs;

        return 0;
}
```

Out of laziness I’ve just put all the code inline inside each class, which has the side effect of suggesting the compiler to inline the functions as well. Sure enough, we can check that it did by looking at some assembly code this time:

```
?Drain@World@@QEAAXXZ PROC           ; World::Drain, COMDAT
; 88   : {
$LN44:
         sub      rsp, 24
; 89   :          for (int x = 0; x < width - 1; ++x)
         mov      eax, DWORD PTR [rcx]
         xor      r11d, r11d
         dec      eax
         mov      r9d, r11d
         test     eax, eax
         jle      SHORT $LN39@Drain
         mov      r8d, 12
         npad     8
$LL4@Drain:
; 90   :                   if (!slices[x + 1].LeakInto(slices[x]))

         mov      rdx, QWORD PTR [rcx+16]
         lea      r10, QWORD PTR [r8+rdx]
         movsd    xmm0, QWORD PTR [r8+rdx-12]
         movsd    QWORD PTR $T1[rsp], xmm0
; 26   :          if (air != 0)

         cmp      DWORD PTR [r10+8], r11d
         jne      SHORT $LN39@Drain
```

This might not be immediately obvious if you’re not used to it, but the highlighted section shows exactly that calls to *LeakInto* (like others) were indeed inlined inside *Drain*.

Was it worth it? Well, once again, I ran the code a few times and it completed at around **59 ms**, or **122 times** faster than the Haskell code. But isn’t that cheating? We know C++ is fast. Can’t we get a 100x speedup using only that crappy (but safer) C# thing?

Explicit parallelism

I’ll start this section by removing an option from the table. One of the answers in the stackoverflow page claims that all the other solutions are not optimized and proposes some imperative code that has complexity $\mathcal{O}$(n log n) but, according to the author, if you have **n** processors you get $\mathcal{O}$(log n) complexity. That’s a completely bogus reasoning because you’re never going to have **n** processors for any given **n**. A good solution for this problem should have complexity that is *linear in n* to begin with, and ideally a performance that increases linearly with the number of processors.

Such a solution, still inspired by the original approach of finding the tallest tower etc., does indeed exist. It has been presented by Guy Steele at a Google Tech Talk back in 2015 (Four Solutions to a Trivial Problem). Guy Steele, by the way, is one of the few people I know who keeps repeating "aggregation is bad, divide and conquer is good", which basically means: scan and fold and zip etc. are bad for parallelism, map/reduce is good. He understands parallelism.

So he proposes an interesting approach, where he can divide the world in sub-worlds, each modeled by a bitonic function (watch his talk for more). It's a really interesting approach, quite math-like too, with a couple of flaws:

- Minor flaws: he explains the algorithm as it would made sense to repeat that decomposition a number of times, getting a tree. That would work very badly in practice, on contemporary CPU and memory architectures, for simple reasons of memory bandwidth and cache line contention. In practice, with today's HW it would be necessary to limit the number of splits, and then just split the list in N linear chunks to be processed in parallel. That's a minor flaw because it's easy to correct.

- Major flaw: it's even more fragile under change than the original code. The bitonic function is strictly tied to the original formulation of the problem. Change the problem (as we did in the previous section) and everything will fall down. No more parallelism. Ouch.

What about the domain-based solution? Draining the world using multiple processors is not easy. It's not undoable, but it's troublesome. The good part is: draining takes just a fraction of the execution time. It doesn't need any speedup. **Building the world takes pretty much all the time, and it's trivial to parallelize.**

Let's understand that better; to build a world we need:

- To find the world height, using max. That's trivial to parallelize using map/reduce.

- Build each slice. Slices are independent, so no synchronization or reduction would be necessary at all. That's an embarrassing parallel part of code!

- No, it isn't true, because to optimize the execution time I calculated the total amount of water in the flooded state *while building the slices*. That's a side effect and won't parallelize trivially, but it's just a sum and is easily amenable to map/reduce.

In practice, we need to face reality as well, that is (as I said above): memory bandwidth and cache line invalidation. You can't just run a number of threads over an array and expect things to scale, **even if you have no synchronization at all in your code**. That's also one of the reasons why compilers will always have a really hard time distributing work among cores in an efficient way. Especially if the language is based on lazy evaluation and the runtime cannot even guess at the number of items to be processed. Anyway, adapting the C# code was simple enough:

- For *max*, I just used the parallel collection support from the library. I haven't tried to roll my own as it was really trivial.

- To parallelize construction, I created a SubWorld class, which will initialize the Slices within a sub-world and calculate the contribution of that sub-world to the total flooded water.

- I then have to wait for the sub-worlds to complete before proceeding (and sum the water from the sub-worlds). That's the reduce step; sub-worlds do not need to synchronize with each other while running (map).

Although I have parameterized the code over the number of threads to use, on my computer the best result was obtained when using only **two** cores. Again, this is not because I need to synchronize, but because of the CPU / memory architecture. A better computer might be able to exploit more cores with the same code. Here is what I got in the end (we need only the code for ParallelWorld and SubWorld, because Slice didn't need any change). To be honest I hacked this code away quickly as a proof of concept, but it works fine.

```
class ParallelWorld
{
    private int width;
    private Slice[] slices;
    private int totalWater;

    public ParallelWorld(int par, params int[] towerHeights)
    {
        totalWater = 0;

        width = towerHeights.Length + 2;
        int worldHeight = towerHeights.AsParallel().Max();
        slices = new Slice[width];

        int x = 0;
        slices[x++] = new Slice(air: worldHeight);

        int slicesPerSubWorld = towerHeights.Length / par;
        SubWorld[] sw = new SubWorld[par];
        for(int i = 0; i < par; ++i)
        {
            int n = i == par - 1 ? towerHeights.Length - slicesPerSubWorld * (par - 1)
                                 : esPerSubWorld;
            sw[i] = new SubWorld();
            sw[i].Run(towerHeights, slices, x, n, worldHeight);
            x += n;
        }

        slices[x] = new Slice(air: worldHeight);

        for(int i = 0; i < par; ++i)
        {
            sw[i].Join();
            totalWater += sw[i].Water();
        }
    }

    public void Drain()
    {
        for(int x = 0; x < width - 1; ++x)
            if(!slices[x + 1].LeakInto(slices[x]))
                break;
            else
                totalWater -= slices[x + 1].Air();


        for(int x = width - 2; x >= 0; --x)
            if(!slices[x].LeakInto(slices[x + 1]))
                break;
            else
                totalWater -= slices[x].Air();
    }

    public int Water()
    {
        return totalWater;
    }
}



class SubWorld
{
    private int totalWater;
    private Thread th;

    private void Build(int[] towerHeights, Slice[] slices,
                       int ofs, int n, int worldHeight)
    {
        for( int i = 0; i < n; ++i )
        {
            int h = towerHeights[ofs + i - 1];
            int w = worldHeight - h;
            totalWater += w;
            slices[ofs+i] = new Slice(wall: h, water: w);
        }
    }

    public void Run(int[] towerHeights, Slice[] slices,
                    int ofs, int n, int worldHeight)
    {
        th = new Thread(() => Build(towerHeights, slices, ofs, n, worldHeight));
        th.Start();
    }

    public void Join()
    {
        th.Join();
    }

    public int Water()
    {
        return totalWater;
    }

}
```

Results are interesting: to get optimal performance, I had to manually force affinity with specific cores, again because of the way CPU and cache line invalidation works. With the "right" setting, the execution time went down from the 130-140 ms of the sequential solution to 78 msec, which is not one half, and doesn't bring us exactly an 100x improvement over Haskell, but it's close enough to call it a win.

Can we do the same with C++? Sure, with an extra bonus. In the C# code, each ParallelWorld has to call *new* to allocate slices. That means there will be concurrent calls to the .NET allocator. Depending on how smart the allocator is, there might be some synchronization going on there. But my C++ code used placement new, which is a way to say: I've already provided the memory for you, just call the constructor. No calls to the allocator. No synchronization required. Win.

In my actual code, again out of laziness, I left the parallelization of max as an exercise for the reader :-) when I ported the ParallelWorld / SubWorld to C++ land. So part of my C++ code was still sequential. Result: even in C++, 2 cores were optimal, choosing the right cores was even more important, and the execution time went down to 43 msec average (in a lucky run I got 38 ms). That's **167x** over Haskell, and it could be better: since std::thread doesn't allow me to create suspended threads, and I didn't want to go through the trouble of creating a pool etc., I'm creating the 2 threads right when I need them, so those 43 msec also account for threads creation. The actual time is therefore slightly smaller. This paper is long enough so I've left out the parallel C++ code, but it's basically just an evolution of the sequential code in the same way as the C# code is an evolution of its sequential predecessor.

Conclusions

You can focus on making your code point-free and control-flow free and math-like and dream about magic compilers spitting out incredibly optimized code for a theoretical machine where there is no such thing as cache line invalidation, or you can focus on the problem domain on one side and on understanding the real machine on the other side.

The former approach will make [some] people look at you in awe. The latter will give you code that responds better to changes in the problem domain, while improving performance and making actual use of multiple cores, even on old rusty languages.

It's a tough choice. Either way, have fun :-).

Post Scriptum: "But I want to do math"

Well, it's ok, there are many chances to apply math:

- Prove that a single pass in enough to converge to a fixed point.

- Prove that if water leaked during a left scan, you don't need to check for leaks during a subsequent right scan (hint: it's because of world-tall walls).

- Given a uniform distribution of tower heights over [0,..H] and a number N of towers, find the average number of Slices we need to process before we hit a world-tall wall (left and right scan). Too easy? Well, what about a different (non-uniform) distribution?

Generally speaking, there is often a chance to solve a problem the smart way using a little math, even in domains that seem somewhat removed from math. As I'm writing these notes, I've just solved a (moderately) complicated problem one of my clients had, which they attempted to solve with equally complicated (and buggy) code. I solved the problem by recognizing that at the bottom of what they were doing there was a rotational symmetry, which induced a finite cyclic group; I mapped that group to $\mathbb{Z}_n$, and solved the problem there. However, in the final code, this is hidden into a bottom-level function, and the rest of the code is speaking the domain language, not the language of cyclic groups. After all, that's a detail that could change as the problem itself evolves, and should not mandate the organization of the code.

Post Post Scriptum: Facts > Flames

If you take the time to write a different solution in Haskell, which:

- Still looks all cool and math-like
- Can easily adapt to changes like the ones I described
- Is not based on the ideas I described
- Can be easily parallelized in a way that works on **real** machines or even better can be made to exploit multicores by the compiler without an explicit design for parallelism
- Is at least as fast as my C# version (let's keep C++ out)

Then I'm totally interested in reading your code and learn more about how you got that. Thanks in advance, anyway.

About me

I've been programming for a long while, I've learnt a few things over time, and I'm occasionally speaking and writing about those things. I'm otherwise busy building stuff, helping companies build better stuff, running, or doing nothing. I'm @CarloPescio on twitter.