Computation at the Speed of Monads

Samuel McAravey

Brigham Young University - Idaho

mcaravey@live.com

7/14/2014

Abstract

Parallelism has been an issue as of late because of the rise of multi-core computers. Over time there have been various solutions both, elegant and clumsy, but it is still hard to create a safe parallel program. This paper will demonstrate how a simple monadic construct can create a program that is monadically guaranteed to execute correct parallelism while taking advantage of CPU power the computer has to offer. This paper will also demonstrate how this same construct tears down some of thee artificial constraints imposed by various programming languages.

I. Introduction

Today multi-core systems are becoming commonplace and developers are having to rely on libraries and packages to parallelize their programs. This has only gotten worse as computers have become more powerful. These libraries and packages don't fix the root problem, rather they are more of a Band-Aid that is painful and difficult for many developers to use. In this paper I will present a solution for the difficulties that we have with parallelism and how it can be made useful for everyday programming.

II. History

Back in the day single-core, single-CPU systems were easy to program. They traditionally had only one thread of execution, and that was all that the developer had to worry about. When parallel libraries started coming out they didn't truly provide parallelism. Rather the operating system simulated parallelism by task switching or by some other means.

The Intel Itanium is a multi-core CPU that failed in the market for several reasons. It is a CPU that the operating systems were able to actually execute in parallel, but it failed because developers did not have the tools available to

help write parallel code. In order to use the Itanium, they would have to mangle their code in such a way that the compiler would be able to split the code up onto separate cores. Most developers are not capable of manually creating parallelized code without the help of tools.

In their book, *Modern processor design : fundamentals of superscalar processors*, Shen and Lipasti stated:

> "However, the holy grail of such research - automated parallelization of serial programs - has yet to materialize. While automated parallelization of certain classes of algorithms has been demonstrated, such success has largely been limited to scientific and numeric applications with predictable flow control (e.g., nested loop structures with statically determined iteration counts) and statically analyzable memory access patterns. (e.g., walks over large multidimensional arrays of float-point data)."

In order to achieve the maximum amount of parallelism and reach this holy grail, we have to solve the parallelism problems that are facing us.

III. The Problem with Parallelism

The problem we run into is to automatically create the most amount of parallelism possible without requiring the developer to wrestle with the code to make it fit into some sort of parallel mold. Once again, this is that holy grail that we are looking for. Before we tackle this problem head-on we need to step back and analyze what is going on:

- In order to use all of the available computing resources we need to parallelize the code.
- In order to parallelize the code we need to break it up into parallel work.
- In order to break the code up into parallel work we need to avoid synchronizing unnecessarily.
- In order to avoid synchronizing unnecessarily we need to find and remove artificial synchronization points.

The first step in solving this problem is to look at what an artificial synchronization point is.

IV. Synchronization Points

We know that there are certain points in our code that forces us to synchronize before moving on. There are the obvious synchronization

points, such as blocking on a mutex, waiting for IO, a long running operation, etc. Yet there are also a couple of very important, but non-obvious synchronization points. The first is a branch, meaning any sort of a jump in the code. An *if* statement is an example of a branching jump, but a function call is also a jump.

A function call is a synchronization point because it requires that all parameters be present before the function can begin execution. This is very important because function calls are everywhere in our code, yet we don't think of them as forcing synchronization upon us. The way we can remove these blocking function calls is to use lazy values.

```
1 void Work(Func<int> x, Func<int> y)
2 {
3    ...
4 }
```

Listing 1: *Function Signature*

Calling a function using lazy values allows us execute the function without having the values of x and y available at the time of call. The values of x and y are only needed when the caller invokes the returned delegate. Because of this it is perfectly possible that the delegate is never called, meaning that x and y may never be needed, and therefore are never evaluated.

This technique also solves the more generic problem with branches: a branch may not need all of the data in order to execute. Let's look at an *if* statement as an example:

```
1 if(getA())
2    return getB();
3 else
4    return getC();
```

Listing 2: *Basic Branch*

With this little example we can see that by using lazy values we can avoid retrieving the values of either B or C. This means that by creating a function that takes only lazy values we can potentially execute a part or all of a function without ever having all of the parameters available. This forces all of the code to be very efficient by executing with only the required

data and nothing more.

What we have done is introduce partial execution into our code simply by introducing everything as a lazy value. This is and of itself is very powerful, but it doesn't quite solve the parallelism problem that we have. The reason for this is that the actual calls *getA*, *getB*, and *getC* are all blocking calls. This means that even with partial execution we still have to block while we wait for the values to arrive, which is terrible for parallelism.

In order to avoid blocking what we need to do is provide callbacks to the *getA/B/C* functions. This avoids blocking by asking for code to execute when the value arrives. We can change the previous code like so:

```
getA(callbackA);

void callbackA(bool a)
{
   if(a)
      getB(callbackB);
   else
      getC(callbackC);
}
```

Listing 3: *Branching with Callbacks*

By turning the parameters into lazy functions with callbacks, we are able to avoid blocking calls. One important thing to note is that these callbacks can be called multiple times. In order to handle this we need the *getA/B/C* functions to take another function object parameter to represent completion. Also we need to add another function object parameter to represent errors that may have happened while evaluating the lazy value. If we combine these three function object we get the following interface for our callback object:

```
interface IObserver<T>
{
   void OnNext(T value);
   void OnCompleted();
   void OnError(Exception error);
}
```

Listing 4: *IObserver*

Using this interface, we are able to provide a single callback object to the *getA/B/C* functions that can handle multiple values. Now we want to represent an object that produces values, which we can watch with an observable object:

```
interface IObservable<T>
{
   IDisposable Subscribe(IObserver<T> observer)
}
```

Listing 5: *IObservable*

Here an observable creates values and let's the observer know when it is done. Using these constructs we can accurately represent the creation of values and the use of those values.

V. A New Kind of Function

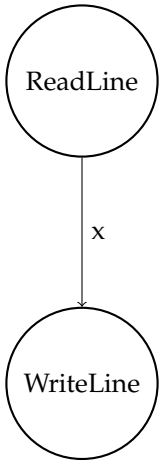
I propose that all function parameters and return values be represented using these observables and observers in place of traditional representations. By doing this we can still treat values as values in our minds, while allowing us to be more explicit and accurate in our program definition. I realize that using these constructs is complicated and can seem like a lot of extra information, but all of this can be hidden behind carefully crafted constructs or a programming language to make things easier for the developer.

The *Observer*, *Observable* pattern is a well known design pattern. It allows an object to produce values and notify an Observer when those values are available. Those who are familiar with the [Reactive Extensions] (Rx) by Microsoft will recognize the *IObserver* interface and the *IObservable* interface. There are several interviews explaining the mathematical origins and the exact reason for the definitions of the *IObserver* and *IObservable* interfaces, neither of which are within the scope of this article [Meijer, 2014]. It is worthwhile to note that these two interfaces are monadic, and as such, all of the benefits of monads apply.

By using the *IObserver* and *IObservable* interfaces we are able to provide the compiler with the information it needs to completely parallelize our programs without us needing to mangle our code. Please note that this will not fix poor algorithms, but it will be able to automatically break up code into work that is parallelizable.

To understand what we are doing from a high level, it's useful to visualize a tree of the

dependencies. Take a simple program that reads in a string and prints it back out to the console. The code and subsequent dependence tree might look like this:



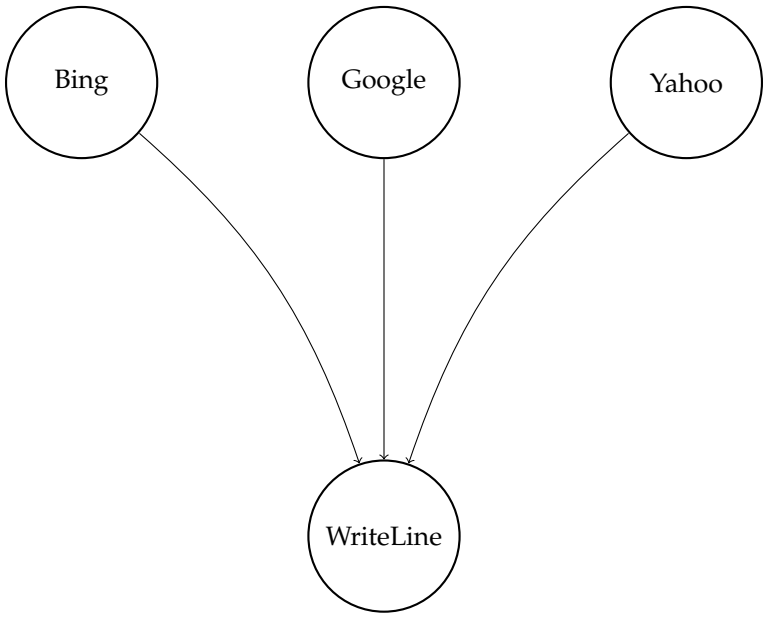

```
IObservable<string> read = Console.ReadLine();
IObservable<Unit> write = Console.WriteLine(read);
write.Subscribe();
```

Listing 6: *ReadLine WriteLine*

Looking at the dependency graph we know that if we simply satisfy the dependency, it does not matter how the code is written. In fact, we can "*call*" the *WriteLine* function before the *ReadLine* function and it will still execute correctly.

This is possible because when we call *WriteLine* we are providing a value which doesn't exist yet, but that will arrive in the future. We can then perform a *ReadLine*, take the result and set the value x with it. Now that the dependency has been fulfilled the *WriteLine* is able to execute and print out to the console.

Another powerful example is the automatic parallelism. Since we are creating dependency trees in our code, the compiler is able to execute separate branches of the tree in parallel. A simple example of this is retrieving data from the web. The following is an example tree and code:




```
using (var client = new HttpClient())
{
   string bing = await client.GetStringAsync(@"http://www.bing.com");
   string google = await client.GetStringAsync(@"http://www.google.com");
   string yahoo = await client.GetStringAsync(@"http://www.yahoo.com");

   Console.WriteLine(bing);
   Console.WriteLine(google);
   Console.WriteLine(yahoo);
}
```

Listing 7: *Basic Web Service Requests*

```
using (var client = new HttpClient())
{
   var obsClient = Observable.Return(client);
   var bingUri = Observable.Return(@"http://www.bing.com");
   var googleUri = Observable.Return(@"http://www.google.com");
   var yahooUri = Observable.Return(@"http://www.yahoo.com");

   // The GetString method is an extension,
   // the details of which are not important here.
   var bing = obsClient.GetString(bingUri);
   var google = obsClient.GetString(googleUri);
   var yahoo = obsClient.GetString(yahooUri);

   var bingWrite = Console.WriteLine(bing);
   var googleWrite = Console.WriteLine(google);
   var yahooWrite = Console.WriteLine(yahoo);

   var result = Observable.WaitForValues(bingWrite, googleWrite, yahooWrite);
   await result;
}
```

Listing 8: *Parallel Web Service Requests*

After running several tests, the observable code appears to executes on average about 20% faster than the asynchronous code. Without having to structure the code in a deformed way, we were able to get speed ups in our code automatically.

VI. Summary

The driver behind using the *IObserver* and *IObservable* interfaces is to allow the maximum amount of parallelism without forcing the developer to think in drastically different ways. We are trying to solve the problems facing us because of parallelism. Let's remind ourselves what we have accomplished with this:

- By removing artificial synchronization points, we are able to avoid synchronizing unnecessarily.
- By avoiding synchronizing unnecessarily, we are able to automatically break up the code into parallel work.
- By automatically breaking up the code into parallel work, we are able to parallelize the code.
- By being able to parallelize the code, we are able to use all of the available computing resources.

With this new kind of function, we are able to maximize the resources provided to us by

the computer with minimal effort. At the moment there is a lot of extra syntax required to accurately express this kind of function in existing languages, but the idea has proven to be very useful at solving the parallelism problems of today.

References

[Meijer, 2014] Erik Meijer (2014). Duality and the End of Reactive `http://channel9.msdn.com/events/Lang-NEXT/Lang-NEXT-2014/Keynote-Duality`

[Reactive Extensions]

`https://rx.codeplex.com/`

List of listings