Concurrency & Reactive programming with F#

… because the future is Immutable!

Writing code that scales to a large number of cores is much easier in the functional style compared to using the typical imperative approach…

Riccardo Terrell

Agenda

- Concurrency & Terminology
- Event & Reactive Programming
- Parallel Programming
- Asynchronous Workflows
- Actor model
- GPGPU Programming
- Async Patterns
- Software Transactional Memory

Something about me – Riccardo Terrell

- Originally from Italy. I moved here ~8 years ago
- +/- 15 years in professional programming
  - C++/VB → Java → .Net C# → C# & F# → ??
- Organizer of the DC F# User Group
- Software Architect at Microsoft
- Passionate in Technology, believer in polyglot programming as a mechanism in finding the right tool for the job
- Lucky husband and pug lover

Objectives

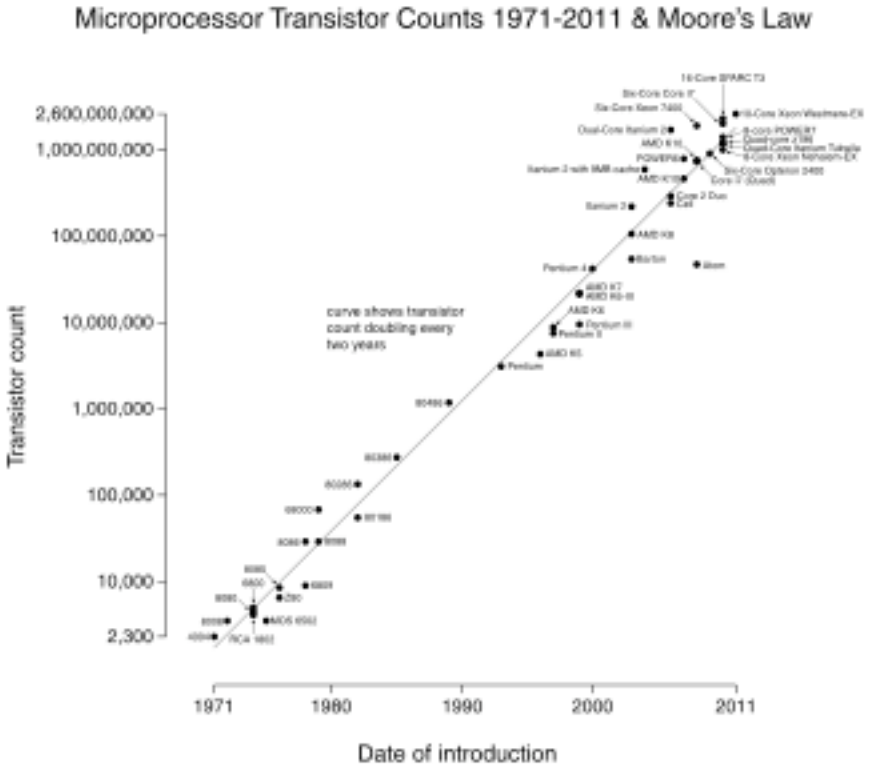
- Immutability and Isolation are your best friends to write concurrent application
  - Use natural isolation, the Actor model is a great to enforce coarse-grained isolation through message-passing
- Asynchronous Programing in F# is easy and declarative
- Concurrency in F# is fully integrated with .NET
- Use Actor Model for high scalable computation

Concurrency & Terminologies

Modern computers come with multiple processors, each equipped with multiple cores, but the single processor is slower than used to be!

Moore's law *(http://en.wikipedia.org/wiki/Moore's_law)*

… to achieve great performances the application must be leveraging a Concurrent Model



Concurrency & Terminologies

- Concurrency is Everywhere
  - Server-side programming for the web or cloud
  - Building a responsive graphical user interface
  - Building a new interactive client application
  - Performing some compute- or I/O-intensive activity

All to attain better performance!

Concurrency & Terminology

Concurrency is the notion of multiple things happening at the same time.

- **Concurrent:** programs with multiple threads of execution, each typically executing different code
- **Parallel:** one or more threads executing simultaneously in order to speed up execution
- **Asynchronous:** programs perform requests that don't complete immediately but that are fulfilled at a later time and where the program issuing the request has to do meaningful work in the meantime
- **Reactive** programming is writing applications in such a way that they respond to events as they occur in real time.

Concurrency & Terminology

- **Concurrent** programming
  - Make interactive programs more scalable (F# Agents, Reactive Extensions)
- **Parallel** programming
  - Run CPU-intensive tasks on multi-core faster (Task Parallel Library (TPL), F# async)
- **Asynchronous** programming
  - Avoid blocking of threads when waiting for I/O (F# asynchronous workflows)
- **Reactive** programming
  - User-interfaces with lots of cheap event handling (F# events, Reactive Extensions)

Concurrency jams to avoid!

Parallel, asynchronous, concurrent, and reactive programs bring many challenges because this programs are nearly always *nondeterministic*

- This makes debugging/testing challenging
- Deadlocking & Livelocking
- Not easy to program
- Concurrency is a double-edged sword

Concurrency… *F# can Help!*

F# advances in concurrency programming

- Immutability, we don't have to protect the "shared state”
- A function call never changes data, it only returns new data
  - Side effect free functions
- Higher-order function and Lazy Evaluation
- Asynchronous Workflow
- Actor-Based concurrency model - MP

Immutability

- Immutability is very important characteristic of FP, is about transformation of state
- Immutable data forces you to use a “transformational” approach

- writing programs using immutable types the only thing a method can do is return a result, it can’t modify the state of any objects
- Immutable data makes the code predictable is easier to work, prevent Bugs
- *Immutability avoid side effects*
- Side effects aren’t all that bad, but unintended side effects are the root of many bugs
- Concurrency is much simpler, as you don’t have to worry about using locks to avoid update conflicts
  - Automatic thread-safe

Immutability

Mutable values cannot be captured by closures!

```
List<Func<int>> actions = new List<Func<int>>();
for (int i = 0; i < 5; i++) {
    actions.Add(() => i * 2);
}
foreach (var action in actions){
    Console.WriteLine(action());
}
```

```
let actions = List.init 5 (fun i -> fun() -> i * 2)
for act in actions do
    printfn "%d " (act())
```

EVENTs

Event & Reactive Programming in F#

- **Events** are used to notify that something happened in the application, is something you can listen to by registering a callback with the event
- **Events** are first class composable entities much like object and functions, I can pass them as arguments to functions and returned as results
- **Events** use combinators to take an existing event and transform it into a new event-stream in a compositional way
- **Event-Streams** can be treats as a collection

Event Combinators

The biggest benefit of using higher-order functions for events is that we can express the flow in a Declarative way

- **What** to do with received data using event **combinators**
- Use functions for working with *event values*

Observalbe Module

```
let myEvent = Event<int>()

myEvent.Publish
    |> Observable.map(fun n -> n.ToString())
    |> Observable.filter(fun n -> n <> "")
    |> Observable.choose(fun n -> Some n)
```

Seq Module

```
let myData = seq {1..10}

myData
    |> Seq.map(fun n -> n.ToString())
    |> Seq.filter(fun n -> n <> "")
    |> Seq.choose(fun n -> Some n)
```

Event map & filter & add

- Filter and trigger events in specific circumstances with `Event.filter`
- Create a new event that carries a different type of value with `Event.map`

```
Event.map    : ('T -> 'R)   -> IEvent<'T> -> IEvent<'R>
Event.filter : ('T -> bool) -> IEvent<'T> -> IEvent<'T>
```

- Register event with `Event.add`

```
Event.add : ('T -> unit) -> IEvent<'Del,'T> -> unit
```

Event merge & scan

- Merging events with `Event.merge`
  - Triggered whenever first or second event occurs
  - Note that the carried values must have same type

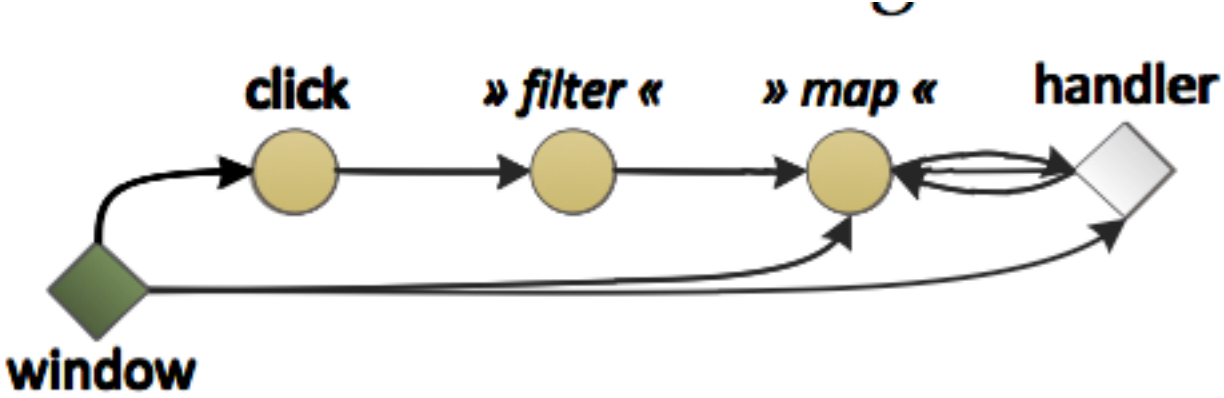
```
IEvent<'T> -> IEvent<'T> -> IEvent<'T>
```

- Creating stateful events with `Event.scan`
  - State is recalculated each time event occurs
  - Triggered with new state after recalculation

```
('St -> 'T -> 'St) -> 'St -> IEvent<'T> -> IEvent<'St>
```

Event are IObservable… *almost*

- Memory leak ☹
  - IEvent does not support *removing* event handlers *(RemoveHandler on the resulting event, it leaves some handlers attached… leak!)*
  - IObservable is able to remove handlers - **IDisposable**



Event are IObservable… *almost*

- Memory leak ☹
  - IEvent does not support *removing* event handlers *(RemoveHandler on the resulting event, it leaves some handlers attached… leak!)*
  - IObservable is able to remove handlers
- Event.scan
  - Event.scan can have attached multiple handlers to the resulting event and they will see the same state
  - IObservable creates a "new state" for every attached handler, it keeps separate state for each handler
  - the main difference is in the **statfullness** - if you want to share state, you can use the Event module - implementing the same using Observable is possible but harder
- .NET Compatibility, if you want to declare events usable from C# then you need to create properties of type IEvent and so you need to use Event combinators

Observable in F#

- Event<'T> interface inherits from IObservable<'T>
  - We can use the same standard F# Events functions for working with Observable

```
Observable.filter : ('T -> bool) -> IObservable<'T> -> IObservable<'T>
Observable.map    : ('T -> 'R)   -> IObservable<'T> -> IObservable<'R>
Observable.add    : ('T -> unit)   -> IObservable<'T> -> unit
Observable.merge  : IObservable<'T> -> IObservable<'T> -> IObservable<'T>


      Observable.subscribe  : IObservable<'T> -> IDisposable
```

Accessing F# events from .NET languages

- Events in F# are values of type
  - `Event<'T>` implementation for the `IEvent<'T>`
  - `Event<'Delegate,'Args>` implementation for the delegate type following .NET standard
- Create the events using new Event<DelegateType, Args> instead of new Event<Args>
- CLIEventAttribute instructs

  F# to generate .NET event (+= & -=)

```
[<CLIEventAttribute>]
member x.NameChanged = nameChanged
```

Wait asynchronously for Event

- `Async.AwaitObservable & Async.AwaitEvent` operations
  - Create workflow that waits for Events Asynchronously

```
AwaitObservable : IObservable<'T> * IObservable<'U>
                  -> Async<Choice<'T, 'U>>
```

```
let! evt = Async.AwaitObservable
             (main.MouseLeftButtonDown, main.MouseMove)
match evt with
| Choice1Of2(up) ->    // Left button was clicked
| Choice2Of2(move) ->  // Mouse cursor moved }
```

Event & Reactive Programming

THREADING

Threading

Is the smallest unit of processing that can be scheduled by an Operation System…

http://en.wikipedia.ork/wiki/Thread_(computing)

- *Threads share the memory heap*
- *Threads can communicate by writing to this shared memory*
- *Each Thread receives its own stack space ~1 Mb*

Threads can be evil!

- Spawning multiple threads on a single-CPU~*system won't hurt anything*~, but the processor can only execute one of those threads at a time
- Spawning new threads can be costly

  each thread has its own stack to track
- Possible race conditions …and Deadlocks
- Hard to stop (Thread.Abort)
- Complicated programming model

  for synchronization (Mutex, Semaphore…
- Waiting for result?

Threading – Race Conditions

A race condition is when you have two threads trying to read or write the same reference at the same time

```
let sumArray (arr : int[]) =
    let total = ref 0
    let thread1Finished = ref false
    ThreadPool.QueueUserWorkItem(
        fun _ -> for i = 0 to arr.Length / 2 - 1 do
                    total := arr.[i] + !total
                 thread1Finished := true
    ) |> ignore

    let thread2Finished = ref false

    ThreadPool.QueueUserWorkItem(
        fun _ -> for i = 0 to arr.Length / 2 - 1 do
                    total := arr.[i] + !total
                 thread2Finished := true
    ) |> ignore

    while !thread1Finished = false ||
          !thread2Finished = false do
          Thread.Sleep(0)
    !total
```

```
let lockedSumArray (arr : int[]) =
    let total = ref 0
    let thread1Finished = ref false
    ThreadPool.QueueUserWorkItem(
        fun _ -> for i = 0 to arr.Length / 2 - 1 do
                    lock (total) (fun () -> total := arr.[i] +
                    thread1Finished := true
    ) |> ignore

    let thread2Finished = ref false

    ThreadPool.QueueUserWorkItem(
        fun _ -> for i = arr.Length / 2 to arr.Length - 1 do
                    lock (total) (fun () -> total := arr.[i] +
                 thread2Finished := true
    ) |> ignore

    while !thread1Finished = false ||
          !thread2Finished = false do
          Thread.Sleep(0)
    !total
```

Dead Lock!

Threading in F#

There are three concepts from the functional world that are essential for parallel computing

1. declarative programming style
2. working with immutable data structures
3. side effect-free functions are *important*

The code becomes more declarative when using immutable data

Imperative

```
let imperativeSum numbers =
    let mutable total = 0
    for i in numbers do
        let x = square i
        total <- total + x
    total
```

declarative

```
let functionalSum numbers =
    numbers
    |> Seq.map square
    |> Seq.sum
```

parallel

```
let functionalSum numbers =
    numbers
    |> PSeq.map square
    |> PSeq.sum
```

Threading

Full .NET Integration with

- *System.Threading & System.Threading.Task*
- Parallel Library (TPL) execute tasks (primitive units of work) in parallel
- Parallel LINQ (PLINQ) used for writing data parallel code
- *Reactive Extensions*

```
let pfor nfrom nto f =
   Parallel.For(nfrom, nto + 1, Action<_>(f)) |> ignore

[| 1000..10000000 |]
 |> Array.Parallel.map (isPrime)

ThreadPool.QueueUserWorkItem(new WaitCallback(printNumbers), box 5)

let spawnTask action =
    Task.Factory.StartNew(action)
```

Threading

DEMO ☺

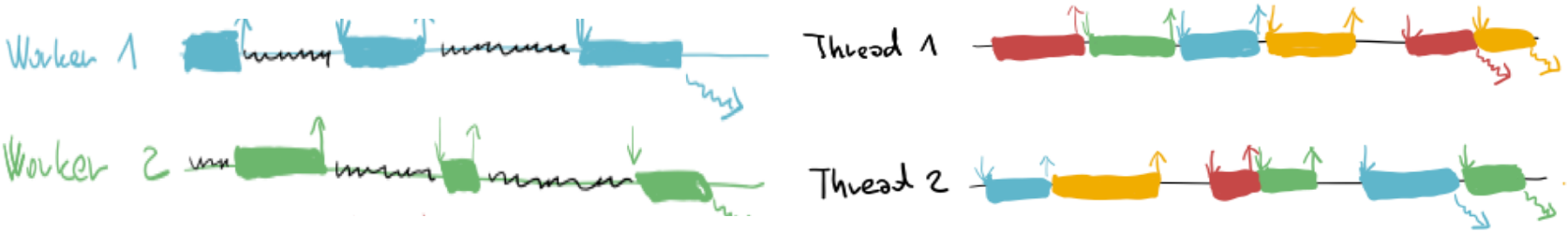
ASYNC

Asynchronous Workflows

- Software is often I/O-bound, it provides notable performance benefits
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disk
- Network and disk speeds increasing slower
- Not Easy to predict when the operation will complete (no-deterministic)

Classic Asynchronous programming

- We're used to writing code linearly



- Asynchronous programming requires decoupling Begin from End
- Very difficult to:
  - Combine multiple asynchronous operations
  - Deal with exceptions, cancellation and transaction

Synchronous Programming

- Synchronous I/O or user-interface code

```
let wc = new WebClient()
let data = wc.DownloadData(url)
outputStream.Write(data, 0, data.Length)
```

- Blocks thread while waiting
  - Does not scale
  - Blocking user interface – when run on GUI thread
  - Simple to write – loops, exception handling etc

Classic Asynchronous Programming

```
let wc = new WebClient()
wc.DownloadDataCompleted.Add(fun e ->
  outputStream.BeginWrite( e.Result, 0, e.Result.Length,
      (fun ar -> outputStream.EndRead(ar)), null))
wc.DownloadDataAsync(url)
```

- Writing the code that performs asynchronous operations is difficult to implement using the current techniques
- Two different programming models
  - BeginFoo & EndFoo methods
  - FooCompleted & FooAsync using events
- **Operation completes in different scope**

Classic Asynchronous Programming

```
let openFileCallback() =
    let fs = new FileStream(@"C:\Program Files\..,
                            FileMode.Open, FileAccess.Read,
                            FileShare.Read)
    let data = Array.create (int fs.Length) 0uy
    let callback ar =
        let bytesRead = fs.EndRead(ar)
        fs.Dispose()
        printfn "Read Bytes: %i, First bytes were: %i %
                        %i ..."
            bytesRead data.[1] data.[2] data.[3]
    fs.BeginRead(data, 0, data.Length,
                        (fun ar -> callback ar), null) |> ignore
```

Anatomy of Asynchronous Workflows

```
let getLength url = async {
  let wc = new WebClient()
  let! data = wc.AsyncDownloadString(url)
  return data.Length }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Anatomy of Asynchronous Workflows

```
let openFileAsynchronous =
    async { use  fs = new  FileStream(@"C:\Program Files\..., …)
            let  data = Array.create (int fs.Length) 0uy
            let!  bytesRead = fs.AsyncRead(data, 0, data.Length)
            do  printfn "Read Bytes: %i, First bytes were:
            %i %i %i ..." bytesRead data.[1] data.[2] data.[3] }
```

- Async defines a block of code we would like to run asynchronously
- We use let! instead of let
  - let! binds asynchronously, the computation in the async block waits until the let! completes
  - While it is waiting it does not block
  - No program or OS thread is blocked

Anatomy of Asynchronous Workflows

```
let ProcessImageAsync(i) =
  async {   use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
            let! pixels = inStream.AsyncRead(numPixels)
            let pixels' = ProcessImage(pixels, i)
            use outStream = File.OpenWrite(sprintf "Image%d.done" i)
            do! outStream.AsyncWrite(pixels') }
let ProcessImagesAsync() =
  let tasks = [ for i in 1..numImages -> ProcessImageAsync(i) ]
  let parallelTasks = Async.Parallel tasks
  Async.RunSynchronously parallelTasks
```

Anatomy of Asynchronous Workflows

```
async { let! image = AsyncRead ”bugghina.jpg"
        let image2 = f image
        do! AsyncWrite image2 ”dog.jpg"
        do printfn "done!"
        return image2 }
```

```
async.Delay(fun () ->
    async.Bind(ReadAsync ”bugghina.jpg", (fun image ->
        let image2 = f image
        async.Bind(writeAsync ”dog.jpg",(fun () ->
            printfn "done!"
            async.Return())))))
```

Async - Exceptions

If an exception from an async workflow isn't caught the exception bubble up and eventually bring down the whole process!

To catch unhandled exceptions from async workflows, you can use

- Async.StartWithContinuations
- Async.Catch

```
asyncTaskX
|> Async.Catch
|> Async.RunSynchronously
|> function
   | Choice1Of2 result     -> printfn "The async operation completed with %A" result
   | Choice2Of2 (ex : exn) -> printfn "Exception thrown: %s" ex.Message
```

Async – Parallel

Creates an asynchronous computation that executes all the given asynchronous computations queueing each as work items and using a fork/join pattern.

```
let rec fib x = if x <= 2 then 1 else fib(x-1) + fib(x-2)

let fibs =
    Async.Parallel [ for i in 0..40 -> async { return fib(i) } ]
    |> Async.Catch
    |> Async.RunSynchronously
    |> function
        | Choice1Of2 result -> printfn "Successfully %A" result
        | Choice2Of2 exn -> printfn "Exception occurred: %s" exn.Message
```

Async - Cancellation

The Asynchronous workflows can be cancelled!

Cancelling an async workflow is a request …
The task does not immediately terminate

- *Async.TryCancelled()*
- *CancellationToken()*

```
let computation         = Async.TryCancelled(cancelableTask, cancelHandler)

let cancellationSource = new CancellationTokenSource()
Async.Start(computation, cancellationSource.Token)

cancellationSource.Cancel()
```

Async - StartWithContinuations

- What do we do if our parallel code throws an exception?
- What do we do if we need to cancel our parallel jobs?
- How can we safely update the user with correct information once we have our results?
  - If we want more control over what happens when our async completes (or fails) we use Async.StartWithContinuations

```
let asyncOp (label:System.Windows.Forms.Label) filename =
    Async.StartWithContinuations(
        async { use outputFile = System.IO.File.Create(filename)
                do! outputFile.AsyncWrite(bufferData)  },
        (fun _ -> label.Text <- "Operation completed."),
        (fun _ -> label.Text <- "Operation failed."),
        (fun _ -> label.Text <- "Operation canceled."))
```

Async – Task Parallel .NET

TPL Tasks integrated in .NET Async Workflows

```
let getStreamData (uri:string) =
  async { let request = WebRequest.Create uri
          use! response = request.AsyncGetResponse()
          return [use stream = response.GetResponseStream()
                  use reader = new StreamReader(stream)
                  while not reader.EndOfStream
                    do yield reader.ReadLine()] }

let result = Async.CreateAsTask <| getStreamData myUri
do result.Start()
let resultValue = result.Value
```

Async – Limitations

Executing code in parallel there are numerous factors to take into account

- The number of processor cores
- Processor cache coherency
- The existing CPU workload
- There is no throttling of executing threads to ensure an optimal usage
- For CPU-level parallelism, use the .NET Task Parallel Library

Async

DEMO 

Actor Model

Concurrent Model Programming

An Agent is an independent computational entity which contains a queue, and receives and processes messages

It provides immutability and isolation

(it enforces coarse-grained isolation through message-passing)

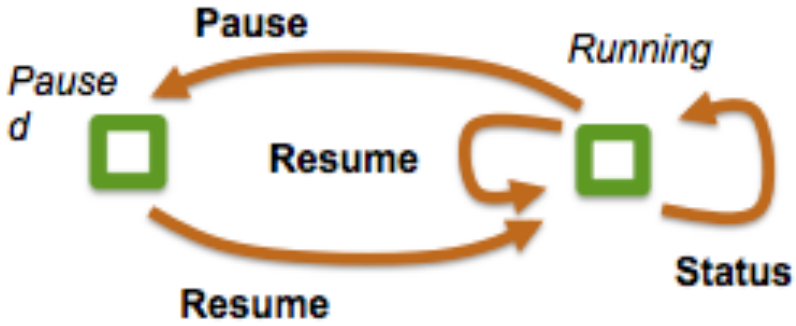
What are F# Agents?

The Actor model is a model of concurrent computation using actors which is characterized by dynamic creation of actors, inclusion of actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

[Wikipedia]

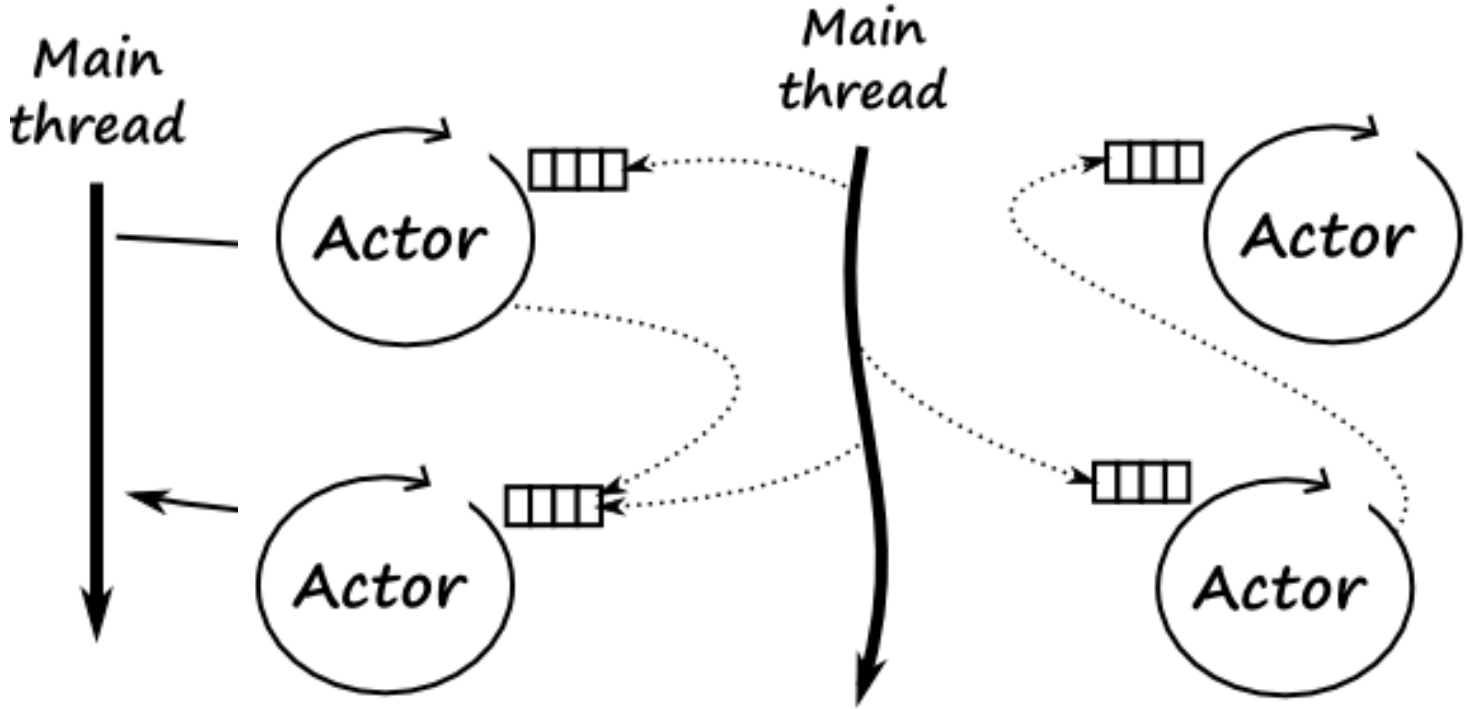
What are F# Agents?

- Receiving status updates with pause
  - Handle all messages when *Running*
  - Does not handle **Status** when *Paused*



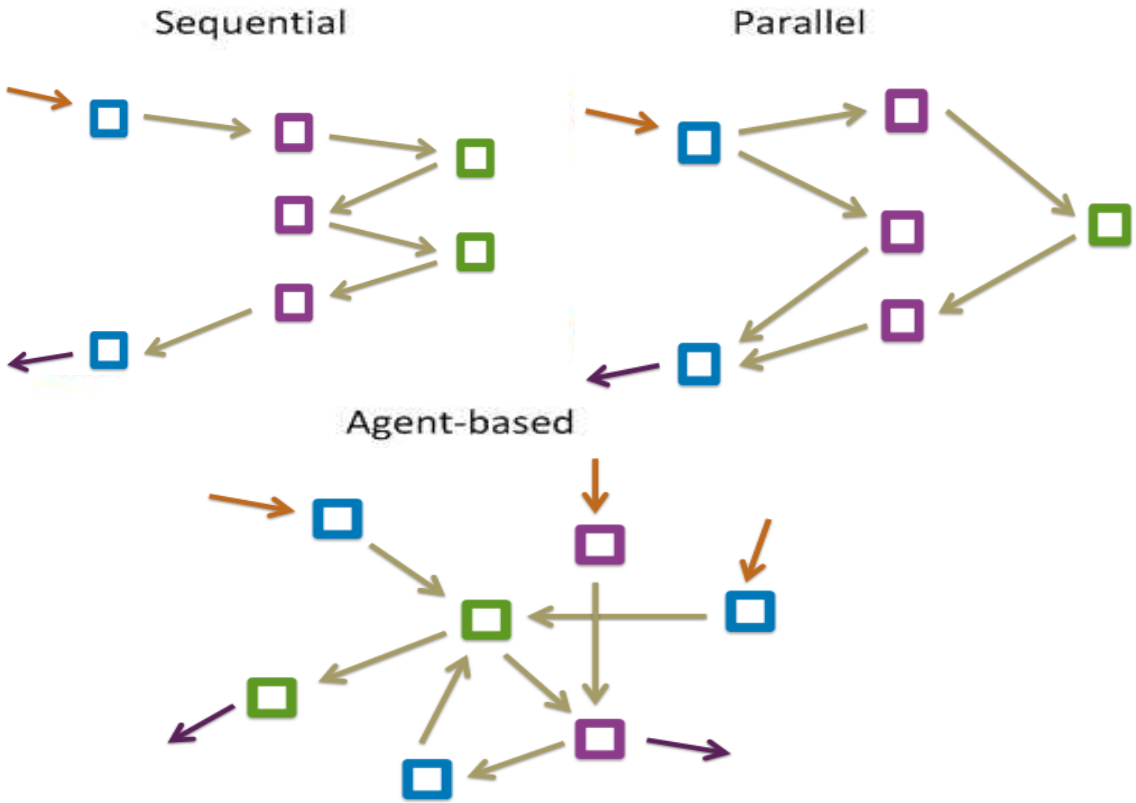
- Multi-state agents use state machines
  - Easy to implement as recursive functions
  - Some states may leave messages in the queue

What are F# Agents?



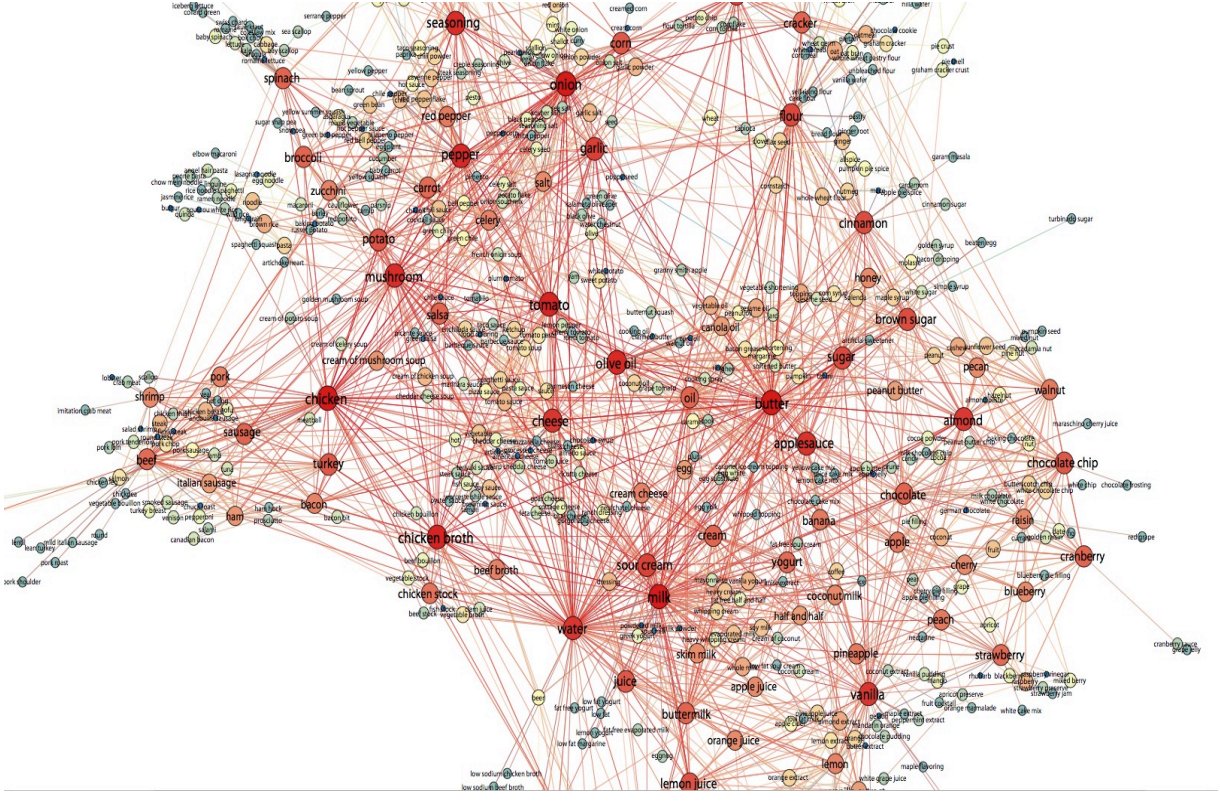
Agent Model

What does a system of actors look like?



What is an Agent?

What does a system of actors look like?



How does an Agent look?

What does a system of actors look like?

What Agent can do? Agents **perform actions**

- Buffering Messages
- Asynchronous Execution
- Determine what to do with income message
- Change its behavior for the next messages
- Communicate and notify other actors and create more Agents
- Send reply to the sender of a message
- Notify other Agents
- Do calculations and update state

Mutable and Immutable state

- **Mutable** state
  - Accessed from the body
  - Used in loops or recursion
  - Mutable variables (ref)
  - Fast mutable collections

```
Agent.Start(fun agent -> async {
  let names = ResizeArray<_>()
  while true do
    let! name = agent.Receive()
    names.Add(name) })
```

- **Immutable** state
  - Passed as an argument
  - Using recursion (**return!**)
  - Immutable types
  - Can be returned from Agent

```
Agent.Start(fun agent ->
  let rec loop names = async {
    let! name = agent.Receive()
    return! loop (name::names) }
 loop [])
```

Declaring messages

- Agents handle multiple messages
  - Message type using discriminated union

```
type CacheMessage<'T> =
  | Add of string * 'T
  | Get of string * AsyncReplyChannel<option<'T>>
  | Reset
```

- Safety guarantees
  - Agent will be able to handle all messages

Actor Model

- **Message** carries input and a callback

```
type Message = Incremet of int
             | Fetch of AsyncReplyeChannel<int>
```

- **Reply** using the callback object

```
let! name, rchan = agent.Receive()
                   rchan.Reply("Hello " + name) })
```

- **Asynchronous** communication

```
let! s = echo.PostAndAsyncReply
             (fun ch -> “Bugghina", ch)
```

Actor Model

```
let hello = Agent.Start(fun agent -> async {
  while true do
    let! name = agent.Receive()
    printfn "Hello %s"  name
    do! Async.Sleep 500 })

hello.Post(“Bugghina")
```

- Single instance of the body is running
- Waiting for message is asynchronous
- Messages are queued by the agent

Actor Model methods

- *Post* - asynchronously posts message in *MailboxProcessor* message queue.
- *PostAndReply* - synchronously sends message to MailboxProcessor and awaits for reply. *PostAndReply* optionally accept timeout parameter. If reply wasn't received after timeout expiration then TimeoutException is raised
- *TryPostAndReply* - similar to *PostAndReply* but returns None if method doesn't return in given timeout
- *PostAndAsyncReply* - sends message to *MailboxProcessor* and returns computation that await the reply.
- PostAndTryAsyncReply - similar to *PostAndAsyncReply* but returns None if methods doesn't return in given timeout

Scan - TryScan

Actor can be in a state in which wait for specific Message(s) and *Scan* (*or TryScan*) a message by looking through messages in arrival

```
let agent = Agent.Start(fun agent ->
        (*...*)
        let! msg = agent.Scan(fun Resume -> reurn! ...
        (*...*)
)
```

Encapsulating Actors

```
type internal OnePlaceMessage<'T> =
  | Put of 'T
  | Get of AsyncReplyChannel<'T>

type OnePlaceAgent<'T>() =
  let agent = Agent.Start(fun agent -> (* ... *))
  member x.Put(value) = agent.Post(Put value)
  member x.AsyncGet() = agent.PostAndAsyncReply(Get)
```

- Ordinary methods for encapsulating `Post`
- Asynchronous methods when waiting for a reply

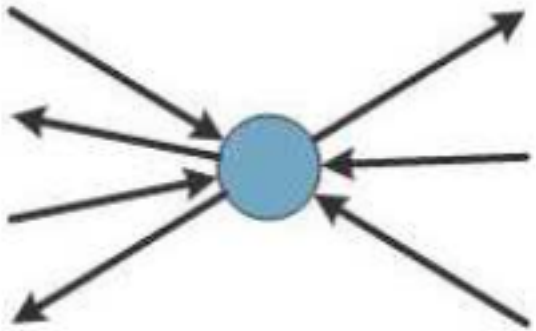
Error Handling

```
let supervisor =
   Agent<System.Exception>.Start(fun inbox ->
     async { while true do
                let! err = inbox.Receive()
                printfn "an error occurred in an agent: %A" err })

let agent =
   new Agent<int>(fun inbox ->
     async { while true do
                let! msg = inbox.Receive()
                if msg % 1000 = 0 then
                    failwith "I don't like that cookie!" })

agent.Error.Add(fun error -> supervisor.Post error)
agent.Start()
```

- We all make mistakes!

Agent-based Patterns

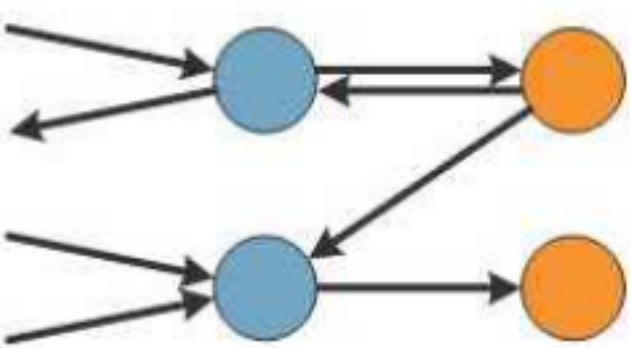
- Worker Agent
  - useful when the application needs to run some stateful computations on a shared state



```
let chat = Agent.Start(fun agent ->
  let rec loop elements = async {
    let! msg = agent.Receive()
    match msg with
    | SendMessage text ->
        // Add message to the list & continue
        let element = XElement(XName.Get("li"), text)
        return! loop (element :: elements)
    (..............)
    }
  loop [] )
```

Agent-based Patterns

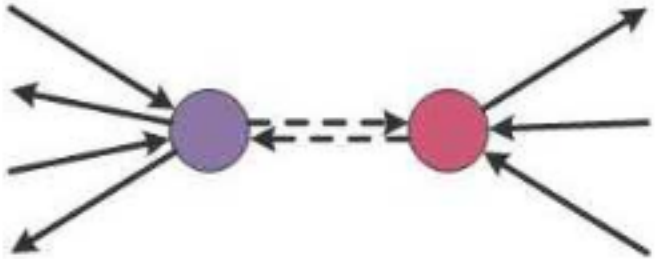
- Layered Agent
  - is useful when we need to perform some pre-processing or checking before calling another agent to do the actual work



```
let caching = Agent.Start(fun agent -> async {
  let table = Dictionary<string, string>()
  while true do
    let! msg = agent.Receive()
    match msg with
    | Add(url, html) -> table.Add(url, html)
    | Get(url, repl) ->
        if table.ContainsKey(url) then
          repl.Reply(Some table.[url])
        else
          repl.Reply(None)
```

Agent-based Patterns

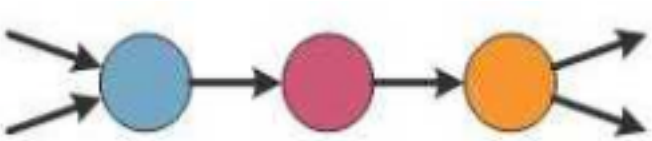
- Proxy Agent
  - is useful when the actual agent that implements the functionality cannot be directly accessed or needs to be protected in some way



```
[<AbstractClass>]
type AgentRef<'a>(id:string) =
    abstract Receive : unit -> Async<'a>
    abstract Post : 'a -> unit
    abstract PostAndTryAsyncReply : (IAsyncReplyChannel<'b> -> 'a)

type Agent<'a>(id:string, comp, ?token) =
    inherit AgentRef<'a>(id)
    let agent = MailboxProcessor<'a>>

    override x.Post(msg:'a) = agent.Post(msg)
    override x.PostAndTryAsyncReply(builder) = agent.PostAndTryAsy

    override x.Receive() = agent.Receive()
```

Agent-based Patterns

- Pipeline Processing
  - is useful when you want to process data in multiple steps



```
let inputs = new BlockingQueueAgent<_>(length)
let step1 = new BlockingQueueAgent<_>(length)

let loadData = async {
  while true do
    let value = loadInput()
    do! inputs.AsyncAdd(value) }

let processData = async {
  while true do
    let! inp = inputs.AsyncGet()
    let! out = processStep1 inp
    do! step1.AsyncAdd(out) }

Async.Start(loadData)
Async.Start(processData)
```

Actor Model

DEMO ☺

Actor Patterns

BarrierAsync - .Net Barrier

```
int count = 0;

// Create a barrier with three participants
Barrier barrier = new Barrier(3);
barrier.AddParticipants(1);

// This is the logic run by all participants
Action action = () =>
{
    Interlocked.Increment(ref count);
    barrier.SignalAndWait();
};

Parallel.Invoke(action, action, action, action);
```

BarrierAsync

```
type BarrierAsync(n) =
    let checkPhase(participants, nReplies, replies) =...
    let transition (participants, nReplies, replies) msg =...
            type BarrierInstruction =
    let cancelT                 | SignalAndWait of (unit -> unit)      ce()
    let agent =                 | AddParticipant
                                | RemoveParticipant

                loop(n, 0, [])), cancelToken.Token)

    interface IDisposable with
        member x.Dispose() = cancelToken.Cancel()

    member x.AsyncSignalAndWait() =
          agent.PostAndAsyncReply(fun reply -> SignalAndWait reply.Reply)

    member x.AddParticipant() = agent.Post AddParticipant
    member x.RemoveParticipant() = agent.Post RemoveParticipant
```

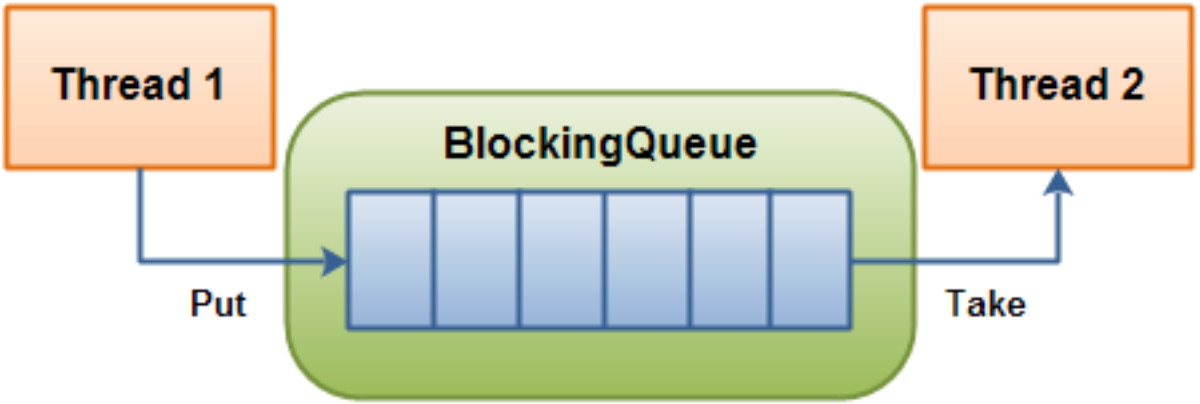
BarrierAsync

```
let transition (participants, nReplies, replies) msg =
      match msg with
      | SignalAndWait reply ->
          checkPhase (participants, nReplies + 1, reply::replies)
      | AddParticipant ->
          participants + 1, nReplies, replies
      | RemoveParticipant ->
          checkPhase (participants+1, nReplies+1, replies)
```

BarrierAsync

```
for nAgents in [1; 1; 10; 100; 1000; 10000; 100000] do
  let makeAgent _ =
    new MailboxProcessor<_>(fun inbox ->
      let rec loop n id =
        async { let! id = inbox.Receive()
                let n = n+1
                if n=nMsgs then
                  do! barrier.AsyncSignalAndWait()
                else
                  return! loop n id }
      loop 0 [])

  async { do! barrier.AsyncSignalAndWait() } |> Async.RunSynchronously
```

AsyncBoundedQueue



AsyncBoundedQueue

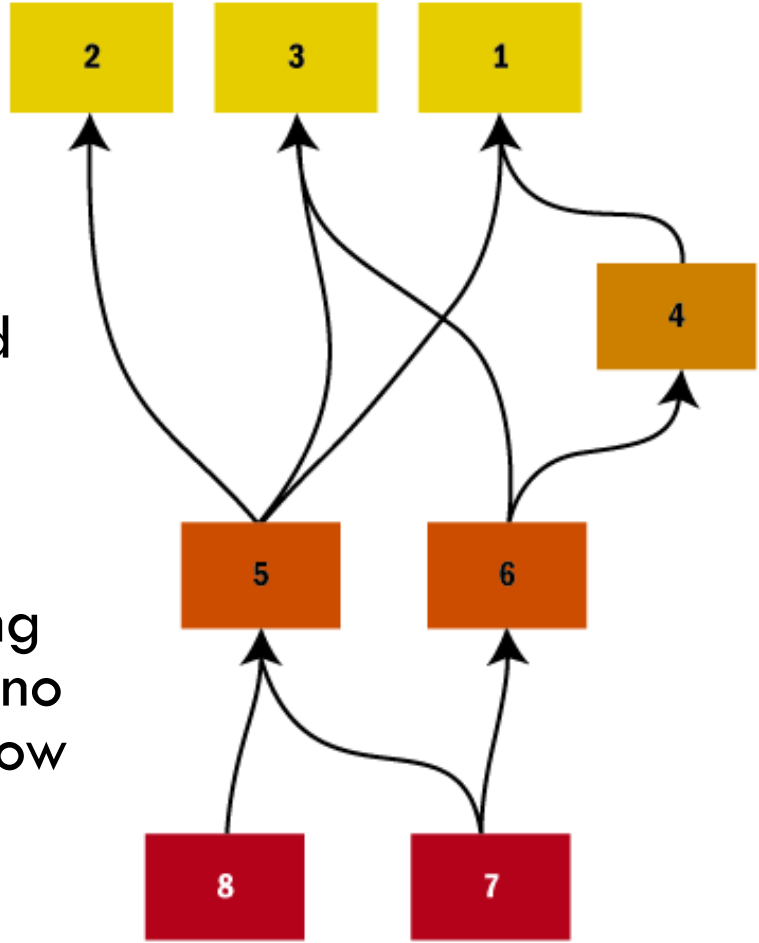
```
let
    member __.AsyncEnqueue x =
        agent <-! (fun reply -> Enqueue(x, reply.Reply))
    member __.AsyncDequeue() =
        agent <-! (fun reply -> Dequeue reply.Reply)
                                                    ue (x, reply)
    interface System.IDisposable with               eply
        member __.Dispose() =
          cancelToken.Cancel()
          (agent :> System.IDisposable).Dispose()

        balance()
```

Directed Acyclic Graph

Wiki

DAG is a direct graph with no directed cycles.

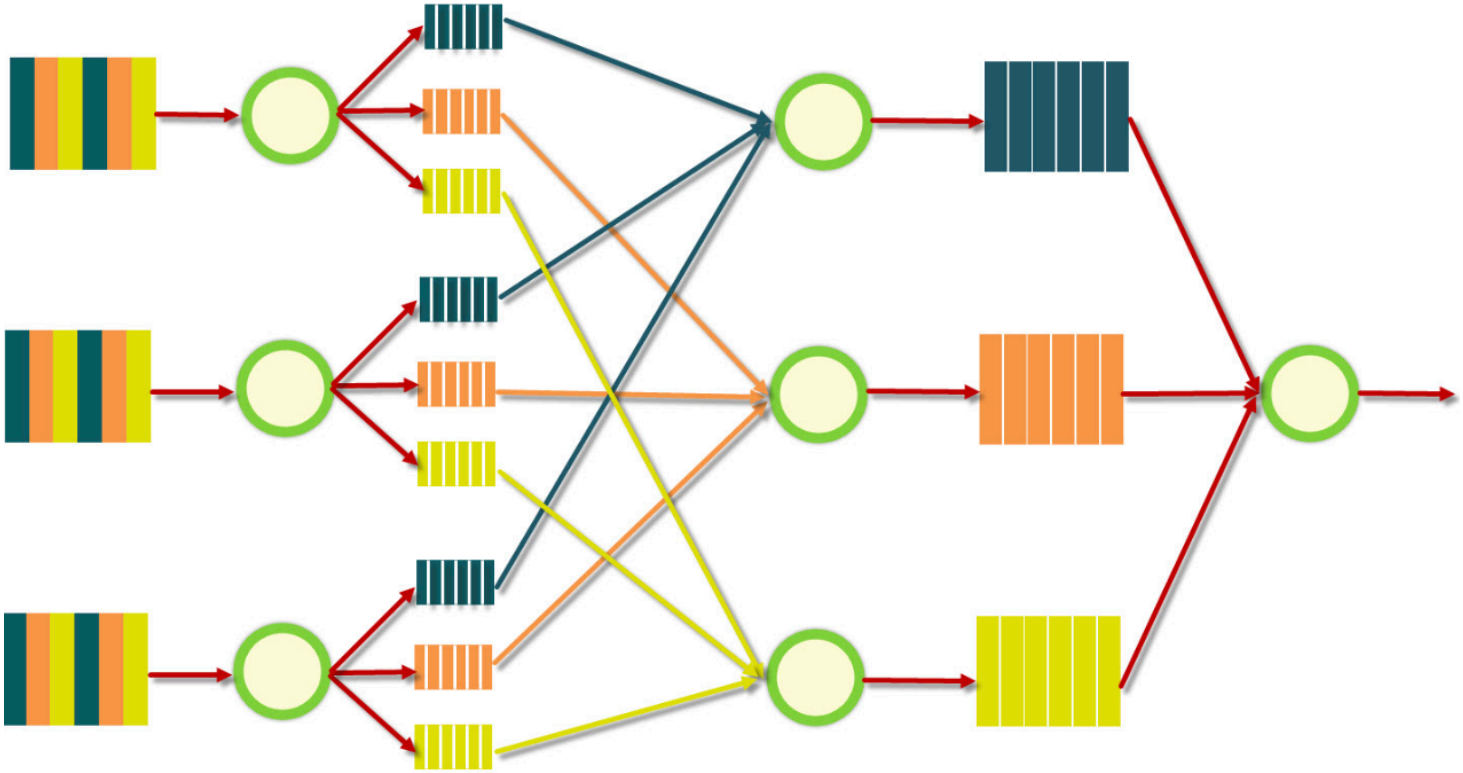
It is formed by a collection of vertices and direct edges, each edge connecting of vertex to another, such that there is no way to start at some vertex ***V*** and follow a sequence of edges that eventually loops back to ***V*** again.



Directed Acyclic Graph

```
type IDAGManager =
    [<CLIEventAttribute>]
    abstract member OnOperationComplete : IEvent<OperationData>
    abstract member Execute : unit -> unit
    abstract member AddOperation : int -> (unit -> unit) -> int array

let dagManager = AsyncDAG.DAGManager()
dagManager.OnOperationComplete.Add(fun op -> printfn "Completed %d" op.Id)
dagManager.AddOperation(1, acc1, 3)
dagManager.AddOperation(2, acc2, 1)
dagManager.AddOperation(3, acc3)
dagManager.AddOperation(4, acc4, 2, 3)
dagManager.Execute()
```

Map >> Reduce



Map >> Reduce

- **Map**
  Function written by the user to take an input pair and produce a result of intermediate key/value pairs. The intermediate values are then grouped with the same intermediate key and passed to the Reduce function.

- **Reduce**
  Function also written by the user to take the intermediate key and sequence of values for that key. The sequence of values are then merged to produce a possibly smaller set of values. Zero or one output value is produced per our Reduce function invocation. In order to manage memory, an iterator is used for potentially large data sets.

Map >> Reduce

is a programming model for processing and generating large data sets.

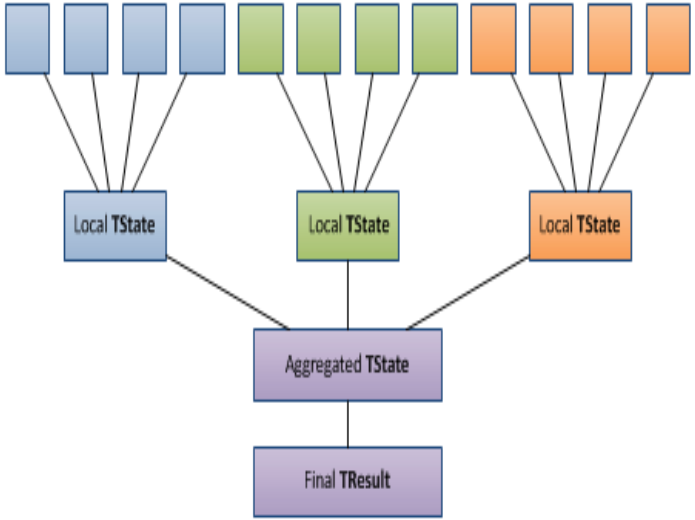
The programming model is based upon five simple concepts:

1. Iteration over input
2. Computation of key/value pairs from each input
3. Grouping of all intermediate values by key
4. Iteration over the resulting groups
5. Reduction of each group

Map >> Reduce

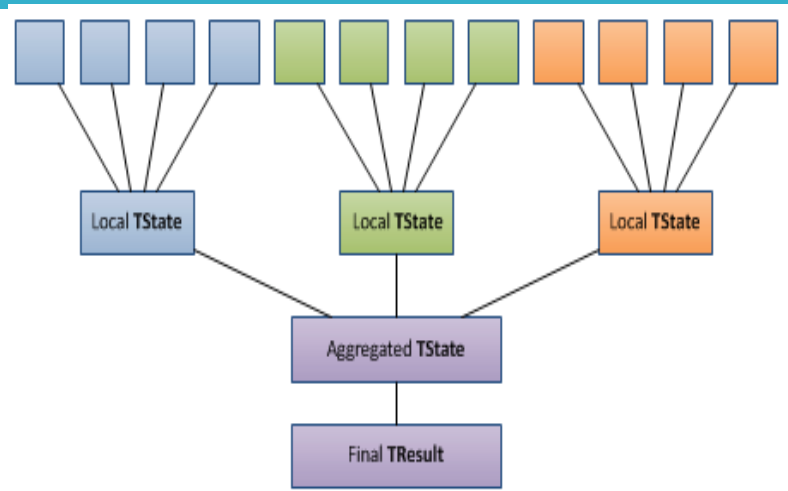
```
let data = [| 1..100 |]

let reduceFunction (list:seq<int>) = list |> Seq.sum
let map () =
    [0..9]
    |> Seq.map (fun i -> i * 10,(i + 1) * 10 - 1)
    |> Seq.map (fun (a,b) -> async { let! sum = async { return reduceFunction (data.[a..b]) }
                                                        |> Async.StartChild
                                    return! sum })
let reduce (seq:seq<Async<int>>) =
    seq |> Seq.sumBy Async.RunSynchronously

let mapReduce = map() |> reduce
```

Map >> Reduce



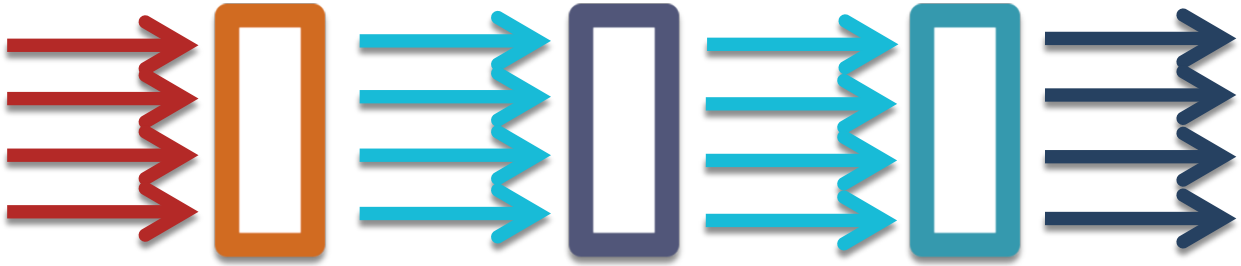

```
val mapReduce :
    initf:    (unit -> 'TState) ->                    // (1)
    processf: ('TState -> 'TInput -> 'TState) ->      // (2)
    mergef:   ('TState -> 'TState -> 'TState) ->      // (3)
    resultf:  ('TState -> 'TResult) ->                // (4)
    input:    pseq<'TInput> ->      // Input data sequence
              'TResult              // Calculated result
```

Map >> Reduce



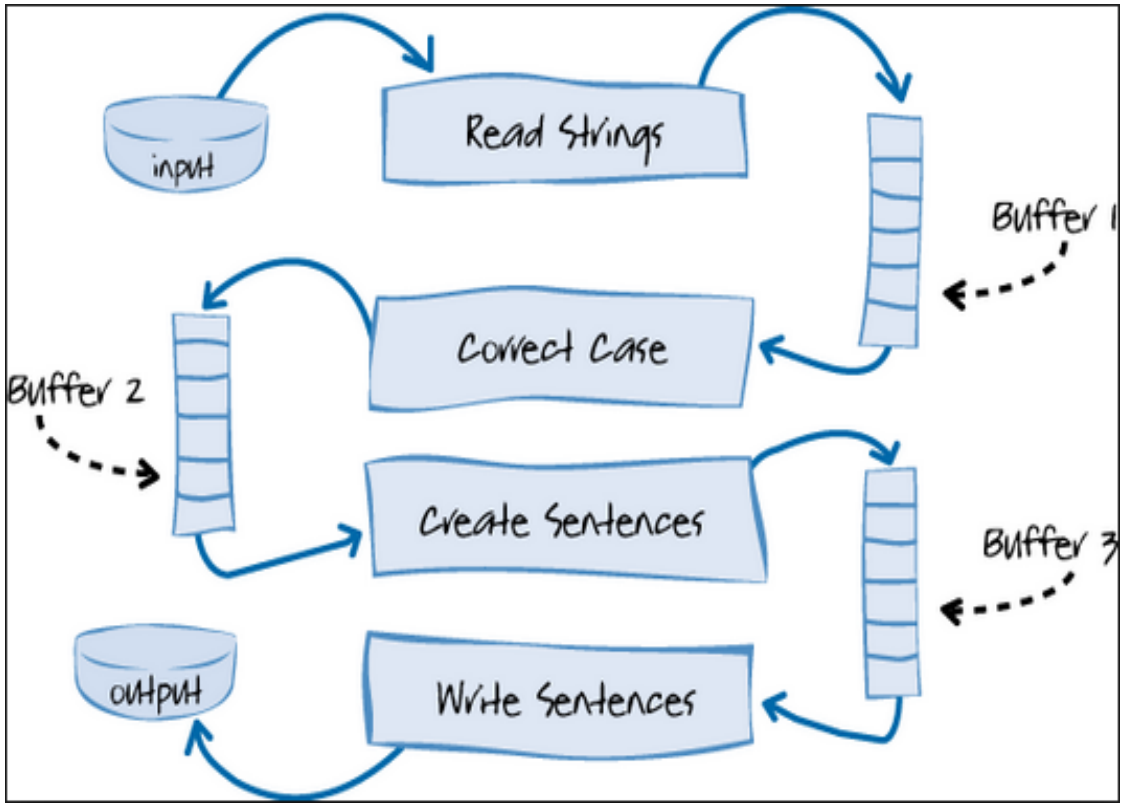

```
let mapReduce   (inputs:seq<'in_key * 'in_value>)
                (map:'in_key -> 'in_value -> seq<'out_key * 'out_value>)
                (reduce:'out_key -> seq<'out_value> -> seq<'reducedValues>)
                outputAgent
                M R partitionF
```

Pipeline Processing



- Pipeline according to Wikipedia:
  - A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements

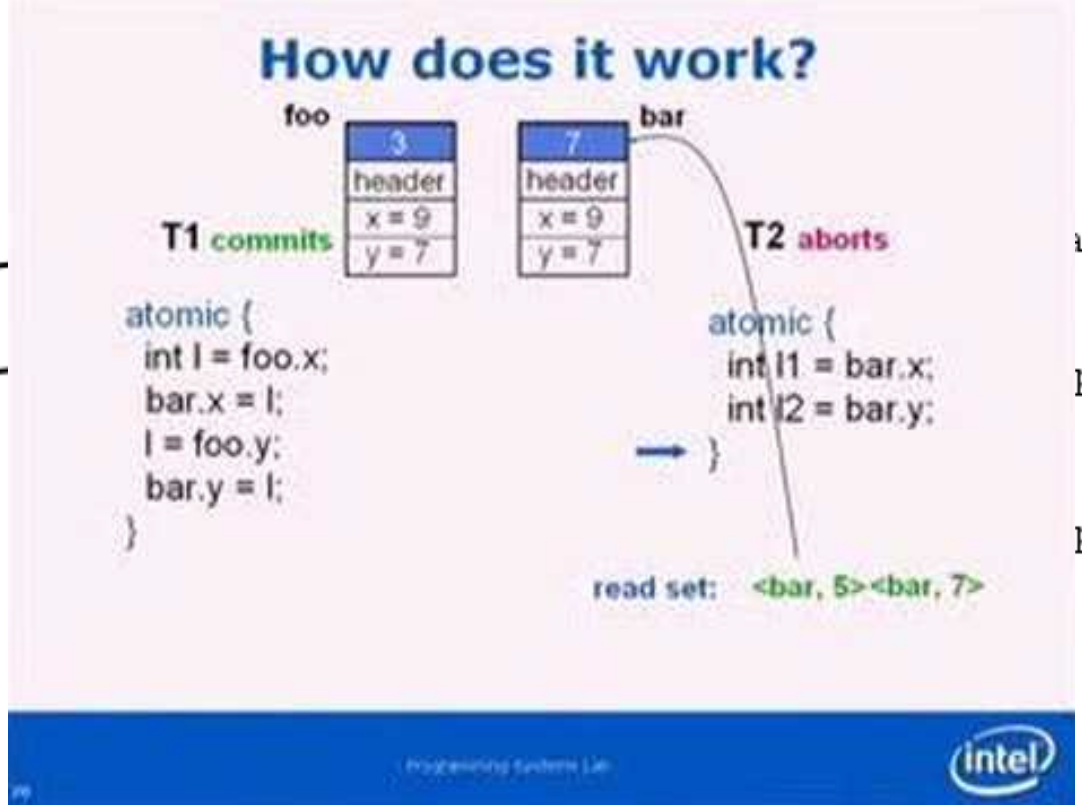
Pipeline Processing



- Values processed in multiple steps
  - Worker takes value, processes it, and sends it
  - Worker is blocked when source is empty
  - Worker is blocked when target is full
  - Steps of the pipeline run in parallel


Stm

Software Transactional Memory



Software Transactional Memory

```
type Stm<'T> = TLog -> 'T
val newTVar : value:'T -> TVar<'T>
val readTVar : ref:TVar<'T> -> trans:TLog -> 'T
val writeTVar : ref:TVar<'T> -> value:'T -> trans:TLog -> unit
val retry : un
val orElse : a                                              'T
val atomically
type StmBuilde
  class
    new : unit
    member Bin                                              > Stm<'g>
    member Com                                              >
    member Del
    member Let : p:'d * rest:('d -> Stm<'e>) -> Stm<'e>
    member Return : x:'i -> Stm<'i>
    member ReturnFrom : m:Stm<'h> -> Stm<'h>
    member Zero : unit -> Stm<'a>
  end
```

```
|> mapM_ (fun i ->
    stm {
        let! x' = readTVar x
        let! y' = readTVar y
        do! check (x' = y')
        do! writeTVar x (x' + 1)
        do! writeTVar y (y' + 1) })
|> atomically
```

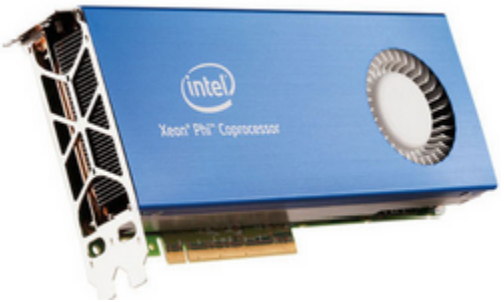
Software Transactional Memory

Hardly Successful

Software Transactional Memory

- **Instead of using TM**
  - Use natural isolation, the Actor model is a great alternative
  - Enforce coarse-grained isolation through message-passing
  - Immutability is your best friend

GPU

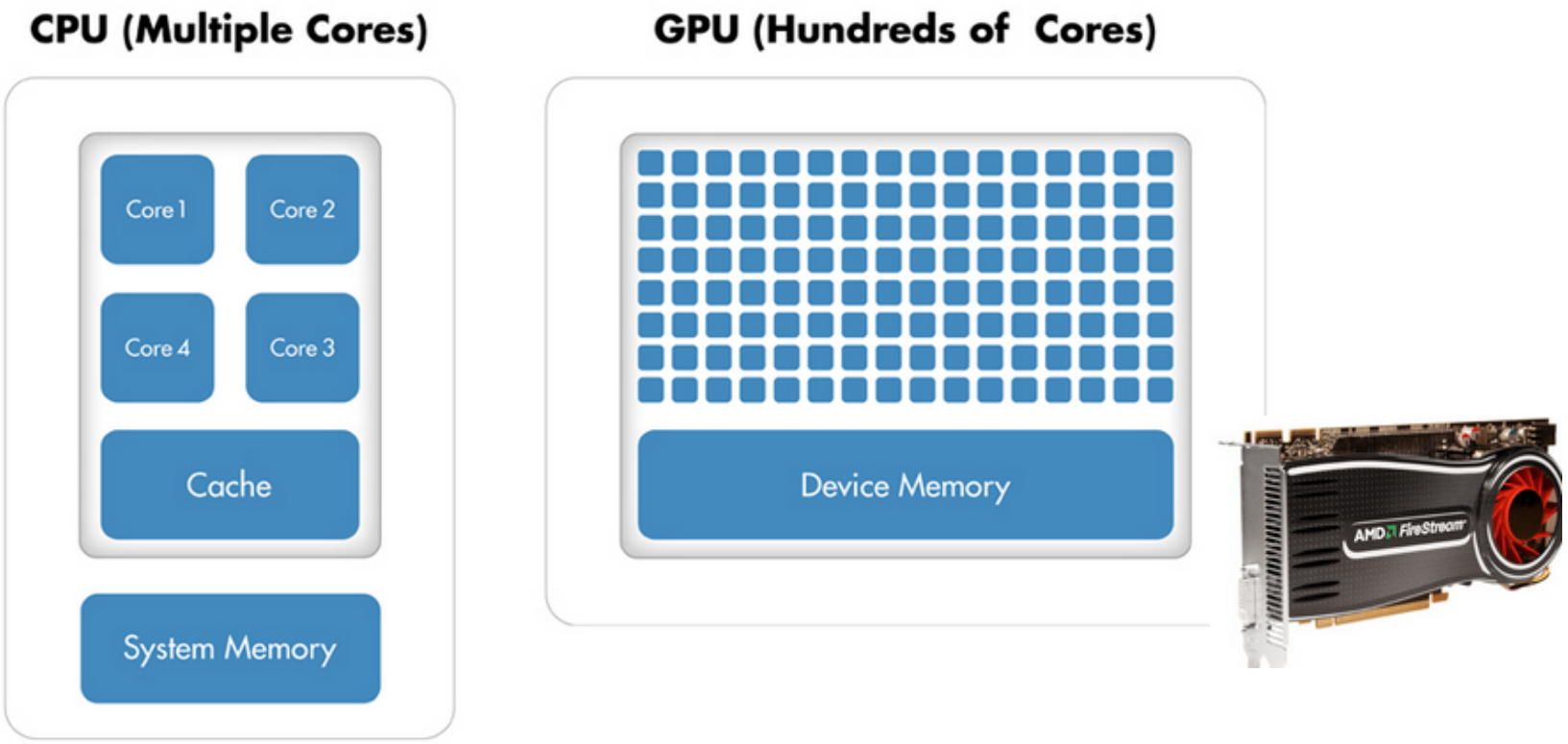


GPU

- Wikipedia defines GPU as follows
  - A graphics processing unit (GPU), is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the building of images in a frame buffer intended for output to a display. **Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel.** In a personal computer

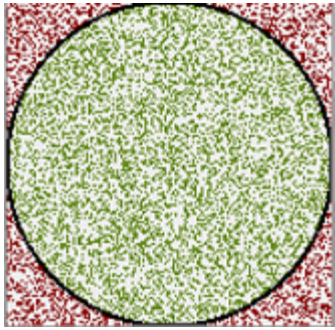
- Wikipedia defines GPGPU as follows
  - General-purpose computing on graphics processing units (GPGPU) is the means of using a graphics processing unit (GPU), which typically handles computation only for computer graphics, **to perform computation in applications traditionally handled by the central processing unit (CPU).** Additionally, the use of multiple graphics cards in one computer, or large numbers of graphics chips, further **parallelizes the already parallel nature of graphics processing.**

GPU


CPU (Multiple Cores)
Core 1
Core 2
Core 4
Core 3
Cache
System Memory
GPU (Hundreds of Cores)
Device Memory
AMD FireStream

GPU – MSFT Accelerator

- Microsoft Accelerator
  - The project Microsoft Research Accelerator is a .Net library for developing array-based computations and executing them in parallel on multi-core CPU or more interestingly, using GPU shaders
  - Accelerator handles all the details of parallelizing and running the computation on the selected target processor, including GPUs and multicore CPUs



GPU – Accelerator Usage

```
open Microsoft.ParallelArrays

type PA = Microsoft.ParallelArrays.ParallelArrays
type FPA = Microsoft.ParallelArrays.FloatParallelArray
```

FPA type represents a computation that returns an array of floats
PA type represents the static class which contains operations for working with computations

GPU – Accelerator Usage

```
let input = FloatParallelArray(nums)
let sum = ParallelArrays.Shift(input, 1) +
                ParallelArrays.Shift(input, -1)
let output = sum / 3.0f
```

Accelerator library exposes various types for creating data-parallel computations.
FloatParallelArray represents a computation that returns a 1D array of floats
Shift method moves elements of the array to the left or to the right

GPU – Accelerator Usage

```
let target = new DX9Target()
let target = new X64MultiCore()

let run(target:Target) (fP:FloatParallelArray) =
        target.ToArray2D(fp)
```

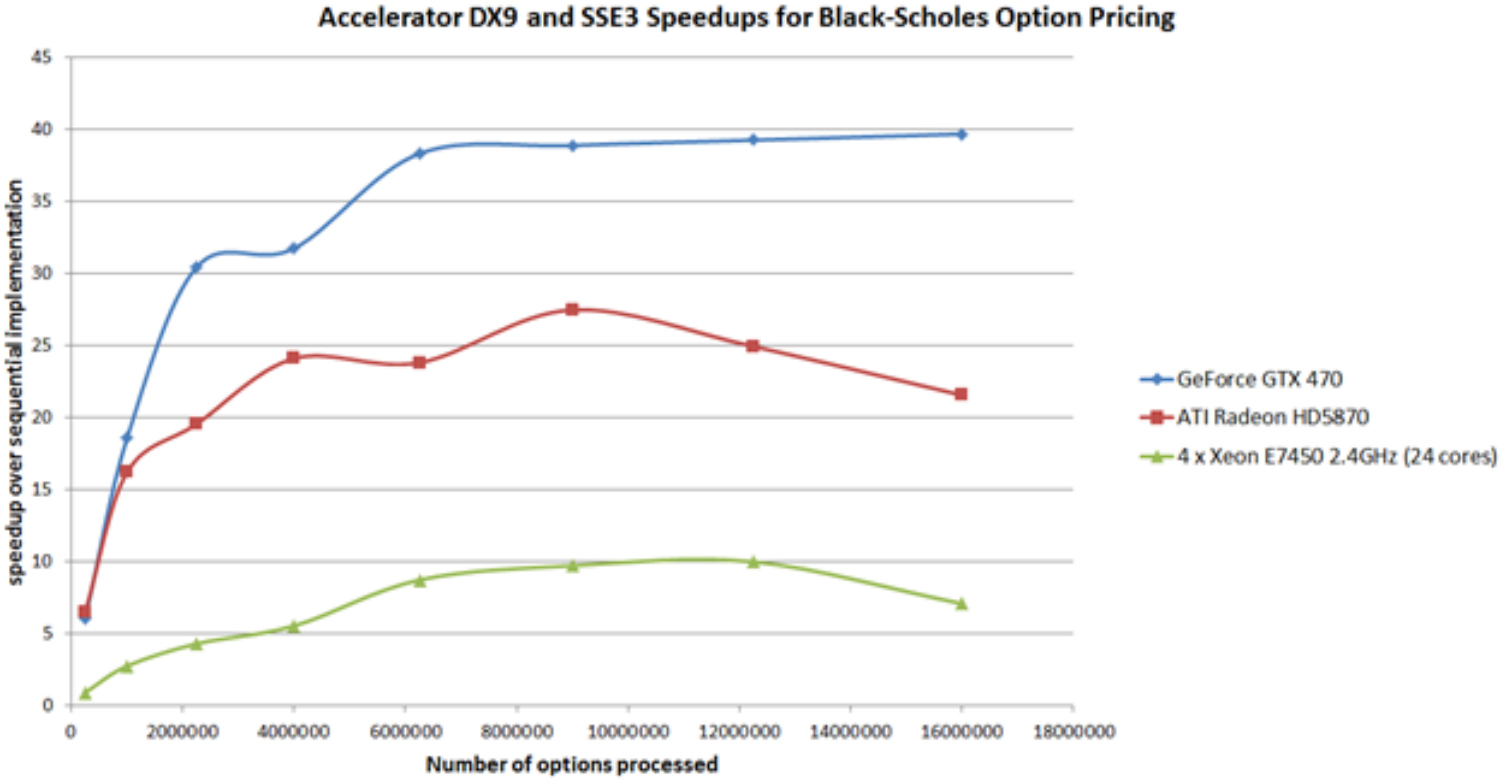
DX9Target performs data-parallel operations using shaders on your GPU (massively parallel)

X64MultiCore performs data-parallel operations using multi-core 64bit CPU.

GPU – MSFT Accelerator

Sample	F# matrix	Preprocessing	X64Multicore	DX9Target	Max. speedup
Life	290ms	~1000ms	40ms	130ms	7.3x
Blur	3690ms	~2000ms	100ms	190ms	36.9x
Pi	1590ms	~850ms	490ms	560ms	3.2x
Rotate	1250ms	~1490ms	N/A	220ms	5.7x

GPU – MSFT Accelerator



GPU – Code Quotations

- **Strengths**
  Data parallelism is ideal whenever you're faced with a problem where large amounts of numerical data needs to be processed. It's particularly appropriate for scientific and engineering computing and for simulation. Examples include fluid dynamics, finite element analysis, n-body simulation, simulated annealing, ant-colony optimization, neural networks, and so on.

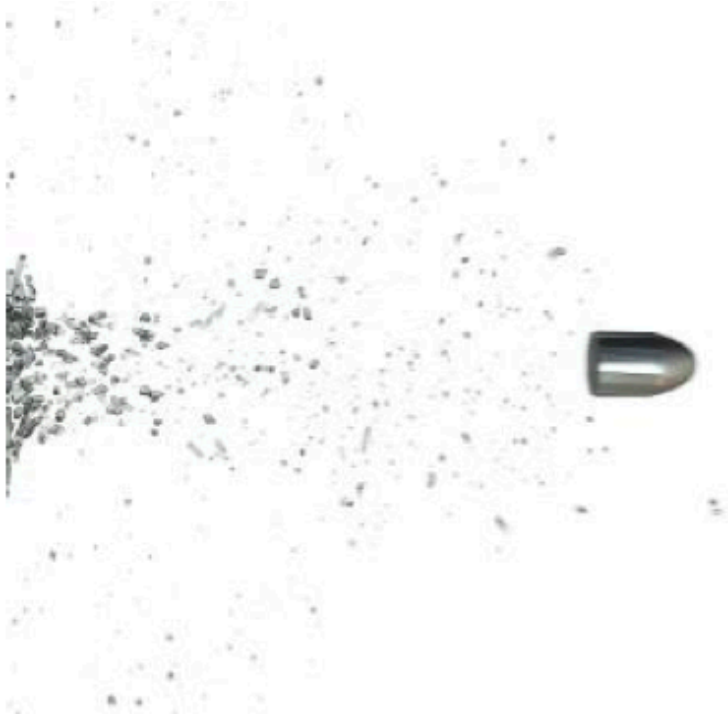
- **Weaknesses**
  Within its niche, data-parallel programming in general, and GPGPU programming specifically, is hard to beat. But it's not an approach that lends itself to all problems. In particular, although it is possible to use these techniques to create solutions to nonnumerical problems (natural language processing, for example), doing so is not straightforward—the current toolset is very much focused on number-crunching

DEMO ☺

Summary

- Immutability and Isolation are your best friends to write concurrent application
  - Use natural isolation, the Actor model is a great to enforce coarse-grained isolation through message-passing
- Asynchronous Programing in F# is easy and declarative
- Concurrency in F# is fully integrated with .NET
- Use Actor Model for high scalable computation

Not a Silver Bullet!!!



Q & A ?

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

github.com/DCFsharp

meetup.com/DC-fsharp/

@DCFsharp

rterrell@microsoft.com