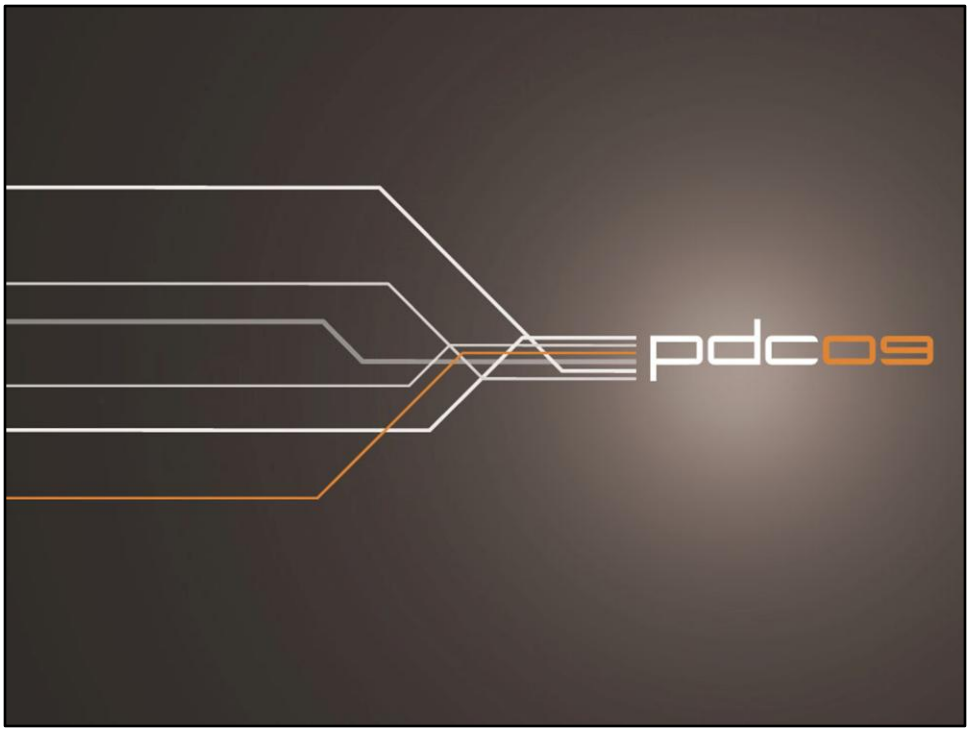


Introduction

Who am I.

Patterns Primer -> Patterns on multi-core -> Patterns on HPC

Herb already talked a bit about patterns.

How to think about an application and apply patterns to it.

Drill into some key patterns we'll be seeing during the rest of the workshop.

This talk isn't exhaustive.

Also… SOA and Azure applications

Searching for Parallelism

Finding exploitable parallelism

- Organizing by Tasks
- Organizing by Data
- Organizing by Ordering

How do you think about your application to best bring out it's underlying parallelism?

You'll note that all these patterns have references.

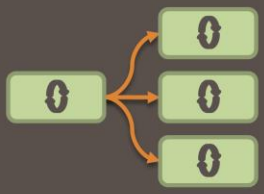
We didn't just make them up!

Patterns have been found to work in a wide variety of situations.

--
http://www.sxc.hu/photo/649877

Organizing by Tasks

- Linear or recursive?
  - Task parallel
  - Divide and conquer
- Enough tasks?
  - Too many – thrashing
  - Too few – under utilization
- Dependencies between tasks
- Scheduling work to tasks



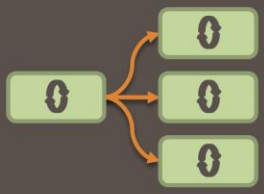
We're going to come back to dependencies and scheduling when we look at the fork/join pattern in a few minutes.

Stephen has lots to say about "delightfully parallel" and how to minimize dependencies to get there.

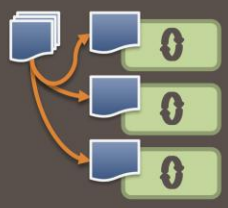
Example – Linear: Running a series of statistical models with different initial criteria
Example – Recursive: Trees

Organizing by Tasks

- Work per task?
  - Small workloads
  - Variable workloads
- Dependencies between tasks?
  - Removable
  - Separable
  - Read only or read/write



Organizing by Data



- Linear or recursive?
  - Geometric decomposition
  - Recursive data
- Data "chunk" size?
  - Too big – under utilization
  - Too small – thrashing
- Chunk layout?
  - Cache and cache line size
  - False cache sharing

Stephen will be talking about some of these implications in detail as it applies to efficiently using the cache.

Richard will be covering how this maps onto a message passing approach where message size and serialization overhead are important.

Example – Linear: Divide an image up and process individual lines or rectangles.

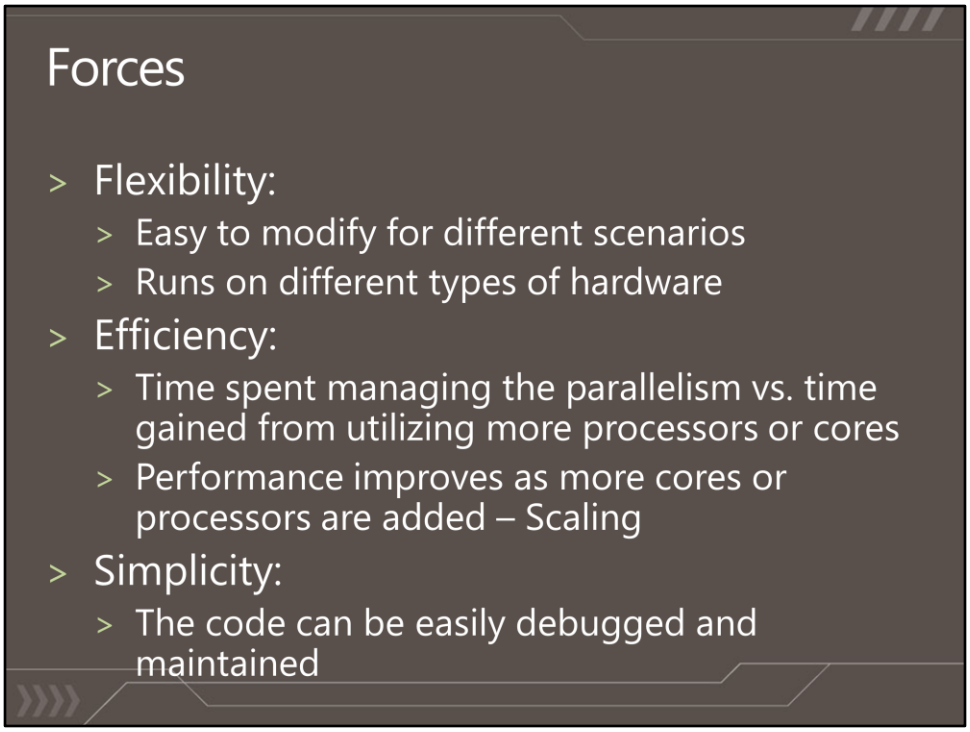
Organizing by Task Ordering

- Linear and irregular orderings
  - Pipeline
  - Asynchronous Agents
- Task constraints
  - Temporal: A → B
  - Simultaneous: A ↔ B
  - None: A B
- External constraints
  - I/O read or write order
  - Message or list output order

Identifying tasks with no constraints is important.

You can use these as “padding” to improve utilization while waiting on other dependencies.

Example: Image/data processing pipelines.



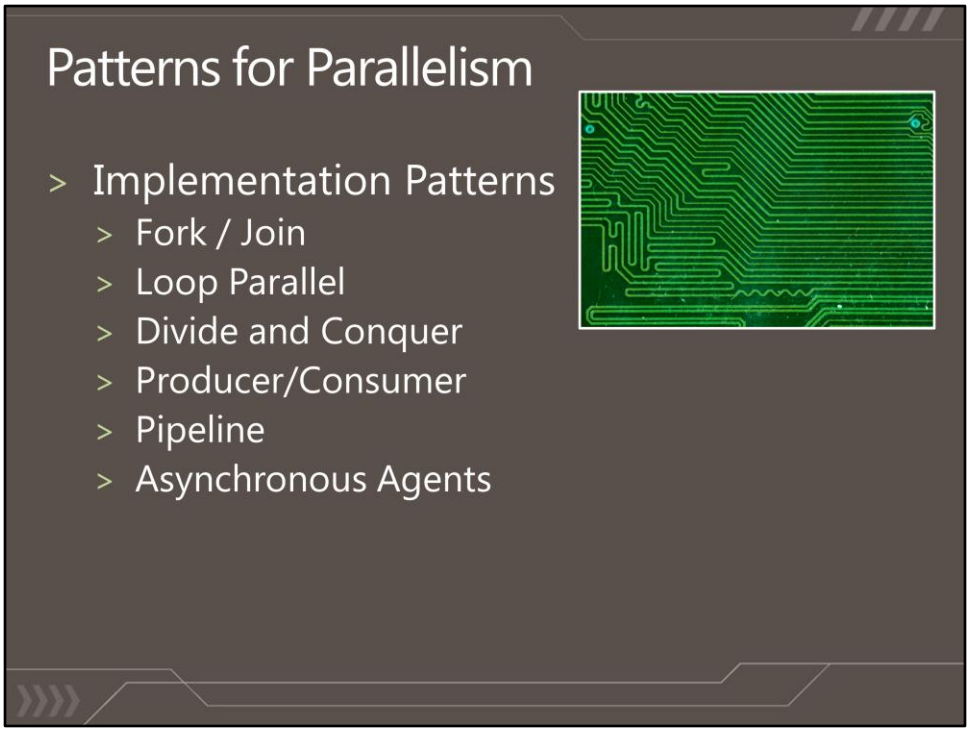
Some of these are mutually exclusive!

It's easy to think of "Simplicity" as an afterthought. Bad Idea!

Code spends more time being read than written.
Applications outlast several generations of hardware.

One way to improve efficiency is usually to add more work.
This improves the communication/management/setup vs. compute ratio

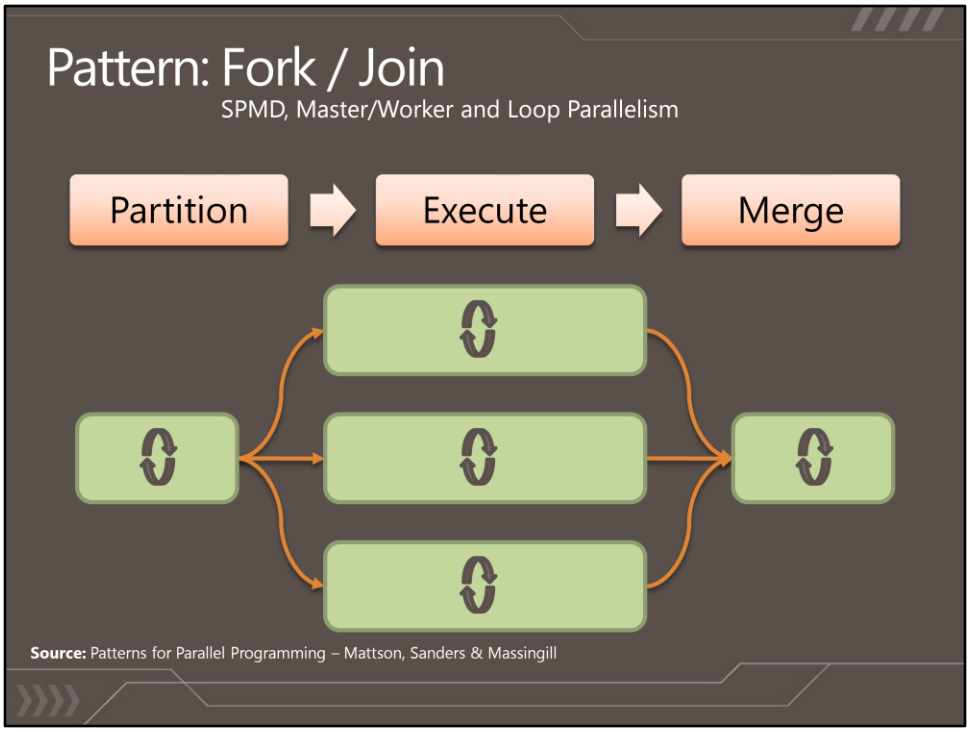
Example: Highly efficient code will (probably) be less flexible and harder to maintain.



These are just some of the common patterns. I'll be covering these from a high level here.

Stephen and **Richard** will be drilling down into how they are implemented and used.

--
http://www.sxc.hu/photo/694300



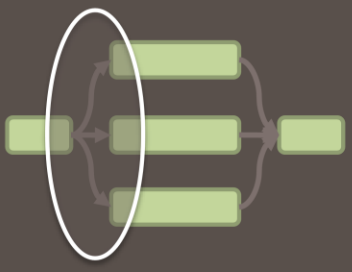
Going to cover this in some detail as in many ways it embodies many of the challenges we'll see in all the other patterns.

Fork / Join Example

```
Parallel.Invoke(
     () => ComputeMean(),
     () => ComputeMedian()
);

parallel_invoke(
     [&] { ComputeMean(); },
     [&] { ComputeMedian(); },
);
```

Partitioning

- How do we divide up the workload?
  - Fixed workloads
  - Variable workloads
- Workload size
  - Too large – hard to balance
  - Too small – communication may dominate



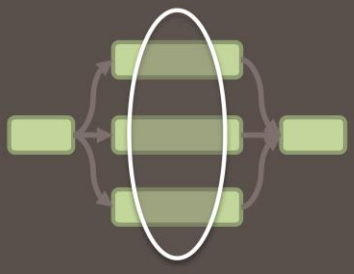
Workload size is particularly important on distributed systems as communication may be costly.

Amortizing latency by sending larger messages may be the right thing to do.

Richard will be talking more about this later.

Workload Balancing

- Static allocation:
  - By blocks
  - By index (interleaved)
  - Guided
- Dynamic work allocation (known and unknown task sizes)
  - Task queues
  - Work stealing



What if you don't know the size of the task?

Example: Calculating in individual pixel on a Mandelbrot is a variable amount of work depending on it's final "color".

In this case static allocation will mean that you end up waiting for the slower tasks.

Sharing State and Synchronization

- Don't share!
- Read only data
- Data isolation
- Synchronization

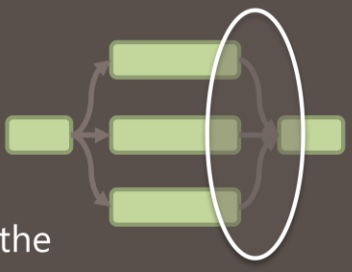
Locking is an anti-pattern here (James Reinders)

F# Immutability

Possible to end up with serial performance (or worse)
If you have to resort to locks or shared data – ask, is there a better way?

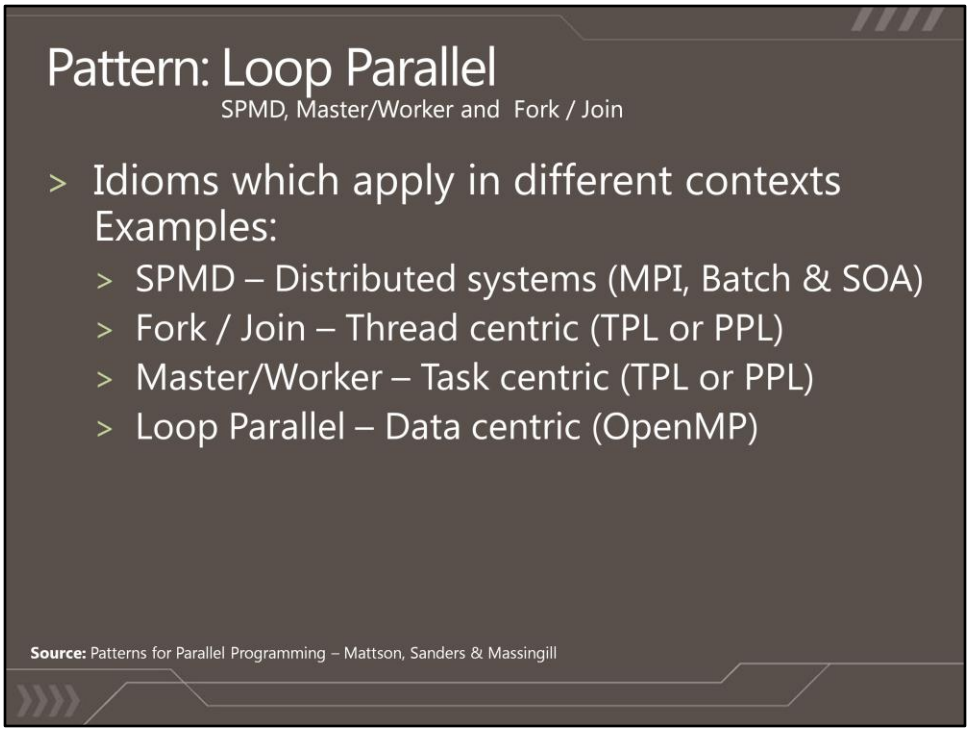
Merging

- Delay the "final answer"
  - Calculate sub-problem result per task
  - Merge results later
  - Possible to also parallelize the merge
- Reduces need for locking
- MapReduce pattern



Source: MapReduce: Simplified Data Processing on Large Clusters – Dean & Ghemawat

Stephen and **Richard** will both be talking about Map Reduce later.



What about loop parallel?

Why am I mentioning it? A very common pattern!

Loop Parallel Examples

```
Parallel.For (0, acc.Length, i =>
{
    acc[i].UpdateCreditRating();
});

#pragma omp parallel for
for (int i = 0; i < len; i++)
{
    acc[i].UpdateCreditRating();
}
```

Loop parallel and fork/join are very similar

Different ways of thinking about the same problem.

Whole tools—like OpenMP—have been built around this pattern!

If indices of acc[] inside our loop share data think parallel tasks rather than loops.

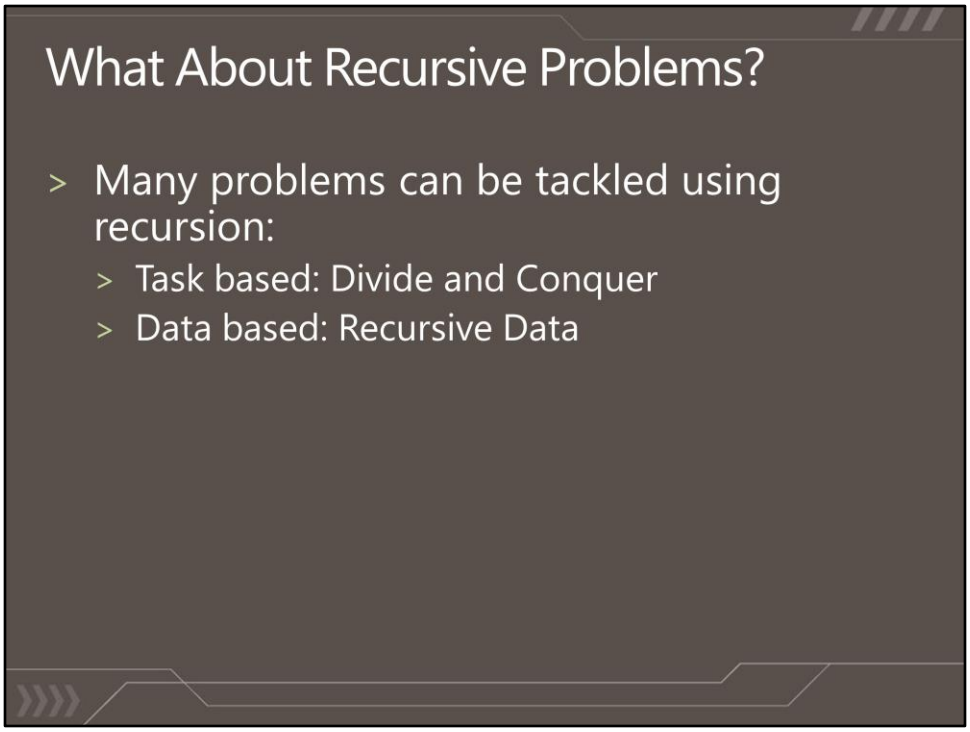
Anti-Pattern: just go through your application parallelizing loops!

Stephen will be showing some more examples of loop parallelism and Parallel.For later.

Loop Parallel With PLINQ

```
var accountRatings =
    accounts.AsParallel()
        .Select(item =>
            CreditRating(item))
        .ToList();
```

Stephen will be talking more about PLINQ later.



Key linking content…

So loop parallel and fork join are two linear ways of thinking about tasks and data.

How about recursion?

Many problems can be tackled using a divide and conquer strategy where the program tackles sub-problems.

We're going to talk about Divide and Conquer next…

Pattern: Divide and Conquer

- Recursive Task Parallel
- Divide problem up into sub-problems

Source: Patterns for Parallel Programming – Mattson, Sanders & Massingill

Trees are a classic divide and conquer strategy.

Workload Balancing

- Deep trees – thrashing
  - Limit the tree depth
- Shallow trees – under utilization

Limit tree depth.

Divide and Conquer Example

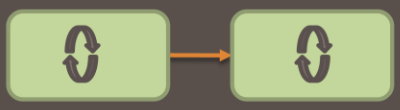
```
static void Walk<T>(Tree<T> root, Action<T>
action)
{
    if (root == null) return;
    var t1 = Task.Factory.StartNew(() =>
        action(root.Data));
    var t2 = Task.Factory.StartNew(() =>
        Walk(root.Left, action));
    var t3 = Task.Factory.StartNew(() =>
        Walk(root.Right, action));
    Task.WaitAll(t1, t2, t3);
}
```

Note: This is a simple example with no limiting. For deep trees we might end up with lots of tasks (thrashing).

Stephen will be showing more sophisticated trees.

Pattern: Producer/Consumer

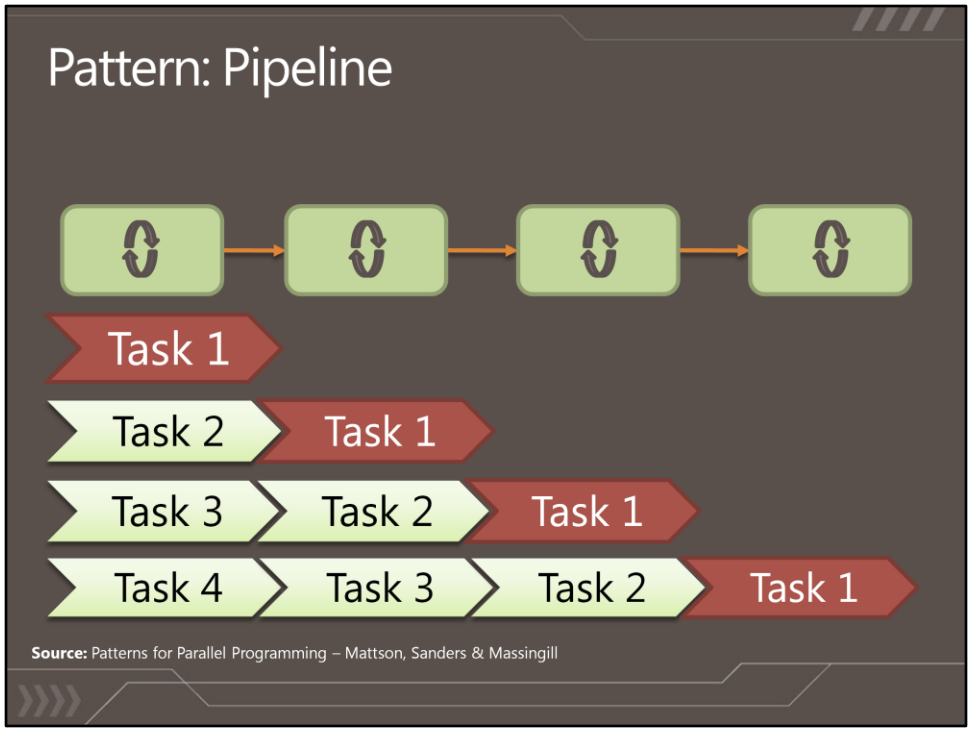
- Organize by Ordering



- Producers… produce!
  - Block when buffer full
- Consumers… consume!
  - Block when buffer empty

Source: http://en.wikipedia.org/wiki/Producer-consumer_problem

"Bounded buffer problem"



A series of producer/consumers

Pipeline Examples

```
Get-ChildItem C:\ |
    Where-Object {$_.Length -gt 2KB} |
    Sort-Object Length
```

MS-DOS, Unix and now PowerShell all allowed you to pipe the output of one process into another.

Workload Balancing

- Pipeline length
  - Long – High throughput
  - Short – Low latency
- Stage workloads
  - Equal – linear pipeline
  - Unequal – nonlinear pipeline

Passing Data Down the Pipe

- Message passing
  - Buffering
  - Message ordering
  - Message size
- Shared queue(s)
  - Large queue items – under utilization
  - Small queue items – locking overhead

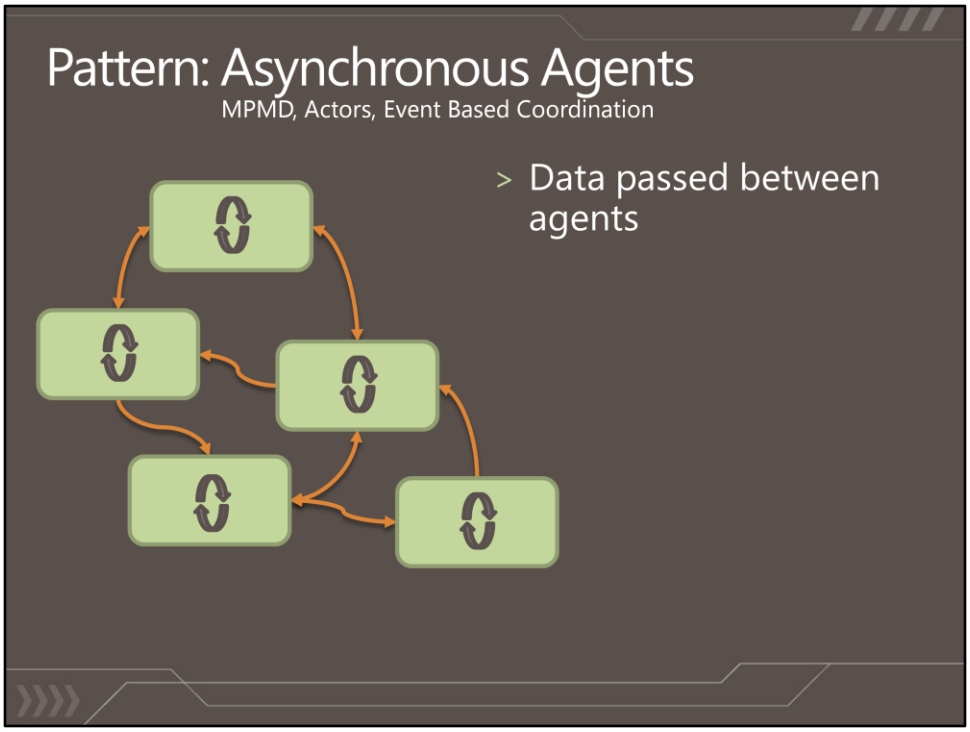
Two approaches…

Messages (MPI) much higher communication overhead

Shared Queue (TPL/PPL)

Pipeline Examples

```
var input = new BlockingCollection<string>();

var readLines = Task.Factory.StartNew(() =>
{
    try {
        foreach(var line in
                File.ReadAllLines(@"input.txt"))
            input.Add(line);
    }
    finally { input.CompleteAdding(); }
});

var writeLines = Task.Factory.StartNew(() =>
{
    File.WriteAllLines(@”output.txt",
        input.GetConsumingEnumerator());
});

Task.WaitAll(readLines, writeLines);
```



Mentioning this largely because it seems like one of the obvious ways to think about parallelism.

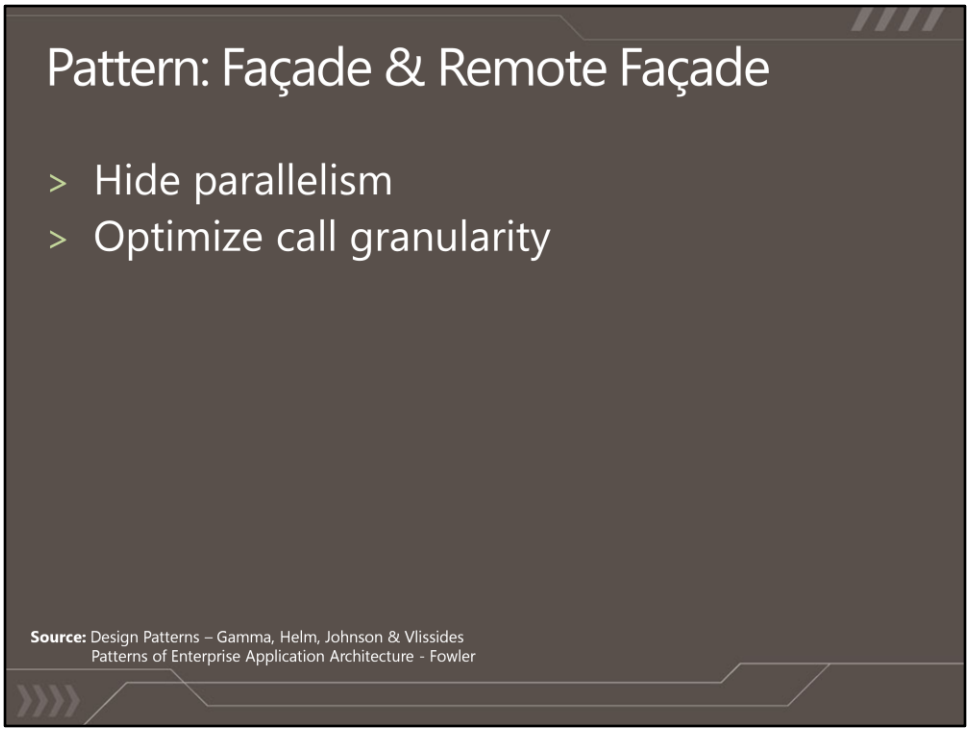
You can write applications this way but it's hard!

Note how data moves in all sorts of directions.

This is a recipe for deadlocks, livelocks and all manner of other difficulties.

Supporting Patterns

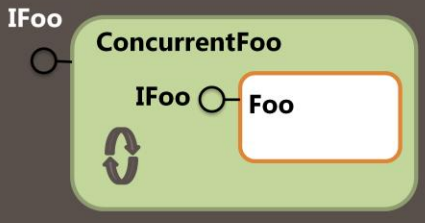
- "Gang of Four" Patterns
  - Façade
  - Decorator
  - Repository
- Shared Data Patterns
  - Shared Queue



Richard will be explaining how may parallel implementations on HPC SOA

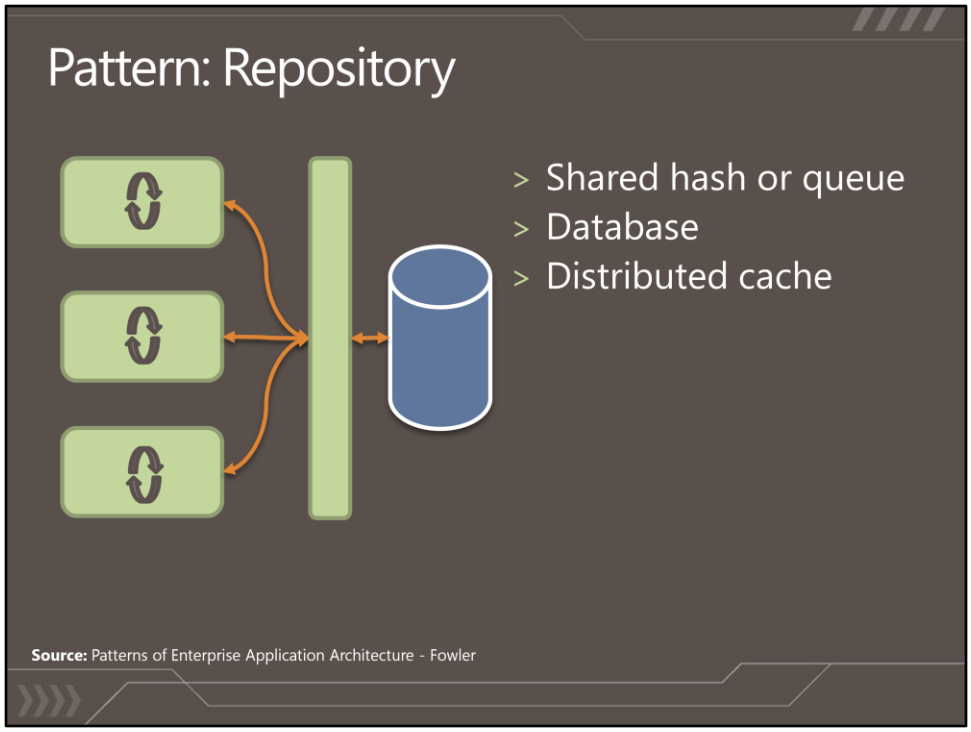
Pattern: Decorator

- Encapsulate parallelism
  - Calling code is parallel agnostic
- Add parallelism to an existing (base) class

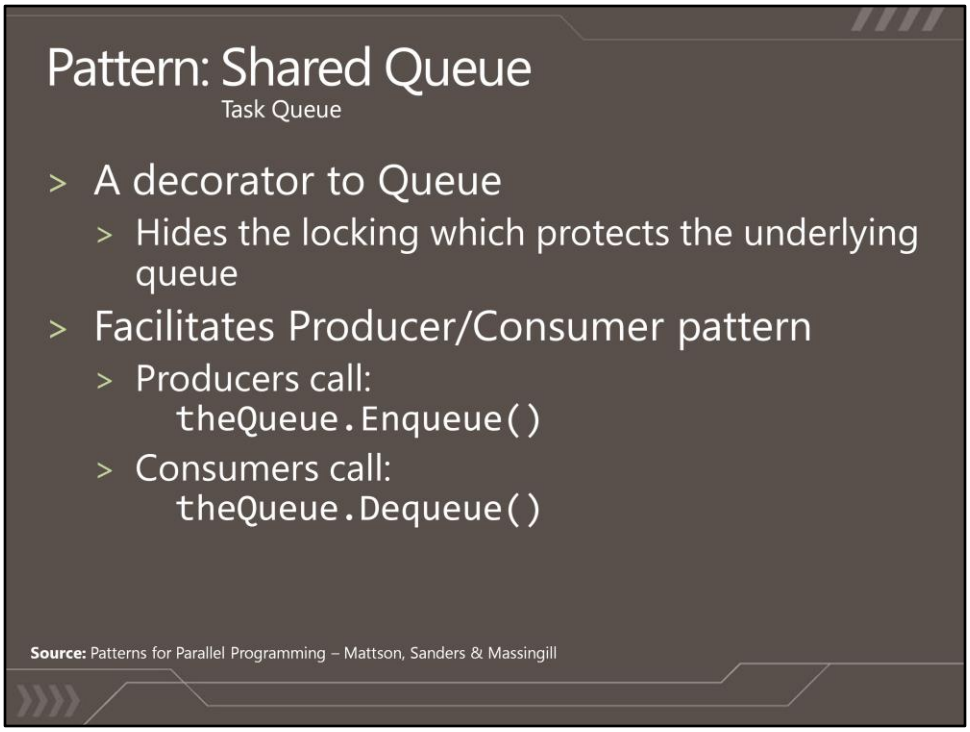


Source: Design Patterns – Gamma, Helm, Johnson & Vlissides

Stephen will be showing an example of a ThreadedWriteStream decorator which extends the stream classes to offload writes onto another thread.



Control access to shared data



The PPL and TPL both provide implementations of shared queue and several other shared/concurrent collections

- ConcurrentQueue
- ConcurrentStack
- ConcurrentDictionary
- BlockingCollection
- ConcurrentBag

Richard will also be showing how HPC SOA uses queues in it's task scheduling.

Shared Queue Example

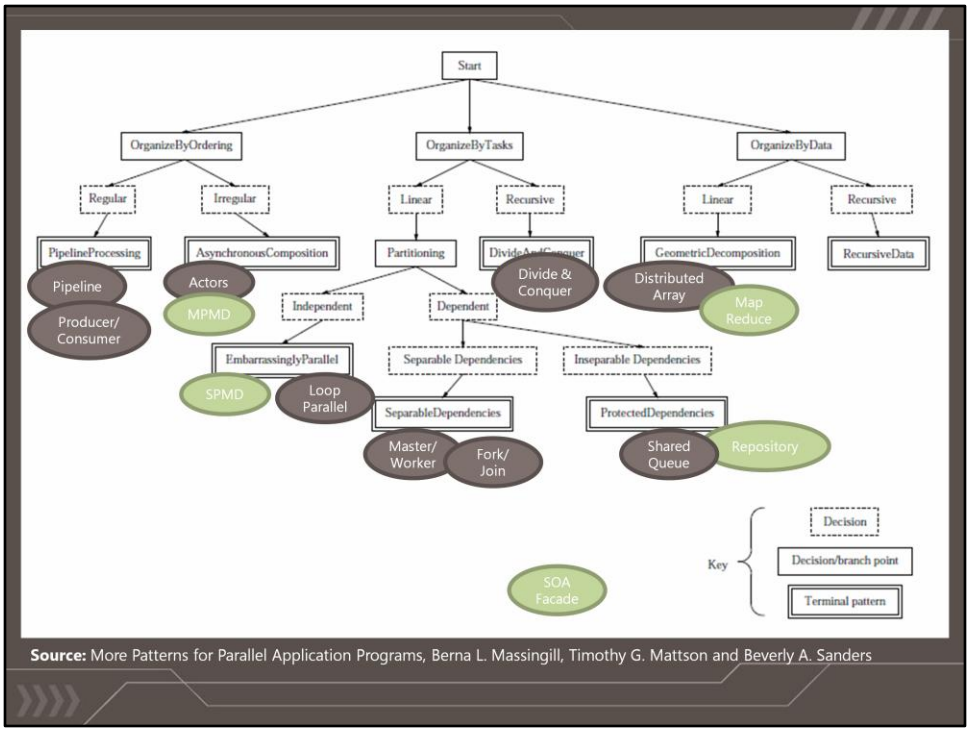
```
var results =
    new ConcurrentQueue<Result>();

Parallel.For(0, 1000, (i) =>
{
    Result result = ComputeResult(i);
    results.Enqueue(result);
});
```

Note that shared queue's behavior differs from that of a serial queue.
The order of items is no longer guaranteed!

Patterns of Patterns

- ConcurrentQueue<T>
  - Implements: IProducerConsumerCollection<T>
- Wraps Queue<T>
  - No IQueue<T> interface (not strictly a decorator)

See the same patterns occurring over and over again.
Patterns build on each other.



Source: More Patterns for Parallel Application Programs, Berna L. Massingill, Timothy G. Mattson and Beverly A. Sanders

We've seen a lot of different patterns today.

After lunch we'll see many of them applied to both multi-core and HPC platforms.

Conclusions



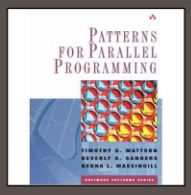
What's this?... A Hammer
What's this?... A Nail
What's this?... A cute bunny

When someone shows you a hammer everything starts to look like a nail.

Don't do this with a hammer, for the bunny's sake.
Don't do this with parallelism or patterns for everyone's sake. USE THEM WISELY!

--
http://www.sxc.hu/photo/604247
http://www.sxc.hu/photo/1101239
http://www.sxc.hu/photo/786448

Want to Find Out More?



Books

- Patterns for Parallel Programming – Mattson, Sanders & Massingill
- Design Patterns – Gamma, Helm, Johnson & Vlissides
- Head First Design Patterns – Freeman & Freeman
- Patterns of Enterprise Application Architecture – Fowler

Research

- A Pattern Language for Parallel Programming ver2.0
- ParaPLOP - Workshop on Parallel Programming Patterns

- My Blog: http://ademiller.com/tech/ (Decks etc.)

Stock photos from: http://www.sxc.hu/

The other speakers will also have links and other resources specific to the content of their talks.