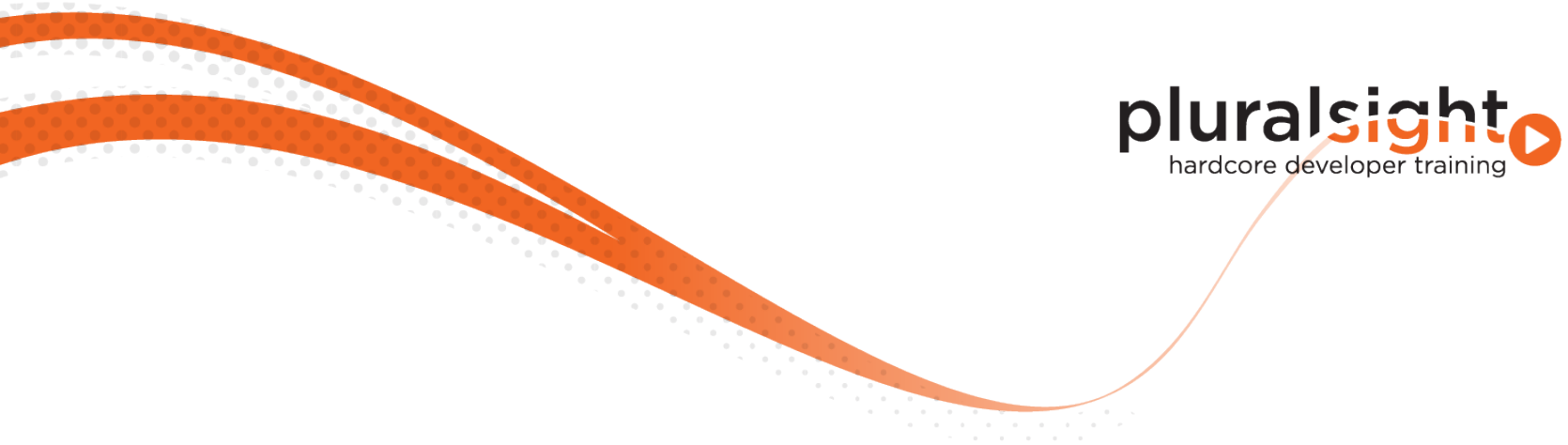
Designs and Patterns for Parallel Programming

Joe Hummel, PhD

joe-hummel@pluralsight.com
http://blog.joehummel.net/
@joehummel



pluralsight
hardcore developer training

Overview

- **Your presenter: Joe Hummel, PhD**
  - PhD in field of high-performance computing
  - An exciting time to be working in this area…

- **Agenda for this module:**
  - *A few high-level design problems…*
  - *Pipeline and Dataflow*
  - *Using the Concurrent Data Structures provided by the TPL*
  - *Producer-Consumer*
  - *MapReduce and Task Local State*
  - *Parallel LINQ*
  - *Speculative Execution*
  - *Asynchronous Programming Model*

Design Problem #1

Your app needs to perform 100+ CPU-intensive operations, each taking roughly 3-5 minutes. Execution order doesn't matter.

How do you execute the 100+ operations?

Create 100 tasks, 1 per op, with no creation options. ✗

[Since long-running, .NET will over-compensate and create 100 worker threads; since CPU intensive, system will thrash.]

Create 100 tasks, 1 per op, with long-running option. ✗

[.NET creates 100 dedicated threads; since CPU intensive, system will thrash.]

Create 1 task per core, as one finishes create another. ✓

[use WaitAllOneByOne pattern, or MaxDegreeOfParallelism.]

Design Problem #2

Your app needs to perform 20+ I/O operations — e.g. web page downloads. Order doesn't matter, but downloads can take anywhere from a few ms to a few seconds.

How do you download the web pages?

√ Create 1 code task per download, no special creation options.

[Starts one download for each worker thread in pool.]

√ Create 1 façade task per download using FromAsync + APM pattern (web object's Begin/End methods).

[Starts download and returns thread to pool, potentially allowing all 20+ downloads to start.]

√ Create 1 task per core, explicitly or via MaxDegreeOfParallelism.

[Starts one download per core.]

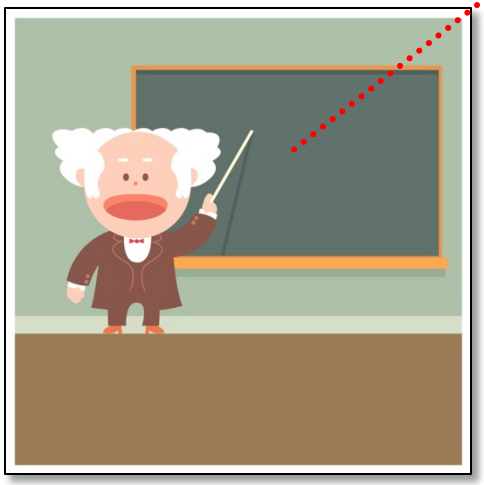
√ Use Producer-Consumer pattern, e.g. dedicating 1-2 tasks to download & remainder to processing.

[Consider when download time << processing time.]

Design Problem #3

You want a logging task that runs for the duration of your app, logging say every 30 seconds.

What does the design look like?



This is harder than it looks :-)

Things to consider in your design:

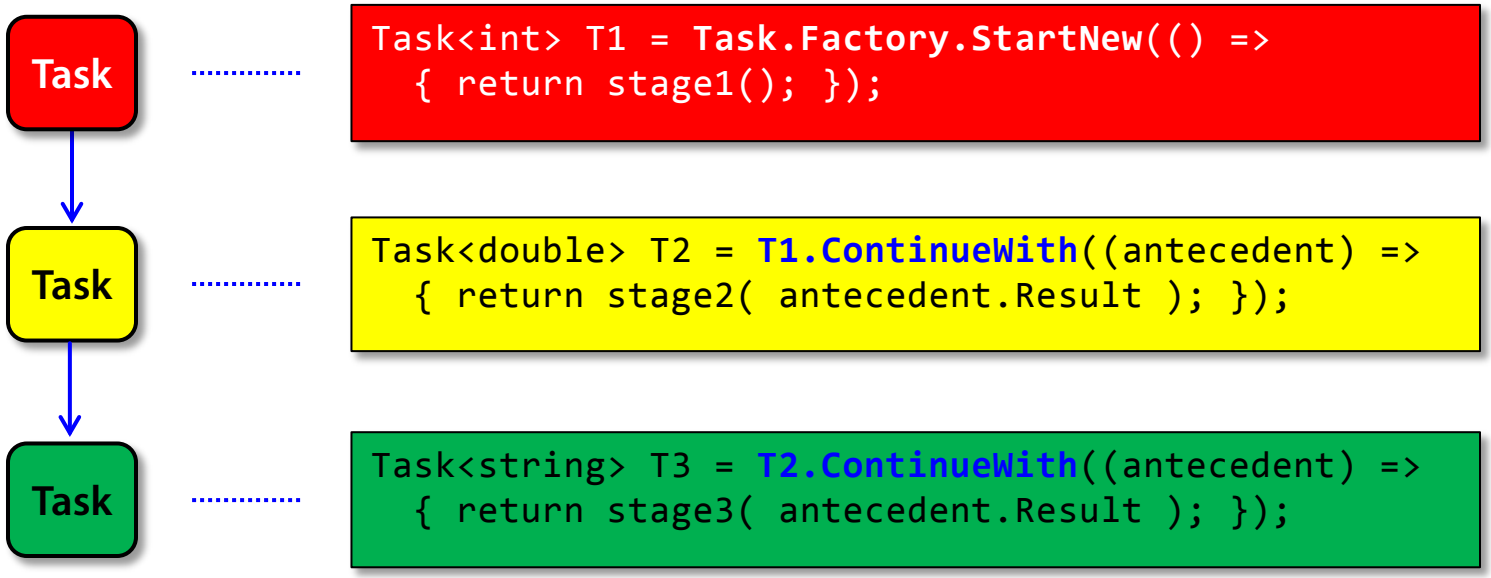
1. *Create with long-running option*
2. *Before app closes you need to join with task and catch any exceptions*
3. *Design clean shutdown of task so app closes (use TPL cancellation?)*
4. *If task can crash, design a way to monitor and restart (check task's `Status` property via a timer, or app's message loop?)*

Patterns

- Pipeline
- Dataflow
- Concurrent data structures
- Producer-Consumer
- Map-reduce
- Parallel LINQ
- Speculative Execution
- Asynchronous Programming Model

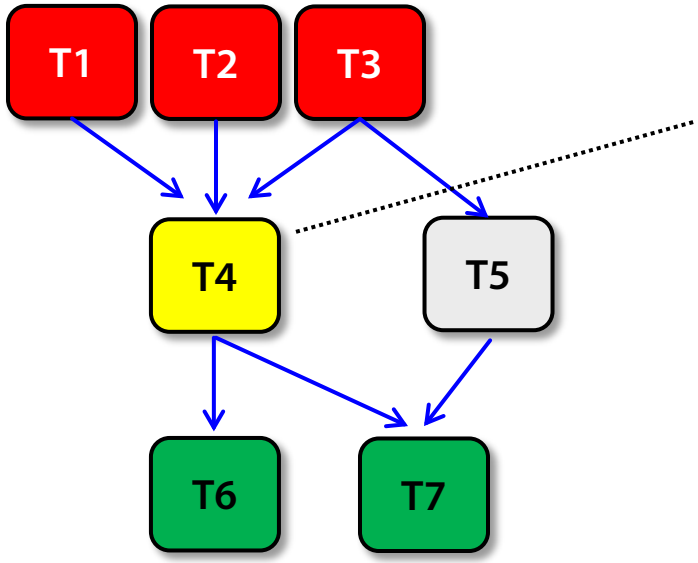
Pipeline

- **Linear flow from one task to another**
  - *image processing, UI updating, workflows…*



Dataflow

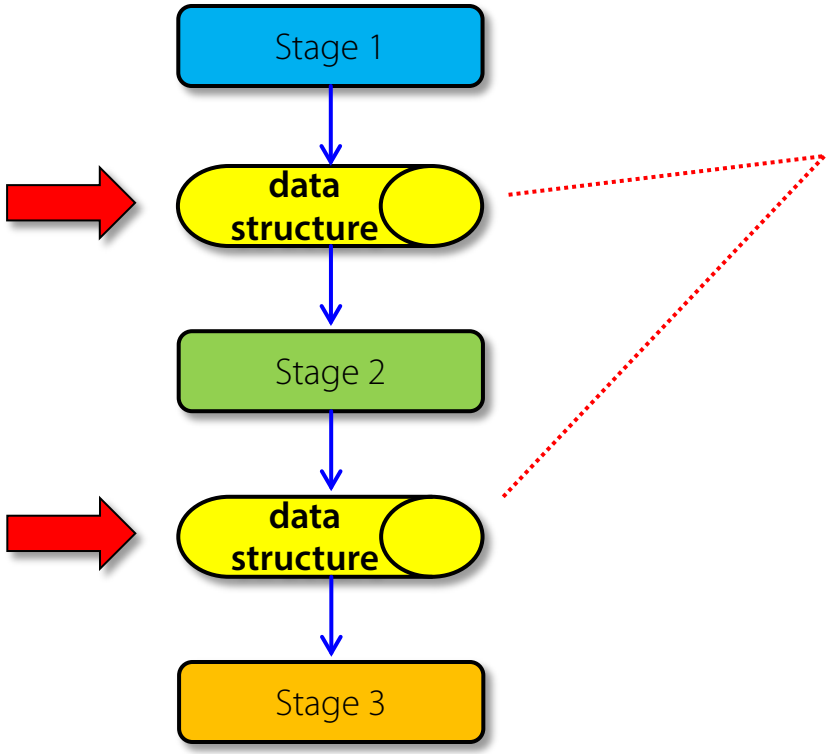
- **Generalized flow** with many-to-one and one-to-many…




```
Task<int> T1 = ...;
Task<int> T2 = ...;
Task<int> T3 = ...;

Task<double> T4 =
  Task.Factory.ContinueWhenAll<int, double>(
     new[] {T1, T2, T3},
     (tasks) =>
     {
        double result;
        foreach(Task<int> t in tasks)
          ...
        return result;
     }
  );
```

Increasing parallelism

- **Thread-safe data structures** allow stages to run **independently**…



bounded or unbounded data structures?

Concurrent Data Structures

- TPL offers a set of thread-safe data structures:
  - `ConcurrentBag<T>`
  - `ConcurrentQueue<T>`
  - `ConcurrentStack<T>`
  - `ConcurrentDictionary<T>`
  - `BlockingCollection<T>`

Bounded in size, blocking tasks when full or empty

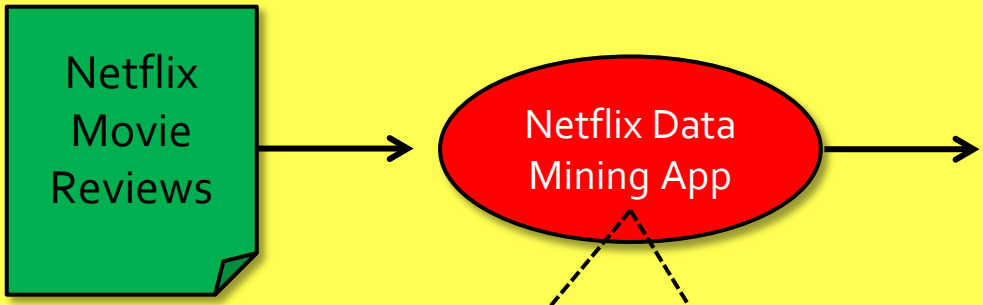
Example: ConcurrentQueue

- **Enqueue?**
  - *safe to call in parallel…*
- **Dequeue?**
  - *typically you check then dequeue…*
  - *this is *still* unsafe: you cannot separate check from dequeue, some other task could grab in-between — yielding race condition!*
  - *Solution? API redesigned with **TryDequeue** method that checks and dequeues in one call*
- **Enumeration?**
  - *Foreach, ToArray, …*
  - *thread-safe by creating snapshots*

```
using System.Collections.Concurrent;

var Q = new ConcurrentQueue<int>();

Task.Factory.StartNew(() =>
  { Q.Enqueue(123); });
Task.Factory.StartNew(() =>
  { Q.Enqueue(456); });


Task.Factory.StartNew(() =>
  {
    if (Q.Count > 0)
      DoWork( Q.Dequeue() );
  });

Task.Factory.StartNew(() =>
  { Work  w;
    if (Q.TryDequeue(out w))
      DoWork(w);
  });

Task.Factory.StartNew(() =>
  { contents = Q.ToArray(); });
```

DEMO

- Netflix data mining using a ConcurrentDictionary…



Users with most reviews (the "top 10" users)

```
305344: 992
387418: 983
   .
   .
   .
```

```
Dictionary<int, int> D = ...;

foreach(string review in file)
{
  userid = parse(review);
  if (!D.Contains(userid))
    D.Add(userid, 1);
  else
    D[userid]++;
}

Sort();
```

Sequential

```
ConcurrentDictionary<int, int> CD = ...;

Parallel.ForEach(file, (review) =>
{
   userid = parse(review);
   CD.AddOrUpdate(userid,
     1,
     (usr, count) => { return count+1; }
   );
});

Sort();
```

Parallel (v1)

Results?

- **Netflix app worked correctly with ConcurrentDictionary**

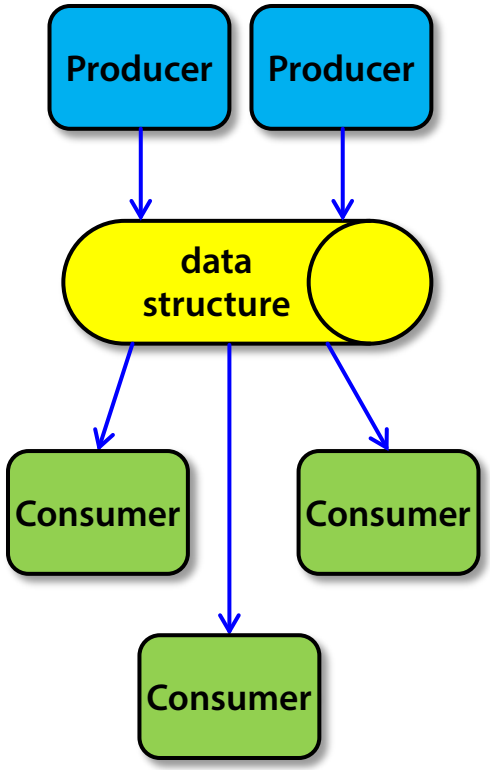
- **Unfortunately, parallel version ran slower**

  - ***In this application**, minimal data processing ==> contention for dictionary*
  - *ConcurrentDictionary does internal synchronization / locking*
  - *Too much contention ==> slower execution…*

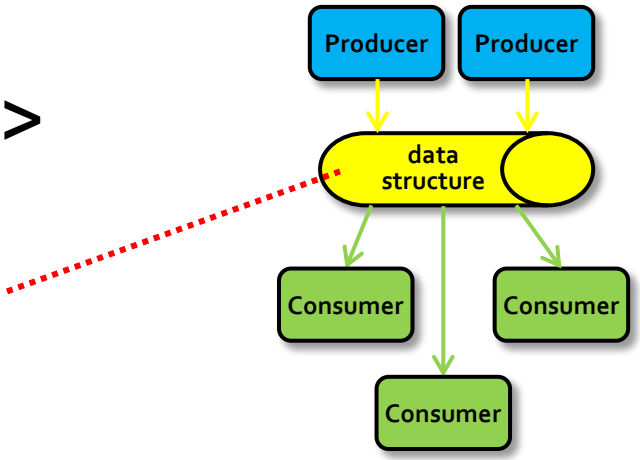
Producer-Consumer

- **Good pattern to use for long-running workloads where data generation speed is very different from data consumption speed**
  - *Data structure throttles faster component…*

- **Example:**
  - **producer(s)** read from disk / network
  - **consumer(s)** process the data



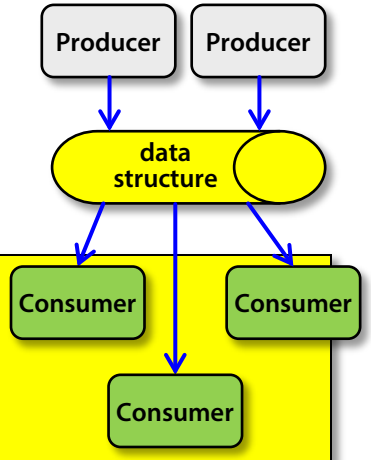
BlockingCollection<>



- **Think fixed-sized collection in a parallel world:**
  - *blocking producers if collection is full*
  - *blocking consumers if collection is empty*

- **Fixed-sized collections are more realistic**
  - *… and typically *improve* performance by throttling faster component!*

Implementation



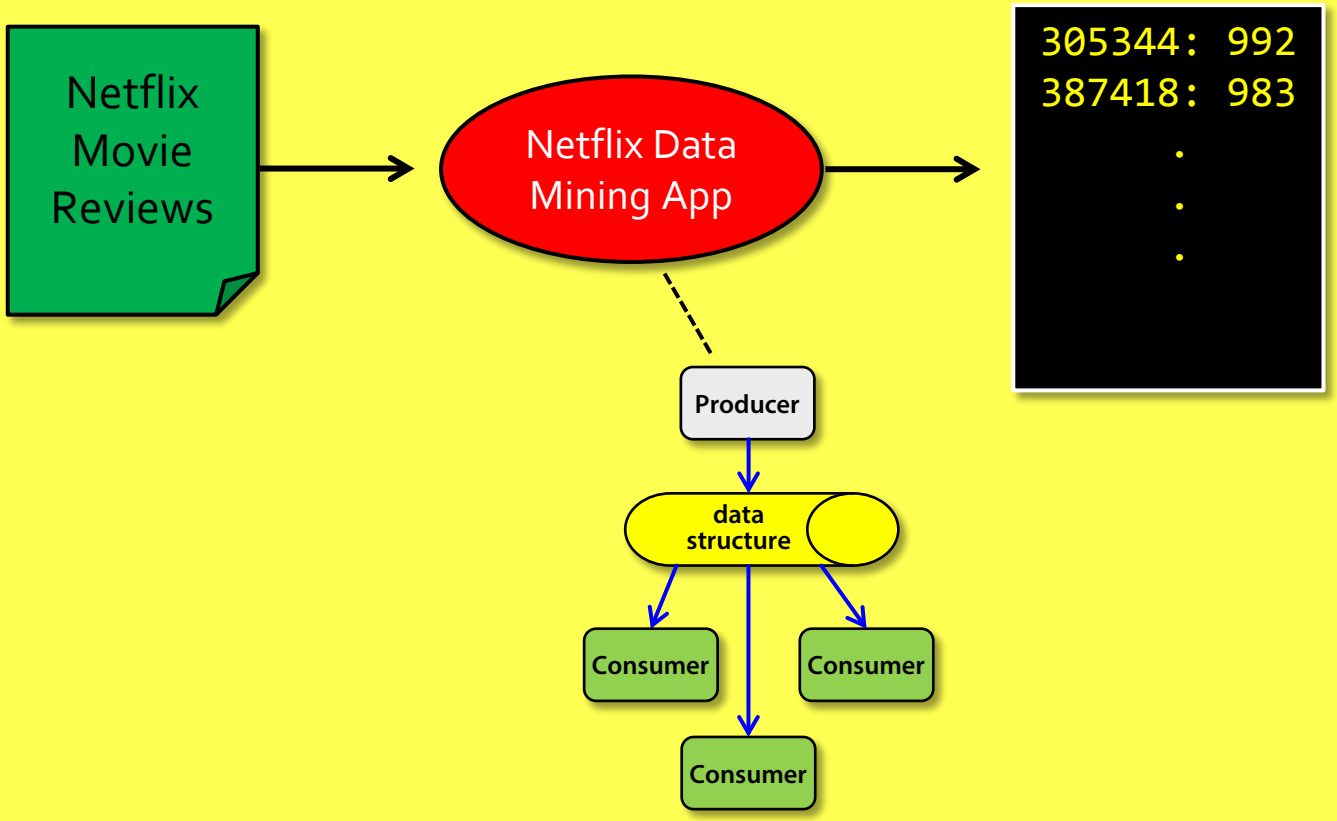

```
int maxCapacity = …;    // collection is a fixed-size:
var workQ       = new BlockingCollection<T>(maxCapacity);

// Create a new factory to produce long-running tasks:
var tf = new TaskFactory( TaskCreationOptions.LongRunning,
                          TaskContinuationOptions.None );
```

```
Task producer = tf.StartNew(() =>
{
   for(…)   // blocks task if Q is full:
     workQ.Add( work );

   // Signal we're done:
   workQ.CompleteAdding();
}
);
```

```
Task consumer = tf.StartNew(() =>
{
   while (!workQ.IsCompleted)
   {
      try {   // blocks task if Q is empty:
        T work = workQ.Take();
        .
        .   // process work item:
        .
      }
      // Exceptions ==> no more work:
      catch(ObjectDisposedException)   { }
      catch(InvalidOperationException) { }
   }
}
);
```

DEMO (v2)

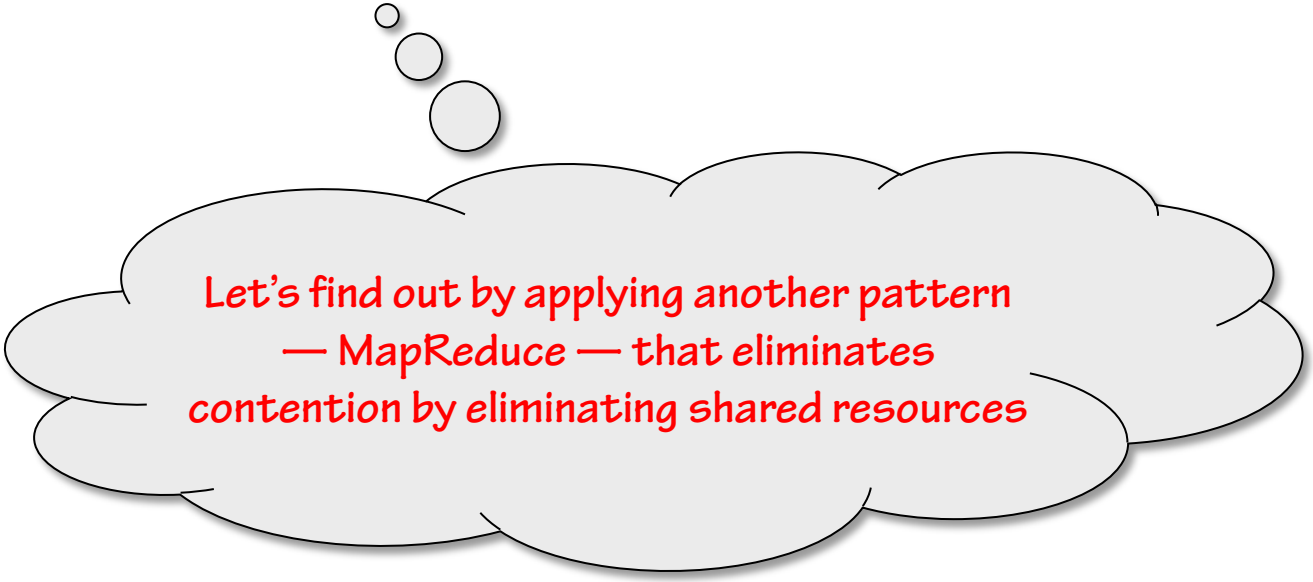
- Netflix data mining using Producer-Consumer…



Results?

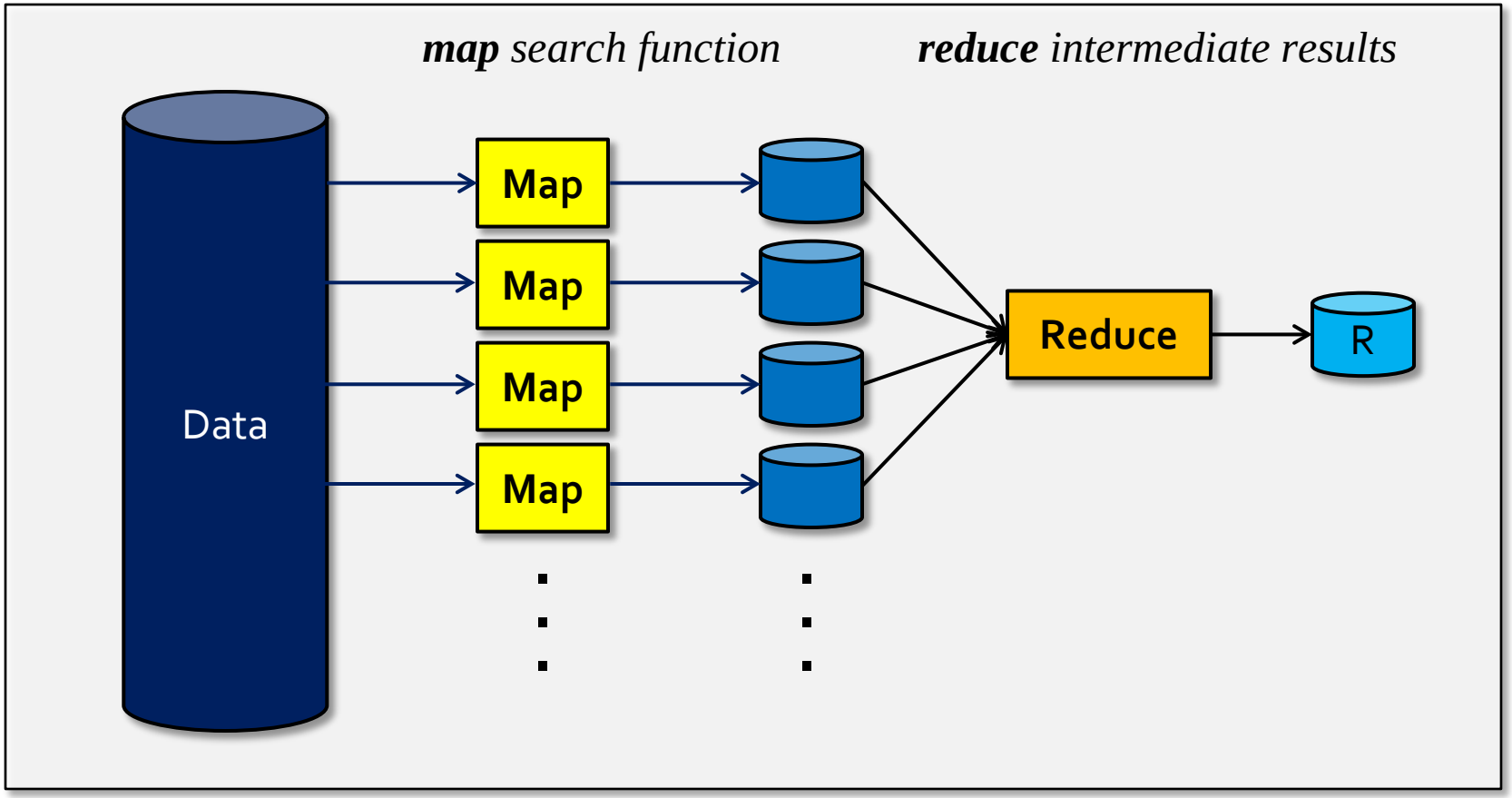
- **Producer-Consumer was correct *and* yielded a performance improvement!**

- **However, improvement was sub-linear**
  - *Bad fit given that I/O time and processing times are roughly the same (fast)?*
  - *Contention for the blocking collection?*



MapReduce

- Commonly used for embarrassingly parallel search / data-mining…



Implementing MapReduce

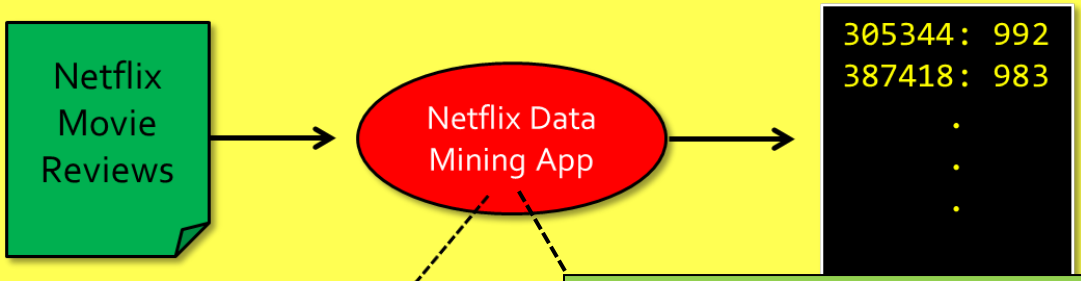
- **Various strategies…**
  - *fire off N Task<T>, use WaitAllOneByOne to reduce results as they finish*
  - *use Parallel.For / .Foreach with Task Local Storage (TLS)*

```
for (int i=0; i<N; ++i)
  tasks.Add(Task.Factory.StartNew<T>(
    (data) => { return map(data); }
  );

while (tasks.Count > 0)   // wait for tasks to finish:
{
   int index = Task.WaitAny( tasks.ToArray() );
   reduce(tasks[index].Result);
   tasks.RemoveAt(index);
}
```

```
Parallel.ForEach(datasource,
  () => { return new TLS(); }, // initializer:

  (datum, …, tls) =>           // task body:
  {
    map(datum, tls);
    return tls;
  },

  (tls) => { reduce(tls); }    // finalizer:
);
```

DEMO (v3)

- Netflix data mining using MapReduce…




```
Dictionary<int, int> D = ...;

foreach(string review in file)
{
  userid = parse(review);
  if (!D.Contains(userid))
    D.Add(userid, 1);
  else
    D[userid]++;
}

Sort();
```

Sequential

```
Dictionary<int, int> result = new Dictionary<int, int>();

Parallel.ForEach(File.ReadLines(infile),
  () => { return new Dictionary<int, int>(); },
  (line, loopControl, localD) =>      // task body:
  {
    userid = parse(line);
    if (!localD.Contains(userid))
      localD.Add(userid, 1);
    else
      localD[userid]++
    return localD;
  },
  (localD) => { lock(result) { merge(result, localD); } }
);
```

same as sequential version

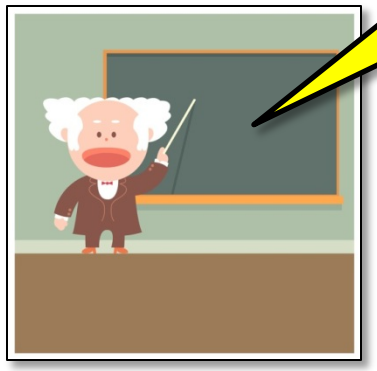
Results?

- **MapReduce yields best performance so far!**

  - *Eliminating contention usually does…*

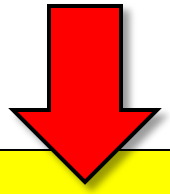
Extra credit for the viewer!

*Take a look at the provided solution "Netflix-Par-FileChunk-LessStrict". It uses MapReduce but improves performance by partitioning the data itself, *and* also by a willingness to accept the occasional imprecise result.*



Parallel LINQ

- Are you a fan of LINQ — Language Integrated Query?
- Then you'll love PLINQ and its support for:
  - *Parallelism*
  - *MapReduce*

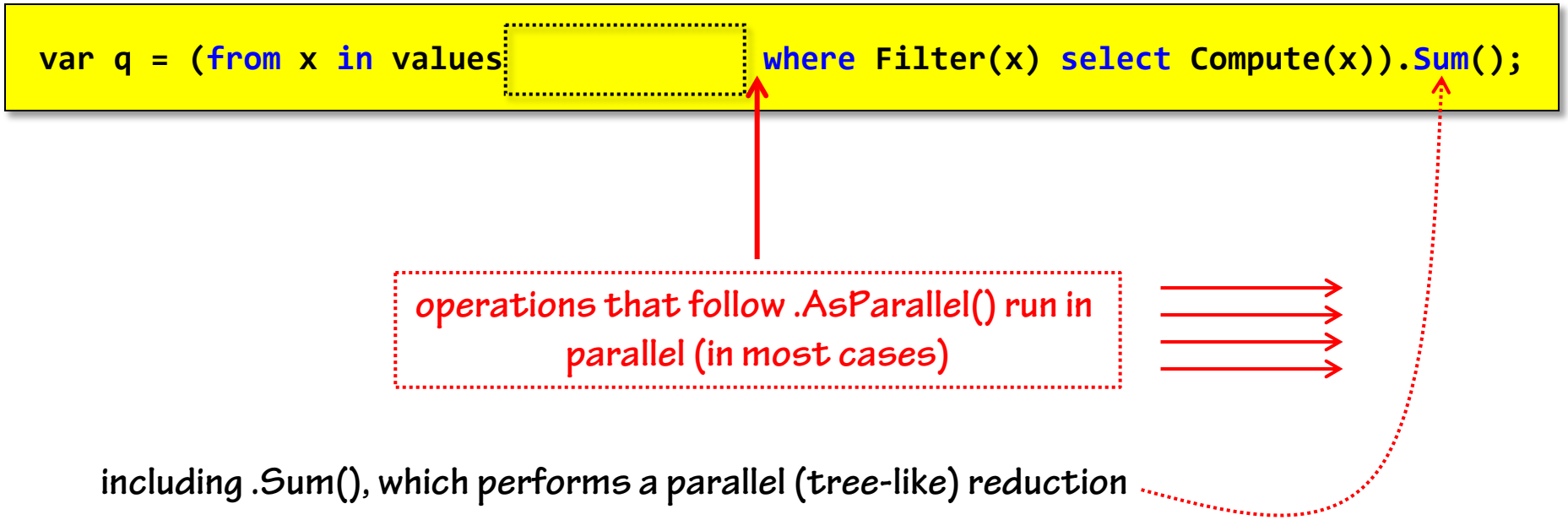


```
var query = ( from x in values.AsParallel()
  where  Filter(x)
  select Compute(x) ).Sum();
```

```
var query = ( from x in values
              where  Filter(x)
              select Compute(x) ).Sum();
```

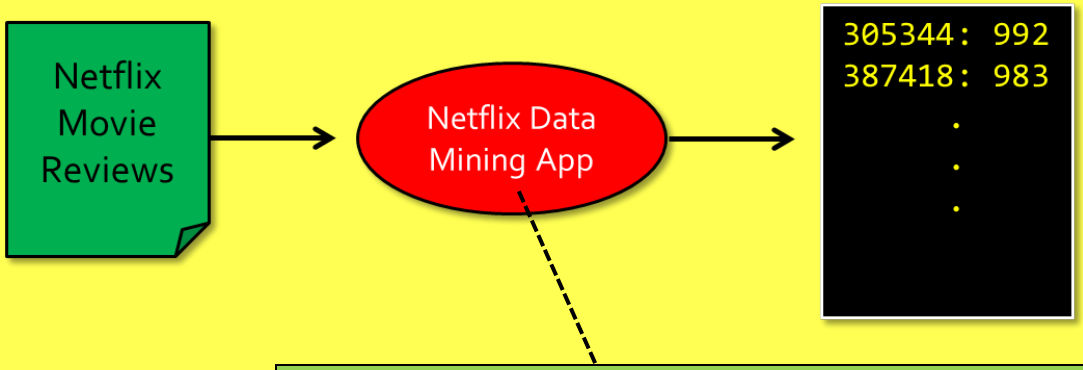
A brief overview…

- LINQ enables querying of IEnumerable data sources
- PLINQ turns an IEnumerable into a ParallelQuery data source
  - *for in-memory collections*



DEMO (v4)

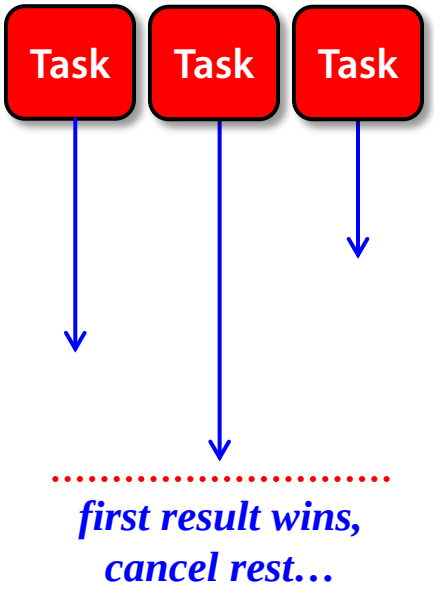
- Netflix data mining using PLINQ…




```
var ReviewsByUser = File.ReadLines(infile).
  Select(line => parse(line)).
  GroupBy( userid => { return userid; } ).
  Select( g => new { UserId = g.Key, NumReviews = g.Count() } );

var Top10 = (from user in ReviewsByUser
  orderby user.NumReviews descending, user.UserId ascending
  select user).Take(10).ToList();
```

Speculative Execution

- When you have multiple sources for generating a result…
- **Example:**
  - *Calling web services…*




```
var cts   = new CancellationTokenSource();
var token = cts.Token;

for (int i=0; i<N; ++i)
  tasks.Add(Task.Factory.StartNew<T>(
    () => { ... },
    token
  );

int winner = Task.WaitAny( tasks.ToArray() );
var result = tasks[winner].Result;

cts.Cancel();   // tell the rest to cancel…
```

Asynchronous Programming Model

- **APM pattern is commonly used for asynchronous operations**
  - *File I/O, network I/O, …*
  - *"**Begin**" starts operation / "**End**" completes operation & harvests result*

- **Various .NET classes support the APM pattern**
  - `FileStream:` *BeginRead / EndRead*
  - `HTTPWebRequest:` *BeginGetResponse / EndGetResponse*
  - `...`

- **Advantage?**
  - *.NET classes start operation on a thread, but then return thread to pool until operation completes*

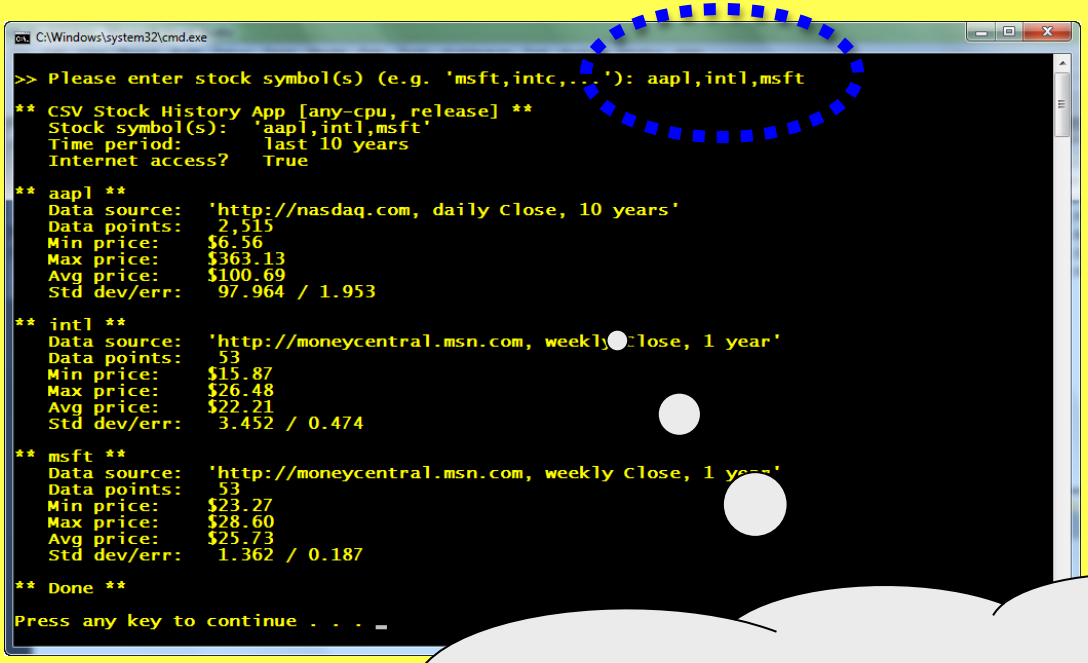
TPL support for APM pattern

- **TPL provides first-class APM support via façade tasks**
  - *Use FromAsync to create task that wraps calls to Begin / End*
  - *Use standard task mechanisms to wait / continue / harvest result*

```
FileInfo fi = new FileInfo(filename);
int bytes   = (int) fi.Length;
byte[] buf  = new byte[bytes];

FileStream fs = new FileStream(filename, FileMode.Open, FileAccess.Read,
                               FileShare.Read, bytes, true /*async*/ );

var T1 = Task<int>.Factory.FromAsync(fs.BeginRead, fs.EndRead, buf, 0, bytes, null);

var T2 = T1.ContinueWith((antecedent) =>
  {
     fs.Close();
     int bytesRead = T1.Result;
     .
     .   // process data in buf:
     .
  });
```

DEMO

- Historical stock data analysis…

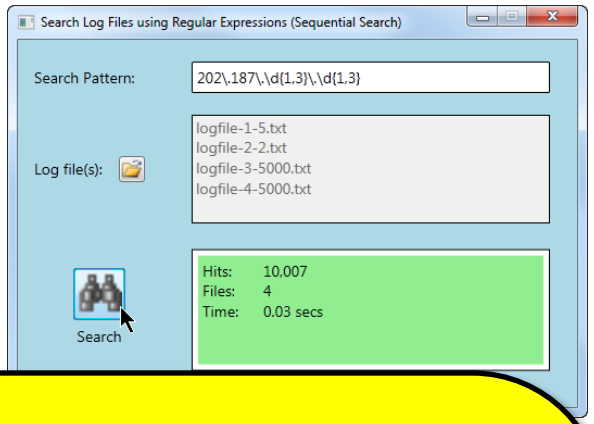


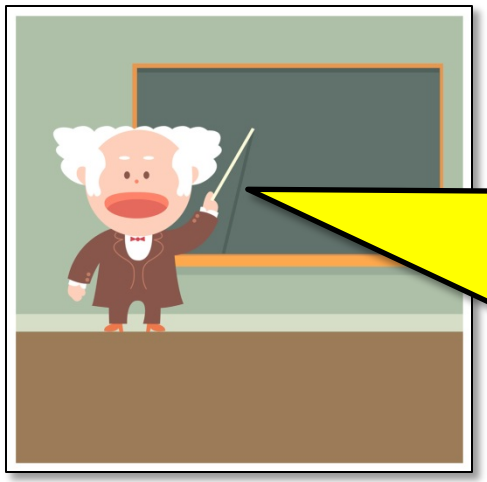
Good example of both patterns:

1. *Speculative execution of downloads*
2. *APM and façade tasks for HTTP requests*

Aside: Parallel I/O

- I/O is *hard* to parallelize
  - *requires support from underlying OS and physical hardware*
  - *sensitive to data sizes*
  - *don't expect much performance from commodity hardware*





Design Challenge!

The first lecture — "Understanding the Dangers of Concurrency" — focused on a Log File Search app. However, none of the provided solutions work for arbitrarily large files (they all fail with an "Out of Memory" exception).

Provided in the "before\" sub-folder of this lecture is a sequential version that works for arbitrarily large files, along with two parallel versions that fail (one does synchronous I/O, and the other does async I/O using the APM). Based on what you've learned in this course, design a correct, high-performing parallel version for arbitrarily large files.

Summary

- **Software design is challenging**
- **Asynchronous and Parallel software is even more challenging :-)**

- **Success measured in terms of both correctness and performance**
- **Increase your odds of success by using patterns:**
  - *Structured parallelism*
  - *Pipleline and Dataflow*
  - *Concurrent data structures*
  - *Producer-Consumer*
  - *MapReduce*
  - *PLINQ*
  - *Speculative execution*
  - *APM*

References

- **Microsoft's main site for all things parallel:**
  - http://msdn.microsoft.com/concurrency
- **LINQ:**
  - LINQ: http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b
  - PLINQ: http://msdn.microsoft.com/en-us/library/dd460688.aspx
  - Parallel.Foreach vs. PLINQ: http://blogs.msdn.com/b/pfxteam/archive/2010/04/21/9997559.aspx
- **I highly recommend the following short, easy-to-read book:**
  - *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*, by C. Campbell, R. Johnson, A. Miller and S. Toub, Microsoft Press

    Online: http://tinyurl.com/tpl-book