Concurrent Programming with Parallel Extensions to .NET

Joe Duffy
Architect & Development Lead
Parallel Extensions

Talk Outline

- Overview
- 5 things about Parallel Extensions
  1. Tasks and futures
  2. Parallel loops
  3. Parallel LINQ
  4. Continuations
  5. Concurrent containers
- What the future holds

Why Concurrency?

"[A]fter decades of single core processors, the high volume processor industry has gone from single to dual to quad-core in just the last two years. Moore's Law scaling should **easily let us hit the 80-core mark** in mainstream processors **within the next ten years** and quite possibly even less."

-- Justin Ratner, CTO, Intel (February 2007)

"If you haven't done so already, **now is the time** to take a hard look at the design of your application, determine what operations are CPU-intensive now or are likely to become so soon, and **identify how those places could benefit from concurrency**."

-- Herb Sutter, C++ Architect at Microsoft (March 2005)

What Changes?

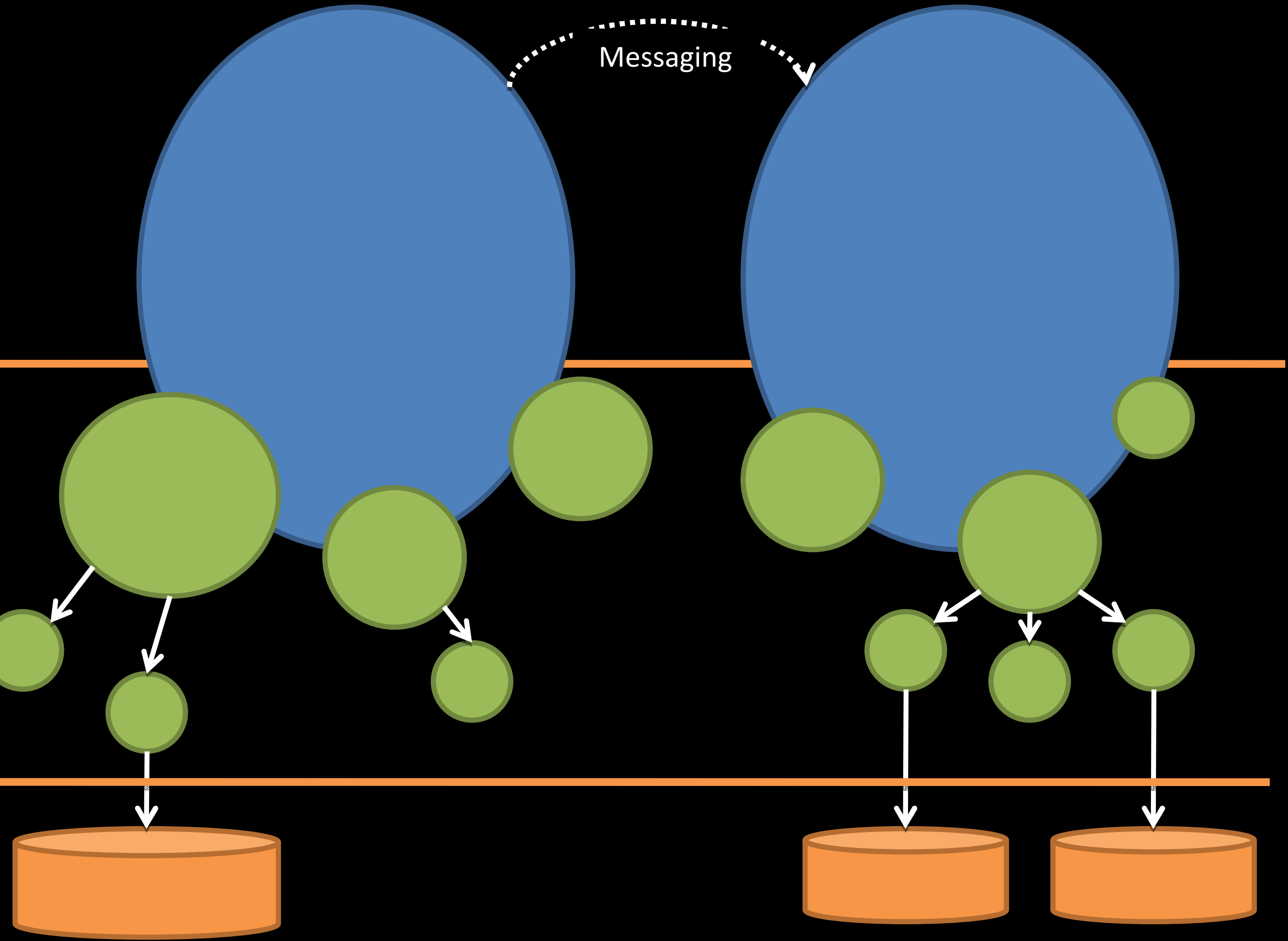
- Familiar territory for servers
  - Constant stream of incoming requests
  - Each runs (mostly) independently
  - So long as IncomingRate > #Procs, we’re good
  - Focus: throughput! => $$$
- Not-so-familiar territory for clients
  - User- and single-task centric
  - Button click => multiple pieces of work(?)
  - Focus: responsiveness! => ☺ ☺ ☺

Finding Parallelism

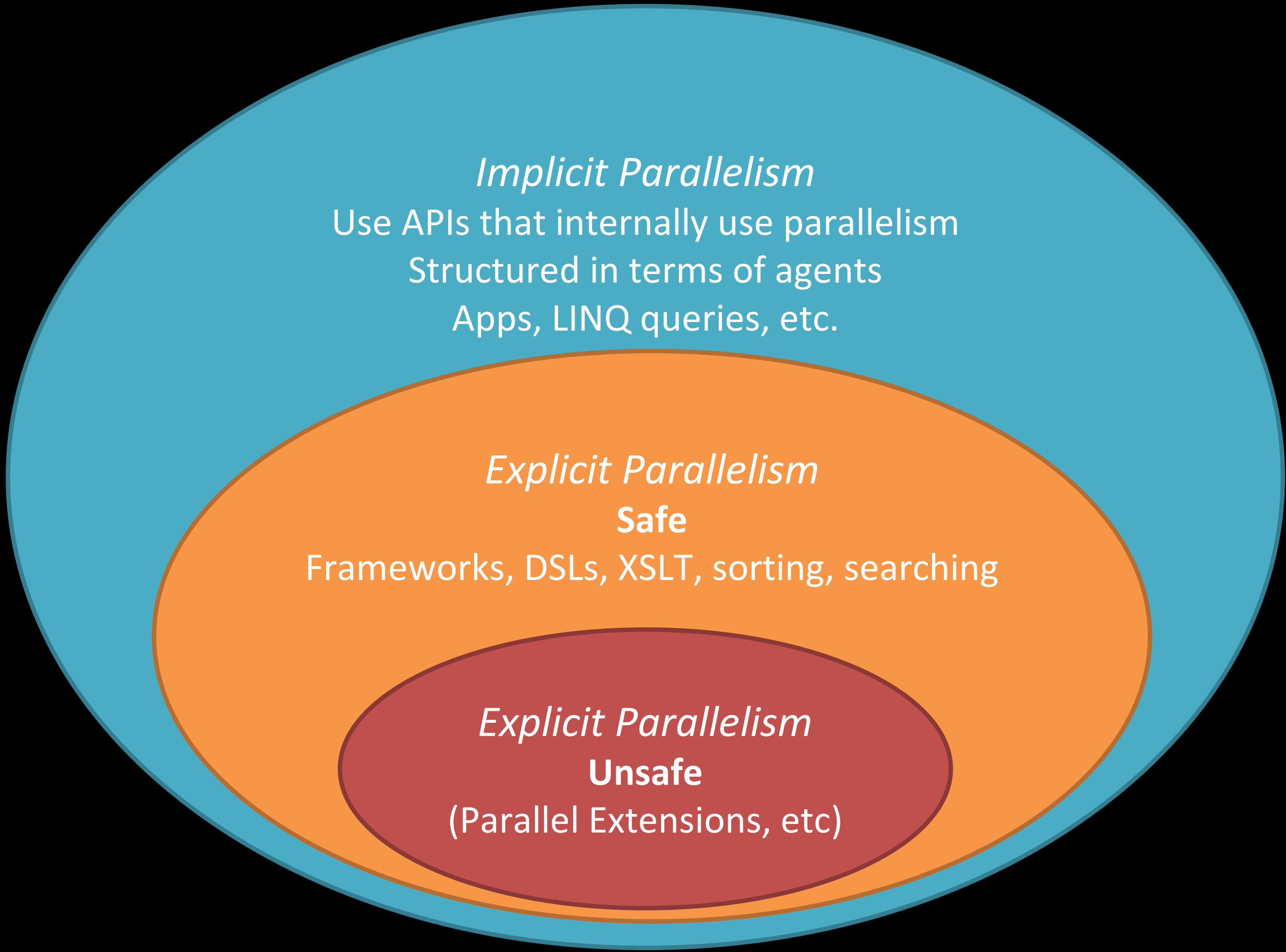
Agents/CSPs
* Message Passing
* Loose Coupling

Task Parallelism
* Statements
* Structured
* Futures
* ~O(1) Parallelism

Data Parallelism
* Data Operations
* O(N) Parallelism



All Programmers Will Not Be Parallel

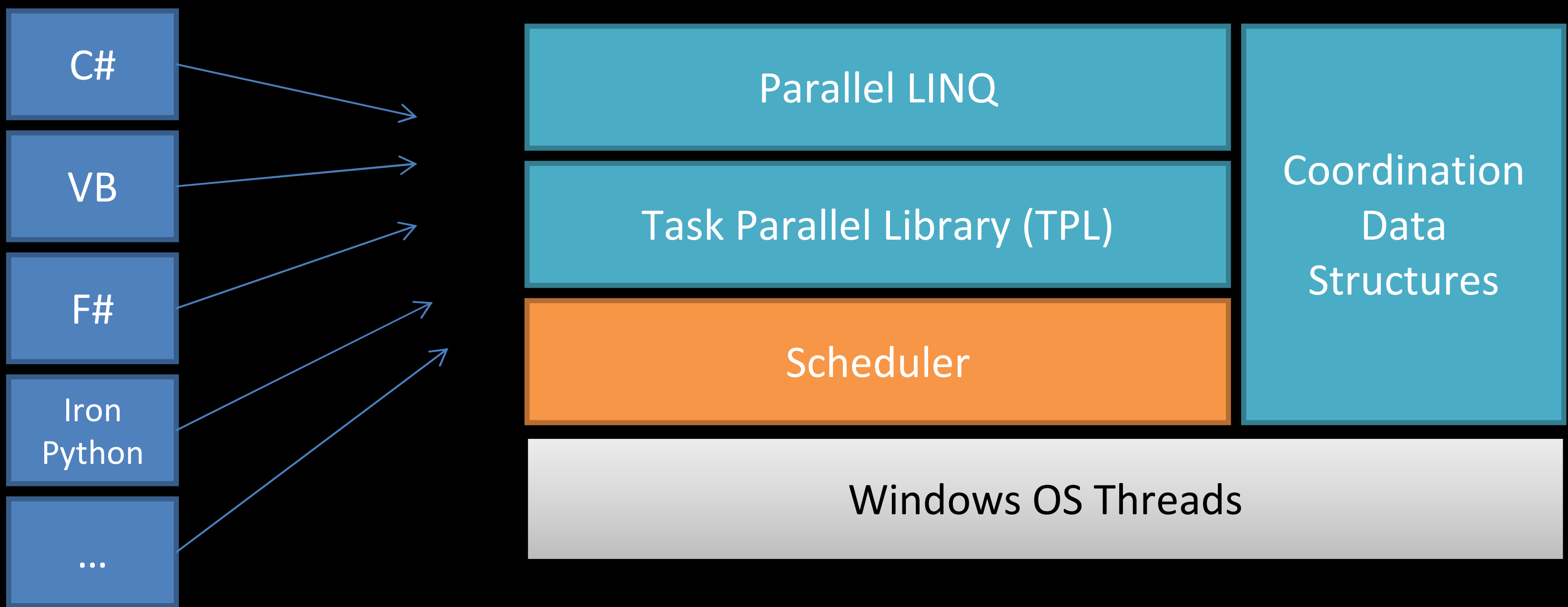


Threading (Today) == 

- It’s C’s fault: thin veneer over hardware/OS
- No logical unit of concurrency
  - Threads are physical
  - ThreadPool is close, but lacks richness
- Synchronization is ad-hoc and scary
  - No structure
  - Patterns (eventually) emerge, but not 1st class
  - Composition suffers
- Platform forces static decision making
  - We’d like sufficient latent parallelism that
  - Programs get faster as cores increase, and ..
  - Programs don’t get slower as cores decrease
- We can do better …

Parallel Extensions to .NET

- New .NET library
  - 1^{st} class data and task parallelism
  - Downloadable in preview form from MSDN
  - System.Threading.dll



API Map

- System.Linq
  - ParallelEnumerable [**PLINQ**]
  - …
- System.Threading [**CDS**]
  - AggregateException
  - CountdownEvent
  - ManualResetEventSlim
  - Parallel [**TPL**]
  - ParallelState [**TPL**]
  - SemaphoreSlim
  - SpinLock
  - SpinWait
  - …
- System.Threading.Collections [**CDS**]
  - BlockingCollection<T>
  - ConcurrentStack<T>
  - ConcurrentQueue<T>
  - IConcurrentCollection<T>
- System.Threading.Tasks [**TPL**]
  - Task
  - TaskCreationOptions (enum)
  - TaskManager
  - Future<T>

#1 Tasks and Futures

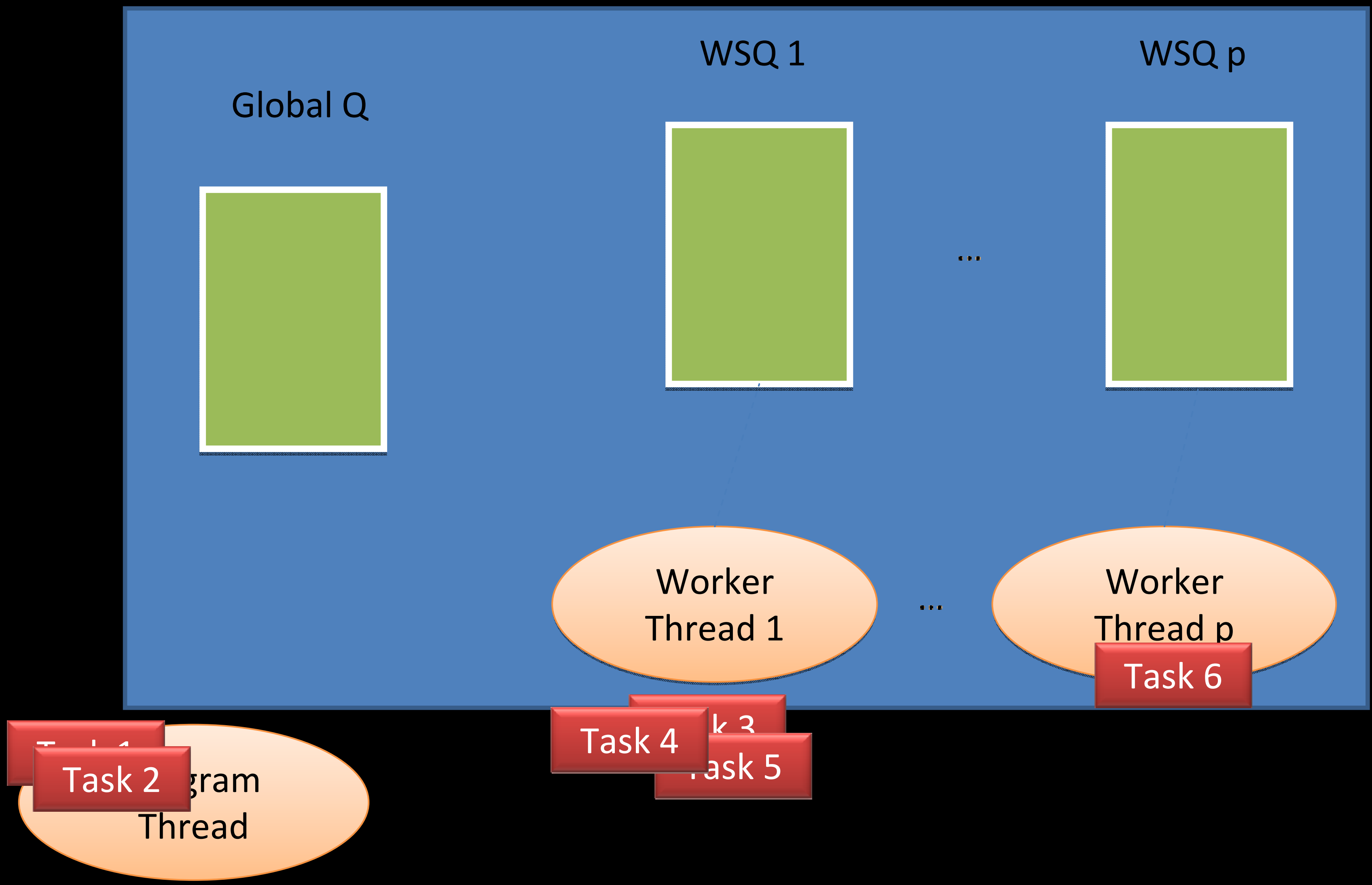
- Task represents a logical unit of work
  - Latent parallelism
  - May be run serially
  - Parent/child relationships
- Future<T> is a task that produces a value
  - Accessing Value will
    - Runs it serially if not started
    - Block if it's being run
    - Return if the value is ready
    - Throw an exception if the future threw an exception
- Can wait on either (Wait, WaitAll, WaitAny)
  - Runs the task "inline" if unstarted

Creating/Waiting

```
Task t1 = Task.Create(() => {
    // Do something.
    Task t2 = Task.Create(() => { … });
    Task t3 = Task.Create(() => { … },
        TaskCreationOptions.DetachedFromParent);
    // Implicitly waits on t2, but not t3.
});
…
t1.Wait();

Future<int> f1 = Future.Create(() => 42);
…
int x = f1.Value;
```

Work Stealing



Cancellation

```
Task t1 = Task.Create(() => {
    Task t2 = Task.Create(() => { … });
    Task t3 = Task.Create(() => { … },
        TaskCreationOptions.RespectParentCancellation);
});
…
t1.Cancel();
```

- t1 unstarted? Cancelled!
- t1 started? IsCancelled = true.
  - t3 unstarted? Cancelled!
  - t3 started? IsCancelled = true.
- (Note: t2 left untouched.)

Applied Use: IAsyncResult Interop

DEMO

#2 Parallel Loops

- Structured patterns for task usage
  - `static void For(`
    `int fromInclusive, int toExclusive, Action<int> body);`
  - `static void ForEach<T>(`
    `IEnumerable<T> source, Action<T> body);`
- Each iteration *may* run in parallel
- Examples
  - `Parallel.For(0, N, i => …);`
  - `Parallel.ForEach<T>(e, x => …);`
- Void return type
  - Must contain side-effects to be useful (beware!)
  - Implies non-interference among iterations

Matrix Multiplication

DEMO

Parallel Loop Reductions

- Ability to write reductions
  - ```
    static void For<TLocal>(
        int fromInclusive, int toExclusive,
        Func<TLocal> init,
        Func<int, ParallelState<Tlocal>> body,
        Action<TLocal> finish);
    ```

- E.g., sum reduction
  - ```
    int[] ns = …;
    int accum = 0;
    Parallel.For(
        0, N, () => 0,
        (i, ps) => ps.Local += ns[i],
        x => Interlocked.Add(ref accum, x));
    ```

Parallel Statement Invokes

- Ability to run multiple statements in parallel
  - `static void Invoke(Action[] actions);`

- Example
  - Parallel.Invoke(

```
Parallel.Invoke(
        () => { x = f(); },
        someAction,
        () => someOtherFunction(z),
        …
  );
```

#3 Parallel LINQ

- Implementation of LINQ that runs in parallel
  – Over in-memory data
  – Arrays, collections, XML, …
- Support for all LINQ operators
  – Maps (Select)
  – Filters (Where)
  – Reductions (Aggregate, Sum, Average, Min, Max, …)
  – Joins (Join)
  – Groupings by key (GroupBy)
  – Existential quantification (Any, All, Contains, …)
  – And more

λ Imperative == !Parallel λ

"Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic. **The assignment statement is the von Neumann bottleneck of programming languages** and keeps us thinking in **word at-a-time** terms in much the same way the computer's bottleneck does."

-- John Backus,
Can Programming be Liberated from the von Neumann Style?
1978 ACM Turing Award Lecture

Just Add AsParallel

- Comprehension syntax
  - Serial:
```
var q = from x in data where p(x) select f(x);
```
  - Parallel:
```
var q = from x in data.AsParallel() where p(x) select f(x);
```
- Direct method calls
  - Serial:
```
Enumerable.Select(
    Enumerable.Where(data, x => p(x)),
    x => f(x));
```
  - Parallel:
```
ParallelEnumerable.Select(
    ParallelEnumerable.Where(data.AsParallel(), x =>p(x)),
    x => f(x));
```

Example: Sequential "Baby Names"

```
IEnumerable<BabyInfo> babies = ...;
var results = new List<BabyInfo>();

foreach (var baby in babies)
{
    if (baby.Name == queryName &&
        baby.State == queryState &&
        baby.Year >= yearStart &&
        baby.Year <= yearEnd)
    {
        results.Add(baby);
    }
}

results.Sort((b1, b2) => b1.Year.CompareTo(b2.Year));
```

Example: Hand-Parallel “Baby Names”

```
IEnumerable<BabyInfo
var results = new L
int partitionsCou                    sorCount;
int remainingCou
var enumerator = b
try {
    using (ManualReset                    etEvent(fals
        for (int i = 0;
            ThreadPool.Queu
                var partialRe
                while(true) {
                    BabyInfo baby;
                    lock (enumerato
                        if (!enumerato
                        baby = enumerato
                    }
                    if (baby.Name == c                          &&
                        baby.Year >=                             {
                            parti
                    }
                }
                lock (resul                    ial
                if (Inter                      iningCou
            });
        }
        done.WaitO
        results.                  CompareTo(b2.Year));
    }
}
finally { if (enumera             able) ((IDisposable)enumerator).Dispo
```

Synchronization Knowledge

nt locking

k of foreach simplicity

Manual aggregation

Tricks

of thread reuse

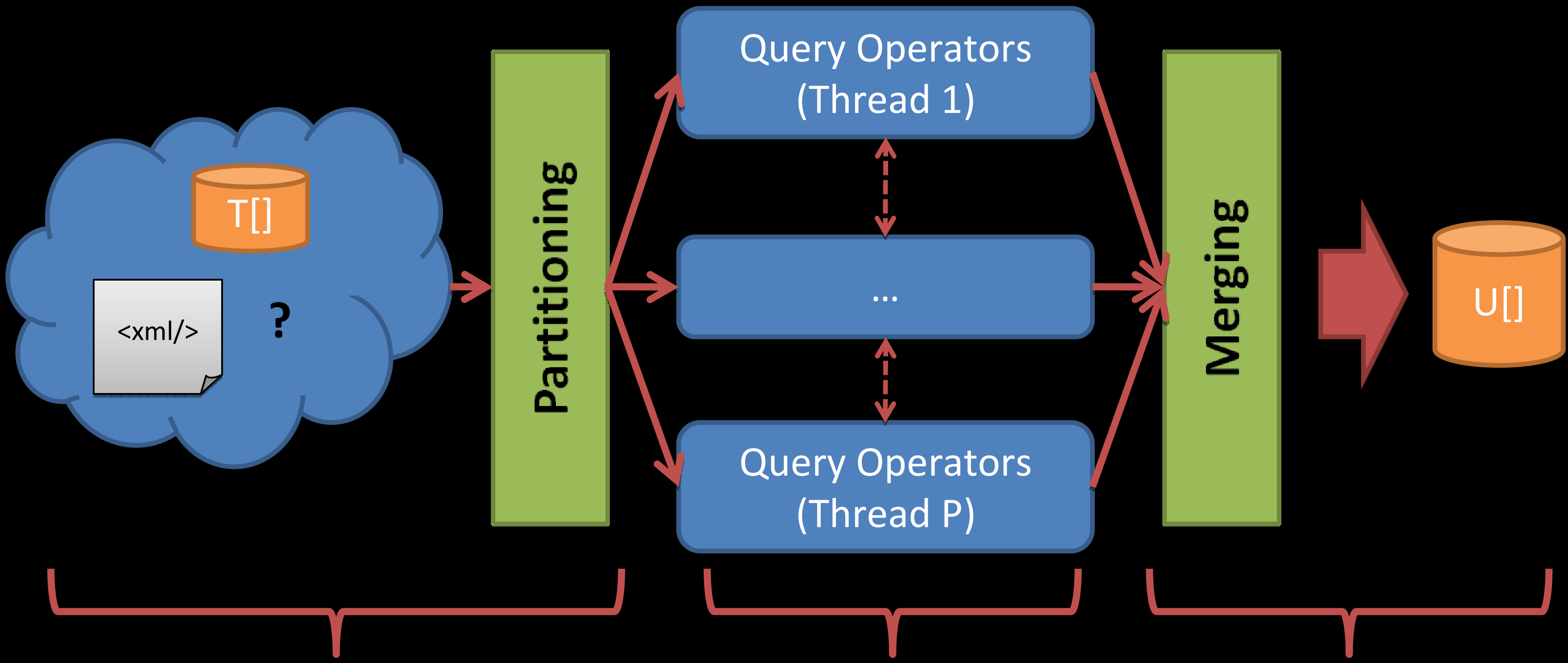
nchronization

Non-parallel sort

Example: “Baby Names” in (P)LINQ

```
var results = from baby in babies.AsParallel()
              where baby.Name == queryName &&
                    baby.State == queryState &&
                    baby.Year >= yearStart &&
                    baby.Year <= yearEnd
              orderby baby.Year ascending
              select baby;
```

Query Execution



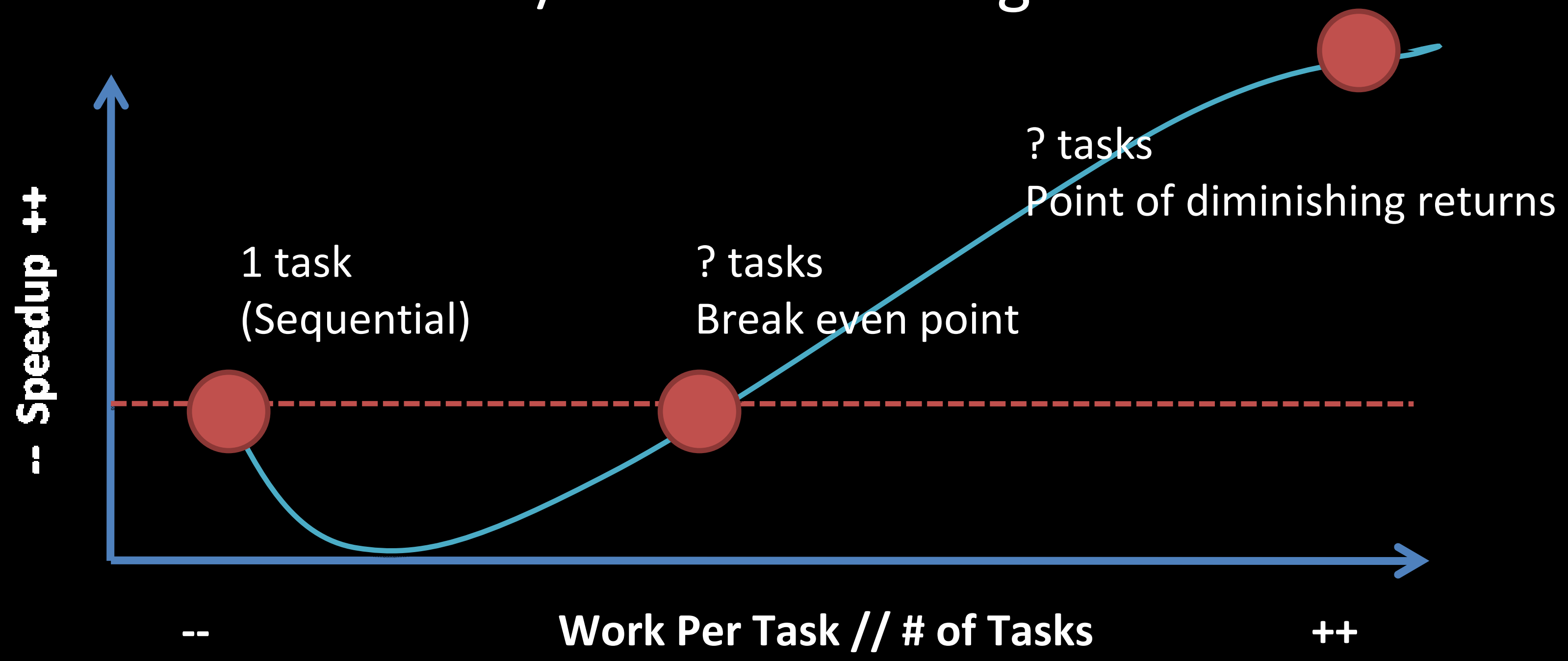
•Data-Source Specific Partitioning
•“Fork”

•Parallel Region
•Minimal Communication

•“Join”
•Union / Sort / Reduction / …

When to “Go Parallel”? (TPL+PLINQ)

- There is a cost; only worthwhile when
  - Work per task/element is large, and/or
  - Number of tasks/elements is large



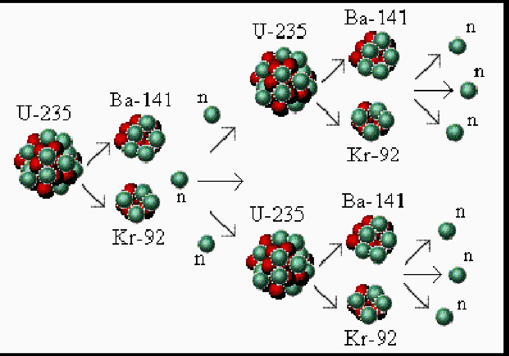
Break Even Point

DEMO

#4 Continuations

- Blocking is bad
  - Holds up a thread (~1MB stack, etc.)
  - Unblocking cannot be throttled (stampedes, cache thrhasing)
  - Requires a “spare” thread to keep the system busy
- Yet non-blocking is hard
  - Manual continuation passing style (CPS)
  - Can’t transform the whole stack
- TPL lets you choose
  - Wait blocks
  - ContinueWith doesn’t

ContinueWith



- Simple "event handler" style

```
Task t1 = Task.Create(() => …);
Task t2 = t1.ContinueWith(t => …);
```

- Only when certain circumstances occur

```
Task t1 = Task.Create(() => …);
Task t2 = t1.ContinueWith(t => …,
    TaskContinuationKind.OnCancelled);
```

- Dataflow chaining

```
Future<int> t1 = Future.Create(() => 42);
Future<string> t2 = t1.ContinueWith(
    t => t.Value.ToString());
string s = t2.Value; // "42"
```

#5 Concurrent Containers

- Coordination often happens with lists
  - OS: runnable queues
  - Producer/consumer: queues
  - Messages to be dispatched
  - Etc.
- Several containers "out of the box"
  - In the System.Threading.Collections namespace
  - ConcurrentStack<T> - lock free LIFO stack
  - ConcurrentQueue<T> - lock free FIFO queue
- More to come:
  - ConcurrentBag<T> - unordered work stealing queues
  - ConcurrentDictionary<K,V> - fine grained locking, lock free reads
  - Etc.

Lock Free Stack

DEMO

Blocking Collection

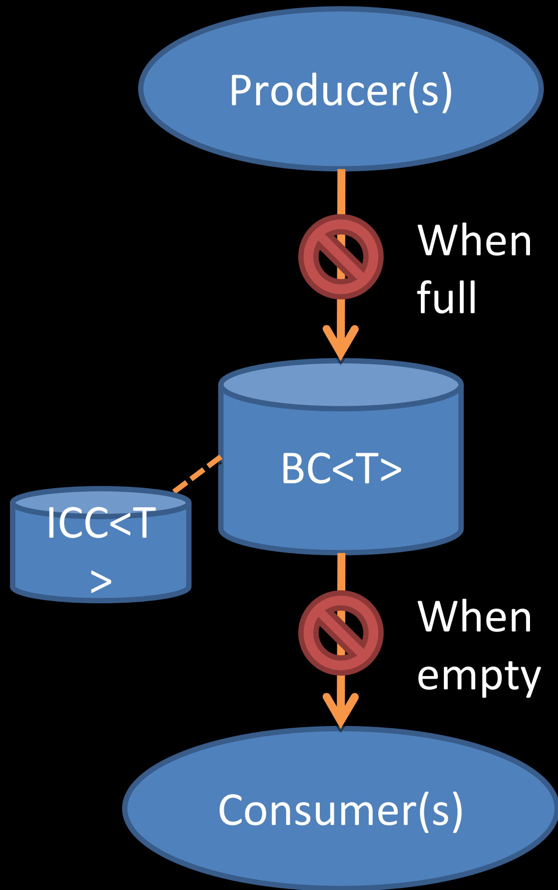
- N producers and M consumers
- Automatic blocking when empty

```
var bc = new BlockingCollection<T>();
T t1 = bc.Remove(); / / If empty, waits.
T t2;
if (bc.TryRemove(ref t2)) …;
```

- Optional bounding when full

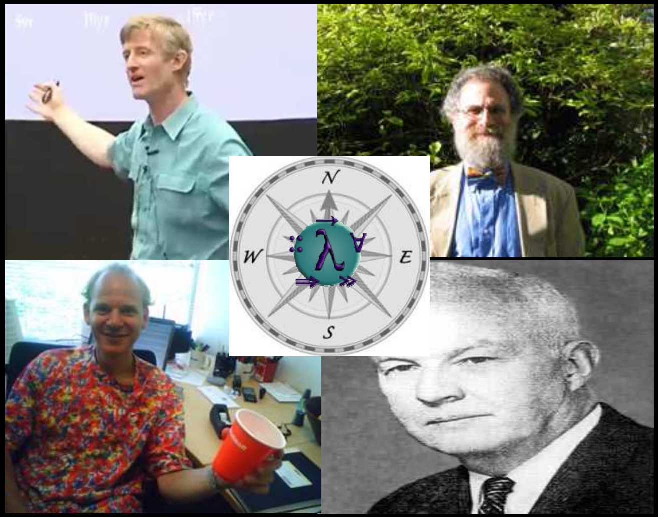
```
var bc = new BlockingCollection<T>(1000);
T e = …;
bc.Add(e);
if (be.TryAdd(e)) …;
```

- Can wrap any IConcurrentCollection<T>
  - Stack and queue both implement it
  - Defaults to queue if unspecified



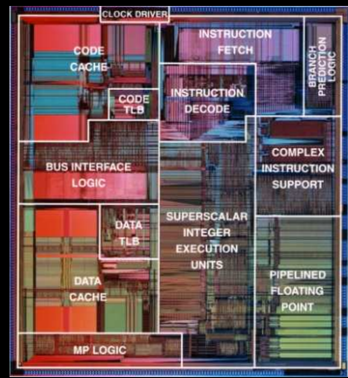
The Future: Programming Models

- Safety
  - Major hole in current offerings (sharp knives)
  - Three key themes
    - Functional: immutability and purity
    - Safe imperative: isolated
    - Safe side-effects: transactions
  - Haskell is the One True North
- Patterns
  - Agents (CSPs) + tasks + data
  - 1st class isolated agents
  - Continue to raise level of abstraction: what, not how



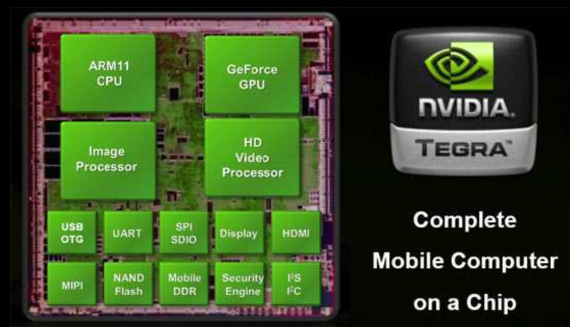
The Future: Efficiency and Heterogeneity

- Efficiency
  - "Do no harm" O(P) >= O(1)
  - More static decision-making vs. all dynamic
  - Profile guided optimizations
- The future is heterogeneous
  - Chip multiprocessors are "easy"
  - Out-of-order vs. in-order
  - GPGPU' (fusion of X86 with GPU)
  - Vector ISAs
  - Possibly different memory systems



+

=~



In Conclusion

- Opportunity and crisis
  - Competitive advantage for those who figure it out
  - Less incentive for the client platform otherwise
- Technologies are immature
  - Parallel Extensions is still only a preview
  - And even that is one small step …
  - Even client hardware of 5-10 yrs is unsettled
- Architects and senior developers pay attention
  - Can make a real difference **today** in select places
  - But not yet for broad consumption
  - 5 year horizon
  - Time to start thinking and experimenting

Q&A

- Thanks!
- Team site: http://msdn.microsoft.com/concurrency/ (With CTP download!)
- Team blog: http://blogs.msdn.com/pfxteam/
- My blog: http://www.bluebytesoftware.com/blog/
- **Book is out in Oct 2008**