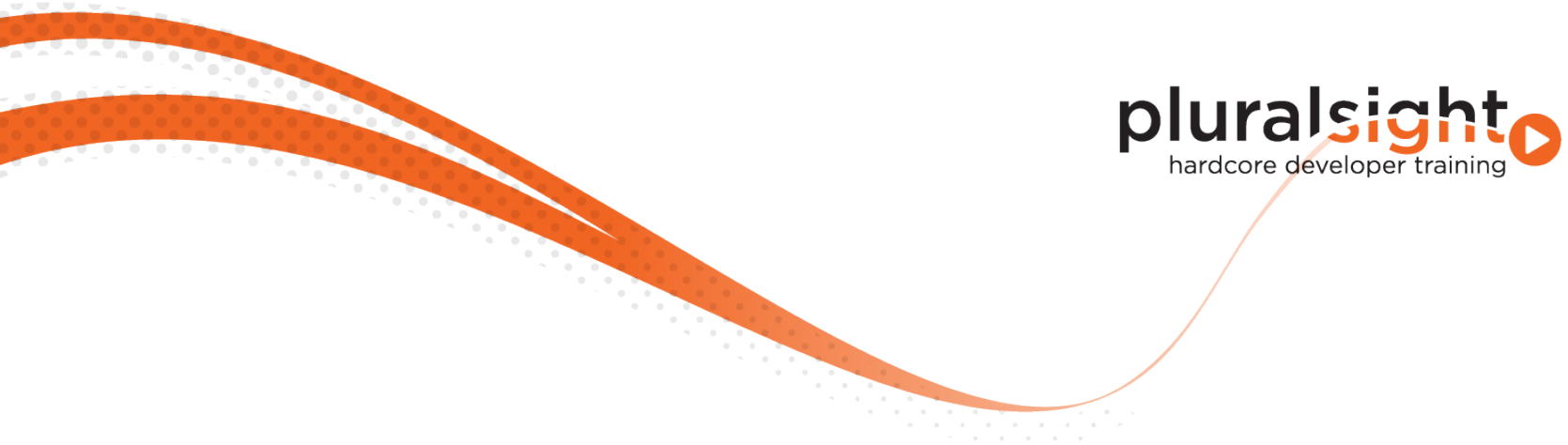
Execution Model and Types of Parallelism

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:**
  - *Execution model*
  - *Tasks vs. Threads vs. Cores*
  - *Types of parallelism:*
    - *data, task, dataflow, embarrassingly parallel*
  - *TPL support:*
    - *Parallel.For, .ForEach, .Invoke, and more*

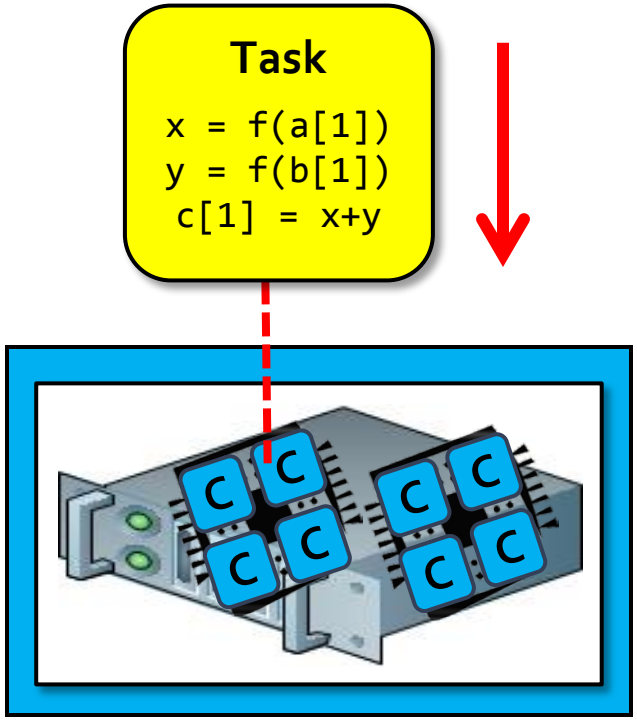
Tasks, tasks, tasks

- Our job as developers is to express work as tasks
- .NET's job is to execute these tasks intelligently

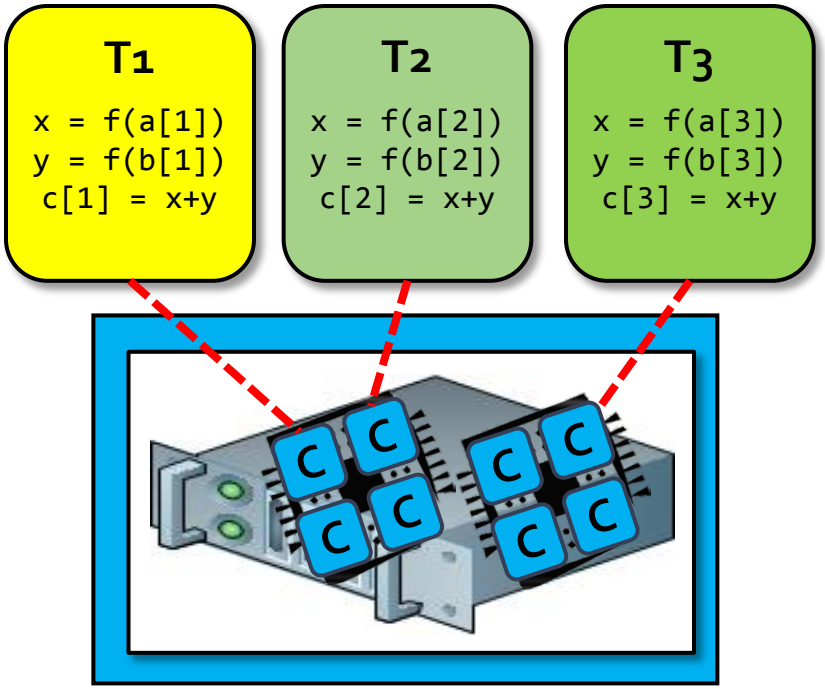
- Let's drill-down on the task execution model in .NET 4…

Execution model

- A task executes sequentially:



- Multiple tasks may execute in parallel:



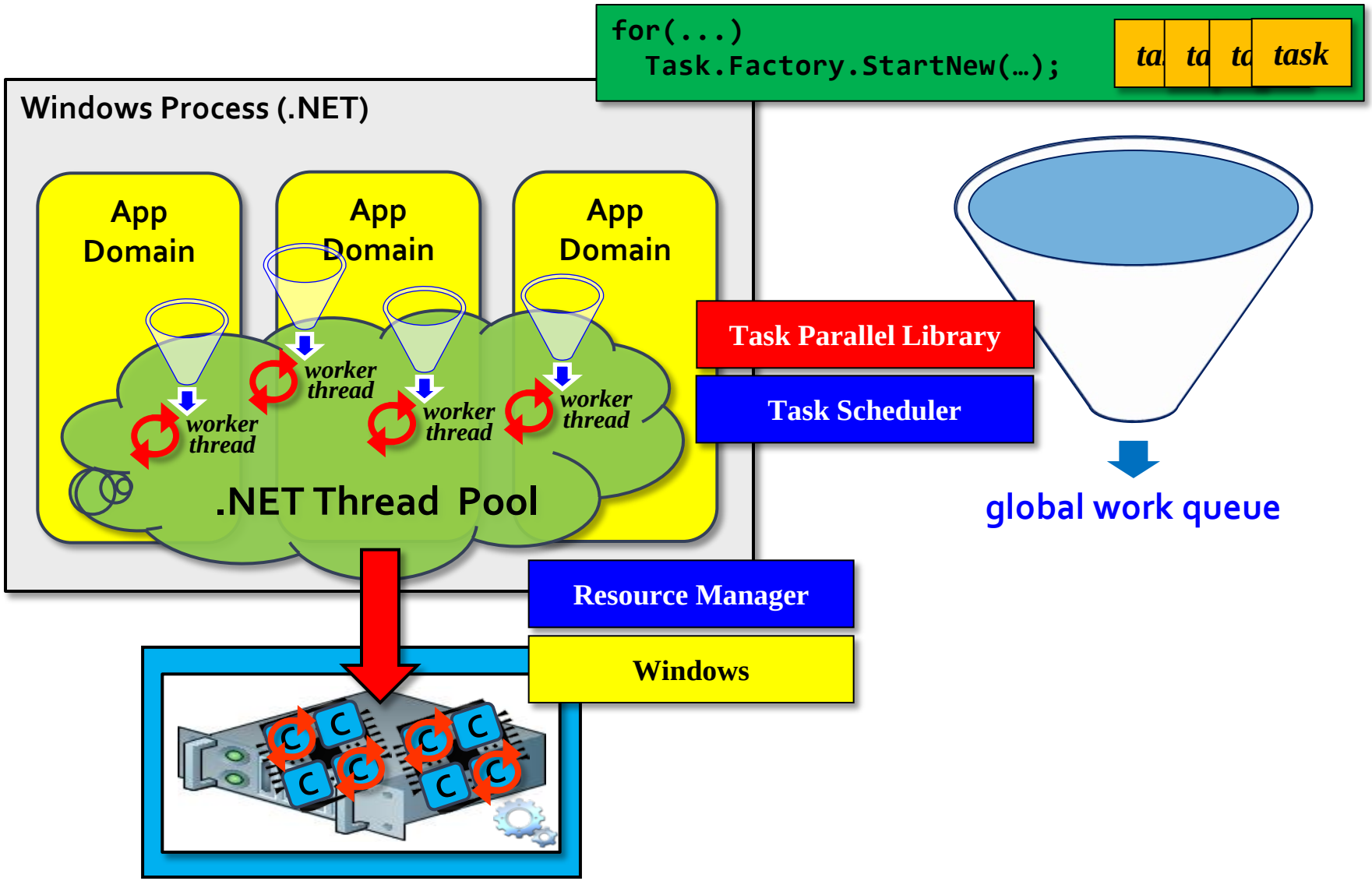
Task granularity

- **Tasks are lightweight objects**
  - *but there's still a cost associated with creating & manipulating*
- **What is minimum granularity to offset this cost?**

```
Task t = Task.Factory.StartNew( () =>
  {
    .
    .
    .
  }
);
```

Task execution should take at least 200-300 hundred CPU cycles…

Tasks vs. Threads vs. Cores



Want to customize task scheduler?

- Add task priorities?
- Use threads from your own thread pool?

- Supply your own task scheduler…

```
class MyTaskScheduler : TaskScheduler
{
    .
    .
    .
}
```

```
TaskScheduler myTS = new MyTaskScheduler();
TaskFactory   myTF = new TaskFactory(myTS);

Task t = myTF.StartNew( () => {...} );
```

For more details and examples, see:
http://msdn.microsoft.com/en-us/library/ee789351.aspx
http://code.msdn.microsoft.com/ParExtSamples

Default task scheduler — observations…

- **A single, global work queue does not scale**

  ==> ***local queues reduce contention***

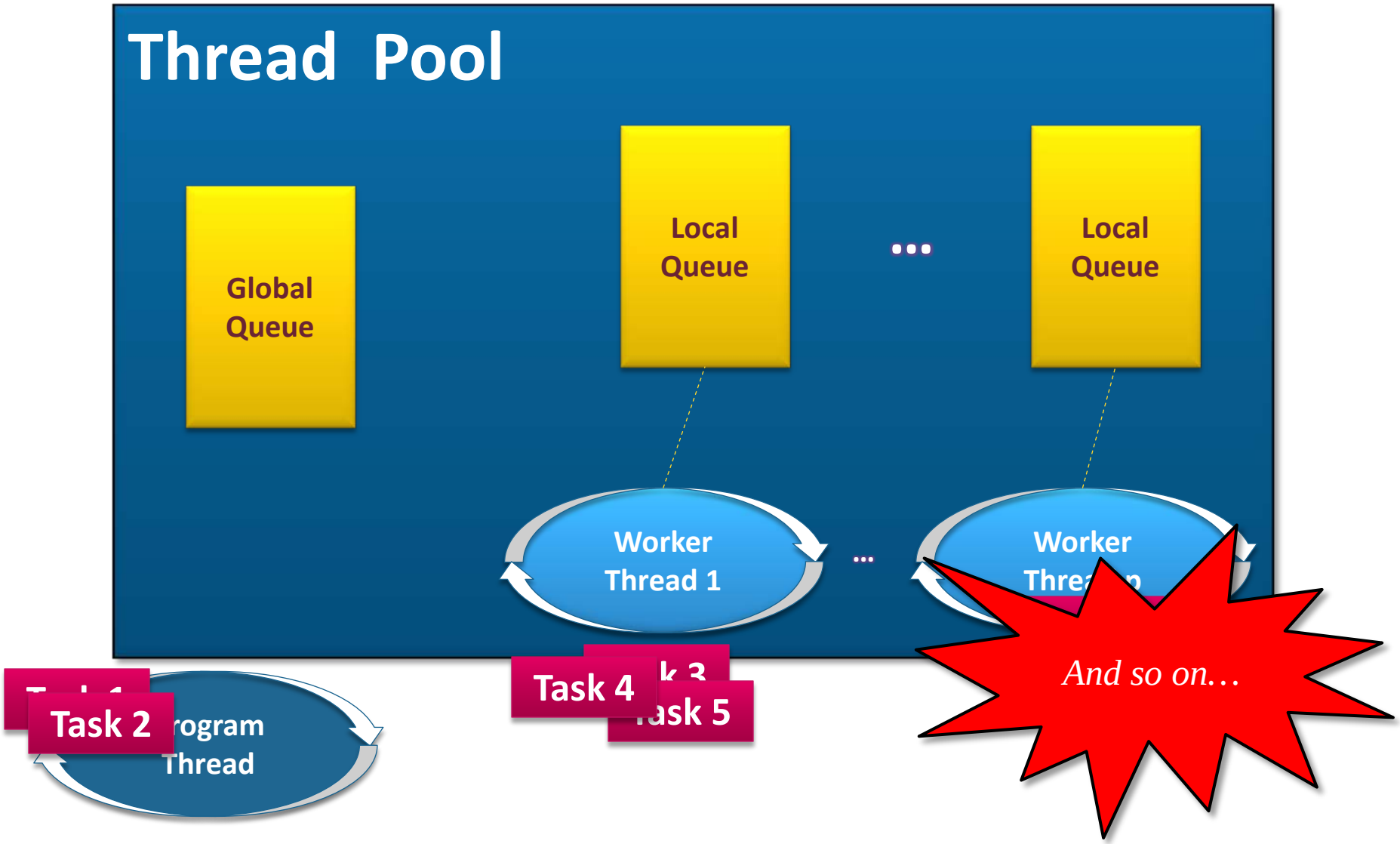
- **Responds dynamically to demand**
  - *a thread is dedicated to a particular tasks until task completes – fast execution*
  - *if task code blocks & waits, its thread blocks & waits – not available to others*

  ==> ***creates new worker threads to maintain task completion rate***

- **Dynamically balances the workload**

  ==> ***"work-stealing" of tasks from other queues for load-balancing***

Work-stealing



Task scheduler assumptions

(1) *Tasks are short-lived — at most 1-2 seconds.*

(2) *Execution order doesn't matter — random order of execution is okay.*

What if execution order matters?

- If you want to execute tasks in order of creation…
- … create tasks with fairness option:

```
Task.Factory.StartNew(
    () => { /* long running code */ },
    TaskCreationOptions.Fairness
);
```

.NET adds task to global queue, never to worker's local queue…

Guarantees that tasks will *start* in order they are created, but tasks could finish in *any* order

Slight increase in cost — global queue carries higher synchronization overhead.

What if task is long-running?

- Where "long" ==> greater than a few seconds…
- … create task with long-running option:

```
Task.Factory.StartNew(
    () => { /* long running code */ },
    TaskCreationOptions.LongRunning
);
```

.NET creates a non-worker pool thread, dedicates thread to this task…

Beware of cost — roughly 200,000 cycles to create, 100,000 to retire, and every context switch is 6,000-8,000. Plus memory cost for stack space.

DEMO

- Suppose we have 100 long-running computations to perform.

What happens if we start 100 tasks with no creation options?

Every couple seconds a new thread is created and a task started…

What happens if we start 100 tasks with long-running option?

Very quickly, 100 threads are created and 100 tasks started…

If computations are CPU-intensive, system becomes over-subscribed with too many threads competing for cores…

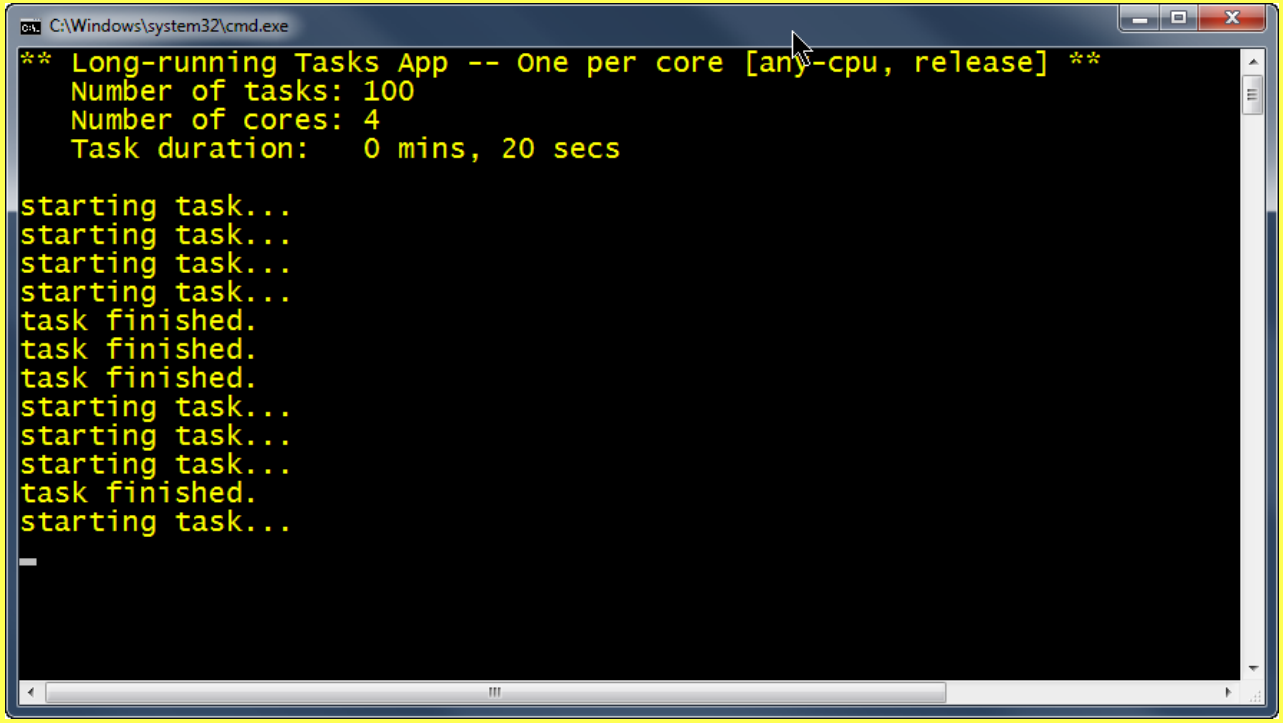
Solution?

- **It's okay to create hundreds or thousands of tasks…**
- **It's *not* okay if those tasks are long-running, CPU-intensive tasks**
  - *you have too many threads competing for cores*
- **Solution?**
  - *Start just T tasks — one per core*
  - *As tasks finish, start new ones…*

```
int numCores = System.Environment.ProcessorCount;
for (int i=0; i<numCores; ++i)
  tasks.Add( Task.Factory.StartNew( () => {...} );

while (tasks.Count > 0)   // wait for tasks to finish:
{
   int index = Task.WaitAny( tasks.ToArray() );
   tasks.RemoveAt(index);
   if (still work to do)
     tasks.Add( Task.Factory.StartNew( () => {...} );
}
```

DEMO

- Correctly executing 100 long-running computations…



Aside — here's an even better way…

- Use parallel for loop with option to constrain amount of parallelism
  - *A teaser for what's coming next :-)*

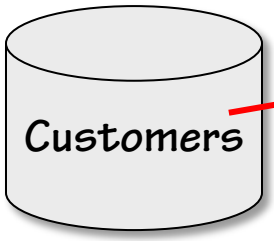
```
using System.Threading.Tasks;

var options = new ParallelOptions();
options.MaxDegreeOfParallelism = System.Environment.ProcessorCount;


Parallel.For(0, N, options, (i) => { /*long-running computation*/ });
```

Common types of parallelism…

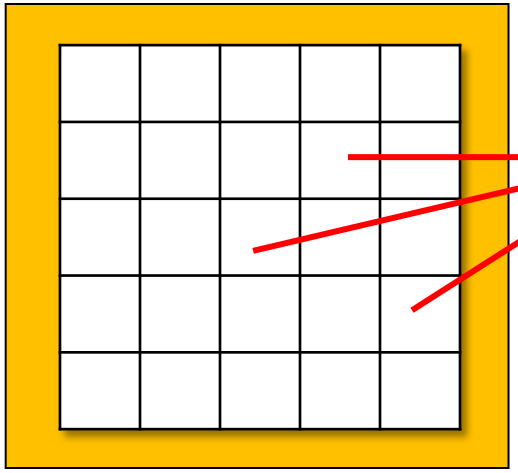
- Data
- Task
- Dataflow
- Embarrassingly parallel

(1) Data parallelism

- **Def**: the same operation executed across different data.



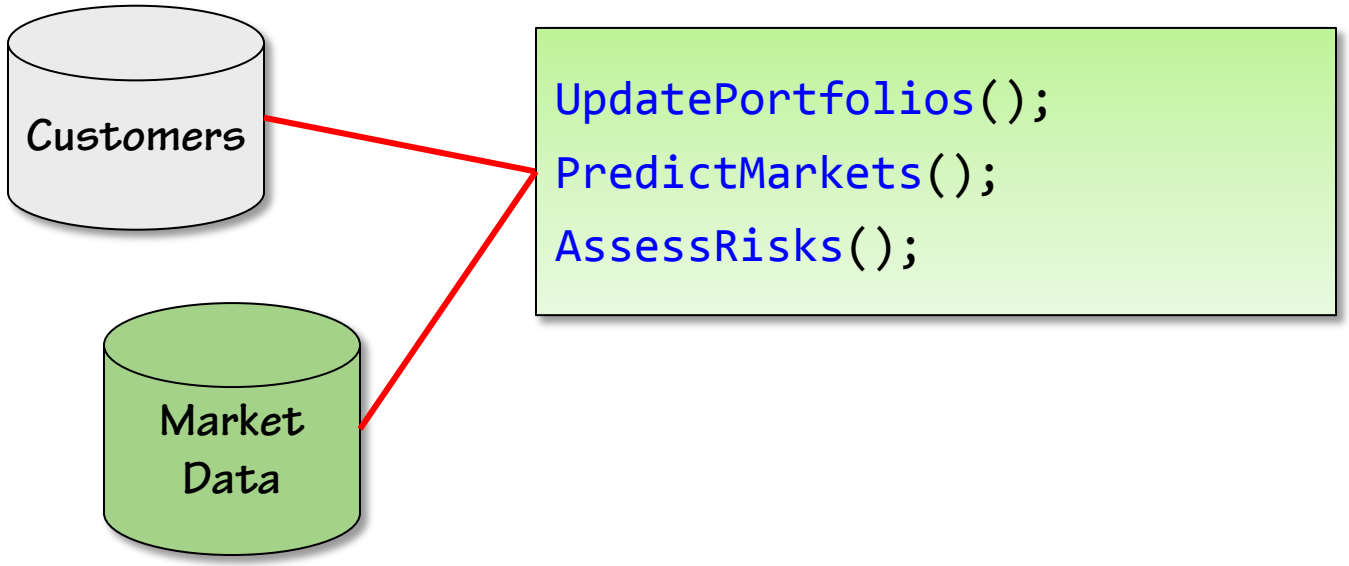

```
foreach(Customer c in Customers)
  UpdatePortfolio(c);
```



```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    A[i,j] = sqrt(c * A[i,j]);
```

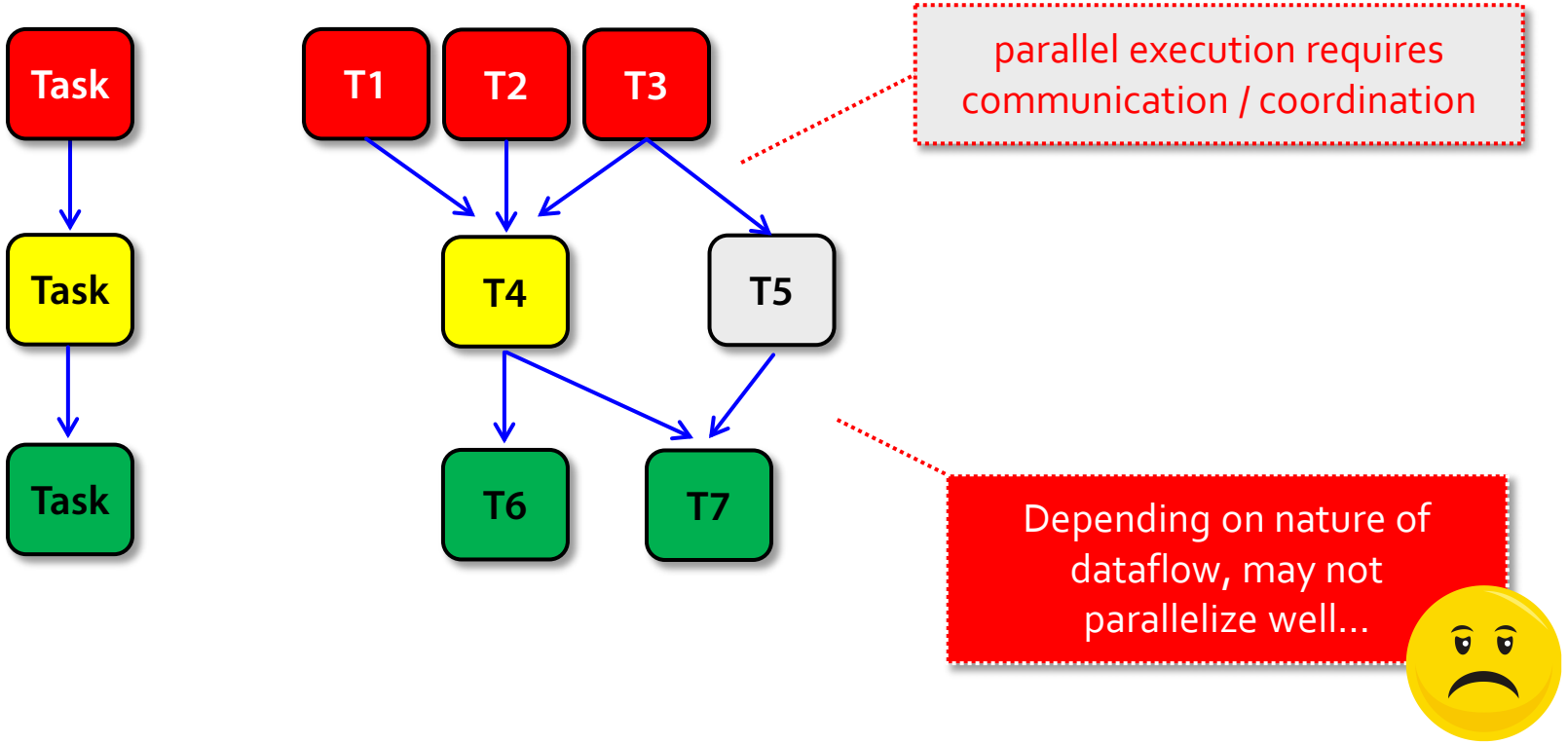
(2) Task parallelism

- **Def:** <u>different</u> operations executed across the same or different data.



(3) Dataflow parallelism

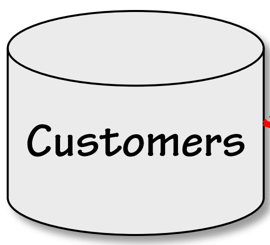
- **Def: when operations depend on one another.**
  - *data "flows" from one operation to another*



(4) Embarrassingly Parallel

- **Def**: **a problem is embarrassingly parallel if the computations are <u>entirely independent</u> of one another.**

*==> *no* data flow*



Original code

```
foreach(Customer c in Customers)
  UpdatePortfolio(c);
```

```
Parallel.Foreach(Customers, (c) =>
  {
      UpdatePortfolio(c);
  });
```

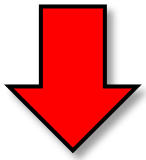
Also referred to as "delightfully parallel", best type of parallelism!

TPL support

- **Dataflow:**
  - *next lecture "Designs and Patterns"*
- **Data parallelism:**
  - `Parallel.For`
  - `Parallel.Foreach`
- **Task parallelism:**
  - `Parallel.Invoke`

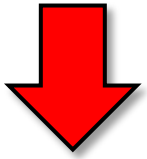
Parallel class

```
for(i=0; i<N; i++)
  DoWork(i);
```



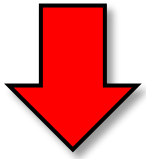
```
Parallel.For(0, N,
  (i) =>
  {
    DoWork(i);
  }
);
```

```
foreach(var e in ds)
  DoWork(e);
```



```
Parallel.ForEach(ds,
  (e) =>
  {
    DoWork(e);
  }
);
```

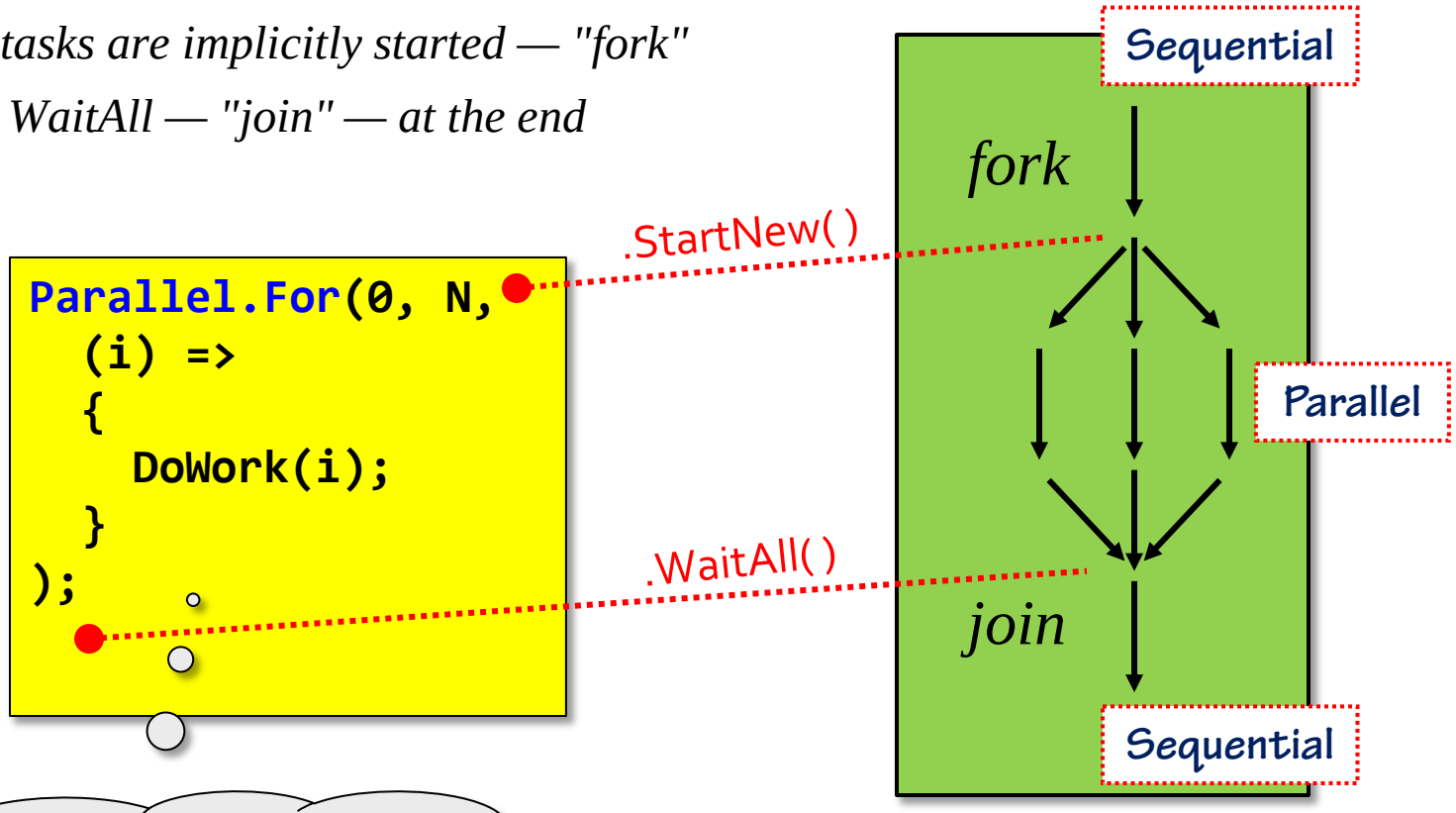
```
Task1();
Task2();
Task3();
```



```
Parallel.Invoke(
  () => Task1(),
  () => Task2(),
  () => Task3()
);
```

Structured parallelism

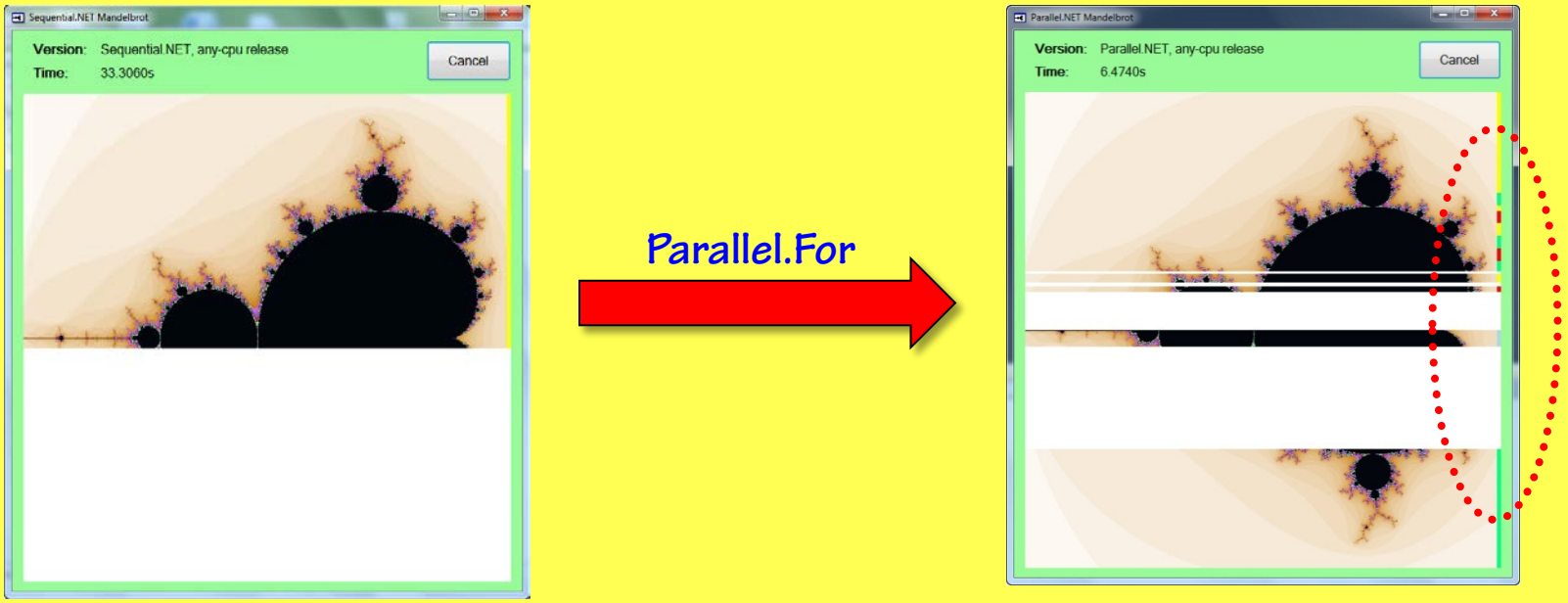
- **Parallel class uses "*fork-join*" pattern**
  - *a set of tasks are implicitly started — "fork"*
  - *implicit WaitAll — "join" — at the end*



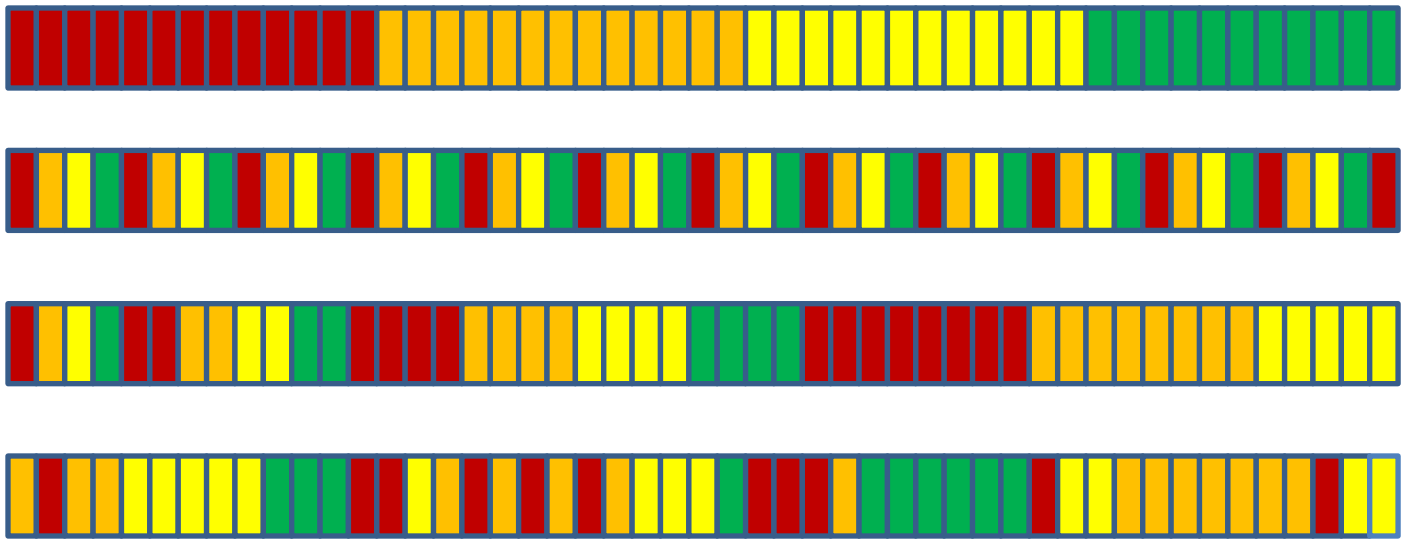
When in doubt, use this pattern —
easiest to understand & apply…

DEMO

- Exploiting data parallelism in Mandelbrot app



Data partitioning

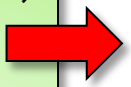


- **Parallel.For & .Foreach "chunk" the data for good load-balancing**
  - *chunks start small, then grow in size*
  - *assumes loop body does significant computation to offset cost of small chunks*
  - *not so good for simple loop bodies --- **use something else?***

Custom data partitioning

- Parallel.Foreach supports custom partitioners
- Example:
  - *partitioning simpler loop bodies for efficient parallelization*

```
for (i=0; i<N; ++i)
{ /*simple*/ }
```



```
using System.Collections.Concurrent;
using System.Threading.Tasks;

Parallel.Foreach( Partitioner.Create(0, N), /*range-like*/
  (range) =>
  {
       for (int i=range.Item1; i<range.Item2; ++i)
       { /*simple*/ }
  });
```

For more details and examples, see:

http://msdn.microsoft.com/en-us/library/dd997411.aspx
http://www.drdobbs.com/go-parallel/article/224600406
http://code.msdn.microsoft.com/ParExtSamples

Exception handling

- How are exceptions handled by the Parallel class?
- If an unhandled exception is thrown by .For / .Foreach / .Invoke, then
  - *tasks allowed to finish their current iteration / work*
  - *exception(s) are deferred and re-thrown as AggregateException*

```
try
{
  Parallel.For(0, N, (i) =>
    {
      .
      .
      .
    }
  );
}
catch(AggregateException ae)
{ /*handle*/ }
```

throw new ApplicationException("oops");

Break out of loop — from *within*

- You can break out of Parallel.For / .Foreach
- Task(s) finish current iteration, then exit
  - *call **Stop** if you want to stop loop ASAP*
  - *call **Break** if you want earlier iterations to complete (0..M-1)*

```
ParallelLoopResult lr = Parallel.For(0, N, (i, loopControl) =>
  {
    if (some condition becomes true)
      loopControl.Stop();
    else
      DoWork(i);
  }
);

if (!lr.IsCompleted)
  // loop did not run to completion:
```

Cancel loop — from *outside*

- Task(s) finish current iteration, then exit
  - ***beware**: an exception is thrown!*

```
void StartButton_Click(…)
{
  var cts =
    new CancellationTokenSource();

  Task.Factory.StartNew(
    () => { DoLoop(cts); });
}

void CancelButton_Click(…)
{
  cts.Cancel();   // stop loop:
}
```

```
void DoLoop(CancellationTokenSource cts)
{
  var options = new ParallelOptions();
  options.CancellationToken = cts.Token;

  try {
    Parallel.For(0, N, options, (i) =>
    {
      DoWork(i);
    });
  }
  catch(OperationCanceledException oce) { /*loop canceled*/  }
  catch(AggregateException ae)          { /*something else*/ }
}
```

DEMO

- Mandelbrot app with cancellation…

Summary

- **Your job is to create tasks**
- **.NET's job is to execute those tasks efficiently**

- **Task Parallel Library provides a versatile, efficient execution engine**
  - *load-balancing task scheduler*
  - *intelligent resource manager*

- **Identify the type of parallelism in your application:**
  - Data, Task, Dataflow, Embarrassingly Parallel
- **Use higher-level abstractions where appropriate:**
  - `Parallel.For, .Foreach, .Invoke`

References

- **Microsoft's main site for all things parallel:**
  - http://msdn.microsoft.com/concurrency
- **MSDN technical documentation:**
  - http://tinyurl.com/pp-on-msdn

- **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