Lightweight Concurrent Tasks

Lecture 1.d

Acknowledgements

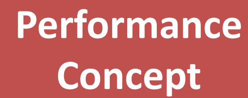
- Authored by
  - Thomas Ball, MSR Redmond

What We've Seen So Far

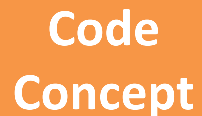
- Parallel.For/ForEach
  - data parallelism over integer range or IEnumerable
  - automatic generation of tasks
  - dynamic partitioning and load balancing

- Parallel.For doesn't address many problems
  - Irregular matrix algorithms
  - Algorithms over trees, graphs
  - …

Concepts



- Wavefront computation



- Task
  - Task Status
  - Parent/child relationship
  - Task Result
  - Task Continuation

Levenshtein Edit Distance

- Find minimal distance between strings s1 and s2
  - via character insertions, deletions, and substitutions
- Example of dynamic programming
  - break down problems into smaller problems
  - cache (“memoize”) results of subproblems
- Other examples
  - longest common subsequence, matrix-chain multiplication, sequence alignment

Edit Distance and Wavefront

i>0 and j>0

```
dist[i, j] =
  (s1[i - 1] == s2[j - 1])
    ? dist[i - 1, j - 1]
    : 1 + min(dist[i - 1, j],
              min(dist[i, j - 1],
                  dist[i - 1, j - 1]));
```

dist

s1

s2

		S	A	T	U	R	D	A	Y
	0	1	2	3	4	5	6	7	8
S	1								
U	2								
N	3								
D	4								
A	5								
Y	6								

(i-1,j-1)	(i,j-1)
(i-1,j)	**(i,j)**

CS2

Edit Distance and Wavefront

		S	A	T	U	R	D	A	Y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
U	2	1	1	2	2	3	4	5	6
N	3	2	2	2	3	3	4	5	6
D	4	3	3	3	3	4	3	4	5
A	5	4	3	4	4	4	4	3	4
Y	6	5	4	4	5	5	5	4	3

Slide 7

CS2 Maybe a slide showing why Parallel.For is not quite what we want?
Caitlin Sadowski, 7/8/2010

TaskFactory and Task

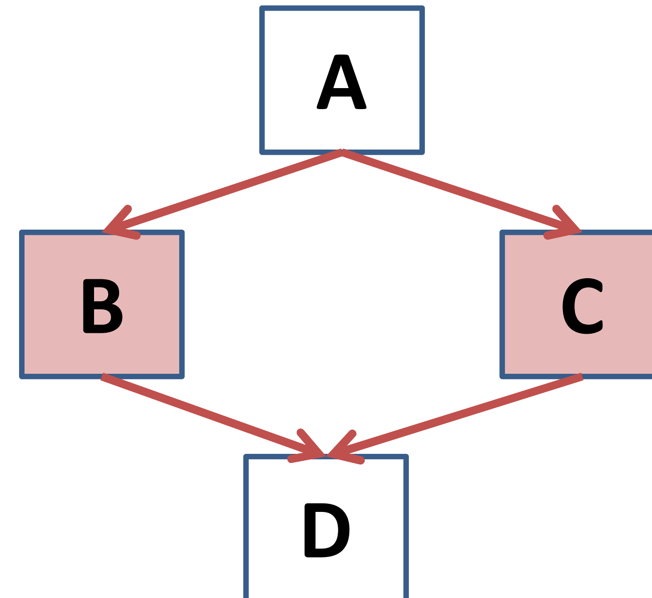
- TaskFactory
  - `Task StartNew(Action)`
  - `Task ContinueWhenAll(Task[], Action<Task[]>)`

- Task
  - `static TaskFactory Factory`
  - `Task ContinueWith(Action<Task>)`
  - `void Wait()`

Wavefront on 2x2 Matrix

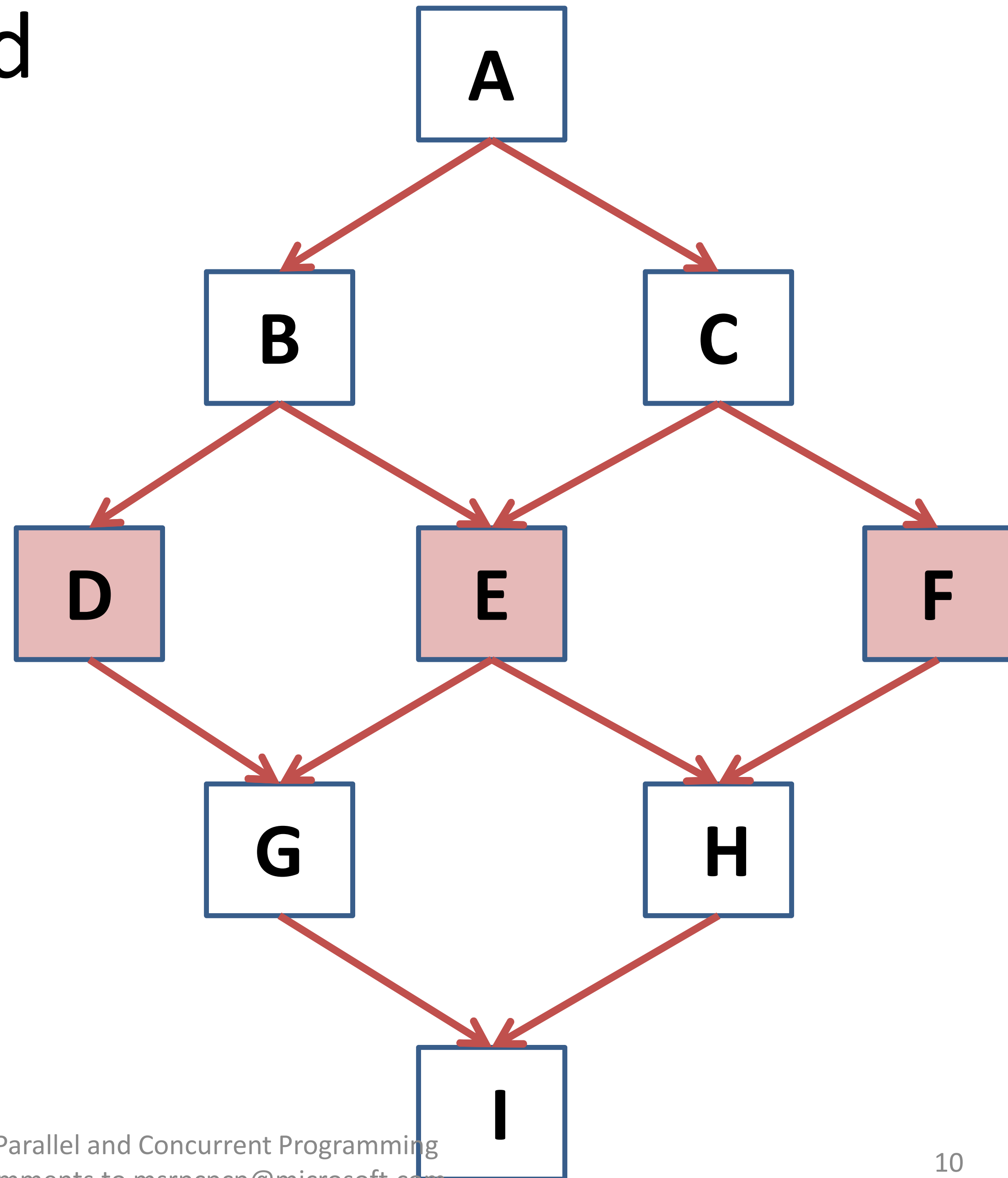
A	C
B	D

```
Task A = Task.Factory.StartNew(actionA);
Task B = A.ContinueWith(actionB);
Task C = A.ContinueWith(actionC);
Task D = Task.Factory.ContinueWhenAll(
      new Task[2] { B, C },
      actionD);
D.Wait();
```



3x3 Matrix and Task Graph

A	C	**F**
B	**E**	H
D	G	I



General Wavefront Algorithm

```
void Wavefront(
  int numRows,
  int numColumns,
  Action<int,int> processCell)
```

ParallelAlgorithms_Wavefront.cs

Edit Distance with Parallel Wavefront

EditDistance.cs

```
// allocation of dist array
Wavefront(numRows, numColumns,
   (int i, int j ) =>
   {
      if (i == 0)
         dist[i,j] = j;
      else if (j == 0)
         dist[i,j] = i;
      else
         dist[i, j] = (s1[i - 1] == s2[j - 1])
            ? dist[i - 1, j - 1]
            : 1 + Math.Min(dist[i - 1, j],

                  Math.Min(dist[i, j - 1],

                              dist[i - 1, j - 1]));
   }
);
```

CS3

Performance of Parallel Wavefront

- Much worse than sequential!
- One Task per entry of the distance matrix
  - Not enough computation per Task
  - Coordination cost of Task allocation, management, and synchronization dominates

Slide 13

CS3 Performance on what exactly is worse than sequential? Does code in previous slide use tasks?
Caitlin Sadowski, 7/8/2010

Blocking For More Work per Task

		S	A	T	U	R	D	A	Y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
U	2	1	1	2	2	3	4	5	6
N	3	2	2	2	3	3	4	5	6
D	4	3	3	3	3	4	3	4	5
A	5	4	3	4	4	4	4	3	4
Y	6	5	4	4	5	5	5	4	3

BlockedWavefront

```
static void BlockedWavefront(
    int numRows, int numColumns,
    int numBlocksPerRow, int numBlocksPerColumn,
    Action<int, int, int, int> processBlock)
```

```
processBlock(start_i, end_i, start_j, end_j);
```

ParallelAlgorithms_Wavefront.cs

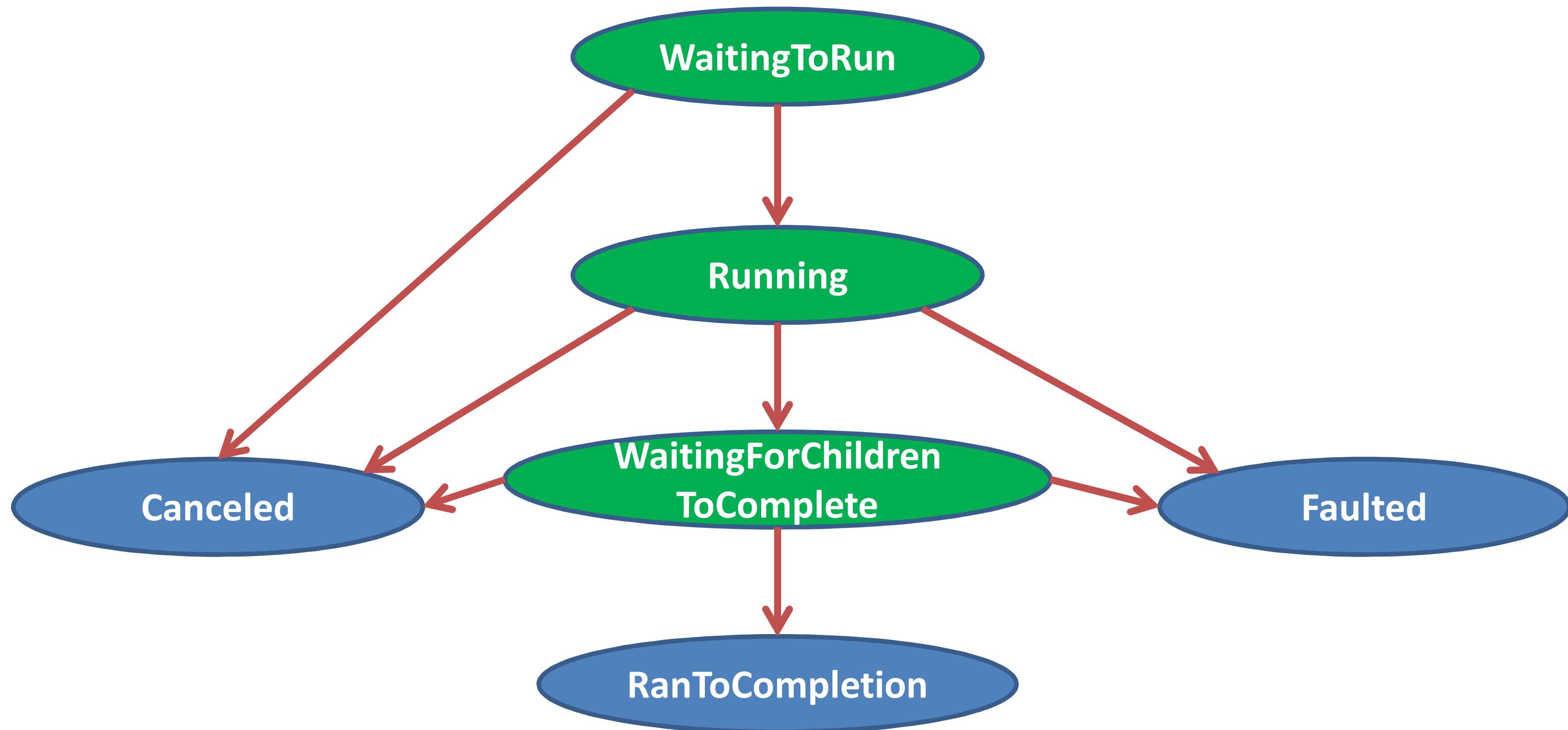
BlockedWavefront

```
// Compute the size of each block.
int rowBlockSize = numRows / numBlocksPerRow;
int columnBlockSize = numColumns / numBlocksPerColumn;
Wavefront(numBlocksPerRow, numBlocksPerColumn, (row, column) =>
{
    int start_i = row * rowBlockSize;
    int end_i = row < numBlocksPerRow - 1 ?
        start_i + rowBlockSize : numRows;
    int start_j = column * columnBlockSize;
    int end_j = column < numBlocksPerColumn - 1 ?
        start_j + columnBlockSize : numColumns;
    processBlock(start_i, end_i, start_j, end_j);
});
```

Tasks, TaskScheduler, ThreadPool

- **Task** represents an asynchronous operation
- **TaskScheduler**
  - is responsible for scheduling of **Tasks**
  - defaults to the .NET 4 **ThreadPool**
  - *more on scheduling in Unit 5*
- **ThreadPool**
  - effectively manages a set of **Threads**
  - *More on thread pool and threads in Unit 5*

Task Status (partial)



CS5

Task Status (partial list)

WaitingToRun	The task is scheduled for execution but has not yet begun running
Running	The task is running but has not yet completed
WaitingForChildrenToComplete	The task has finished executing and is waiting for attached child tasks to complete
RanToCompletion	The task completed execution successfully
Canceled	The task acknowledged cancellation by throwing an OperationCanceledException
Faulted	The task completed due to an unhandled exception

CS5 What's the big picture? How do I query & use these states?
Caitlin Sadowski, 7/8/2010

Nested Tasks

```
static void SimpleNestedTask() {
    var parent = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Outer task executing.");
        var child = Task.Factory.StartNew(() =>
        {
            Console.WriteLine("Nested task starting.");
            Thread.SpinWait(500000);
            Console.WriteLine("Nested task completing.");
        });
    });
    parent.Wait();
    Console.WriteLine("Outer has completed.");
}
```

TaskExamples.cs

Child Tasks

```
static void SimpleNestedTask() {
    var parent = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Outer task executing.");
        var child = Task.Factory.StartNew(() =>
        {
            Console.WriteLine("Nested task starting.");
            Thread.SpinWait(500000);
            Console.WriteLine("Nested task completing.");
        },
            TaskCreationOptions.AttachedToParent);
        );
    });
    parent.Wait();
    Console.WriteLine("Outer has completed.");
}
```

Relationship Between Nested and Child Tasks?

“Nested” doesn’t imply “Child”

“Child” doesn’t imply “Nested”

Task Results (Futures)

```
var cts = new CancellationTokenSource();
Task<int> dataForThefuture = Task.Factory.StartNew(
    () => ComputeSomeResult(), cts.Token);

...

// This will return the value immediately if the Task has already
// completed, or will wait for the result to be available if it’s
// not yet completed.
int result = dataForTheFuture.Result;
```

Task Results (Futures)

```
var cts = new CancellationTokenSource();
Task<int> dataForThefuture = Task.Factory.StartNew(
    () => ComputeSomeResult(), cts.Token);

...

// Cancel it and make sure we are made aware of any exceptions
// that occurred.
cts.Cancel();
dataForTheFuture.ContinueWith(t => LogException(dataForTheFuture),
    TaskContinuationOptions.OnlyOnFaulted);
```

Tasks and Exceptions

- If a nested task throws an exception, it must be observed or handled directly in the outer task just as with any non-nested task
- If a child task C throws an exception, the exception is automatically propagated to the parent task (via `C.Wait()`or `C.Result`)

CS4

Difference between Nest and Child Tasks

Category	Nested Tasks	Attached Child Tasks
Outer task (parent) waits for inner tasks to complete.	No	Yes
Parent propagates exceptions thrown by children (inner tasks).	No	Yes
Status of parent (outer task) dependent on status of child (inner task).	No	Yes

CS4 So, whats the big picture? What are some examples why nested vs. child makes more sense, and visa versa?
Caitlin Sadowski, 7/8/2010

CS6

TaskContinuationOptions (partial)

NotOnRanTo Completion	Continuation task should not be scheduled if its antecedent ran to completion.
NotOnFaulted	Continuation task should not be scheduled if its antecedent threw an unhandled exception
NotOnCanceled	Continuation task should not be scheduled if its antecedent was canceled
OnlyOnlyOn RanToCompletion, OnlyOnFaulted, OnlyOnCanceled	You can guess…

CS6 What about a couple high-level (e.g. with clear motivation) examples?
Caitlin Sadowski, 7/8/2010

Tasks and Cancellation

- Task cancellation is cooperative
- To cancel a parent and all its children in one request, pass the same token to all tasks
  - Parent cancellation doesn't imply child cancellation
  - When child task cancels itself, a TaskCanceledException is propagated to the joining task (see exceptions)

One Task per Element

```
static void MyParallelInvoke(params Action[] actions)
{
    var tasks = new Task[actions.Length];
    for (int i = 0; i < actions.Length; i++)
    {
        tasks[i] = Task.Factory.StartNew(actions[i]);
    }
    Task.WaitAll(tasks);
}
```

MyParallel.cs

One Task per Element, With Child Tasks

```
static T[] MyParallelInvoke<T>(params Func<T>[] functions)
{
    var results = new T[functions.Length];
    Task.Factory.StartNew(() =>
    {
        for (int i = 0; i < functions.Length; i++)
        {
            int cur = i;
            Task.Factory.StartNew(
                () => results[cur] = functions[cur](),
                TaskCreationOptions.AttachedToParent);
        }
    }).Wait();
    return results;
}
```

One Task per Element, With Parallel.For

```
static T[] MyParallelInvoke<T>(
    params Func<T>[] functions
) {
    T[] results = new T[functions.Length];
    Parallel.For(0, functions.Length, i =>
    {
        results[i] = functions[i]();
    });
    return results;
}
```

http://code.msdn.microsoft.com/ParExtSamples

- ParallelExtensionsExtras.csproj
  - Extensions/
    - TaskExtraExtensions.cs
    - TaskFactoryExtensions/
  - ParallelAlgorithms/

Parallel Programming with Microsoft .NET

- Chapter 3 (Parallel Tasks)
- Chapter 6 (Dynamic Task Parallelism)