Imperative Data Parallelism (Performance)

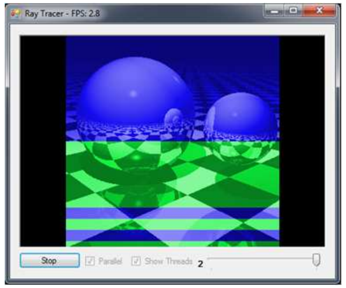
Unit 1.a

Acknowledgments

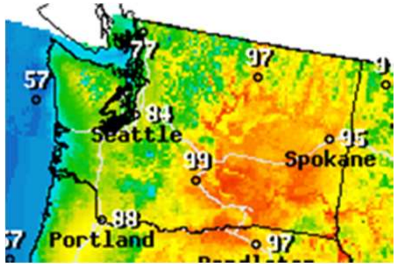
- Authored by
  - Thomas Ball, MSR Redmond
- This slide deck contains material courtesy of
  - Tim Harris, MSR Cambridge

Data Intensive Problems

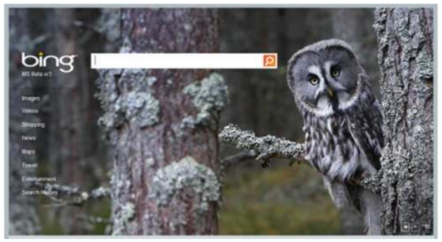
- Rendering in 3D
  - Ray tracing



- Modeling of complex physical systems
  - Weather prediction



- Analysis of massive datasets
  - Web search



tjb3

Data Parallelism

- Why: *speed-up*
- How (greatly simplified):
  - split work into independent similar pieces
  - execute pieces in parallel
  - collect results
- Simple example
  - Count number of words in a set of files

Slide 4

tjb3 This also could have a nice graphical representation, e.g. mapping files to cores.

I usually first describe data parallelism with a file processing example (e.g. counting words in a file); it is easy to see that you could process multiple files in parallel.

This may be a good time to introduce the DAG model of computation.

Tom Ball, 8/10/2010

Concepts

Performance Concept

- Amdahl's law
- Directed-acyclic graph (DAG) execution model
- Work and span

Code Concept

- C# lambdas
- Parallel.For

Correctness Concept

- Happens-before edges

Amdahl's law

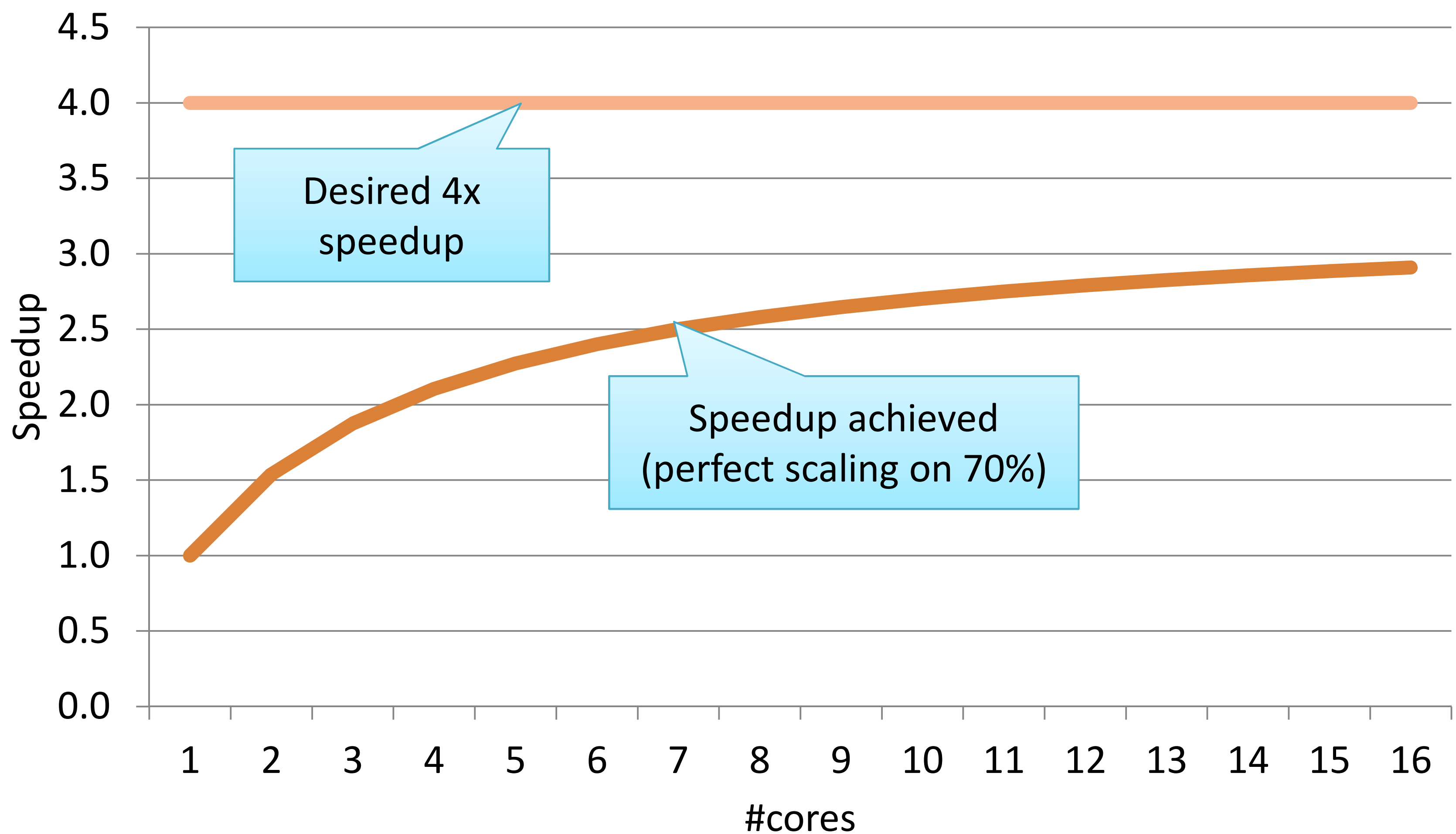
- Sorting takes 70% of the execution time of a sequential program
- You replace the sorting algorithm with one that scales perfectly on multi-core hardware
- How many cores do you need to get a 4x speed-up on the program?

Amdahl's law, $f = 70\%$

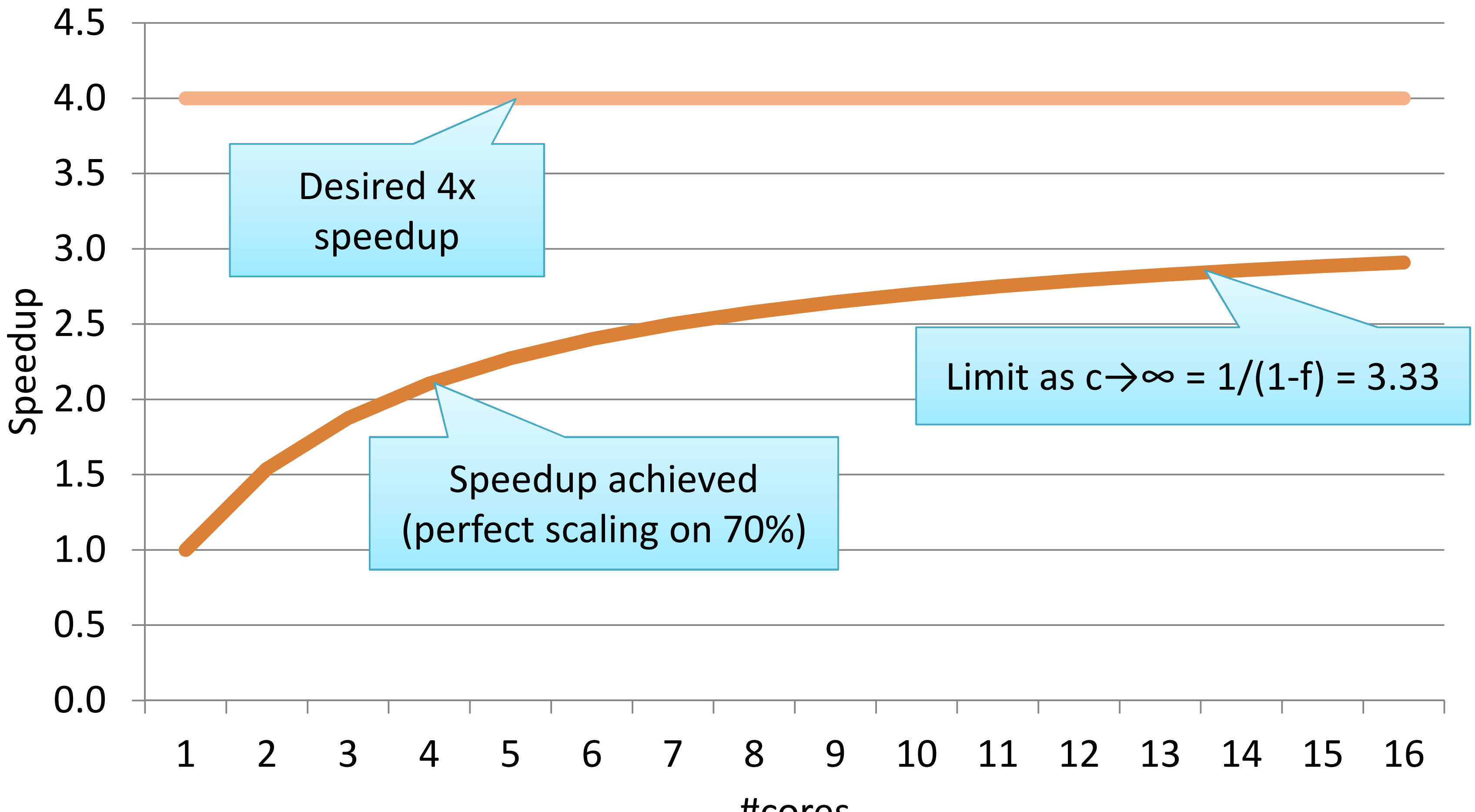
$$Speedup(f, c) = \frac{1}{(1-f) + \frac{f}{c}}$$

f	= the parallel portion of execution
$(1-f)$	= the sequential portion of execution
c	= number of cores used

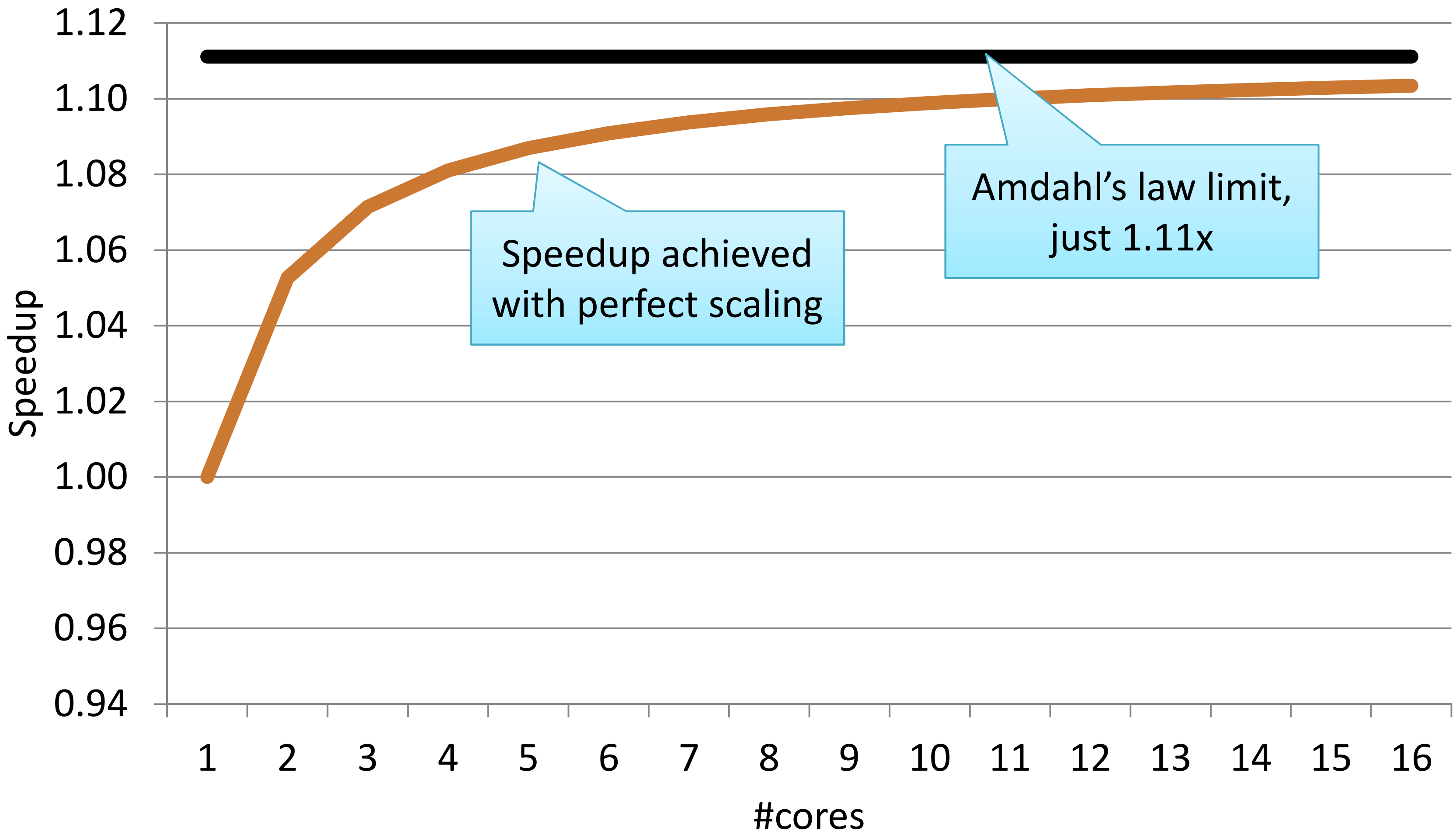
Amdahl's law, $f = 70\%$



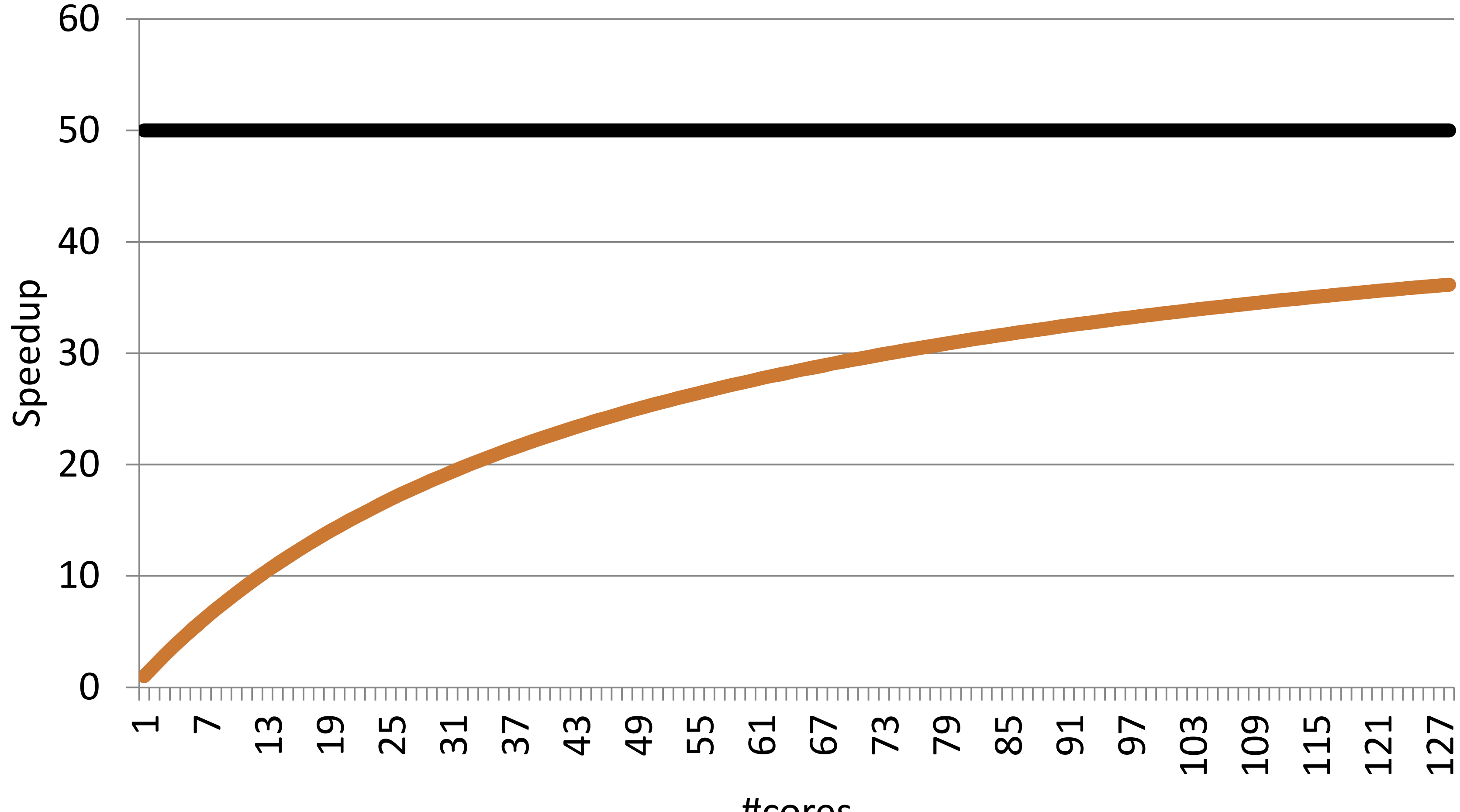
Amdahl's law, $f = 70\%$



Amdahl's law, $f = 10\%$



Amdahl's law, $f = 98\%$



Lesson

- Speedup is limited by <u>sequential</u> code

- Even a small percentage of <u>sequential</u> code can greatly limit potential speedup

CS4 CS6

On The Sunny Side of the Street: Gustafson's Law

Any sufficiently large problem can be parallelized effectively

$$Speedup(f, c) = fc + (1 - f)$$

f	= the parallel portion of execution
$(1 - f)$	= the sequential portion of execution
c	= number of cores used

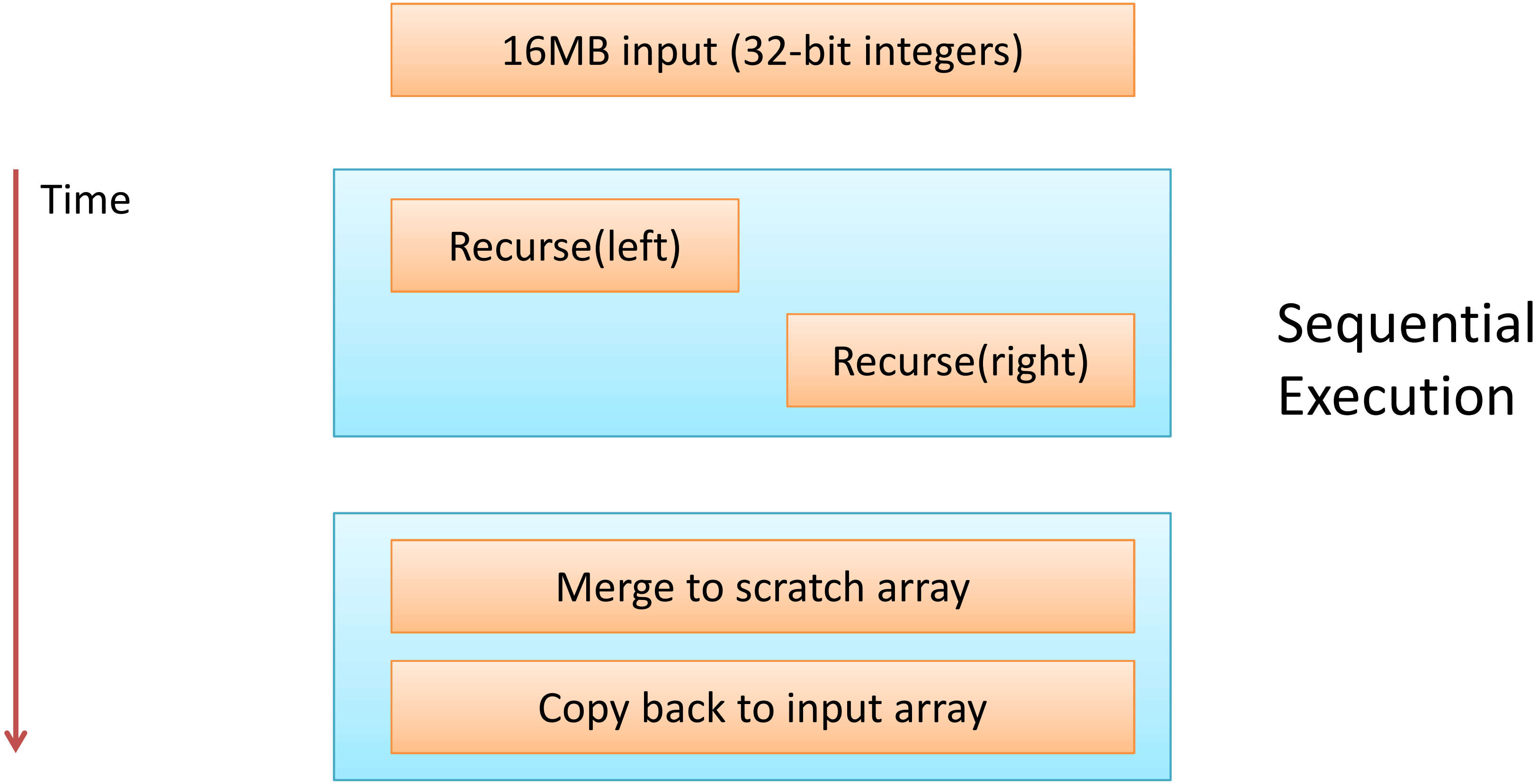
Key assumption: f increases as problem size increases

Slide 13

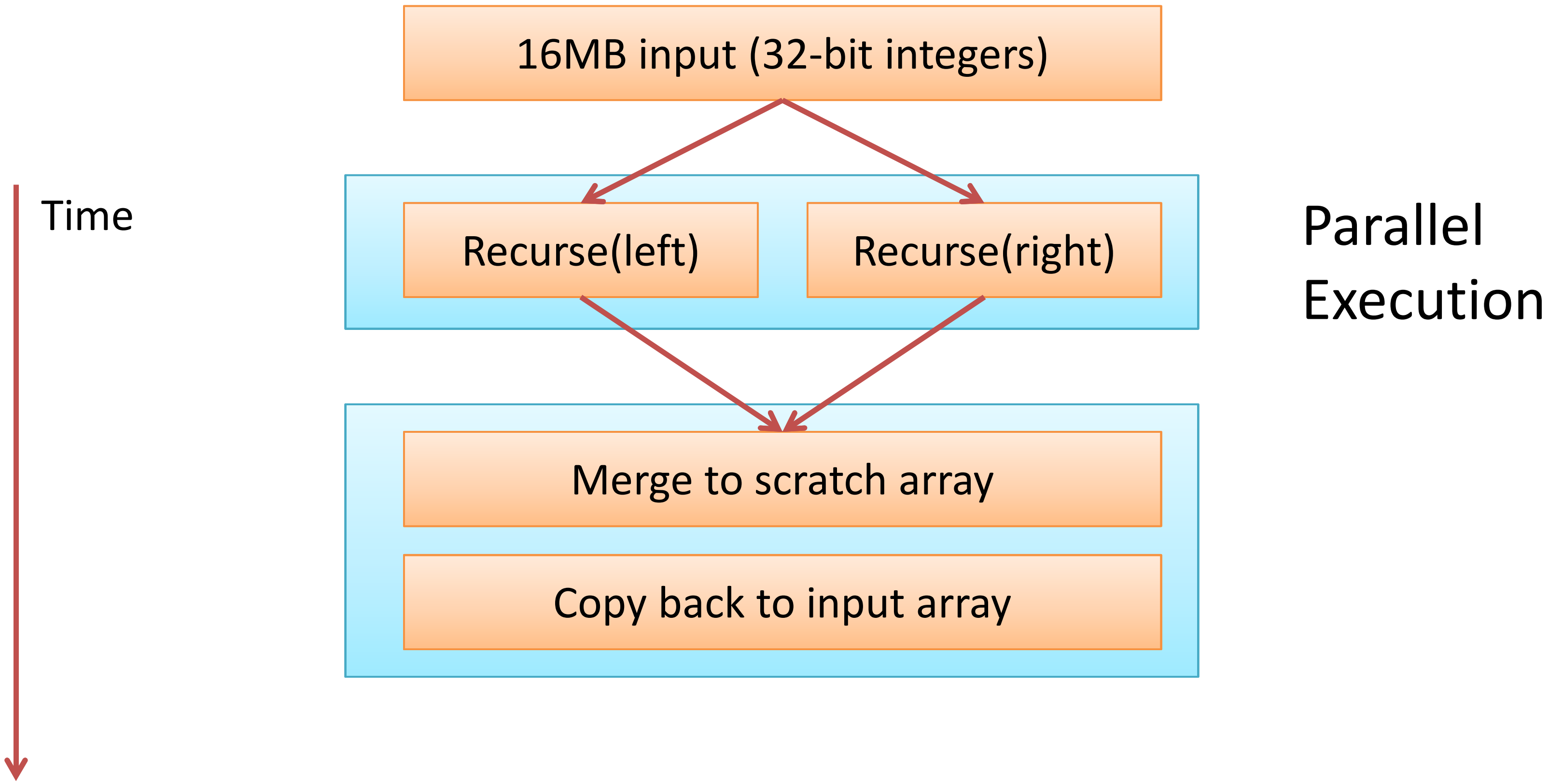
CS4 No examples?
Caitlin Sadowski, 7/7/2010

CS6 How does this sit with Amdahl's law? What is the take away message for this overall collection of slides?
Caitlin Sadowski, 7/7/2010

Sequential Merge Sort

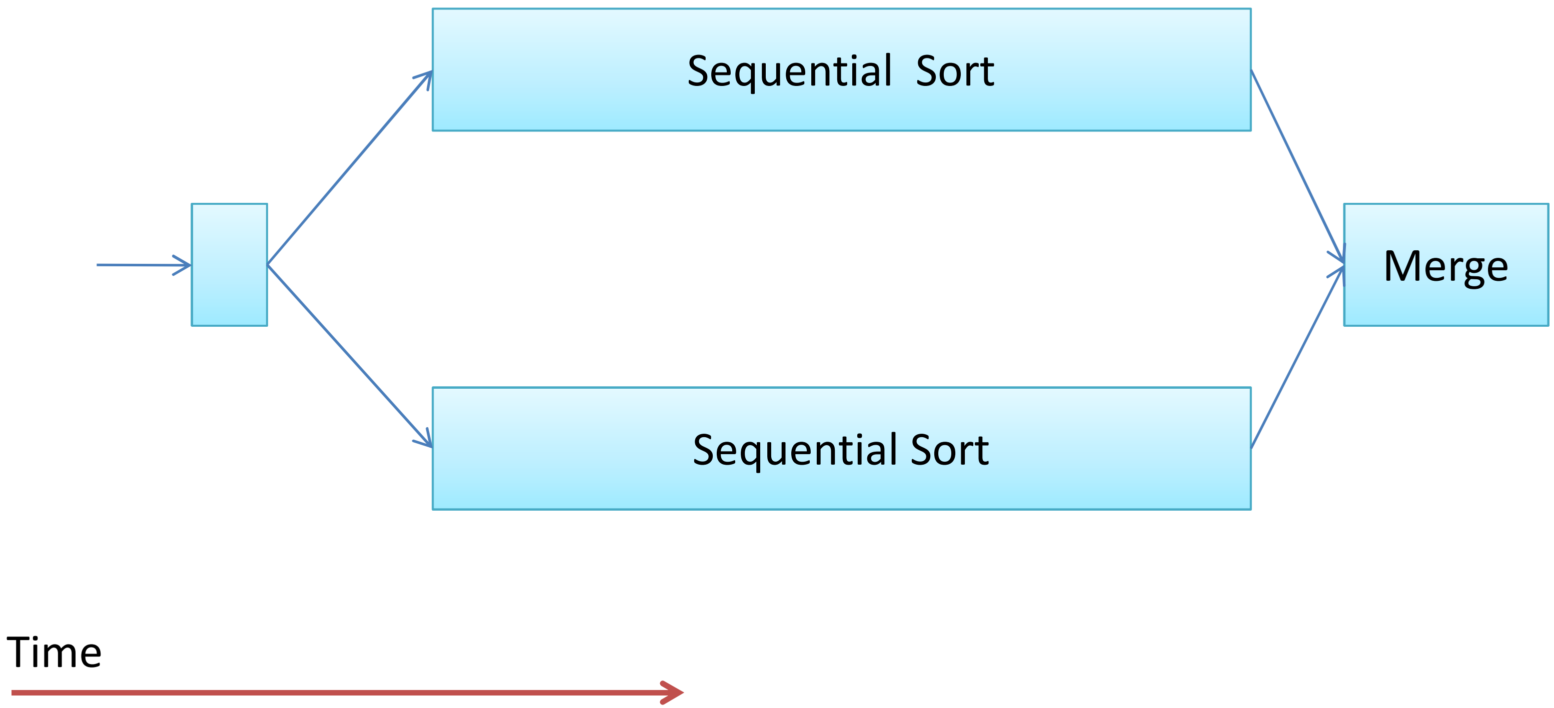


Parallel Merge Sort
(as Parallel Directed Acyclic Graph)



tjb4

Parallel DAG for Merge Sort (2-core)

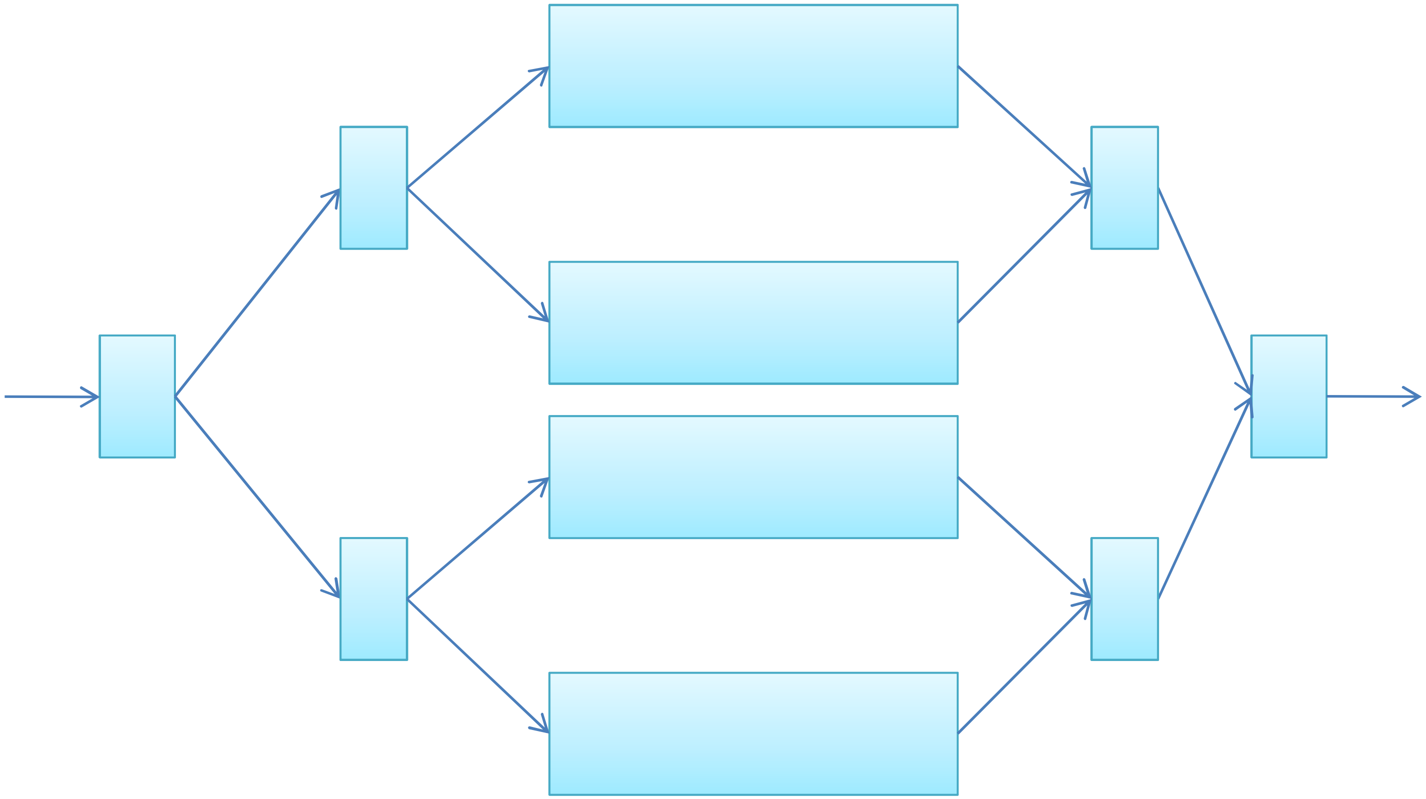


Slide 16

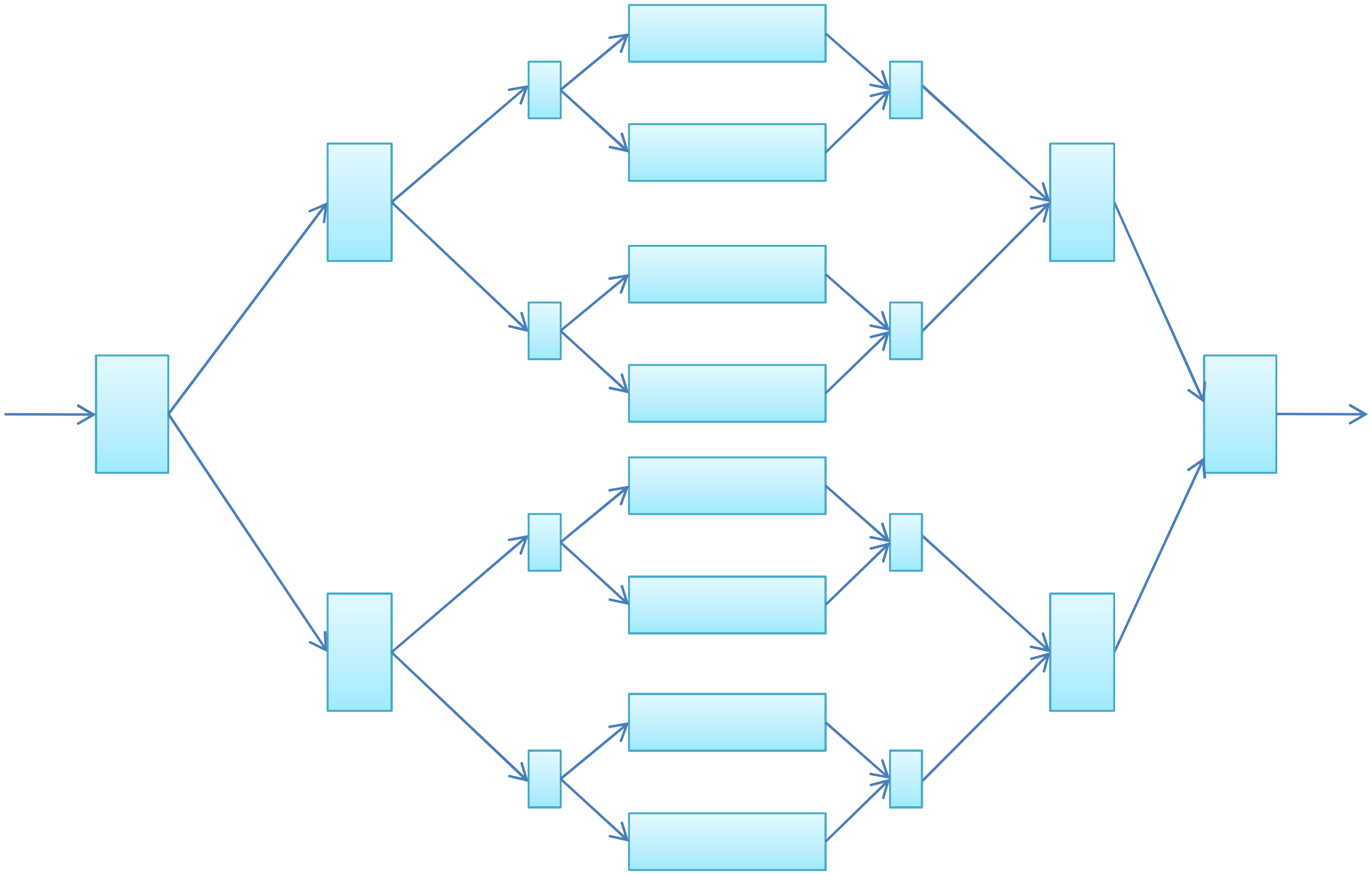
---

tjb4 time always goes down in PPCP - reorient
Tom Ball, 8/10/2010

Parallel DAG for Merge Sort (4-core)



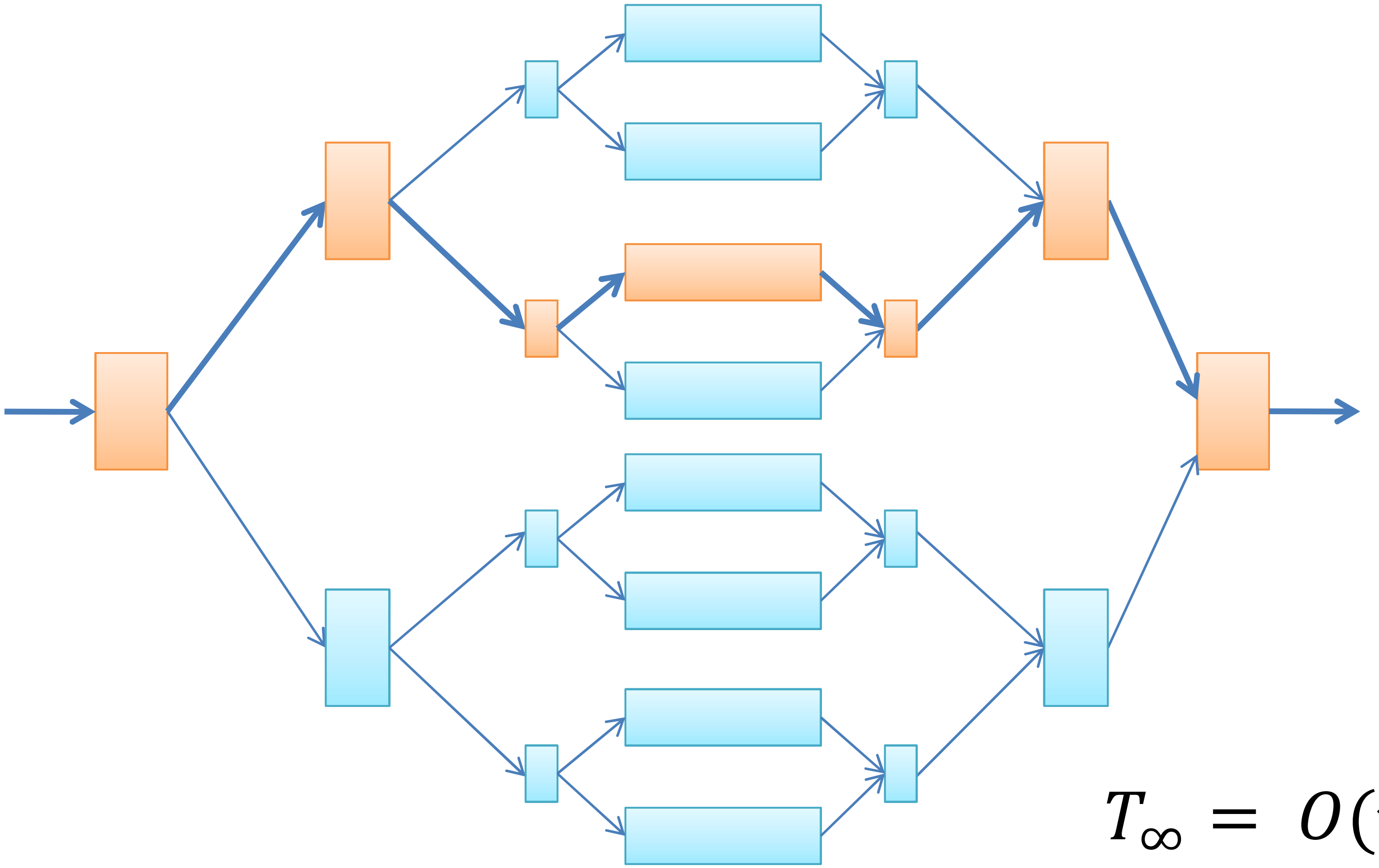
Parallel DAG for Merge Sort (8-core)



Work and Span

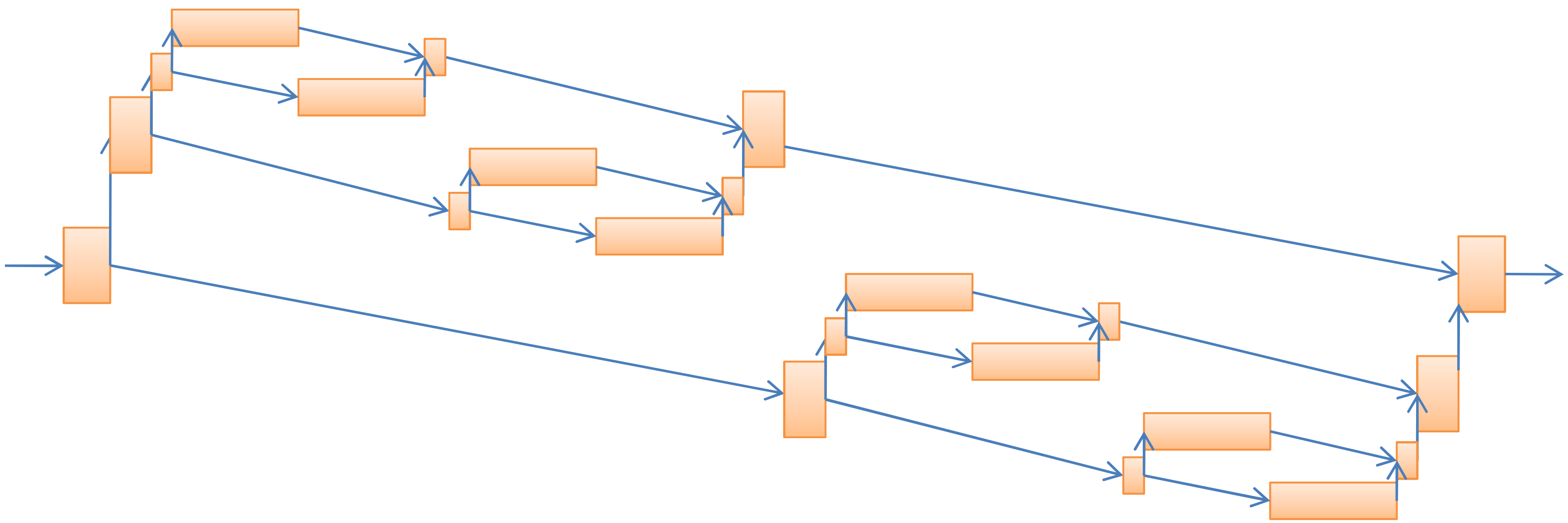
- Work
  - the total number of operations executed by a computation
- Span
  - the longest chain of sequential dependencies (critical path) in the parallel DAG

T_∞ (span): Critical Path Length (Sequential Bottleneck)



$$T_\infty = O(n)$$

T_1 (work): Time to Run Sequentially



$$T_1 = O(n \log n)$$

Work T_1, Span T_∞, and Running Time T_P

T_P = time to run on P processors

Work Law $$\frac{T_1}{P} \leq T_P$$

"speedup is limited by P"

Work T_1, Span T_∞, and Running Time T_P

T_P = time to run on P processors

Speedup $\frac{\boldsymbol{T_1}}{\boldsymbol{T_P}} \leq P$

Work T_1, Span T_∞, and Running Time T_P

T_P = time to run on P processors

Span Law $T_\infty \le T_P$

"speedup also is limited by critical path"

Work T_1, Span T_∞, and Running Time T_P

T_P = time to run on P processors

Speedup $\frac{T_1}{T_P} \leq \frac{\boldsymbol{T_1}}{\boldsymbol{T_\infty}}$ **Parallelism**

"speedup is bounded above
by available parallelism"

Work/Span of Merge Sort (Sequential Merge)

- Work $\boldsymbol{T_1}$: $O(n \log n)$
- Span $\boldsymbol{T_\infty}$: $O(n)$
  - Takes $O(n)$ time to merge n elements

- Parallelism:
  - $\frac{\boldsymbol{T_1}}{\boldsymbol{T_\infty}}$: $O(\log n)$ - really bad!

Main Message

- Analyze the Work and Span of your algorithm
- Parallelism is Work/Span
- Try to decrease Span
  - the critical path
  - a *sequential* bottleneck
- If you increase Span
  - better increase Work by a lot more!

And Now, For Something Completely Different

$$\lambda x.x$$

LambdasAndDelegates.cs

C# Lambda Expressions

- Syntax

```
(input parameters) => expression
(input parameters) => {statement;}
```

- Examples:

```
x => x
(x,y) => x==y
(int x, string s) => s.Length > x
() => { return SomeMethod()+1; }

Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4);
```

Some Useful Delegate Types

- delegate void **Action**()
  - `Action showMethod = () => WriteLine(“Hi!”);`

- delegate void **Action**<T>(T t)
  - `Action<int> showPlusOne = (x) => WriteLine(x+1);`

- delegate U **Func**<T,U>(T t)
  - `Func<int,int> plusOne = (x) => (x+1);`

Things to Know about C# Lambdas

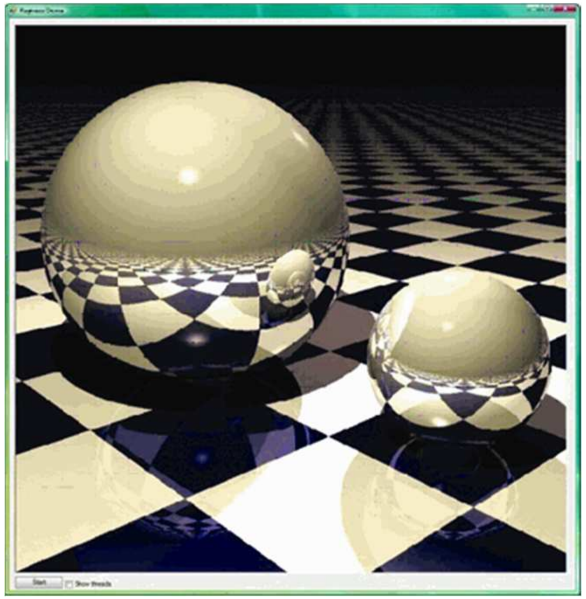
- Lambda is an expression (with no type)
  - Its "signature" is inferred by the compiler by what it's being assigned to.
- Conversion to a delegate type
- Type inference for parameters
- Capture of free variables
  - Locations referenced by free variables are converted to be on the heap ("boxed")

Parallel.For

```
public static ParallelLoopResult For(
    int fromInclusive
  , int toExclusive
  , Action<int> body
  );
```

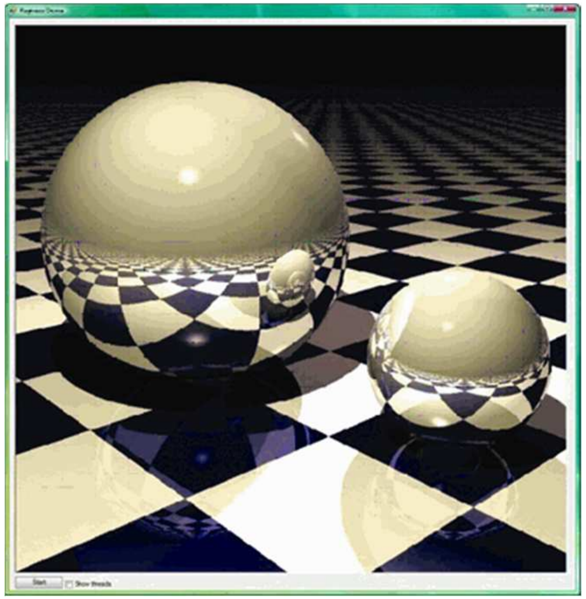
ParallelSamples.cs

PoPP_ClosingOverSharedData.cs



Sequential Ray Tracing

```
void Render(Scene scene, Color[,] rgb)
{
  for (int y = 0; y < screenHeight; y++)
  {
    for (int x = 0; x < screenWidth; x++) {
      rgb[x,y] = TraceRay(new Ray(scene,x,y));
    }
  }
}
```



Parallel Ray Tracing with *Lambda* and **Parallel.For**

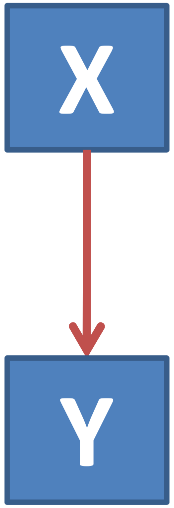
```
void Render(Scene scene, Color[,] rgb)
{
  Parallel.For(0, screenHeight,  (y) =>
  {
    for (int x = 0; x < screenWidth; x++)
    {
      rgb[x,y] = TraceRay(new Ray(scene,x,y));
    }
  });
}
```

Run It!

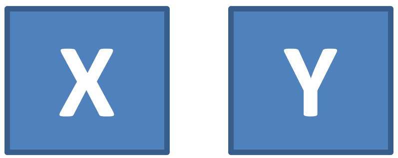
RayTracerTest.cs
ValidateOutput_BasicParallelRaytracer

Thinking About Parallel.For

Parallel DAG Edges (Happens-before) Constrain Execution



- Implementation guarantees
  - X completes execution before Y starts execution
  - All effects of X are visible to Y



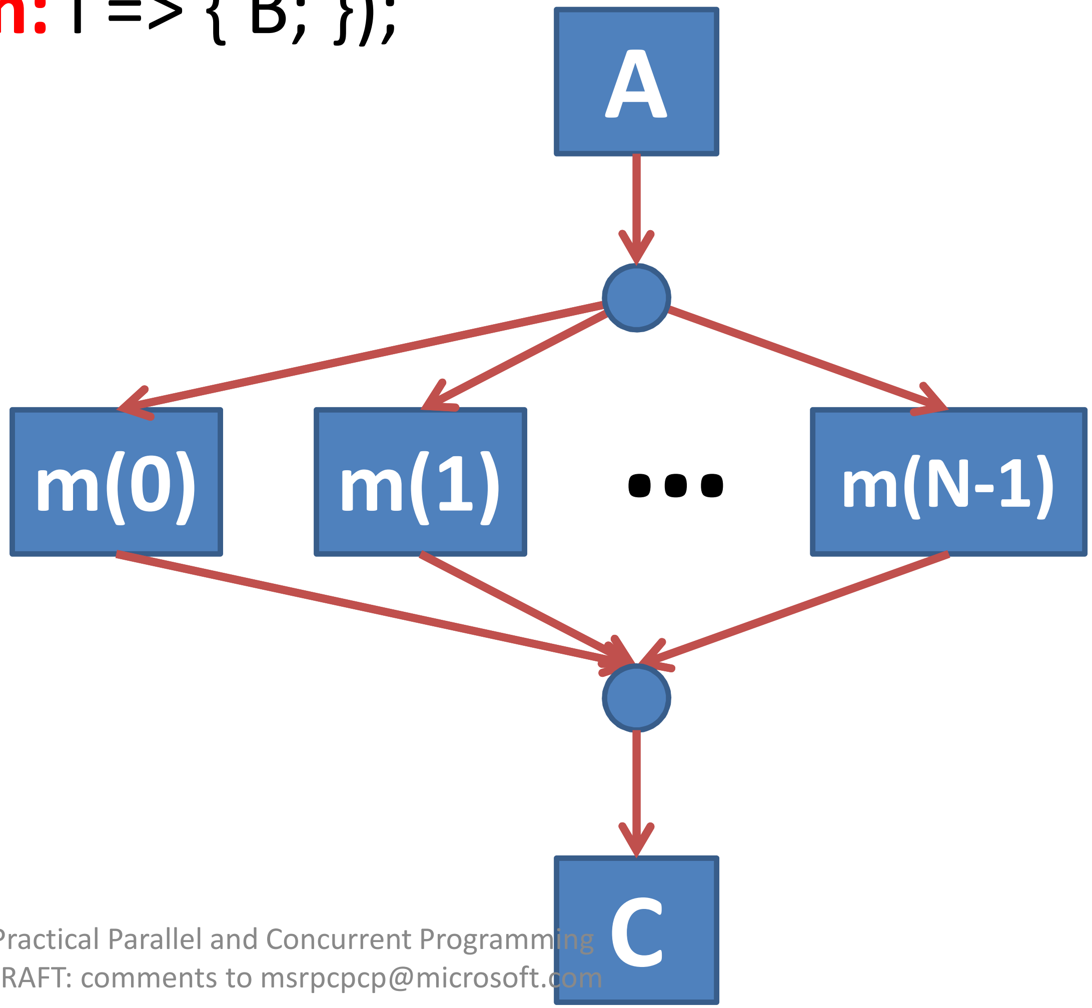
- No directed path between X,Y, no guarantees
  - X,Y could happen in parallel, or
  - X could happen before Y, or
  - Y could happen before X

Parallel.For Happens-before Edges

A;

Parallel.For(0, N, **m:** i => { B; });

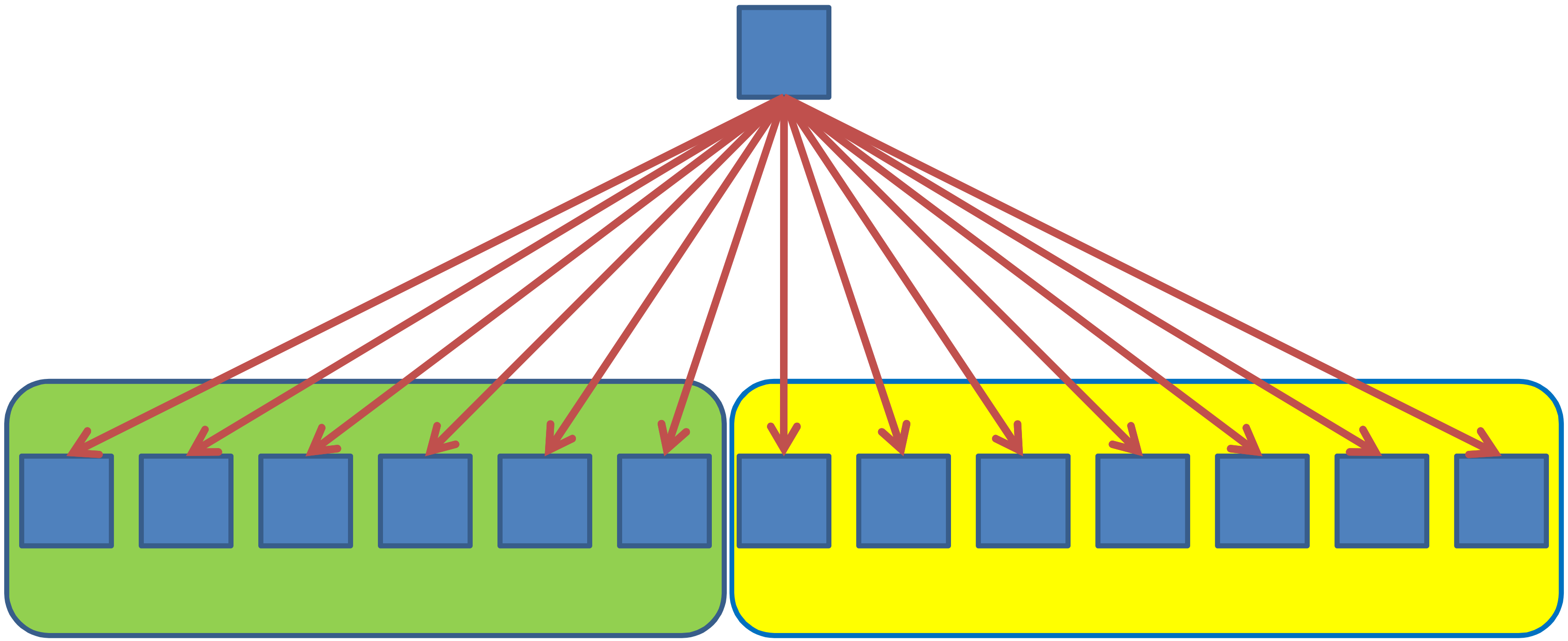
C;



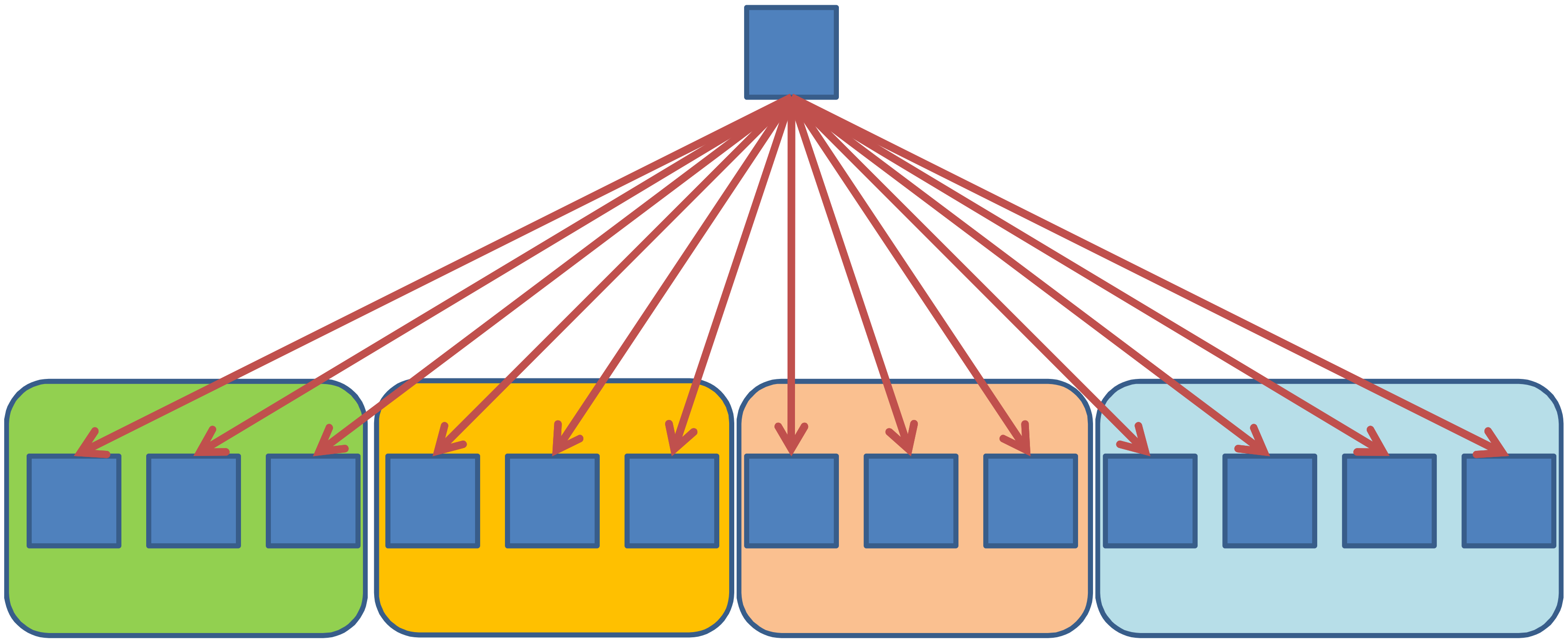
What .NET 4 Does For You

- Parallel.For automatically
  - Assigns work to cores efficiently
  - Dynamically partitions
  - Balances load efficiently
  - Handles exceptions
- And much more…

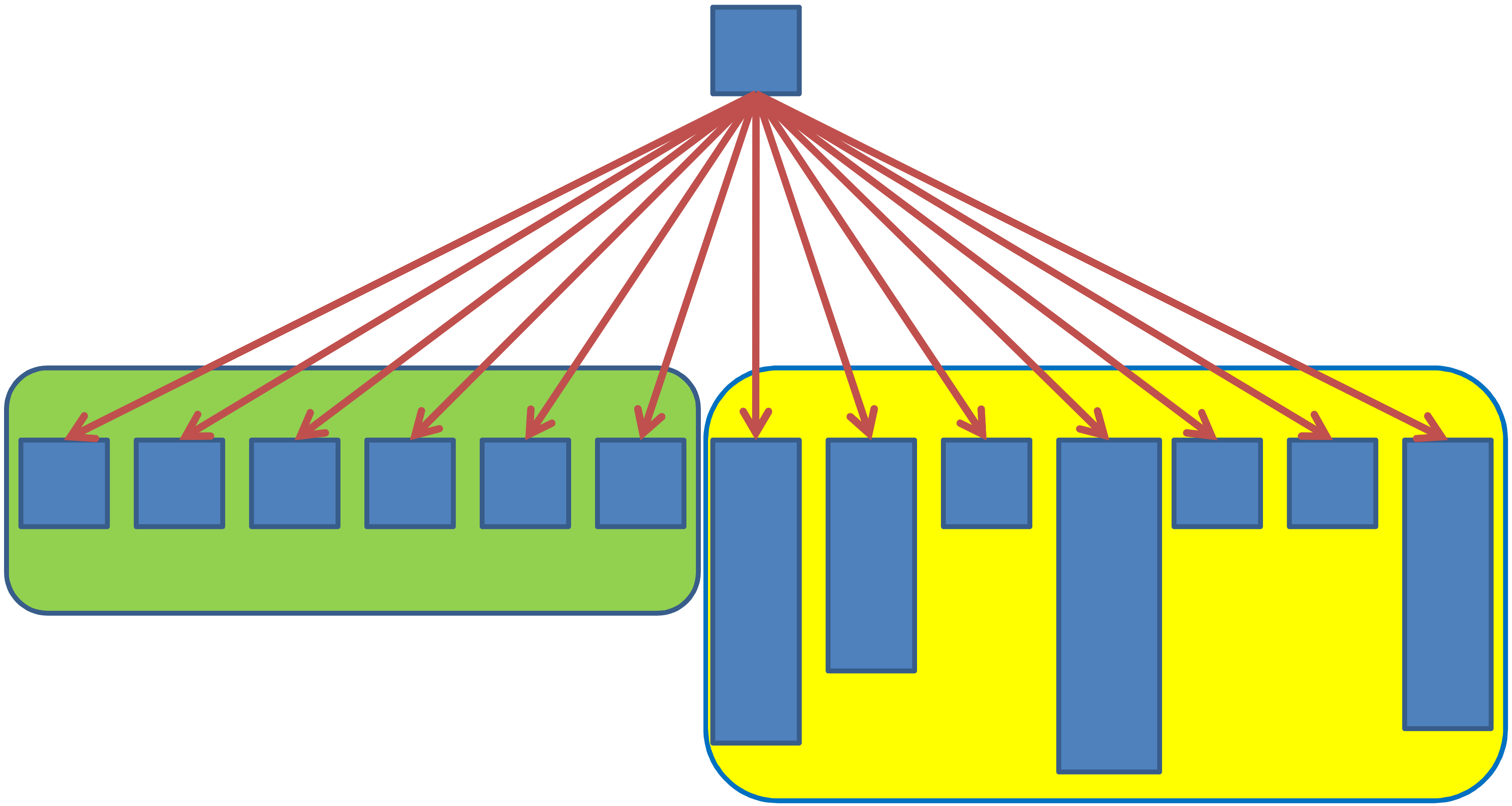
Example Partitioning on Two Cores



Partitioning on Four Cores



Unbalanced Workloads

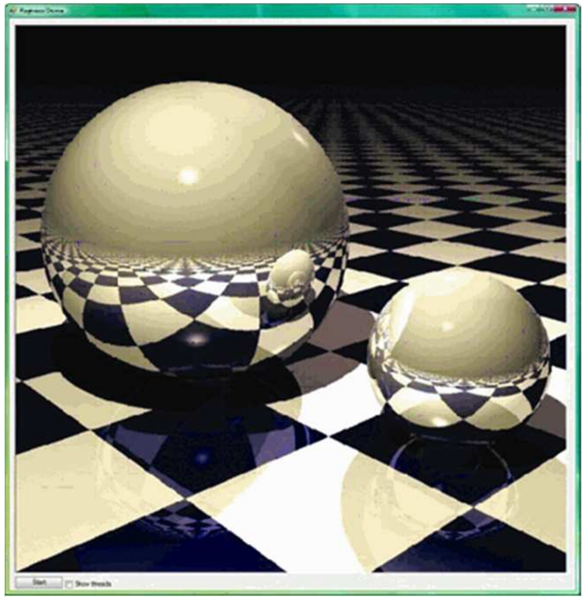


Parallel.For(0, N, i => { S });

- Desirable properties of statement S
  - S does enough work, or
  - N is large enough to compensate

Some Typical Reasons for Performance Problems

- Not enough work per parallel task
  - Overhead of coordinating parallelism dominates
- Insufficient memory bandwidth
  - Limitation of processor architecture
- False sharing
  - Bad cache performance
  - Exposed to memory wall
- Lock contention



Nested Parallel.For

```
void Render(Scene scene, Color[,] rgb)
{
 Parallel.For(0, screenHeight,  (y) =>
 {
  Parallel.For(0, screenWidth,  (x) =>
  {
   rgb[x,y] = TraceRay(new Ray(scene,x,y));
  }
 });
}
```

Is there enough work per parallel task? Experiment and find out!

FineVsCoarseGrainedRayTracer.cs
RayTracerTest.cs

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

- Parallel.For/ForEach
  - Game of Life
  - Blend Images
  - ImageColorizer
  - Morph
  - MandelbrotsFractals
  - ComputePi
  - Strassens

Parallel Programming with Microsoft .NET

- Chapter 1 (Introduction)
- Chapter 2 (Parallel Loops) Parallel.For/ForEach