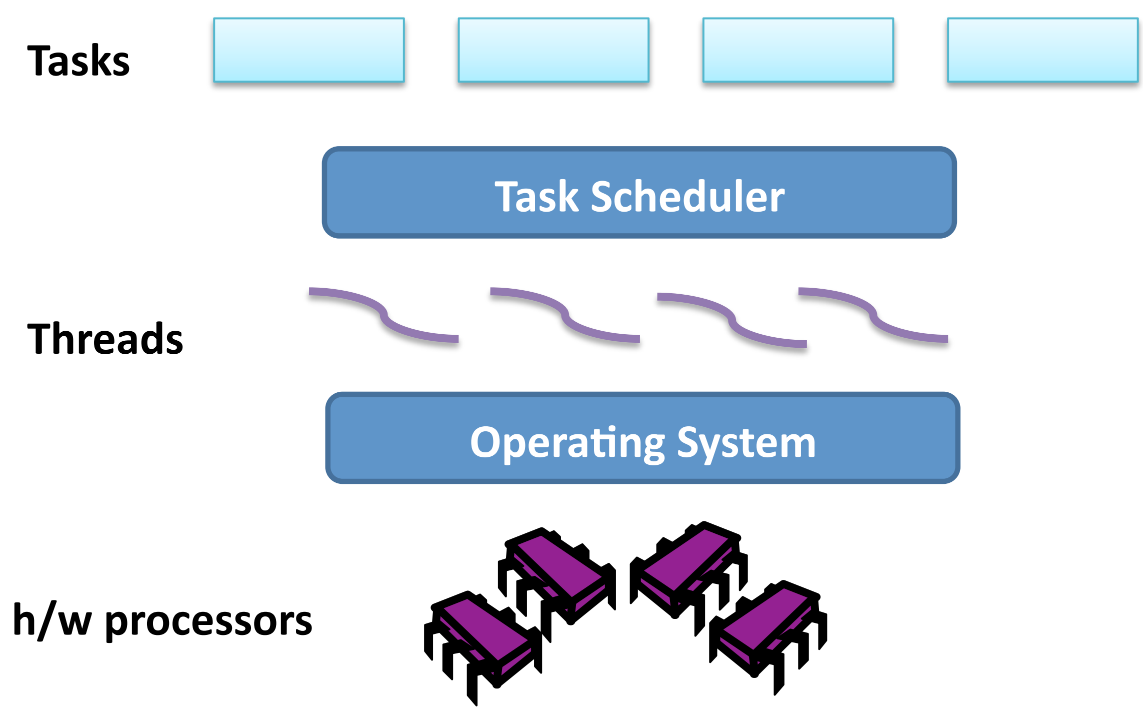
PARALLEL PROGRAMMING ABSTRACTIONS

Tasks vs Threads

- Similar but not the same.



Tasks vs Threads: Main Differences

- Tasks need not run concurrently
  - The task scheduler can schedule them on the same thread

Tasks vs Threads: Main Differences

- Tasks need not run concurrently
  - The task scheduler can schedule them on the same thread
- Tasks do not have fairness guarantees

```
while(t == 0);
Write("hello");
```

```
t = 1;
```

Tasks vs Threads: Main Differences

- Tasks need not run concurrently
  - The task scheduler can schedule them on the same thread
- Tasks do not have fairness guarantees
- Tasks are cheaper than threads
  - Have smaller stacks
  - Do not pre-allocate OS resources

Generating Tasks from Programs

- You don't want programmers to explicitly create tasks
  - Like assembly level programming
- Instead:
  - Design programming language constructs that capture programmers intent
  - Compiler converts these constructs to tasks
- Languages with the following features are very convenient for this purpose
  - Type inference
  - Generics
  - First order anonymous functions (lamdas/delegates)

First-order functions: C# Lambda Expressions

- Syntax

```
(input parameters) => expression
```

- Is similar to :

```
&anon_foo // where foo is declared elsewhere
anon_foo(input parameters){ return expression;}
```

- 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);
```

Sequential Merge Sort With Lambdas

```
MergeSort(int[] a, low, hi){
    if(base_case) …

    int mid = low + (hi-low)/2;

    var f = (l,h) => { MergeSort(a, l, h);}

    f(low, mid-1);
    f(mid, high);

    Merge(a, low, mid, hi);
}
```

Things to Know about C# Lambdas

- Lambda is an expression (with no type)
- 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”)

The Task Abstraction

```
delegate void Action();

class Task {
      Task( Action a );
      void Wait();

      // called by the WSQ scheduler
      void Execute();
}
```

Merge Sort With Tasks

```
MergeSort(int[] a, low, hi){
    if(base_case) …

    int mid = low + (hi-low)/2;

    Task left = new Task(
       delegate{ MergeSort(a, low, mid); } );

    Task right = new Task(
       delegate{ MergeSort(a, mid, hi); } );

    left.Wait();
    right.Wait();

    Merge(a, low, mid-1, hi);
}
```

Parallel.Invoke

```
static void Invoke(params Action[] actions);
```

- Invokes all input actions in parallel
- Waits for all of them to finish

Merge Sort With Parallel.Invoke

```
MergeSort(int[] a, low, hi){
    if(base_case) …

    int mid = low + (hi-low)/2;

    Paralle.Invoke{
      () => { MergeSort(a, low, mid-1); }
      () => { MergeSort(a, mid, hi); }
    }

    Merge(a, low, mid, hi);
}
```

Compare with Sequential Version

```
MergeSort(int[] a, low, hi){
    if(base_case) …

    int mid = low + (hi-low)/2;


            { MergeSort(a, low, mid-1); }
            { MergeSort(a, mid, hi); }


    Merge(a, low, mid, hi);
}
```

Data Parallelism

- Sometimes you want to perform the same computation on all elements of a collection

- For every string in an array, check if it contains "foo"

Parallel.For

```
For(int lower, int upper,
            delegate int (int) body);
```

- Iterates a variable i from lower from to upper
- Calls the delegate with i as the parameter in parallel

Parallel.For

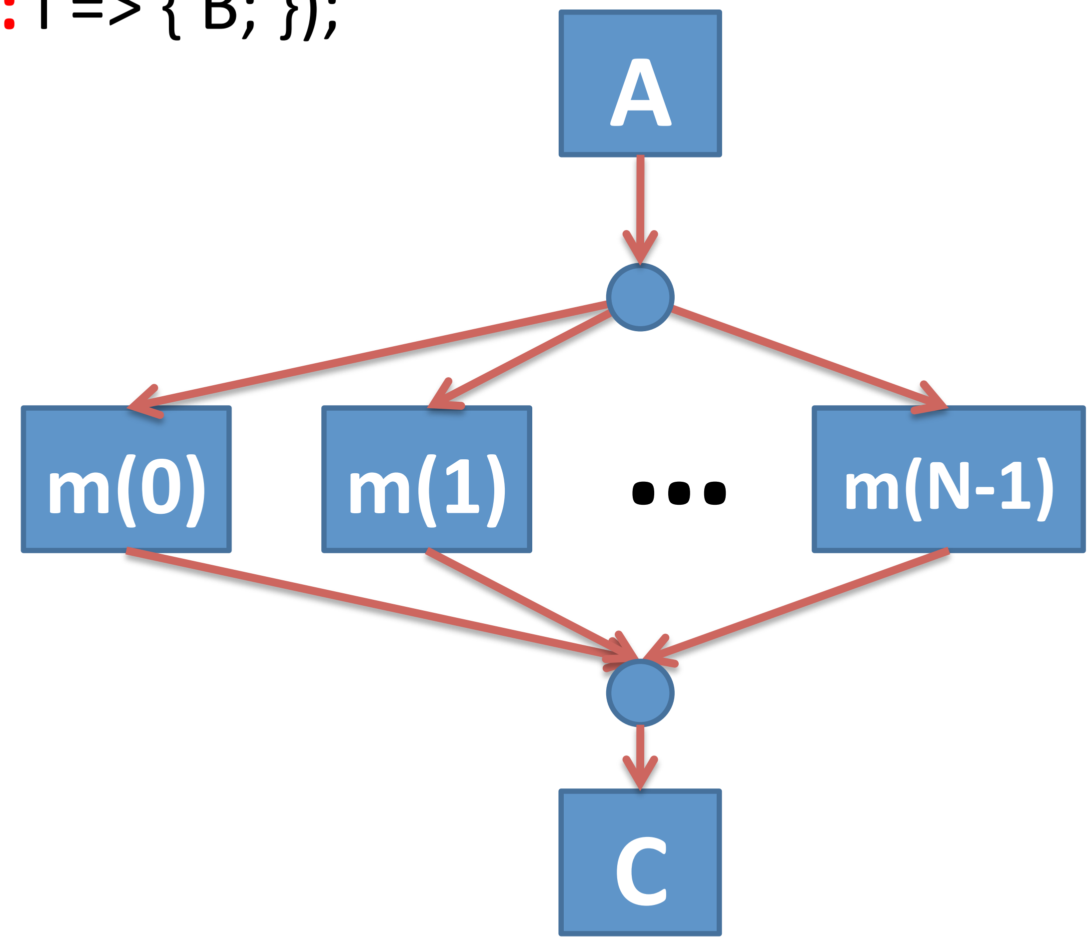
```
// sequential for
for(int i=0; i<n; i++){
   if(a[i].Contains("foo")){DoSomething(a[i]);}
}

//Parallel for
Parallel.For(0, n, (i) => {
   if(a[i].Contains("foo")){DoSomething(a[i]);}
});
```

The DAG created by Parallel.For

A;

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

C;



Paralle.ForEach

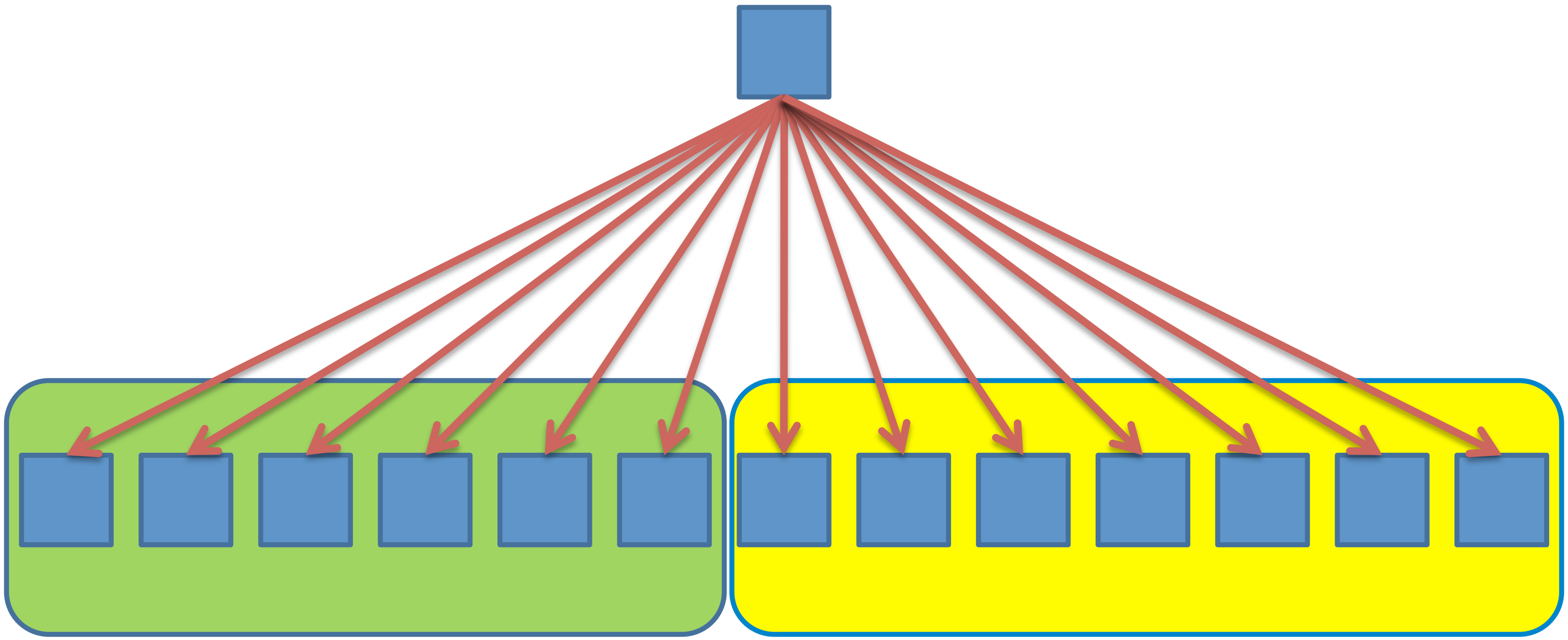
```
ForEach<T>(IEnumerable<T> source,
           delegate void (T) body);
```

- Same as Parallel.For, but iterates over elements of a collection in parallel

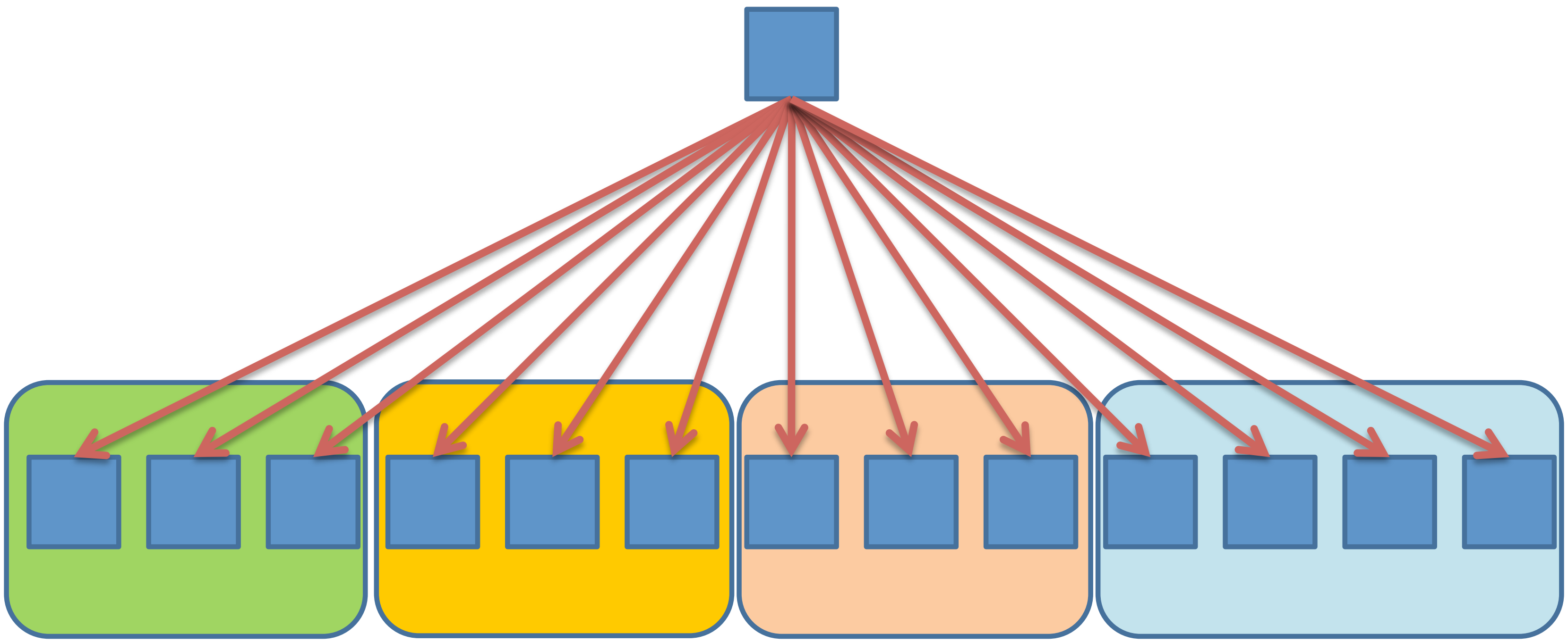
Advantage of High-Level Abstractions

- Makes it easy to add parallelism
  - Explore and experiment with different parallelization strategies
- Language Compiler/Runtime can perform performance optimizations
  - Efficiently create tasks
  - Efficiently distribute tasks to available cores
  - Tight integration with scheduler

Example Partitioning on Two Cores



Partitioning on Four Cores



Advantage of High-Level Abstractions

- Makes it easy to add parallelism
  - Explore and experiment with different parallelization strategies
- Language Compiler/Runtime can perform performance optimizations
  - Efficiently create tasks
  - Efficiently distribute tasks to available cores
  - Tight integration with scheduler
- Provide programmatic features
  - Exceptions
  - Cancelling running tasks

Semantics of Parallel Constructs

- What does this do:

```
Paralle.Invoke{
   () => { WriteLine("Hello"); }
   () => { WriteLine("World"); }
}
```

Semantics of Parallel Constructs

- What does this do:

```
Paralle.Invoke{
   () => { WriteLine("Hello"); }
   () => { WriteLine("World"); }
}
```

- Compare with

```
{ WriteLine("Hello"); }
{ WriteLine("World"); }
```

Semantics of Parallel Constructs

- By writing this program

```
Paralle.Invoke{
  () => { WriteLine("Hello"); }
  () => { WriteLine("World"); }
}
```

- You are telling the compiler that both outcomes are acceptable

Correctness Criteria

- Given a DAG, any linearization of the DAG is an acceptable execution

Correctness Criteria

- Simple criterion:
  - Every task operates on separate data
  - No dependencies other than the edges in the DAG

- We will look at more complex criteria later in the course.