Parallel Programming

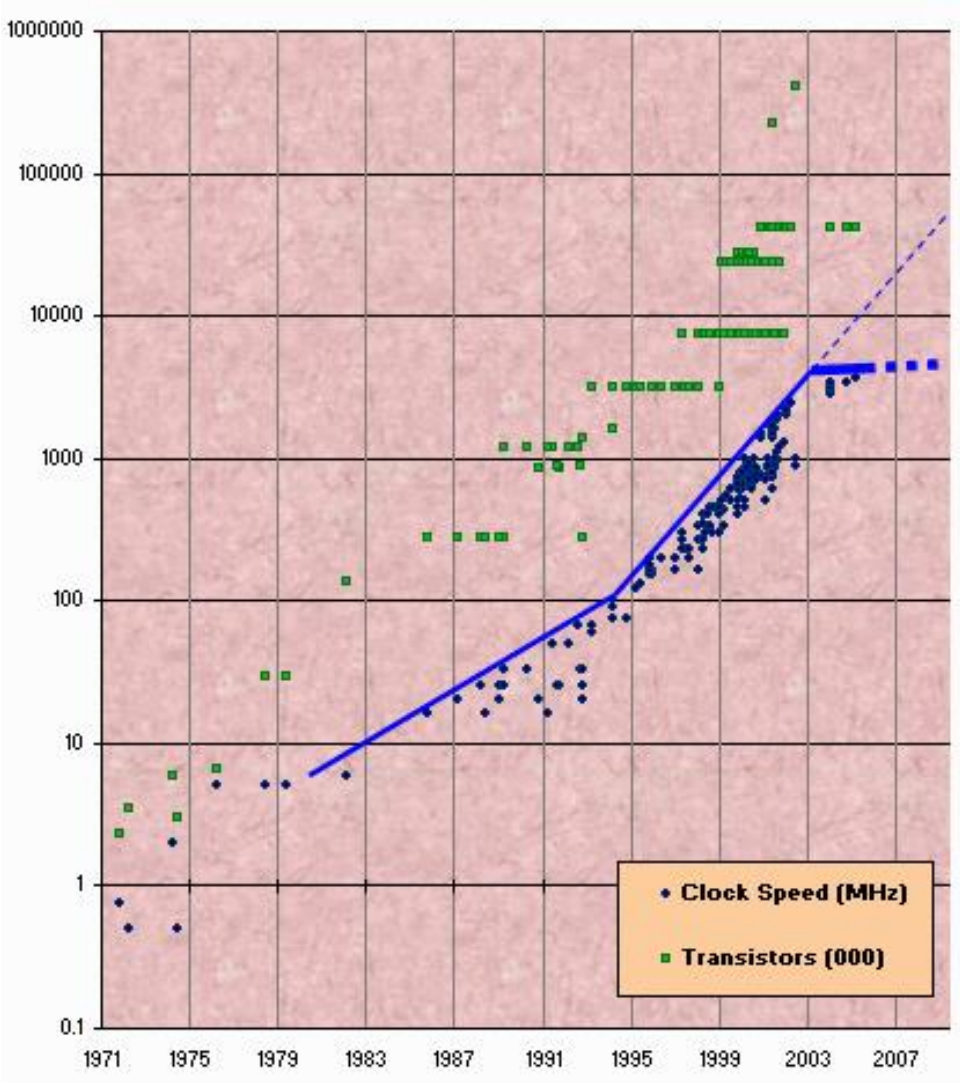
DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Objectives

- **Why program in parallel ?**
- **Introduce Parallel Extensions**
  - Parallel.Do
  - Parallel.For
  - Parallel.ForEach
- **Introduce PLINQ Parallel Linq**

“Free Lunch is over” Herb Sutter, Dr Dobb’s Journal March 2005



Why the need for parallel programming

- **Doesn't MS Word run fast enough ?**
- **For a given time frame X**
  - Data volume increases
    - More detail in games
    - Intellisense
  - Better answers
    - Speech recognition
    - Route planning
    - Word gets better at grammar checking

Setting Expectations

- **Amdahl's Law**
  - If only n% of the compute can be parallelised at best you can get an n% reduction in time taken.
  - Example
    - 10 hours of compute of which 1 hour cannot be parallelised.
    - Minimum time to run is 1 hour + Time taken for parallel part, thus you can only ever achieve a maximum of 10x speed up.
- **Not even the most perfect parallel algorithms scale linearly**
  - Scheduling overhead
  - Combining results overhead

Types of Parallelism, Coarse Grained

- **Coarse grain parallelism**
  - Large tasks, little scheduling or communication overhead
    - Each service request executed in parallel, parallelism realised by many independent requests.
    - DoX,DoY,DoZ and combine results. X,Y,Z large independent operations
  - Applications
    - Web Page rendering
    - Servicing multiple web requests
    - Calculating the payroll

Types of Parallelism, Fine Grained

- **Fine grain parallelism**
  - A single activity broken down into a series of small co-operating parallel tasks.
    - *f*(n=0...X) can possibly be performed by many tasks in parallel for given values of n
    - Input => Step 1 => Step 2 => Step 3 => Output
  - Applications
    - Sensor processing
    - Route planning
    - Graphical layout
    - Fractal generation (Trade Show Demo)

Moving to a Parallel Mind Set

- **Identify areas of code that could benefit from concurrency**
  - Keep in mind Amdahl's law
- **Adapt algorithms and data structures to assist concurrency**
  - Minimise shared state
  - Lock free data structures
- **Utilise class System.Threading.Parallel**
  - Built on top of the Task library

Fork/Join Pattern

- **Task based parallelism**
  - Book Hotel, Book Car, Book Flight
  - Sum X + Sum Y + Sum Z

```
Parallel.Invoke( BookHotel , BookCar , BookFlight );

// Won't get here until all three tasks are complete
```

```
Task<int> sumX = Task.Factory.StartNew(SumX);
Task<int> sumY = Task.Factory.StartNew(SumY);
Task<int> sumZ = Task.Factory.StartNew(SumZ);

int total = sumX.Result+ sumY.Result+ sumZ.Result;
```


Parallel Loops

- **Loops can be successfully parallelised IF**
  - The order of execution is not important.
  - Each iteration is independent of each other.
  - Each Iteration has sufficient work to compensate for scheduling overhead.

Parallel.For

- **Replaces a conventional for loop with a parallel loop**
  - Restrictions
    - Only integer range.
    - Increments only by 1.
- **Loop body supplied by a delegate.**
- **Will the output of both these pieces code be the same ?**

```
for (int i = 0; i < 10; i++)
{
  Console.WriteLine(i);
}
```

```
Parallel.For(0, 10, i =>
{
  Console.WriteLine(i);
});
```

Parallel.For Mechanics

- **Parallel.For utilises multiple tasks**
  - The number of created tasks will vary based on number of cores and monitored throughput.
  - A single task may execute many loop iterations.
  - Each iteration of the loop results in a delegate invocation
- **Since loop iterations run in parallel**
  - All the normal thread safety issues come into play
  - Heavy use of synchronization can negate the full benefit.

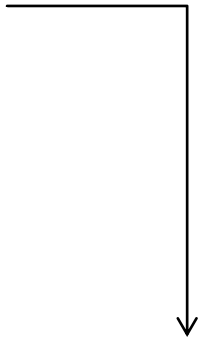
Parallel Aggregation

- **Would the following Parallel.For be a suitable replacement ?**

```
double pi = 1;
double multiplier = -1;

for (int i = 3; i < 100000000; i+=2)  {
   pi += multiplier * (1.0 / (double)i);
   multiplier = multiplier * -1;
}

pi *= 4.0;
```

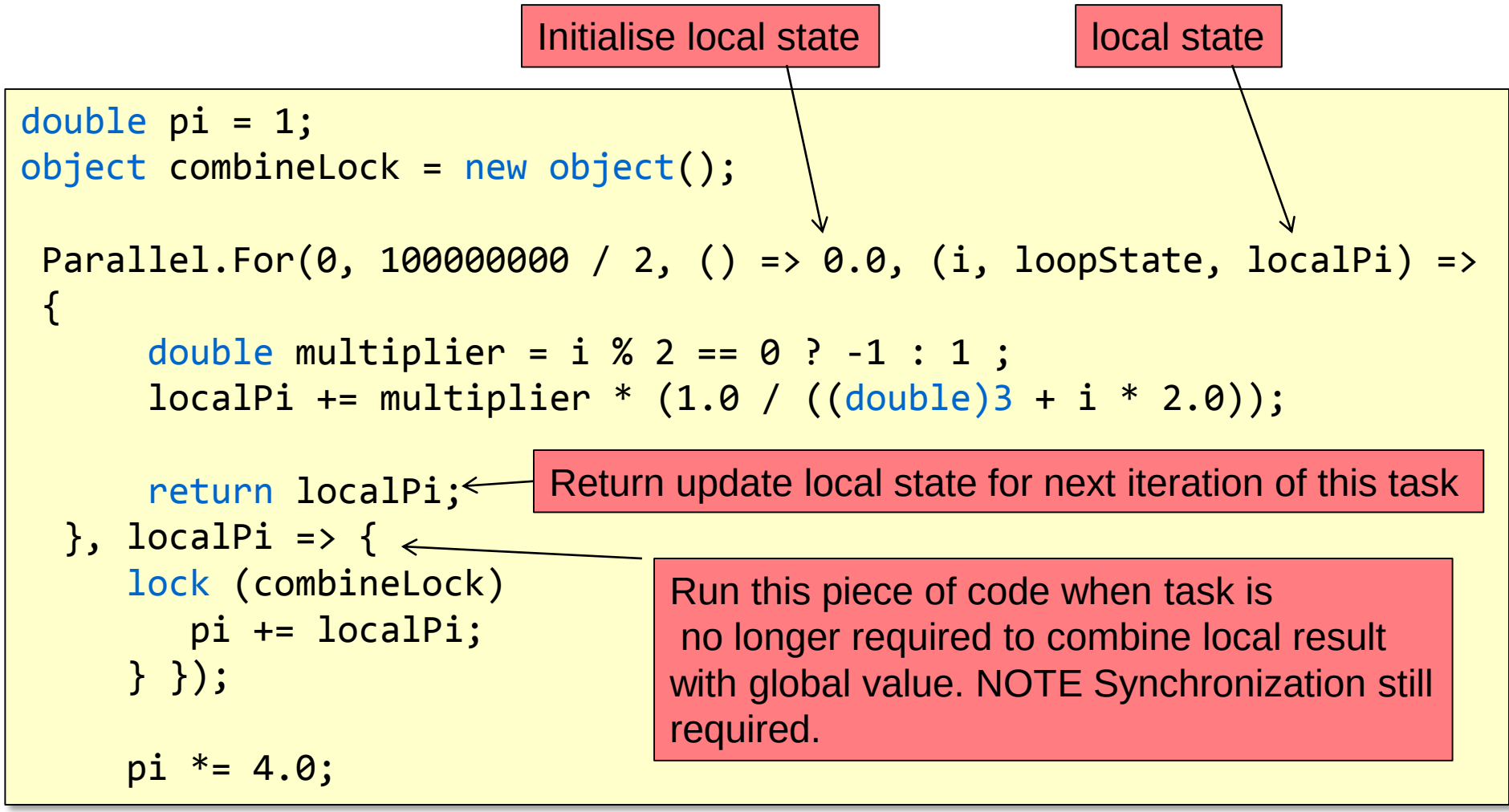


```
Parallel.For(0, 100000000 / 2, i =>
{
   pi += multiplier * (1.0 / ((double)3 + i * 2.0));
   multiplier *= -1;
});
```

Thread Safe Parallel.For

- **Utilising Monitor based synchronization around updating shared state would create an unnecessary bottleneck.**
- **Ideally we want each task to have no shared state and can run freely.**
- **Parallel.For allows each Task to have a piece of local storage.**
  - Utilise local storage to build a partial result.
  - Supply a delegate to be run when task is first used to initialise local result.
  - Supply a delegate to be run when the task is no longer used to combine its local result with a shared global result
    - WARNING...still need to synchronize for this part.

Thread Safe Parallel.For



Parallel.ForEach

- **Similar to Parallel.For but parallelises the consumption of a IEnumerable<T>**
- **IEnumerator<T> is not thread safe so access is guarded**
  - Initially one item is taken per spawned task
  - Attempts to learn the optimal amount to take per task, thus reducing contention.

```
List<int> values = new List<int> { 1, 2, 3, 4, 5 };
Parallel.ForEach(values, i =>
{
   Console.WriteLine(i);
});
```


Parallel Loop Options

- **Parallel Options**
  - CancellationToken
    - Supply your own Cancellation Token, to allow client termination of the loop.
  - Max Degree of Parallelism
    - The maximum number of tasks will run concurrently for the loop.
  - Task Scheduler
    - Select an alternative scheduler.

Alternative Parallel.For

- **Parallel.For is very limited**
  - Only allow int's
  - No steps, just ++
- **Consider using Parallel.ForEach with an iterator method**

```
Parallel.ForEach(Range(1.0, 2.0, 0.1), i =>
{
   Console.WriteLine(i);
});

private static IEnumerable<double> Range(
             double start, double end, double step) {
   for ( double i = start; i < end; i+= step){
                yield return i;
   }
 }
```

My Parallel.For doesn't scale

- **Small loop bodies often don't parallelise**
  - Either they don't scale as expected
  - Or in some cases are slower than sequential version
- **Overhead of task management and delegate invocation cost**
  - As number of cores increases this will become less noticeable.
- **Parallel loops are typically best achieved when you have an inner sequential loop.**

```
for( ;; )
{
  for( ;; )
  {

  }
}
```

Parallelise outer loop to ensure sufficient work per delegate invocation

```
Parallel.For( )
{
  for( ;; )
  {

  }
}
```

Creating Outer loops

- **A single for loop, can be replaced by an outer and inner loop.**

```
Parallel.For (0,5000,i=>
{
  Do(i);
} );
```

```
Parallel.For(0, 50, outerIndex =>
{
  for (int innerIndex = 0; innerIndex < 100; innerIndex++){
   int i = innerIndex + 100 * outerIndex;
   Do(i);
  }
});
```

Parallel.ForEach outer loop

- **Refactoring single loops to outer inner loops can be tedious**
  - Consider using a iterator method to generate a stream of ranges to represent the bounds of the inner for loops

```
Parallel.ForEach(new Range(0, 5000).CreateSubRanges(50), r =>
{
   for (int i = r.Start; i < r.End; i++)
   {
       Do(i)
   }
});
```


Outer loop generation

```
public struct Range    {
   public int Start;
   public int End;

   public IEnumerable<Range> CreateSubRanges(int nRanges)   {

   int subRangeStart = Start;
   int subRangeStep = ((End - Start)+1) / nRanges;
   while (nRanges > 1) {
      yield return new Range()
       { Start = subRangeStart,
         End = subRangeStart + subRangeStep-1 };
      subRangeStart += subRangeStep;
      nRanges--;
    }

    yield return new Range() { Start = subRangeStart, End = End };
}
```


PLinq

- **The AsParallel () extension method on IEnumerable<T>, results in using parallel versions of Where,Select etc.**
  - Works with Linq to Objects and Linq to XML

Utilise Parallel Extension methods
Instead of Enumerable

```
var values = new List<int> { ... };

var results = from val in values.AsParallel()
              where val % 2 == 0
              select val;

foreach (int result in results)
{
     Console.WriteLine(result);
 }
```

PLinq Partitioning

- **PLinq queries start with partitioning**
  - How much data and which bits of data to give each task.
- **Range Partitioning**
  - When the source has an index able interface
  - Queries the IEnumerable<T> to see if its an T[] or IList<T>
- **Chunk Partitioning**
  - Tasks request a chunk of items, the size of chunks typically grow over time.
  - Can produce better load balancing.
  - Used for non index able sources.
- **Utilising Range Partitioning can increase performance.**

ForAll

- **Parallel version of List<T>.ForEach**
  - ForAll takes an action delegate
  - Action is executed in parallel.

```
var values = new List<int> { ... };

values.AsParallel().ForAll(i =>
{
  if (i % 2 == 0)
  {
      Console.WriteLine(i);
  }
});
```


Summary

- **Is the Free Lunch back ?**
  - Utilising Parallel.XXX may bring back a moderate free lunch
  - Still need to understand what is happening under the covers if you want to scale, and have a gut busting lunch.
- **Benchmark code sequentially before attempting to parallelise**
  - Benchmark on a range of cores.
- **Maximise the use of local storage to reduce contention and simplify thread safety.**