Parallel Programming in C#

Hans-Wolfgang Loidl
<H.W.Loidl@hw.ac.uk>

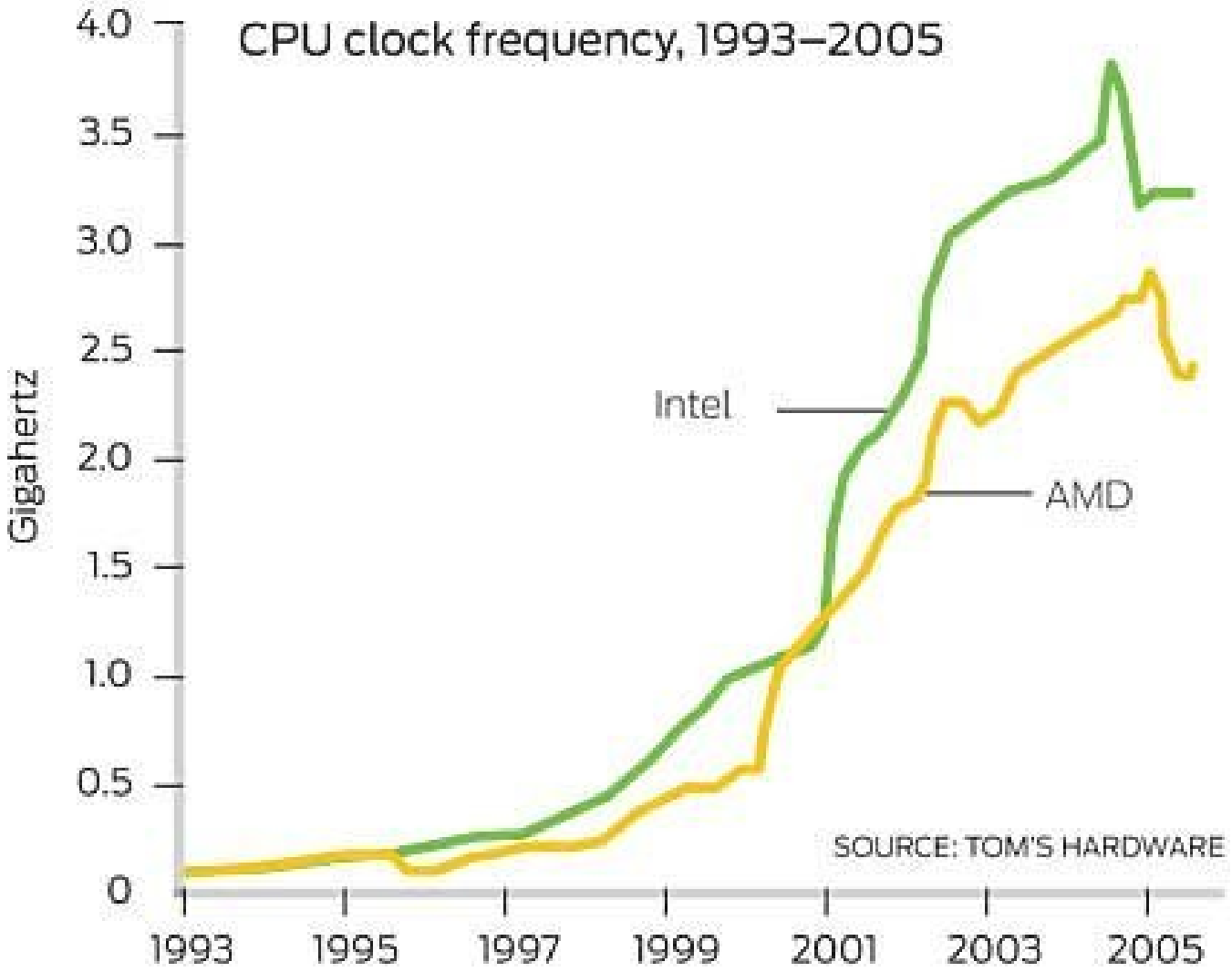
**School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh**

Semester 1 — 2018/19

HERIOT WATT UNIVERSITY

Computers are always too slow!

Clock Rates


CPU clock frequency, 1993–2005
Gigahertz
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
1993
1995
1997
1999
2001
2003
2005
Intel
AMD
SOURCE: TOM'S HARDWARE

The Free Lunch is over!

- **Don't expect your sequential program to run faster on new processors**
- Still, processor technology advances
- BUT the focus now is on *multiple cores per chip*
- Today's desktops typically have 4 cores.
- Latest experimental multi-core chips have up to 1,000 cores[1].

[1] See *"World's First 1,000-Processor Chip"*, University of California, Davis, June 2016

The Free Lunch is over!

- **Don't expect your sequential program to run faster on new processors**
- **Still, processor technology advances**
- **BUT the focus now is on *multiple cores per chip***
- **Today's desktops typically have 4 cores.**
- **Latest experimental multi-core chips have up to 1,000 cores[1].**

[1] See *"World's First 1,000-Processor Chip"*, University of California, Davis, June 2016

The Free Lunch is over!

- **Don't expect your sequential program to run faster on new processors**
- **Still, processor technology advances**
- **BUT the focus now is on *multiple cores per chip***
- **Today's desktops typically have 4 cores.**
- **Latest experimental multi-core chips have up to 1,000 cores[1].**

[1] See *"World's First 1,000-Processor Chip"*, University of California, Davis, June 2016

Options for Parallel Programming in C#

C# provides several mechanisms for par. programming:

Explicit threads with synchronisation via locks, critical regions etc.

- The user gets full control over the parallel code.
- *BUT* orchestrating the parallel threads is tricky and error prone (race conditions, deadlocks etc)
- This technique requires a *shared-memory model*.

Options for Parallel Programming in C#

C# provides several mechanisms for par. programming:

Explicit threads with a message-passing library:

- Threads communicate by explicitly sending messages, with data required/produced, between workstations.
- Parallel code can run on a *distributed-memory architecture*, eg. a network of workstations.
- The programmer has to write code for (un-)serialising the data that is sent between machines.
- *BUT* threads are still explicit, and the difficulties in orchestrating the threads are the same.
- A common configuration is C+MPI.

Options for Parallel Programming in C#

C# provides several mechanisms for par. programming:

OpenMP provides a standardised set of program annotations for parallelism, without explicit threads.

- The annotations provide information to the compiler where and when to generate parallelism.
- It uses a *shared-memory model* and communication between (implicit) threads is through shared data.
- This provides a higher level of abstraction and simplifies parallel programming.
- *BUT* it currently only works on physical shared-memory systems.

Options for Parallel Programming in C#

C# provides several mechanisms for par. programming:

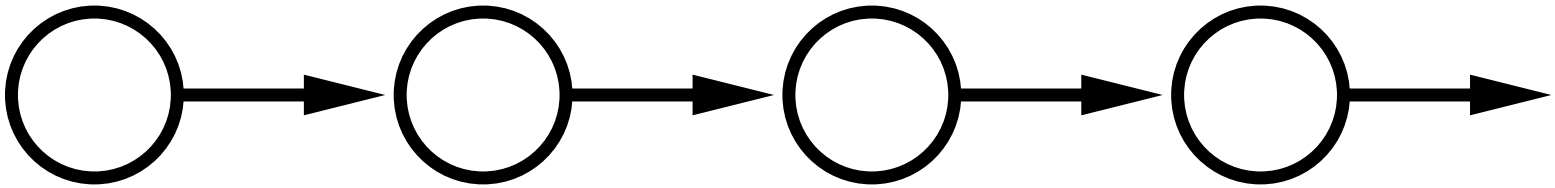
Declarative languages, such as F# or Haskell, do not operate on a shared program state, and therefore provide a high degree if inherent parallelism:

- *Implicit parallelism* is possible, ie. *no* additional code is needed to generate parallelism.
- The compiler and runtime-system automatically introduce parallelism.
- *BUT* the resulting parallelism is often fine-grained and inefficient.
- Therefore, typically annotations are used to improve parallel performance.

Options for Parallel Programming in C#

C# provides several mechanisms for par. programming:

- Imperative and object-oriented programming models are inherently sequential:
  - They describe an algorithm *step-by-step*.

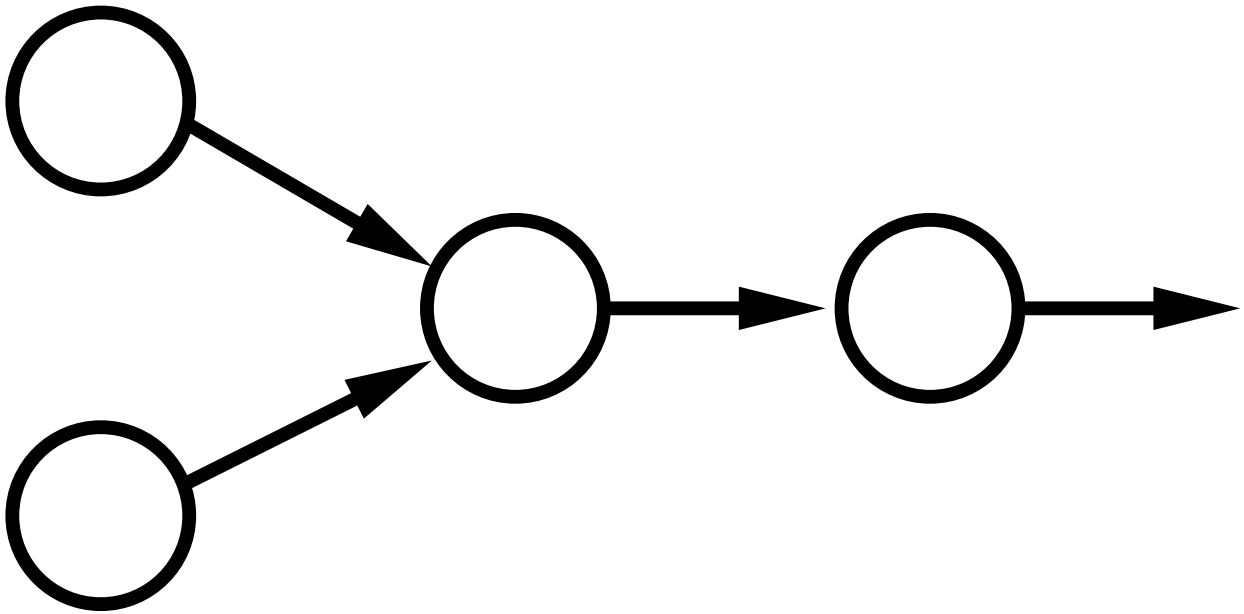


- Parallelising a program often needs re-structuring, is difficult and therefore expensive.

Options for Parallel Programming in C#

C# provides several mechanisms for par. programming:

- Declarative programming models describe *what* to compute, rather than *how* to compute it:
  - The order of computation may be modified



- Parallelising a program does not require restructuring of the code and is much easier.

Options for Parallel Programming in C#

C# provides several mechanisms for par. programming:

Parallel patterns, or skeletons, capture common patterns of parallel computation and provide a fixed parallel implementation. They are a specific instance of *design patterns*.

- To the programmer, most parallelism is implicit.
- The program has to use a parallel pattern to exploit parallelism.
- Using such patterns requires advanced language features, in particular delegates (higher-order functions).

Types of Parallelism in C#

C# supports two main models of parallelism:

- *Data parallelism:* where an operation is applied to each element in a collection.
- *Task parallelism:* where independent computations are executed in parallel.

Parallel Loops in C#

A sequential `for` loop in C#:

```
int n = ...
for (int i = 0; i<=n; i++)
{
   // ...
});
```

Parallel Loops in C#

A parallel `for` loop in C#:

```
int n = ...
Parallel.For(0, n, i =>
{
    // ...
});
```

Parallel Loops in C#

A parallel `for` loop in C#:

```
int n = ...
Parallel.For(0, n, i =>
{
   // ...
});
```

- **The *language construct* `for` is translated into a (higher-order) function *Parallel.For*.**
- **The argument to Parallel.For is an *anonymous method*, specifying the code to be performed in each loop iteration.**
- **The arguments to this anonymous method are the start value, the end value and the iteration variable.**

A Simple Example

We can limit the degree of parallelism like this:

```
var options = new ParallelOptions() {
                  MaxDegreeOfParallelism = 2 };
Parallel.For(0, n, options, i =>
 {
  fibs[i] = Fib(i);
 });
```

Terminating a Parallel Loop

Parallel loops have two ways to break or stop a loop instead of just one.

- Parallel break, *loopState.Break()*, allows all steps with indices lower than the break index to run before terminating the loop.
- Parallel stop, *loopState.Stop()*, terminates the loop without allowing any new steps to begin.

Parallel Aggregates

- The *parallel aggregate* pattern combines data parallelism over a collection, with the aggregation of the result values to an overall result.
- It is parameterised both over the operation on each element as well as the combination (aggregation) of the partial results to an overall results.
- This is a very powerful pattern, and it has become famous as the *Google MapReduce* pattern.

An Example of Parallel Aggregates

```
var options = new ParallelOptions() {
                    MaxDegreeOfParallelism = k};
Parallel.ForEach(seq /* sequence */, options,
                    () => 0,  // The local initial partial result
                    // The loop body
                    (x, loopState, partialResult) =>  {
                       return Fib(x) + partialResult; },
                    // The final step of each local context
                    (localPartialSum) => {
                       // Protect access to shared result
                       lock (lockObject)
                         {
                           sum += localPartialSum;
                         }
                    });
```

Discussion

- The *ForEach* loop iterates over all elements of a sequence *in parallel*.
- Its arguments are:
  - A *sequence* to iterate over;
  - *options* to control the parallelism (optional);
  - a delegate initialising the result value;
  - a delegate specifying the operation on each element of the sequence;
  - a delegate specifying how to combine the partial results;
- To protect access to the variable holding the overall result, a *lock* has to be used.

Another Example of Parallel Aggregates

```
int size = seq.Count / k;  // make a partition large enough to feed
var rangePartitioner = Partitioner.Create(0, seq.Count, size);
Parallel.ForEach(
  rangePartitioner, () => 0, // The local initial partial result
  // The loop body for each interval
  (range, loopState, initialValue) => {
    // a *sequential* loop to increas the granularity of the parall
    int partialSum = initialValue;
    for (int i = range.Item1; i < range.Item2; i++) {
      partialSum += Fib(seq[i]);
    }
    return partialSum; },
  // The final step of each local context
  (localPartialSum) => {
    // Use lock to enforce serial access to shared result
    lock (lockObject) {
      sum += localPartialSum;
    }
  });
```

Discussion

- A *Partitioner* (`System.Collections.Concurrent`) is used to split the entire range into sub-ranges.
- Each call to the partitioner returns an index-pair, specifying a sub-range.
- Each task now works on such a sub-range, using a sequential `for` loop.
- This reduces the overhead of parallelism and can improve performance.

Task Parallelism in C#

- When independent computations are started in different tasks, we use a model of *task parallelism*.
- This model is more general than data parallelism, but requires more detailed control of synchronisation and communication.
- The most basic construct for task parallelism is:

```
Parallel.Invoke(DoLeft, DoRight);
```

- It executes the methods `DoLeft` and `DoRight` in parallel, and waits for both of them to finish.

Example of Task Parallelism

The following code sorts 2 lists in parallel, providing a comparison operation as an argument:

```
Parallel.Invoke( // generate two parallel threads
                 () => ic1.Sort(cmp_int_lt),
                 () => ic2.Sort(cmp_int_gt));
```

Implementation of Task Parallelism

- The implementation of *Invoke* uses the more basic constructs
  - *StartNew*, for starting a computation;
  - *Wait*, *WaitAll*, *WaitAny*, for synchronising several computations.
- Any shared data structure needs to be protected with locks, semaphores or such.
- Programming on this level is similar to explicitly managing threads:
  - it can be more efficient but
  - it is error-prone.

Task Parallelism in C#

- Sometimes we want to start several computations, but need only one result value.
- As soon as the first computation finishes, all other computations can be aborted.
- This is a case of *speculative parallelism*.
- The following construct executes the methods `DoLeft` and `DoRight` in parallel, waits for *the first task* to finish, and cancels the other, still running, task:

```
Parallel.SpeculativeInvoke(DoLeft, DoRight);
```

Futures

- **A *future* is variable, whose result may be evaluated by a parallel thread.**
- Synchronisation on a future is implicit, depending on the evaluation state of the future upon read:
  - If it has been evaluated, its value is returned;
  - if it is under evaluation by another task, the reader task blocks on the future;
  - if evaluation has not started, yet, the reader task will evaluate the future itself
- The main benefits of futures are:
  - Implicit synchronisation;
  - automatic inlining of unnecessary parallelism;
  - asynchronous evaluation
- Continuation tasks can be used to build a chain of tasks, controlled by futures.

Futures

- A *future* is variable, whose result may be evaluated by a parallel thread.
- Synchronisation on a future is implicit, depending on the evaluation state of the future upon read:
  - If it has been evaluated, its value is returned;
  - if it is under evaluation by another task, the reader task blocks on the future;
  - if evaluation has not started, yet, the reader task will evaluate the future itself
- The main benefits of futures are:
  - Implicit synchronisation;
  - automatic inlining of unnecessary parallelism;
  - asynchronous evaluation
- Continuation tasks can be used to build a chain of tasks, controlled by futures.

Futures

- A *future* is variable, whose result may be evaluated by a parallel thread.
- Synchronisation on a future is implicit, depending on the evaluation state of the future upon read:
  - If it has been evaluated, its value is returned;
  - if it is under evaluation by another task, the reader task blocks on the future;
  - if evaluation has not started, yet, the reader task will evaluate the future itself
- The main benefits of futures are:
  - Implicit synchronisation;
  - automatic inlining of unnecessary parallelism;
  - asynchronous evaluation
- Continuation tasks can be used to build a chain of tasks, controlled by futures.

Futures

- A *future* is variable, whose result may be evaluated by a parallel thread.
- Synchronisation on a future is implicit, depending on the evaluation state of the future upon read:
  - If it has been evaluated, its value is returned;
  - if it is under evaluation by another task, the reader task blocks on the future;
  - if evaluation has not started, yet, the reader task will evaluate the future itself
- The main benefits of futures are:
  - Implicit synchronisation;
  - automatic inlining of unnecessary parallelism;
  - asynchronous evaluation
- Continuation tasks can be used to build a chain of tasks, controlled by futures.

Example: Sequential Code

```
private static int seq_code(int a) {
 int b = F1(a);
 int c = F2(a);
 int d = F3(c);
 int f = F4(b, d);
 return f;
}
```

Example: Parallel Code with Futures

```
private static int par_code(int a) {
 // constructing a future generates potential parallelism
 Task<int> futureB = Task.Factory.StartNew<int>(() => F1(a));
 int c = F2(a);
 int d = F3(c);
 int f = F4(futureB.Result, d);
 return f;
}
```

Divide-and-Conquer Parallelism

- Divide-and-Conquer is a common (sequential) pattern:
  - If the problem is atomic, solve it directly;
  - otherwise the problem is *divided* into a sequence of sub-problems;
  - each sub-problem is solved recursively by the pattern;
  - the results are *combined* into an overall solution.

Recall: Binary Search Trees

```
public class Node<T> where T:IComparable {
     // private member fields
     private T data;
     private Node<T> left;
     private Node<T> right;

     // properties accessing fields
     public T Value { get { return data;  }
                      set { data = value; } }
     public Node<T> Left { get { return this.left;  }
                           set { this.left = value; } }
     public Node<T> Right { get { return this.right;  }
                            set { this.right = value; } }
```

Example: Parallel Tree Mapper

```
public delegate T TreeMapperDelegate(T t);

public static void ParMapTree(TreeMapperDelegate f,
                              Node<T> node) {
if (node==null) { return ; }

node.Value = f(node.Value);
var t1 = Task.Factory.StartNew(() =>
                 ParMapTree(f, node.Left));
var t2 = Task.Factory.StartNew(() =>
                 ParMapTree(f, node.Right));
Task.WaitAll(t1, t2);
}
```

Example: Sorting

```
static void SequentialQuickSort(int[] array, int from, int to) {
   if (to - from <= Threshold)   {
     InsertionSort(array, from, to);
   } else {
     int pivot = from + (to - from) / 2;
     pivot = Partition(array, from, to, pivot);
     SequentialQuickSort(array, from, pivot - 1);
     SequentialQuickSort(array, pivot + 1, to);
   }
 }
```

Example: Parallel Quicksort

```
static void ParallelQuickSort(int[] array, int from,
                              int to, int depthRemaining) {
  if (to - from <= Threshold) {
    InsertionSort(array, from, to);
  } else {
    int pivot = from + (to - from) / 2;
    pivot = Partition(array, from, to, pivot);
    if (depthRemaining > 0) {
      Parallel.Invoke(
        () => ParallelQuickSort(array, from, pivot - 1,
                                depthRemaining - 1),
        () => ParallelQuickSort(array, pivot + 1, to,
                                depthRemaining - 1));
    } else {
      ParallelQuickSort(array, from, pivot - 1, 0);
      ParallelQuickSort(array, pivot + 1, to, 0);
    }
  }
}
```

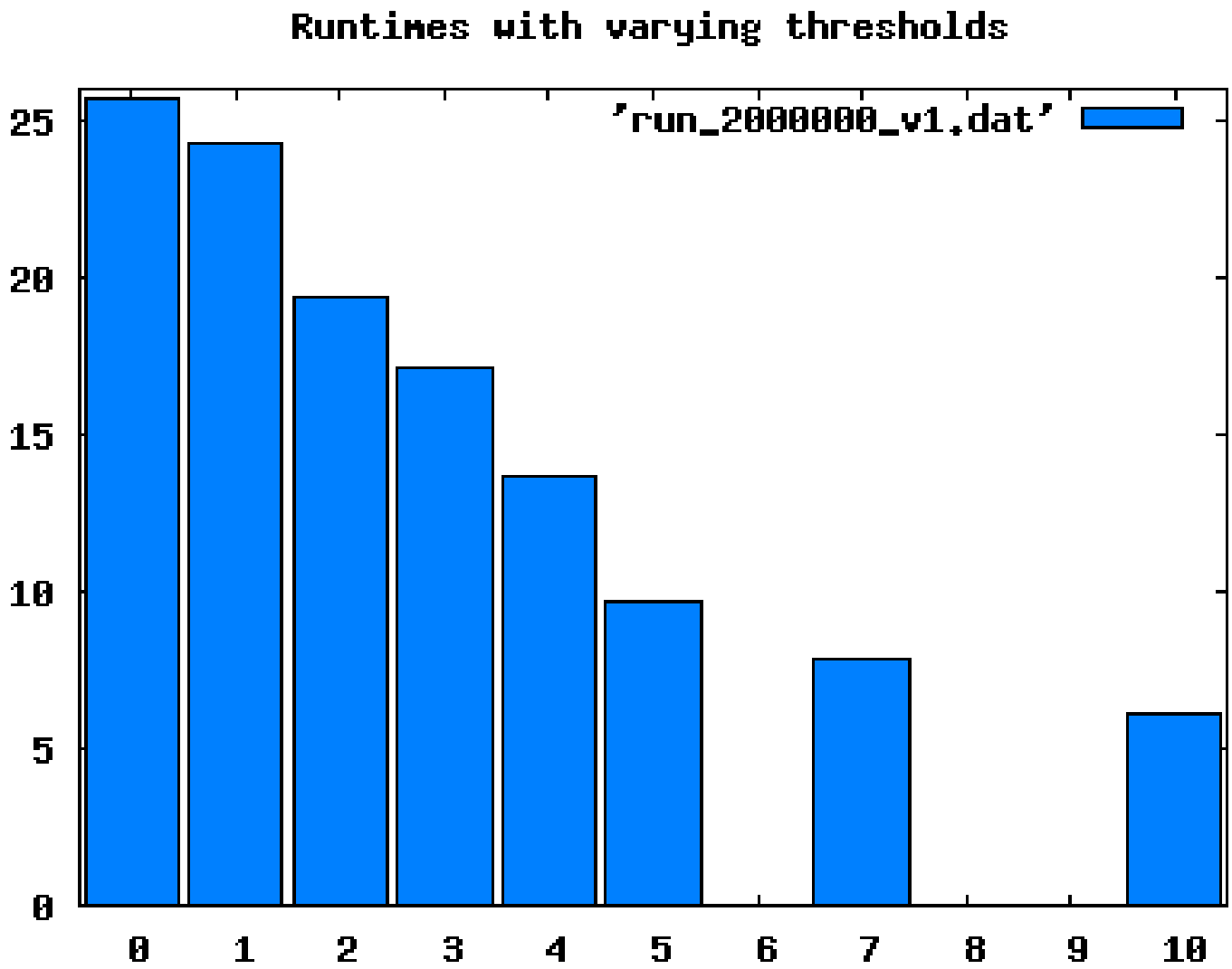
Example: Partition (Argh)

```
private static int Partition(int[] array, int from, int to, int pivot) {
 // requires: 0 <= from <= pivot <= to <= array.Length-1
 int last_pivot = -1;
 int pivot_val = array[pivot];
 if (from<0 || to>array.Length-1) {
   throw new System.Exception(String.Format("Partition: indices out of bounds: from={0}, to={1}, Length={2}",
                                            from, to, array.Length));
 }
 while (from<to) {
  if (array[from] > pivot_val) {
    Swap(array, from, to);
    to--;
  } else {
    if (array[from]==pivot_val) {
      last_pivot = from;
    }
    from++;
  }
 }
 if (last_pivot == -1) {
   if (array[from]==pivot_val) {
     return from;
   } else {
     throw new System.Exception(String.Format("Partition: pivot element not found in array"));
   }
 }
 if (array[from]>pivot_val) {
   // bring pivot element to end of lower half
   Swap(array, last_pivot, from-1);
   return from-1;
 } else {
  // done, bring pivot element to end of lower half
```

Discussion

- An explicit threshold is used to limit the amount of parallelism that is generated (*throttling*).
- This parallelism threshold is not to be confused with the sequential threshold to pick the appropriate sorting algorithm.
- Here the divide step is cheap, but the combine step is expensive; don't expect good parallelism from this implementation!

Performance of Parallel QuickSort



A Comparison: QuickSort in Haskell

```
quicksort :: (Ord a, NFData a) => [a] -> [a]
quicksort [] = []
quicksort [x] = [x]
quicksort (x:xs) = (left ++ (x:right))
        where
        left = quicksort [ y | y <- xs, y < x]
        right = quicksort [ y | y <- xs, y >= x]
```

A Comparison: QuickSort in Haskell

```
quicksort :: (Ord a, NFData a) => [a] -> [a]
quicksort [] = []
quicksort [x] = [x]
quicksort (x:xs) = (left ++ (x:right)) `using` strategy
        where
        left = quicksort [ y | y <- xs, y < x]
        right = quicksort [ y | y <- xs, y >= x]
        strategy result = rnf left `par`
                          rnf right `par`
                          rnf result
```

A Comparison: QuickSort in Haskell

```
quicksort :: (Ord a, NFData a) => [a] -> [a]
quicksort [] = []
quicksort [x] = [x]
quicksort (x:xs) = (left ++ (x:right)) `using` strategy
        where
        left = quicksort [ y | y <- xs, y < x]
        right = quicksort [ y | y <- xs, y >= x]
        strategy result = rnf left `par`
                          rnf right `par`
                          rnf result
```

More on high-level parallel programming next term in F21DP2 "Distributed and Parallel Systems"

Pipelines

- A *pipeline* is a sequence of operations, where the output of the n-th stage becomes input to the $n+1$-st stage.
- Each stage is typically a large, sequential computation.
- Parallelism is achieved by overlapping the computations of all stages.
- To communicate data between the stages a `BlockingCollection<T>` is used.
- This pattern is useful, if large computations work on many data items.

Pipelines

```
var buffer1 = new BlockingCollection<int>(limit);
var buffer2 = new BlockingCollection<int>(limit);

var f = new TaskFactory(TaskCreationOptions.LongRunning,
                        TaskContinuationOptions.None);
var task1 = f.StartNew(() =>
               Pipeline<int>.Producer(buffer1, m, n, inc));
var task2 = f.StartNew(() =>
               Pipeline<int>.Consumer(
                buffer1,
                new Pipeline<int>.ConsumerDelegate(x => x*x),
                buffer2));
var task3 = f.StartNew(() =>
                { result_str =
                   Pipeline<int>.LastConsumer(buffer2, str);
                });
Task.WaitAll(task1, task2, task3);
```

Pipelines: Producer Code

```
public static void Producer(BlockingCollection<T> output, ... ) {
  ...
  try {
    foreach (T item in ...) {
      output.Add(item);
    }
  } finally {
    output.CompleteAdding();
  }
}
```

Pipelines: Consumer Code

```
public static void Consumer(BlockingCollection<T> input,
                            ConsumerDelegate worker,
                            BlockingCollection<T> output) {
 try {
   foreach (var item in input.GetConsumingEnumerable()) {
      var result = worker(item);
      output.Add(result);
   }
  } finally {
      output.CompleteAdding();
  }
}
```

Selecting the Right Parallel Pattern

Application characteristic	Relevant pattern
Do you have *sequential loops* where there's no communication among the steps of each iteration?	The *Parallel Loop pattern*. Parallel loops apply an independent operation to multiple inputs simultaneously.

Selecting the Right Parallel Pattern

Application characteristic	Relevant pattern
Do you need to *summarize data* by applying some kind of combination operator? Do you have loops with steps that are not fully independent?	The *Parallel Aggregation pattern*. Parallel aggregation introduces special steps in the algorithm for merging partial results. This pattern expresses a reduction operation and includes map/reduce as one of its variations.

Selecting the Right Parallel Pattern

Application characteristic	Relevant pattern
Do you have *distinct operations* with well-defined control dependencies? Are these operations largely free of serializing dependencies?	The *Parallel Task pattern*. Parallel tasks allow you to establish parallel control flow in the style of fork and join.

Selecting the Right Parallel Pattern

Application characteristic	Relevant pattern
Does the ordering of steps in your algorithm depend on *data flow constraints*?	The *Futures pattern*. Futures make the data flow dependencies between tasks explicit. This pattern is also referred to as the Task Graph pattern.

Selecting the Right Parallel Pattern

Application characteristic	Relevant pattern
Does your algorithm *divide* the problem domain dynamically during the run? Do you operate on recursive data structures such as graphs?	The *Divide-and-Conquer pattern* (Dynamic Task Parallelism pattern). This pattern takes a divide-and-conquer approach and spawns new tasks on demand.

Selecting the Right Parallel Pattern

Application characteristic	Relevant pattern
Does your application perform a *sequence of operations* repetitively? Does the input data have streaming characteristics? Does the order of processing matter?	The *Pipelines pattern*. Pipelines consist of components that are connected by queues, in the style of producers and consumers. All the components run in parallel even though the order of inputs is respected.

Summary

- The preferred, high-level way of coding parallel computation in C# is through *parallel patterns*, an instance of design patterns.
- Parallel patterns capture common patterns of parallel computation.
- Two main classes of parallelism exist:
  - Data parallelism, which is implemented through parallel For/Foreach loops.
  - Task parallelism, which is implemented through parallel method invocation.
- Tuning the parallel performance often requires code restructuring (eg. thresholding).

Further Reading

Further reading:

- **"Parallel Programming with Microsoft .NET — Design Patterns for Decomposition and Coordination on Multicore Architectures", by C. Campbell, R. Johnson, A. Miller, S. Toub. Microsoft Press. August 2010.**
  `http://msdn.microsoft.com/en-us/library/ff963553.aspx`
- **"Patterns for Parallel Programming", by T. G. Mattson, B. A. Sanders, and B. L. Massingill. Addison-Wesley, 2004.**
- **"MapReduce: Simplified Data Processing on Large Clusters", J. Dean and S. Ghemawat. In *OSDI '04 — Symp. on Operating System Design and Implementation*, pages 137–150, 2004.** `http://labs.google.com/papers/mapreduce.html`

Advertisment

Next term: F21DP2 “Distributed and Parallel Systems”

In this course we will cover parallel programming in

- C+MPI: threads with explicit message passing
- OpenMP: data and (limited) task parallelism
- parallel Haskell: semi-explicit parallelism in a declarative language

Exercise

Produce a parallel implementation, testing the *"Goldbach conjecture":*

> ***Every even integer greater than 2 can be expressed as the sum of two primes.***

For details see:
`http://en.wikipedia.org/wiki/Goldbach%27s_conjecture`
A sample solution is available from the Sample C# source section of the course page.