

Asynchronous and parallel F# & Asynchronous and parallel C#

Peter Sestoft
BSWU 2016-03-17

Agenda

- Why is parallel programming important?
- CPU-bound parallelism in F# 3.0
- I/O-bound parallelism in F# 3.0

- Tasks in .NET 4.0, illustrated with C#
- Asynchronous programming in C# 4.5

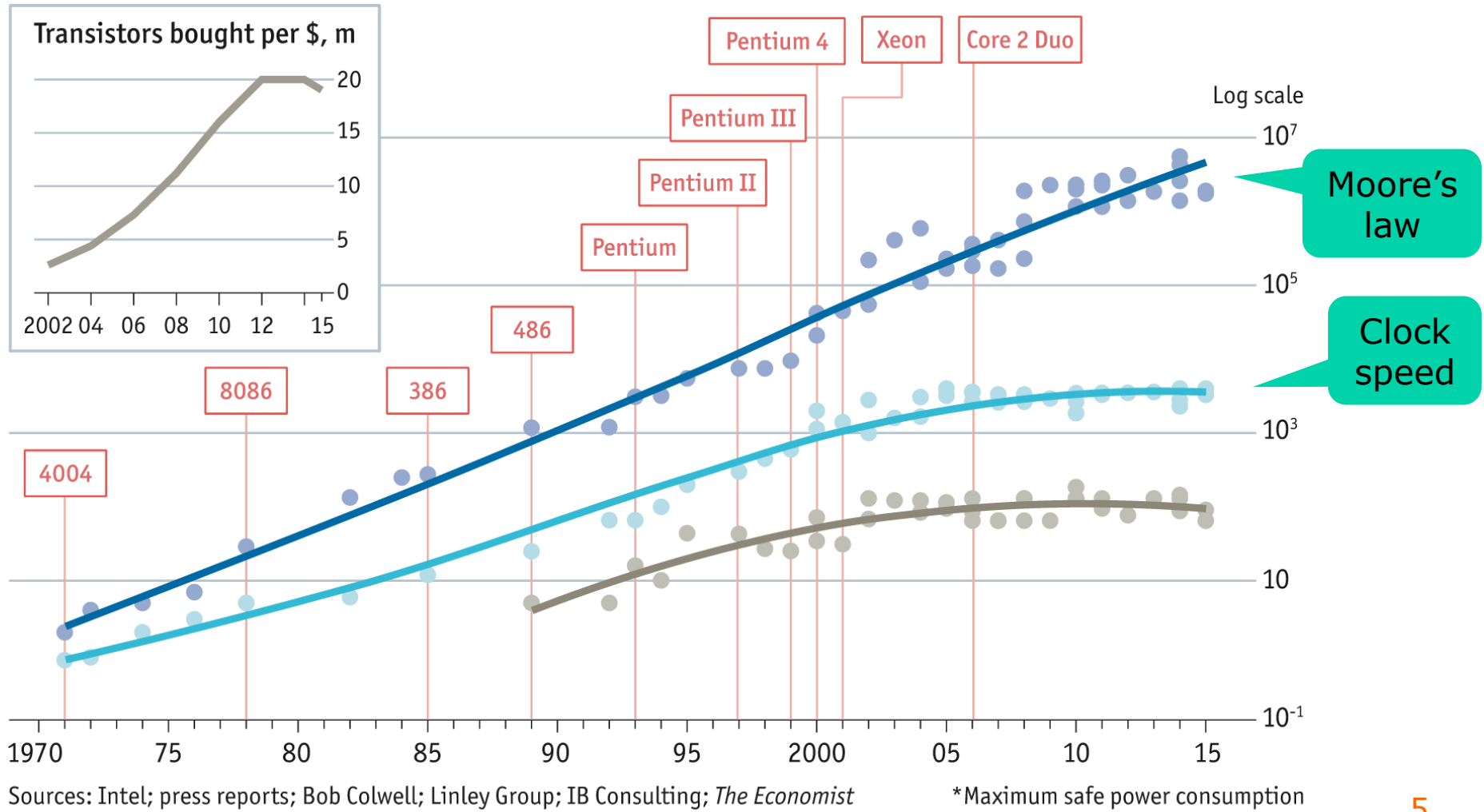
Why parallel programming?

- Until 2004, CPUs became faster every year
 - So sequential software became faster every year
- Today, CPUs clock speed < 3 GHz as in 2004
 - So *sequential* software has not become much faster
- Instead, we get
 - Multicore: 2, 4, 8, ... CPUs on a chip
 - Vector instructions (4 x MAC) built into CPUs
 - Super-parallel Graphics Processing Units (GPU)
 - 384 simple CUDA cores in this 2014 laptop
 - 2880 simple (single-precision) CUDA cores in Nvidia GK110
- Herb Sutter: *The free lunch is over (2005)*
- More speed requires **parallel programming**
 - But parallel programming is **difficult** and **errorprone**
 - ... with existing means: threads, synchronization, ...

The free lunch is over: No more growth in single-core speed

Stuttering

● Transistors per chip, '000 ● Clock speed (max), MHz ● Thermal design power*, w □ Chip introduction dates, selected



Economist, 11 March 2016

Why *functional* parallel programming?

- What is the purpose of synchronization?
 - To avoid conflicting **updates** of **shared data**
- Shared mutable data are
 - hard to understand and use correctly
 - inefficient on multicore machines
 - <http://www.itu.dk/people/sestoft/itu/PCPP/E2015/>
- Functional programming
 - **No** updates to shared data
 - Instead: copying, partial sharing, intermediate data structures, message passing, ...
- Some consensus this is the way forward
 - Erlang, Scala, F#, Java 8 streams, ...

CPU-bound parallel programming in F#

- A slow, CPU-consuming operation:

```
let rec slowfib n =  
    if n<2 then 1.0 else slowfib(n-1) + slowfib(n-2);;
```

```
let fib42 = slowfib(42);;  
// Real: 00:00:02.328, CPU: 00:00:02.328
```

F# 4.0
Mono 4.2.1
MacOS 10.9.5
Intel i7, 4 core

parallelfib.fs

- Computing two Fibonacci numbers:

```
let fibs = [ slowfib(41); slowfib(42) ];;  
// Real: 00:00:03.778, CPU: 00:00:03.778
```

- Doing it in parallel

Two tasks

```
let fibs =  
    let tasks = [ async { return slowfib(41) };  
                async { return slowfib(42) } ]  
    Async.RunSynchronously (Async.Parallel tasks);;  
// Real: 00:00:02.462 CPU: 00:00:03.970
```

More CPU-bound parallel programming in F#

- Computing 43 Fibonacci numbers:

```
let fibs = [ for i in 0..42 do yield slowfib(i) ];;  
// Real: 00:00:06.385, CPU: 00:00:06.386
```

- Doing it in parallel:

```
let fibs =  
    let tasks =  
        [ for i in 0..42 do yield async { return slowfib(i) } ]  
        Async.RunSynchronously (Async.Parallel tasks);;  
// Real: 00:00:02.662, CPU: 00:00:07.352
```

2.4 x faster on a 4-core machine

Dissecting the example

```
let tasks = [ for i in 0..42 do yield async { return slowfib(i) } ]  
Async.RunSynchronously (Async.Parallel tasks);;
```

```
async { return slowfib(i) }
```

Async<float>

An asynchronous task that will produce a float

```
let tasks =
```

Async<float> list

```
[ for i in 0..42 do yield async { return slowfib(i) } ]
```

List of asynchronous tasks that each will produce a float

```
Async.Parallel tasks
```

Async<float []>

An asynchronous task that will produce a float array

```
Async.RunSynchronously (Async.Parallel tasks)
```

float []

A float array

Asynchronous operations in F#

- An `async { ... }` expression produces an asynchronous task, `Async<t>`
 - In `return r` inside `{ ... }` the `r` must have type `t`
- `let! res = e` will run `e` and bind the result to `res` of type `u`, when `e` has type `Async<u>`
- `Async.RunSynchronously(asy)` will run computation `asy` and wait for its completion
- `Async.Parallel(asy)` creates a new asynchronous task that will run all `asy` and return an array of their results

Finding prime factors

- Prime factors of a number

```
factors 973475;;  
val it : int list = [5; 5; 23; 1693]
```

```
Array.init : int -> (int -> 'a) -> 'a []
```

- Prime factors of 0..200000

```
Array.init 200000 factors;;  
Real: 00:00:09.070, CPU: 00:00:09.072  
val it : int list [] =  
  [|[]; []; [2]; [3]; [2; 2]; [5]; [2; 3]; [7]; ... |]
```

- Same, in parallel, 4.6 x faster

```
let factors200000 = Array.Parallel.init 200000 factors;;  
Real: 00:00:01.948, CPU: 00:00:15.124  
val factors200000 : int list [] =  
  [|[]; []; [2]; [3]; [2; 2]; [5]; [2; 3]; [7]; ... |]
```

The number of prime factors

```
let histogram = Array.init 200000 (fun i -> 0)
let incr i = histogram.[i] <- histogram.[i] + 1
Array.iter (fun fs -> List.iter incr fs) factors200000;;
```

```
Real: 00:00:00.114, CPU: 00:00:00.115
```

```
val histogram : int [] =
  [|0; 0; 199988; 99996; 0; 49993; 0; 33330; 0; 0; 0;
    19997; 0; 16665; 0; 0; 0; 12498; 0; 11110; 0; 0; 0;
    9089; 0; 0; 0; 0; 0; 7141; 0; 6665; 0; 0; 0; 0; ... |]
```

- The heavy task, factorization, is parallelized
- The easy task, counting, is sequential
- Compare “imperative” C# version next slide

The same in C#

```
readonly int[] histogram = new int[200000];
static List<int> Factors(int n) {
    List<int> factors = new List<int>();
    int d = 2;
    while (n > 1) {
        if (n % d == 0) {
            factors.Add(d);
            lock (histogram)
                histogram[d]++;
            n /= d;
        } else
            d++;
    }
    return factors;
}
```

... but must lock
for atomic update

Tempting to
count while
factorizing

FactorsParallel.cs

```
List<int>[] factors200000 = new List<int>[200000];
Parallel.For(0, 200000,
    n => { factors200000[n] = Factors(n); });
```

- Exact same performance as parallel F#
 - But easy to forget locking => wrong results!!

Better: Functional parallel C#

- Parallelize the hard work:

```
var factors200000 =  
    Enumerable.Range(0, range)  
        .AsParallel()  
        .Select(i => Factors(i));
```

- Then do the fast work sequentially:

```
int[] histogram = new int[range];  
foreach (List<int> factors in factors200000)  
    foreach (int factor in factors)  
        histogram[factor]++;
```

- Learn functional programming (also)
to become better C# and Java developers
– Not least for parallel programming

More concurrency: I/O-bound parallel programming in F#

- Let us find the sizes of some homepages

```
let urls = ["http://www.itu.dk"; "http://www.diku.dk";  
           ...];;
```

```
let lengthSync (url : string) =  
  
    let wc = new WebClient()  
    let html = wc.DownloadString(Uri(url))  
  
    html.Length;;
```

```
lengthSync("http://www.diku.dk");;
```

```
[ for url in urls do yield lengthSync url];;
```

async.fs

Doing it in parallel, even with just 1 CPU

Not optimal

- Because *the webservers* work in parallel

```
let lens =  
  let tasks = [ for url in urls do yield async { return lengthSync url }  
  Async.RunSynchronously(Async.Parallel tasks);;
```

- Better: Let I/O system deal with responses:

```
let lengthAsync (url : string) =  
  async {  
    printf ">>>%s>>>\n" url  
    let wc = new WebClient()  
    let! html = wc.AsyncDownloadString(Uri(url))  
    printf "<<<%s<<<\n" url  
    return html.Length  
  };;
```

async.fs

```
let lens =  
  let tasks = [ for url in urls do yield lengthAsync url]  
  Async.RunSynchronously(Async.Parallel tasks);;
```

Why not `async { ... lengthSync ... }`?

- The thread will block while waiting for synchronous call `wc.DownloadString(...)`
- Better use `wc.AsyncDownloadString(...)` which is asynchronous
 - Will send a web request
 - Will release the calling thread
 - When a response arrives, it will continue computation (maybe on a different thread)
- So can have many more active requests than there are threads
 - Very bad to have more than 500 threads
 - But 50,000 async concurrent requests is fine

Parallel and asynchronous C#

- The `async { ... }` concept arose in F# 2.0
- The C# and .NET people adopted it
 - And changed it somewhat
- It is part of .NET 4.5 and C# 4.5

Reminder: C# delegates, lambdas

Types

```
delegate R Func<R>();  
delegate R Func<A1,R>(A1 x1);  
...  
delegate void Action();  
delegate void Action<A1>(A1 x1);  
...
```

```
unit -> R  
A1 -> R
```

```
unit -> unit  
A1 -> unit
```

Expressions

```
Func<int> fun1 = delegate() { return 42; };  
Func<int> fun2 = () => 42;  
Func<int,double> fun3 = x => x*Math.PI;  
int r1 = fun1() + fun2();  
double r2 = fun3(2);
```

```
Action act1 = delegate() { Console.WriteLine("Hello!"); };  
Action act2 = () => { Console.WriteLine("Hello!"); };  
Action<int> act3 = x => { r1 += x; };  
act1(); act2(); act3(42);
```

Parallel.For in .NET via C#

- Example: 50x50 matrix multiplication

```
for (int r=0; r<rRows; r++)  
    for (int c=0; c<rCols; c++) {  
        double sum = 0.0;  
        for (int k=0; k<aCols; k++)  
            sum += A[r,k]*B[k,c];  
        R[r,c] = sum;  
    }
```

Sequential,
5575 ms/mult

Example168.cs

```
Parallel.For(0, rRows, r => {  
    for (int c=0; c<rCols; c++) {  
        double sum = 0.0;  
        for (int k=0; k<aCols; k++)  
            sum += A[r,k]*B[k,c];  
        R[r,c] = sum;  
    }  
});
```

Parallel,
1800 ms/mult
4-core Xeon

What does Parallel.For do

```
Parallel.For(0, rRows, r => {  
    for (int c=0; c<rCols; c++) {  
        double sum = 0.0;  
        for (int k=0; k<aCols; k++)  
            sum += A[r,k]*B[k,c];  
        R[r,c] = sum;  
    }  
});
```

Delegate of
type
Action<int>

Parallel.For(m, n, body)
executes **body(m)**, **body(m+1)**, ..., **body(n-1)**
in some order, possibly concurrently

Parallel.Invoke

```
static double SlowFib(int n) { ... heavy job ... }
```

- Assume we need to compute this:

```
double result = SlowFib(40) * 3 + SlowFib(43);
```

- Use Invoke to compute in parallel:

```
double fib40 = 0.0, fib43 = 0.0;  
Parallel.Invoke(delegate { fib40 = SlowFib(40); },  
                delegate { fib43 = SlowFib(43); });  
double result = fib40 * 3 + fib43;
```

Example169.cs

- Sanity check: What is the best speed-up this can give?

Parallel.For for web access

- Get a protein's amino acid sequence from NCBI:

```
static String NcbiEntrez(String query) {
    byte[] bytes = new WebClient().DownloadData(new Uri(...));
    return ASCIIEncoding.ASCII.GetString(bytes);
}
static String NcbiProtein(String id) {
    return NcbiEntrez("efetch.fcgi?db=protein&id=" + id);
}
```

Example170.cs

- Get many proteins in parallel:

```
static String[] NcbiProteinParallel(params String[] ids) {
    String[] res = new String[ids.Length];
    Parallel.For(0, ids.Length,
        i => { res[i] = NcbiProtein(ids[i]); });
    return results;
}
```

This is thread-safe. Why?

Locking

- Try to put results into an array list (wrong):

```
ICollection<String> results = new List<String>();  
Parallel.For(0, ids.Length,  
    i => { String res = NcbiProtein(ids[i]);  
          results.Add(res);  
    });
```

Multiple concurrent updates, so wrong results

- Need to lock on the array list:

```
ICollection<String> results = new List<String>();  
Parallel.For(0, ids.Length,  
    i => { String res = NcbiProtein(ids[i]);  
          lock (results)  
            results.Add(res);  
    });
```

Why not inline `res` in the call:
`results.Add(NcbiProtein(ids[i]))?`

Asynchronous actions; GUI example

- Actions may block the GUI thread
 - Eg long-running computations
 - Eg access to network, disk, remote server
- Asynchronous actions avoid this problem

```
b1.Click += delegate(Object sender, EventArgs e)
{
    b1.Enabled = false;
    b1.Text = "(Computing)";
    Console.WriteLine("\nComputing SlowFib({0}) = ", n);
    double result = SlowFib(n++);
    Console.WriteLine(result);
    b1.Text = "Next Fib";
    b1.Enabled = true;
};
```

Gui.cs

General tasks for C# asynchrony

- Class Task
 - Asynchronous activity that returns no result
 - Typically created from an Action delegate
 - Executed on a *thread pool* by a *task scheduler*
 - ... so can execute many tasks on few threads
 - A *task* is not a *thread*
- Class Task<T> subclass of Task
 - Asynchronous activity that returns result of type T
 - Typically created from a Func<T> delegate
 - Called a *future* by Lisp and Java people

Operations on Task and Task<T>

- Task.Run(Action act)
 - started Task that executes `act()`
- Task.Run(Func<T> fun)
 - started Task<T> that executes `fun()`, gives its result
- Task.Delay(ms)
 - started task that delays for ms milliseconds
- t.Wait()
 - block until t is complete
- t.Result (when t is Task<T>)
 - block until t is complete and then return its result
- t.ContinueWith(Action<Task> cont)
 - task that executes `cont(t)` when t completes
- t.ContinueWith<U>(Func<Task,U> cont)
 - task that executes `cont(t,u)` when t completes with u

A task to compute SlowFib

Example171.cs

- Create Task<double> from delegate:

```
static Task<double> SlowFibTask(int n) {  
    return Task.Run(() => SlowFib(n));  
}
```

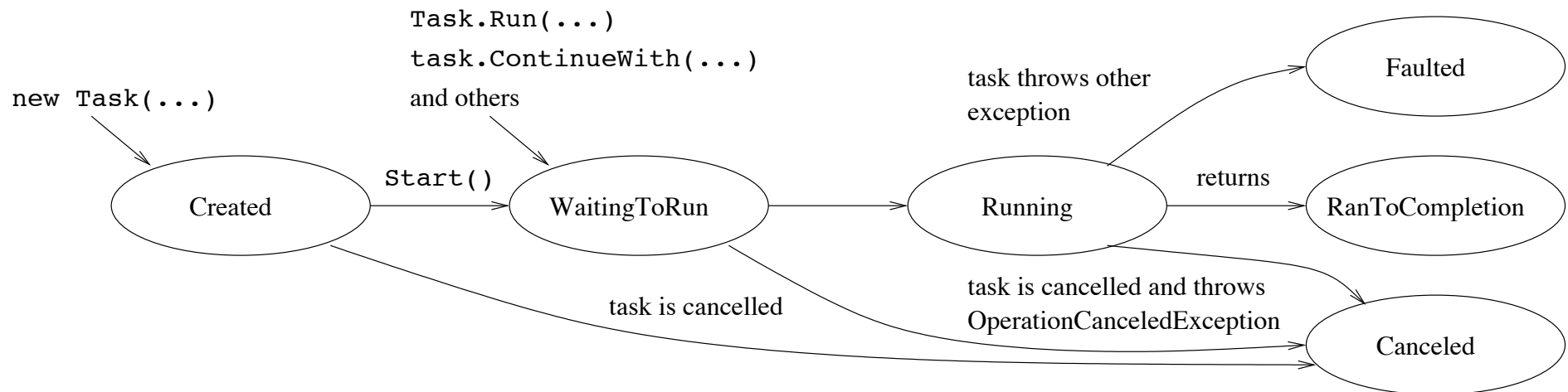
type Func<double>

- Returns a task, that when run, will compute SlowFib(n)
- How to use the task:

```
Task<double> task = SlowFibTask(n);  
... task may now be running ...  
Console.WriteLine(task.Result);
```

Task states (task.Status)

- RanToCompletion = terminated successfully
- Faulted = task threw exception
- Canceled = was cancelled, acknowledged it
- Completed = any of the above



Tasks for web access

- Read bytes, then convert to String:

```
static Task<String> NcbiEntrezTask(String query) {  
    return new WebClient().DownloadDataAsync(new Uri(...))  
        .ContinueWith((Task<byte[]> task) =>  
            ASCIIEncoding.ASCII.GetString(task.Result));  
}
```

Example172.cs

A continuation!

- The result of the method is a started task `t`
- The task performs the download asynchronously
- When the download completes,
 - the download task is bound to `task`
 - the `task.Result` byte array is transformed to a String and becomes the result of the task `t`

Much neater
on slide 38

```
static Task<String> NcbiProteinTask(String id) {  
    return NcbiEntrezTask("efetch.fcgi?...&db=protein&id="+id);  
}
```

Aggregate task operations (C# 4.5)

- `Task.WhenAll(params Task[] ts)`
 - task that completes when *all* of tasks `ts` complete (aka concurrency "barrier")
- `Task.WhenAll(params Task<T>[] ts)`
 - task that completes when all of `ts` complete, returning a `T[]` containing their results
- `Task.WhenAny(params Task[] ts)`
- `Task.WhenAny(params Task<T>[] ts)`
 - task that completes when *any* of the `ts` complete, returning one of the `ts` that completed

Tasks for parallel web access

- Get many proteins in parallel

```
static Task<String[]> NcbiProteinParallelTasks(String[] ids) {  
    IEnumerable<Task<String>> tasks  
        = from id in ids select NcbiProteinTask(id);  
    return Task.WhenAll(tasks);  
}
```

Example172.cs

- How to use it:

```
ShowResult(NcbiProteinParallelTasks("P01308", ...).Result);
```

```
>gi|124617|sp|P01308.1|INS_HUMAN RecName: Full=Insulin; ...  
MALWMRLLPLLALLLWGPDPAAAFVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAEDLQVGQVELGG  
GPGAGSLQPLALEGSLQKRGIVEQCCTSICSLYQLENYCN
```

```
>gi|12643972|sp|P01315.2|INS_PIG RecName: Full=Insulin; ...  
MALWTRLLPLL...
```

Implementing task timeouts

Example173.cs

- Use `WhenAny` to await task or a `Delay`:

```
static Task<double> SlowFibTimeoutTask(int n) {  
    Task<double> slow = SlowFibTask(n);  
    return Task.WhenAny(slow, TaskEx.Delay(1000))  
        .ContinueWith<double>((Task<Task> task) =>  
            task.Result==slow ? slow.Result : -1  
        );  
}
```

- When the `slow` task or the `Delay` completes so does the `WhenAny` task
- The `WhenAny` task gets bound to variable `task`
- The `task.Result` is the completed task
 - If the `slow` task completed, return its result
 - Otherwise the `slow` task timed out, return -1

Much neater
on slide 39

Task cancellation

- One cannot "kill", "stop" or "suspend" a task
- But one can *request* cancellation, and the task can check for and *acknowledge* (or not)

```
public static void ComputeTask(Cancellation token) {  
    for (int i=0; i<100000000; i++)  
        token.ThrowIfCancellationRequested();  
}
```

Check and
acknowledge

```
CancellationTokenSource cts = new CancellationTokenSource();  
Cancellation token = cts.Token;  
Task task = Task.Run(() => ComputeTask(token), token);
```

Running

```
cts.Cancel();  
// task.Wait();
```

Canceled

Would throw AggregateException
containing TaskCanceledException

Exceptions in tasks

- An exception `exn` thrown by a task is not propagated to the task's creator
- Instead
 - the task is moved to state `Faulted`
 - `t.Wait()` and `t.Result` will throw an `AggregateException` containing `exn`
 - `WaitAll` collects thrown exceptions from subtasks

Tasks versus threads

- A task is executed on a *task scheduler*
 - Typically many tasks run on a few threads
 - Because tasks may be blocked not on CPU work but input/output, GUI, net, GPU, ...
 - A task typically takes up few resources (just a representation of what to do when resumed)
- A *thread* might be used to represent a task
 - But a thread takes up many more resources
 - Each thread has a method call stack in the VM
 - Many threads slow down garbage collection
- The default task scheduler uses ThreadPool
 - and very clever work-stealing queues for tasks

Asynchronous methods (C# 4.5)

- Tasks allow compositional asynchrony
- But using `ContinueWith` gets rather hairy
- C# 4.5 has asynchronous methods
 - Declared using `async` keyword
 - Must return `Task` or `Task<T>` or `void`
 - May contain `await e` where `e` is a task
 - The rest of the method is the continuation of `e`
- Implementation of asynchronous method:
 - the compiler rewrites it to a state machine
 - much like `yield return` in iterator methods

Asynchronous web download

- Declare the method `async`
- Use `await` instead of `ContinueWith(...)`

```
static async Task<String> NcbiEntrezAsync(String query) {  
    byte[] bytes = await new WebClient().DownloadDataAsync(...);  
    return ASCIIEncoding.ASCII.GetString(bytes);  
}
```

- Use as before, or from other `async` methods:

```
static async Task<String> NcbiProteinAsync(String id) {  
    return await NcbiEntrezAsync("efetch.fcgi?...&id=" + id);  
}
```

```
static async Task<String[]> NcbiProteinParallelAsync(... ids) {  
    var tasks = from id in ids select NcbiProteinAsync(id);  
    return await Task.WhenAll(tasks);  
}
```

Timeout rewritten with async/await

- Much clearer than the ContinueWith version:

```
static async Task<double> SlowFibTimeoutAsync(int n) {  
    Task<double> slow = SlowFibTask(n);  
    Task completed = await Task.WhenAny(slow, Task.Delay(1000));  
    return completed == slow ? slow.Result : -1;  
}
```

Example173.cs

- Use as before ...

Composing asynchronous methods

- An NCBI PubMed query is done in two phases
 - First do an `esearch` to get a WebKey in XML
 - Then use `efetch` and the WebKey to get results
- To do this asynchronously using Task and ContinueWith would be quite convoluted
- Rather easy with asynchronous methods:

```
static async Task<String> NcbiPubmedAsync(String term) {  
    String search = String.Format("esearch.fcgi?... ", term);  
    XmlDocument xml = new XmlDocument();  
    xml.LoadXml(await NcbiEntrezAsync(search));  
    XmlNode node = xml["eSearchResult"];  
    String fetch = String.Format("...&db=Pubmed&WebEnv={1}", ...  
                                node["WebEnv"].InnerText);  
    return await NcbiEntrezAsync("efetch.fcgi?...&" + fetch);  
}
```

Example172.cs

Composability, general timeout

- Async methods can be further composed, eg
 - do all tasks asynchronously using WhenAll
 - do some task asynchronously using WhenAny
 - do task, subject to timeout
 - etc
- A general timeout task combinator

```
static async Task<T> Timeout<T>(Task<T> task, int ms, T alt) {  
    if (task == await Task.WhenAny(task, Task.Delay(ms)))  
        return task.Result;  
    else  
        return alt;  
}
```


Rules for C# asynchronous methods

- Cannot have out and ref parameters
- If the method's return type is Task
 - it can have no value-returning `return e; stmts.`
- If the method's return type is Task<T>
 - then all paths must have a `return e; stmt.` where `e` has type T
- In an `await e` expression,
 - if `e` has type Task then `await e` has no value
 - if `e` has type Task<T> then `await e` has type T

References

- The importance of parallel programming
 - Free Lunch is Over: <http://www.gotw.ca/publications/concurrency-ddj.htm>
- F# 3.0 asynchronous programming
 - <http://msdn.microsoft.com/en-us/library/dd233250.aspx> (Asynch Workfl)
 - <http://msdn.microsoft.com/en-us/library/ee353679.aspx> (WebClient)
 - <http://tomasp.net/blog/csharp-fsharp-async-intro.aspx>
 - http://en.wikibooks.org/wiki/F_Sharp_Programming/Async_Workflows
- F# parallel programming
 - <http://tomasp.net/blog/fsharp-parallel-samples.aspx>
 - <http://tomasp.net/blog/fsharp-parallel-plinq.aspx>
 - <http://tomasp.net/blog/fsharp-parallel-aggregate.aspx>
- C# parallel (4.0) and asynchronous (5.0) programming
 - Sestoft: C# Precisely 2nd ed chapters 22 and 23
 - Microsoft technical notes, see refs. in C# Precisely chapter 34
- John Hughes: *Why functional programming matters*, 1990
 - Erlang Factory talk, San Francisco, 11 March 2016
<https://www.youtube.com/watch?v=Z35Tt87pIpg>