

Parallelism I

COS 326

Speaker: Andrew Appel

Princeton University



Parallelism:

Doing many things at the same time
instead of one after the other.



UNDERSTANDING TECHNOLOGY TRENDS



Moore's Law

Moore's Law: *The number of transistors you can put on a computer chip doubles (approximately) every couple of years.*

Consequence for most of the history of computing: *All programs double in speed every couple of years.*

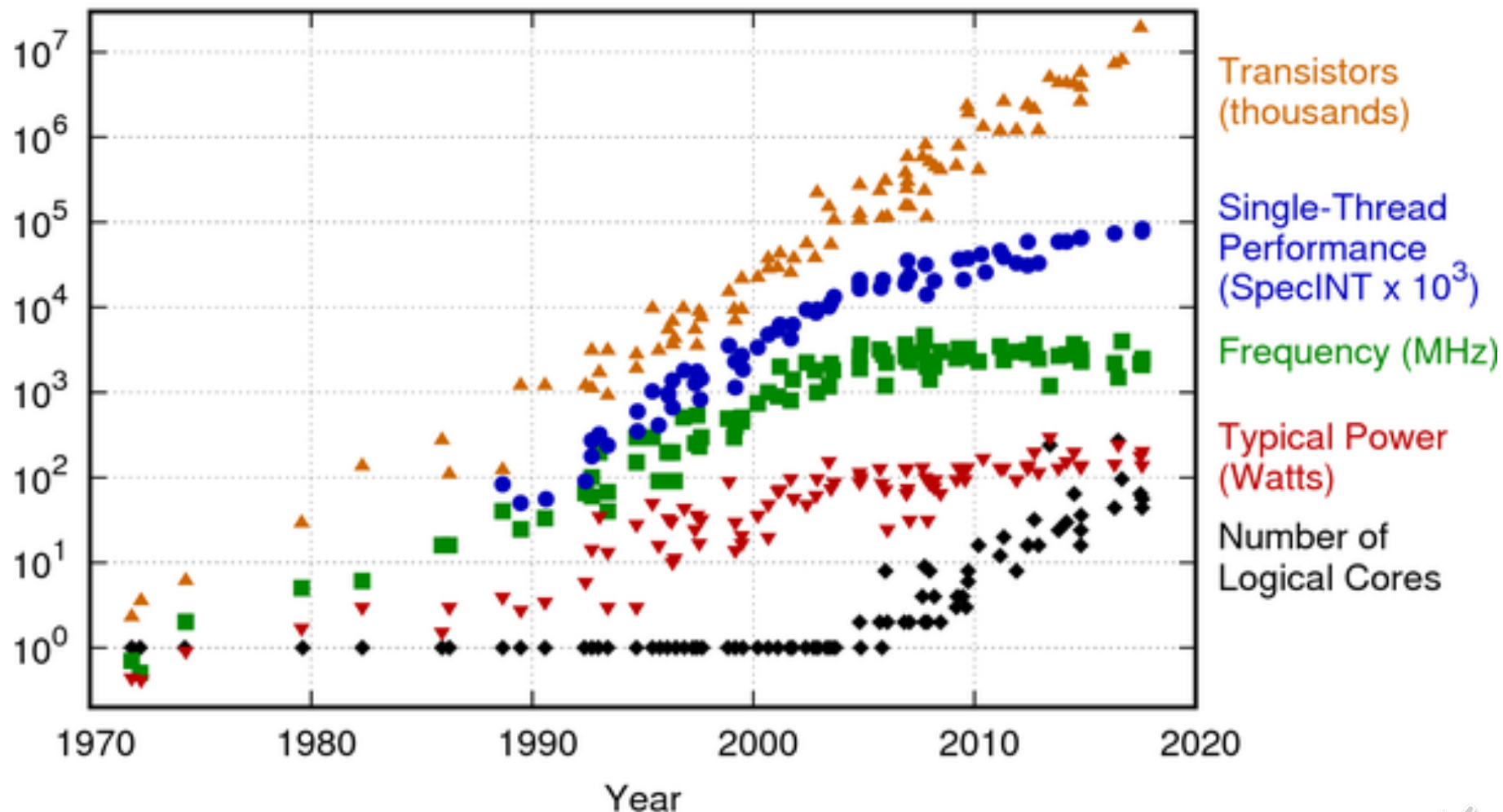
- Why? Hardware designers are wicked smart.
- They have been able to use those extra transistors to (for example) double the number of instructions executed per time unit, thereby processing speed of programs

Consequence for application writers:

- *watch TV for a while and your programs optimize themselves!*
- new applications thought impossible became possible because of increased computational power



42 Years of Microprocessor Trend Data



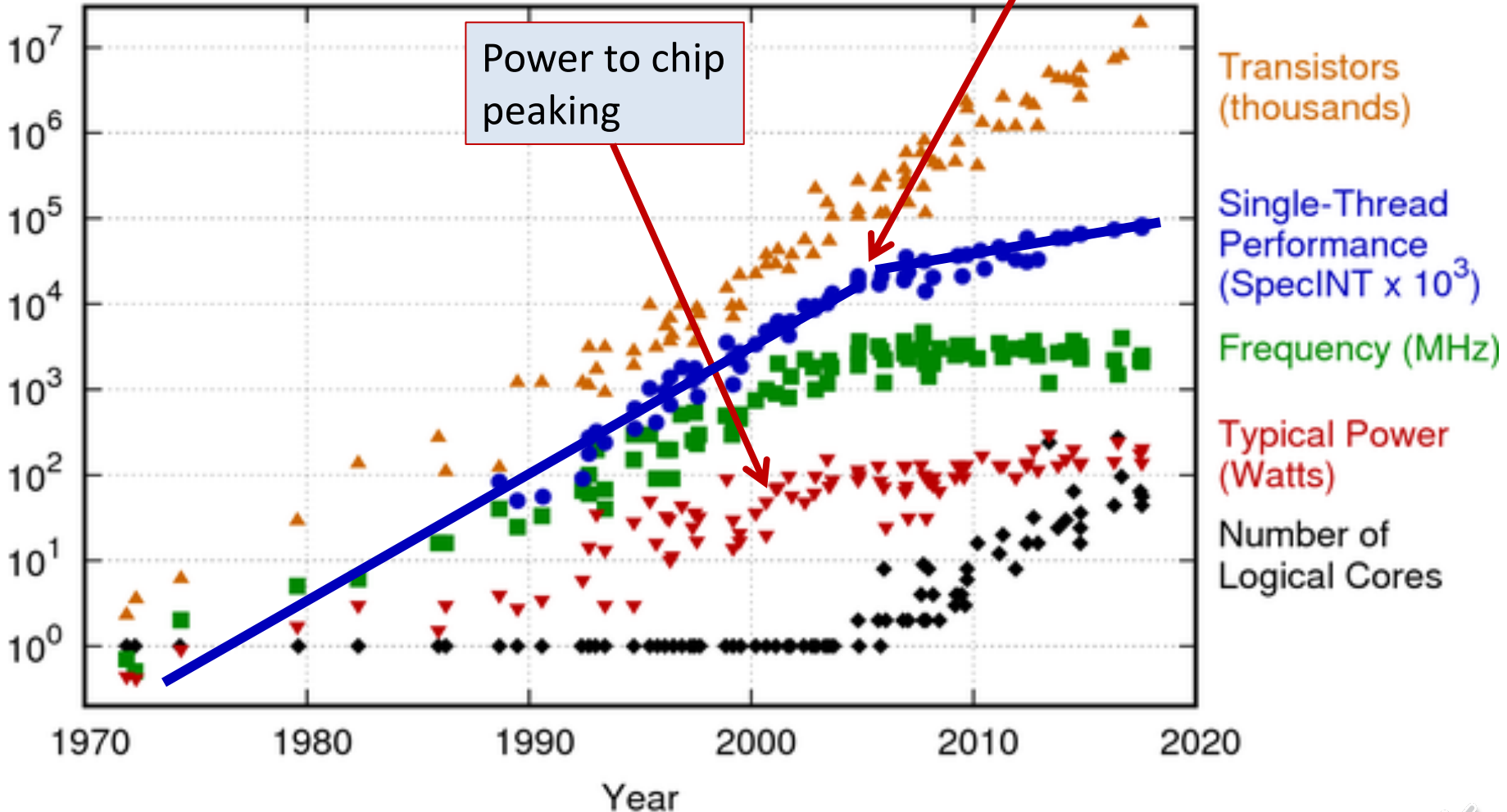
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp



42 Years of Microprocessor Trend Data

Darn! Intel engineers no longer optimize my programs while I watch TV!

Power to chip peaking



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp



Parallelism

Why is it particularly important (today)?

- Roughly every other year, a chip from Intel would:
 - halve the feature size (size of transistors, wires, etc.)
 - double the number of transistors
 - [until 2003] double the clock speed
 - this drove the economic engine of the IT industry (and the US!)
- No longer able to double clock or cut voltage: a processor won't get any faster!
 - (so why should you buy a new laptop, desktop, etc.?)
 - power and heat are limitations on the clock
 - errors, variability (noise) are limitations on the voltage
 - but we can still pack a lot of transistors on a chip... (at least for another 10 to 15 years.)



So...

Instead of trying to make your CPU go faster, Intel (and AMD, ARM, etc.) have been to packing more CPUs onto a chip.

- a few years ago: dual core (2 CPUs).
- a little more recently: 4, 6, 8 cores.
- Intel is testing 48-core chips with researchers now.
- Within 10 years, you'll have ~1024 Intel CPUs on a chip.

In fact, that's already happening with graphics chips (eg, Nvidia).

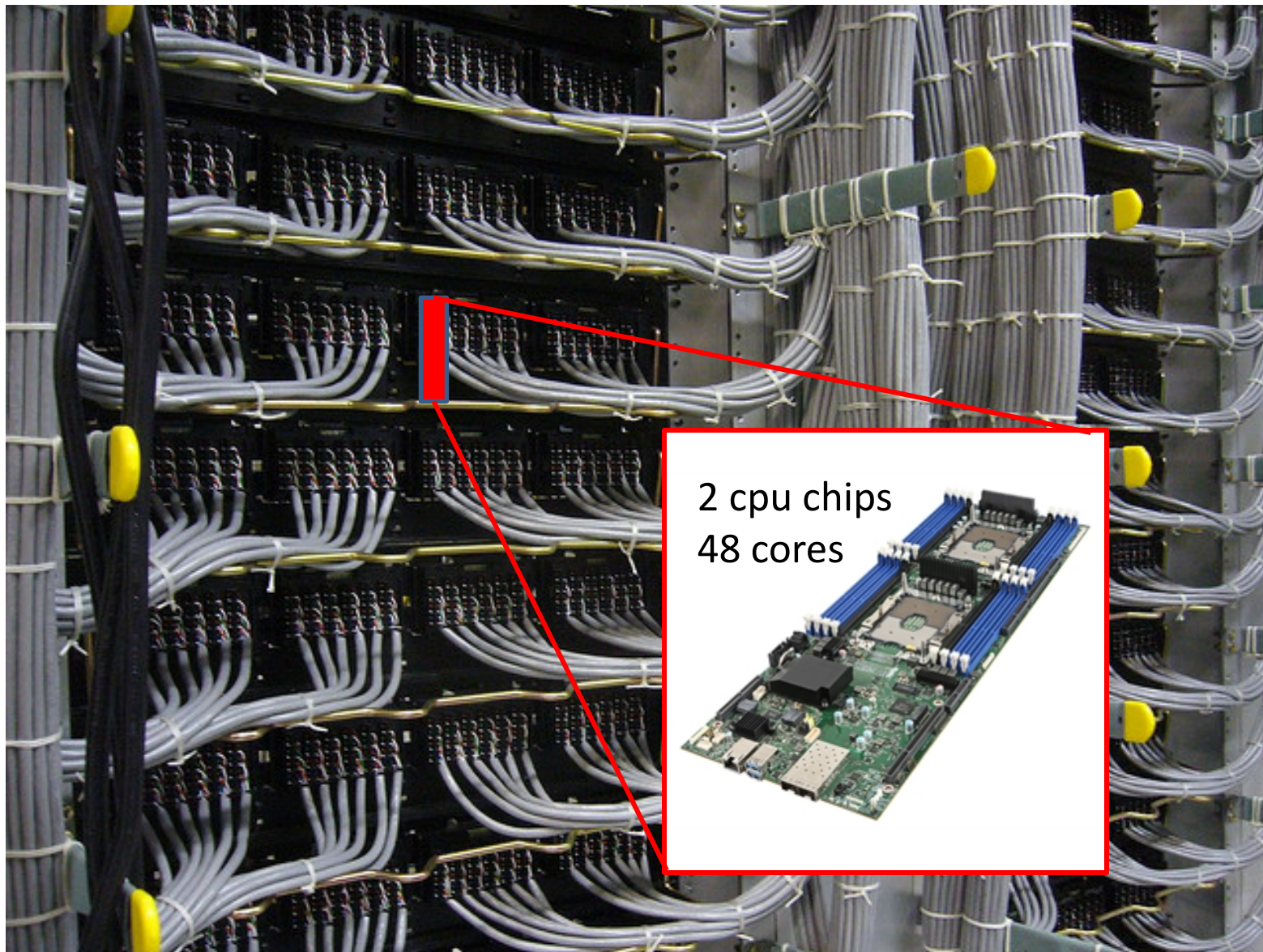
- really good at simple data parallelism (many deep pipes)
- but they are *much* dumber than an Intel core.
- and right now, chew up a *lot* of power.
- watch for GPUs to get “smarter” and more power efficient, while CPUs become more like GPUs.



But there's more: Data Centers



Data Centers: *Lots* of Connected Computers!



2 cpu chips
48 cores



Data Centers: Lots of Connected Computers

Computer containers for plug-and-play parallelism:



80,000 servers ?
x 20 cores/server ?
= 1.6 million cores?



How many servers?
Trade secret!
How does Microsoft
estimate how many
servers Google has?



Data Centers

10s or 100s of thousands of computers

All connected together

Motivated by new applications and scalable web services:

- let's catalogue all N billion webpages in the world
- let's allow anyone in the world to search for the page they need
- let's process that search in less than a second

It's Amazing!

It's Magic!



Sounds Great!

So my old programs will run 2x, 4x, 48x, 256x, 1024x faster?

No way!



Sounds Great!

So my old programs will run 2x, 4x, 48x, 256x, 1024x faster?

No way!

Single core ==> multicore means new algorithms

- *without work & thought, our programs don't get any faster at all*
- *it takes ingenuity to generate efficient parallel algorithms from sequential ones*



Unfortunately

Most parallel and concurrent programming models are far harder to work with than sequential ones:

- *They introduce nondeterminism*
 - the root of (almost all) evil
 - program parts suddenly have many different outcomes
 - they have different outcomes *on different runs*
 - debugging requires considering *all of the possible outcomes*
 - horrible *heisenbugs* hard to track down
- *They are nonmodular*
 - module A implicitly influences the outcomes of module B
- *They introduce new classes of errors*
 - race conditions, deadlocks
- *They introduce new performance/scalability problems*
 - busy-waiting, sequentialization, contention,

Fortunately

You are taking a class on functional programming ...



Fortunately

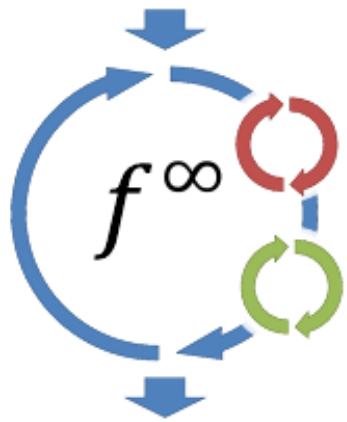
You are taking a class on functional programming ...

With a well designed parallel functional programming language (or library),

- Correctness is pretty easy to reason about because the parallel program has the same (purely functional) semantics as the sequential program
- Programs are deterministic
- Programs don't have race conditions
- But can still be tricky to reason about the *cost* of parallel functional programs



Frameworks for Parallel Functional Programming



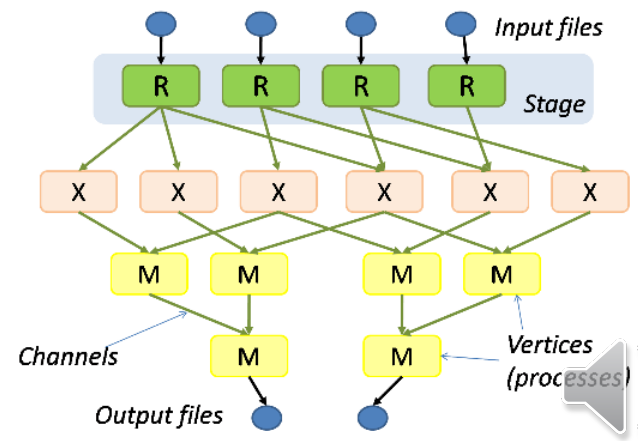
Naiad



Pig



Dryad



INTRODUCING PARALLELISM



Flavors of Parallelism

Data Parallelism

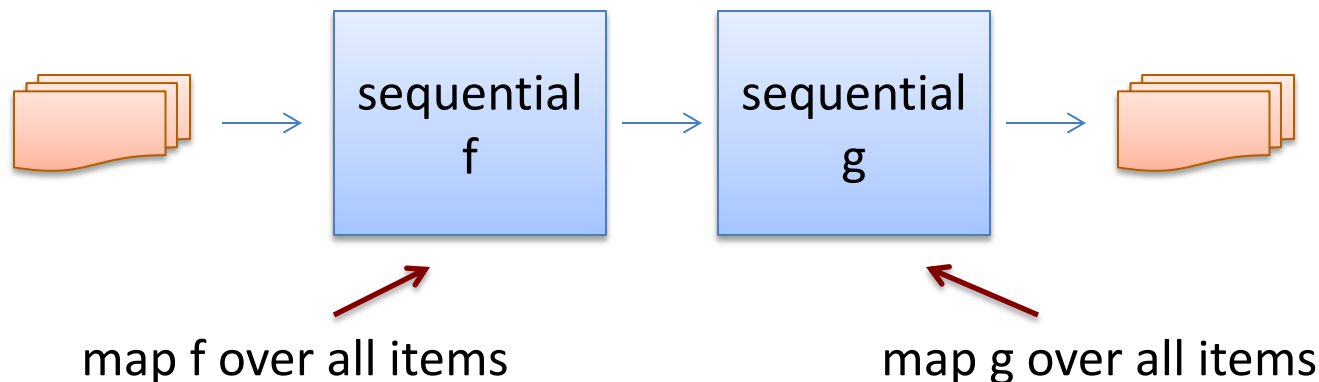
- same computation being performed on a *collection* of independent items
- e.g., adding two vectors of numbers

Task Parallelism

- different computations/programs running at the same time
- e.g., running web server and database

Pipeline Parallelism

- assembly line:



Parallelism vs. Concurrency

Parallelism: performs many tasks *simultaneously*

- **purpose:** improves throughput
- **mechanism:**
 - many independent computing devices
 - decrease run time of program by utilizing multiple cores or computers
- eg: running your web crawler on a cluster versus one machine.

Concurrency: mediates multiparty access to shared resources

- **purpose:** decrease response time
- **mechanism:**
 - switch between different threads of control
 - work on one thread when it can make useful progress; when it can't, suspend it and work on another thread
- eg: running your clock, editor, chat at the same time on a single CPU.
 - OS gives each of these programs a small time-slice (~10msec)
 - often *slows* throughput due to cost of switching contexts
- eg: don't block while waiting for I/O device to respond, but let another thread do useful CPU computation



**THREADS:
A CONVENTIONAL PARALLEL
PROGRAMMING MODEL**



Threads: A Warning

Concurrent Threads with Locks: the classic shoot-yourself-in-the-foot concurrent programming model

- all the classic error modes

Why Threads?

- almost all programming languages will have a threads library
 - OCaml in particular!
- you need to know where the pitfalls are
- the assembly language of concurrent programming paradigms
 - we'll use threads to build several higher-level programming models



Threads

A thread is an abstraction of a processor.

- programmer (or compiler) decides that some work can be done in parallel with some other work, e.g.:

```
let _ = compute_big_thing() in  
let y = compute_other_big_thing() in  
...
```

- we *fork* a thread to run the computation in parallel, e.g.:

```
let t = Thread.create compute_big_thing () in  
let y = compute_other_big_thing () in  
...
```

Intuition in Pictures

```
let t = Thread.create f () in  
let y = g () in  
...
```



processor 1

time 1 `Thread.create`
time 2 `execute g ()`
time 3 ...

processor 2

`(* doing nothing *)`
`execute f ()`
...



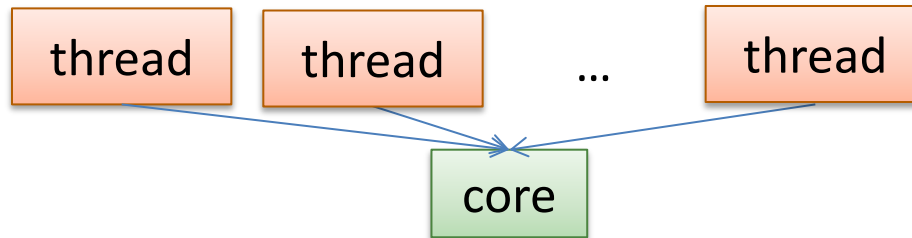
Of Course...

Suppose you have 2 available cores and you fork 4 threads. In a typical multi-threaded system,

- the operating system provides *the illusion* that there are an infinite number of processors.
 - not really: each thread consumes space, so if you fork too many threads the process will die.
- it *time-multiplexes* the threads across the available processors.
 - about every 10 msec, it stops the current thread on a processor, and switches to another thread.
 - so a thread is really a *virtual processor*.

OCaml, Concurrency and Parallelism

Unfortunately, even if your computer has 2, 4, 6, 8 cores, OCaml cannot exploit them. It multiplexes all threads over a single core



Hence, OCaml provides concurrency, but not parallelism. *Why?* Because OCaml (like Python) has no parallel “runtime system” or garbage collector. Other functional languages (Haskell, F#, ...) do.

Fortunately, when thinking about *program correctness*, it doesn't matter that OCaml is not parallel -- I will often pretend that it is.

You can hide I/O latency, do multiprocess programming or distribute tasks amongst multiple computers in OCaml.



Multicore Ocaml

<https://github.com/ocaml-multicore/ocaml-multicore>

A concurrent and shared-memory parallel extension of the OCaml compiler.

Multicore OCaml cleanly separates abstractions for concurrency (overlapped execution) from parallelism (simultaneous execution). Concurrency is expressed through [effect handlers](#) and parallelism through [domains](#). Much of the work in supporting shared memory parallelism is the development of a mostly-concurrent, generational, mark-and-sweep collector that strikes a balance between single-threaded performance and feature backwards compatibility, and multicore scalability.

OCaml Labs



Coordination

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t  
  
let t = Thread.create f () in  
let y = g () in  
...
```

How do we get back the result that `t` is computing?

First Attempt

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
  match !r with
    | Some v -> (* compute with v and y *)
    | None -> failwith "impossible"
```

What's wrong with this?



Second Attempt

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

Two Problems

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
    | Some v -> v
    | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

First, we are *busy-waiting*.

- consuming CPU without doing something useful.
- CPU could either be running a useful thread/program or power down.

Two Problems

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
let rec wait() =
  match !r with
  | Some v -> v
  | None -> wait()
in
let v = wait() in
  (* compute with v and y *)
```

Second, an operation like **`r := Some v`** may not be *atomic*.

- **`r := Some v`** requires us to copy the bytes of **`Some v`** into the **`ref r`**
- we might see part of the bytes (corresponding to **`Some`**) before we've written in the other parts (e.g., **`v`**).
- So the waiter might see the wrong value.

Atomicity

Consider the following:

```
let inc(r:int ref) = r := !r + 1
```

Atomicity

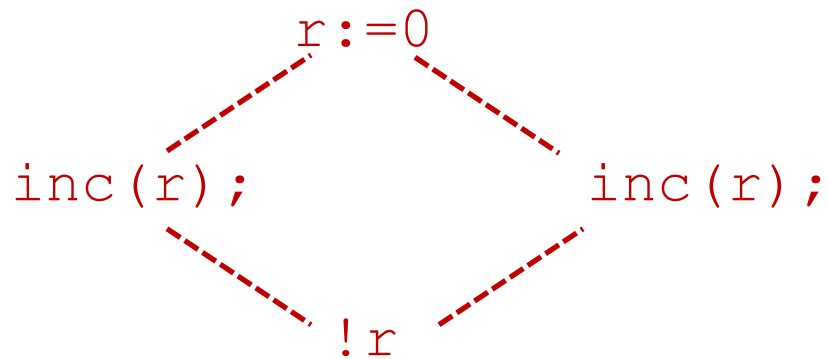
Consider the following:

```
let inc(r:int ref) = r := !r + 1
```

and suppose two threads are incrementing the same ref r:

Thread 1

Thread 2



If r initially holds 0, then what's the final value of r?

Atomicity

The problem is that we can't see exactly what instructions the compiler might produce to execute the code.

It might look like this:

Thread 1

$x_1 := \text{mem}[r]$

$x_1 := x_1 + 1$

$\text{mem}[r] := x_1$

Thread 2

$x_2 := \text{mem}[r]$

$x_2 := x_2 + 1$

$\text{mem}[r] := x_2$

Atomicity

Furthermore, we don't know the order in which instructions in thread 1 and thread 2 are executed.

On a single processor, they may be interleaved.

Thread 1

$x_1 := \text{mem}[r]$

$x_1 := x_1 + 1$

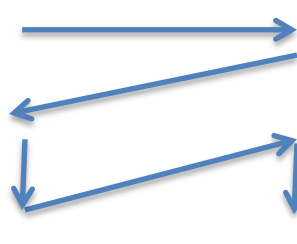
$\text{mem}[r] := x_1$

Thread 2

$x_2 := \text{mem}[r]$

$x_2 := x_2 + 1$

$\text{mem}[r] := x_2$



Atomicity

Furthermore, we don't know the order in which instructions in thread 1 and thread 2 are executed.

With this interleaving, what's the final value of r ?

Thread 1

$x_1 := \text{mem}[r]$

$x_1 := x_1 + 1$

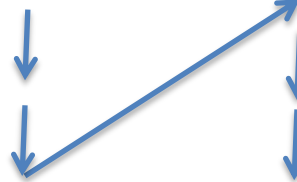
$\text{mem}[r] := x_1$

Thread 2

$x_2 := \text{mem}[r]$

$x_2 := x_2 + 1$

$\text{mem}[r] := x_2$



Atomicity

Furthermore, we don't know the order in which instructions in thread 1 and thread 2 are executed.

With this **interleaving**, what's the final value of r ?

Thread 1

$x_1 := \text{mem}[r]$

$x_1 := x_1 + 1$

$\text{mem}[r] := x_1$

Thread 2

$x_2 := \text{mem}[r]$

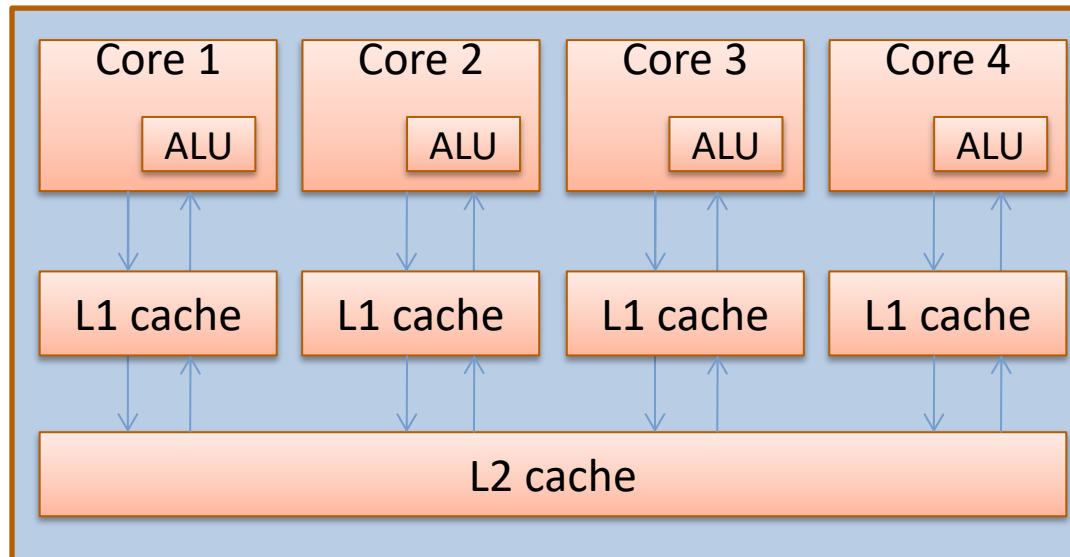
$x_2 := x_2 + 1$

When you see the word "interleaving", beware!

Real Machines

Today's multicore processors don't even have *sequentially consistent* memory models.

That means that we can't even assume that what we will see corresponds to *some* interleaving of the threads' instructions!



Beyond the scope of this course.

Synchronization

```
let inc(r:int ref) = r := !r + 1
```

thread 1

```
lock (s) ;  
inc r;  
unlock (s)
```

thread 2

```
lock (s) ;  
inc r;  
unlock (s)
```

The solution is to add *synchronization* instructions to ensure that certain program sequences behave as if atomic.

But programming with general locks is painful and error-prone.



Recall our Problem

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t  
  
let t = Thread.create f () in  
let y = g () in  
...
```

How do we get back the result that t is computing?

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
  Thread.join t ;
  match !r with
  | Some v -> (* compute with v and y *)
  | None -> failwith "impossible"
```

One Solution (using join)

```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
  Thread.join t ;
match !r with
| Some v -> (* compute with v and y *)
| None -> failwith "impossible"
```

Thread.join t causes the current thread to *wait* until the thread t terminates.

One Solution (using join)

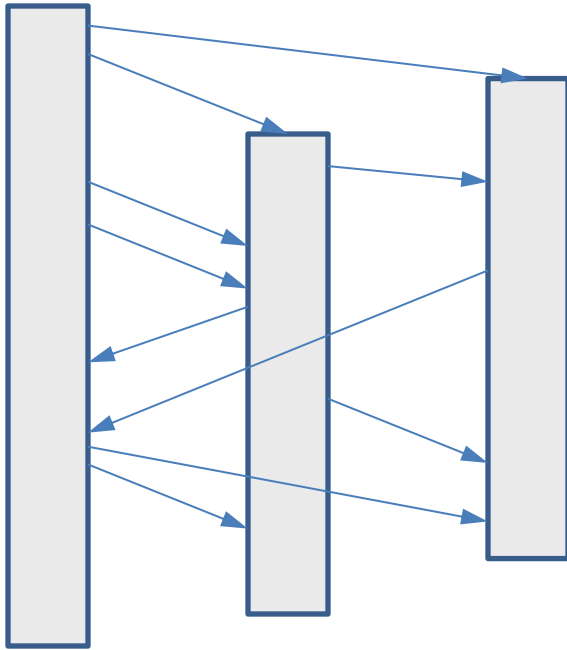
```
let r = ref None
let t = Thread.create (fun _ -> r := Some(f ())) in
let y = g() in
  Thread.join t ;
match !r with
| Some v -> (* compute with v and y *)
| None -> failwith "impossible"
```

Synchronization

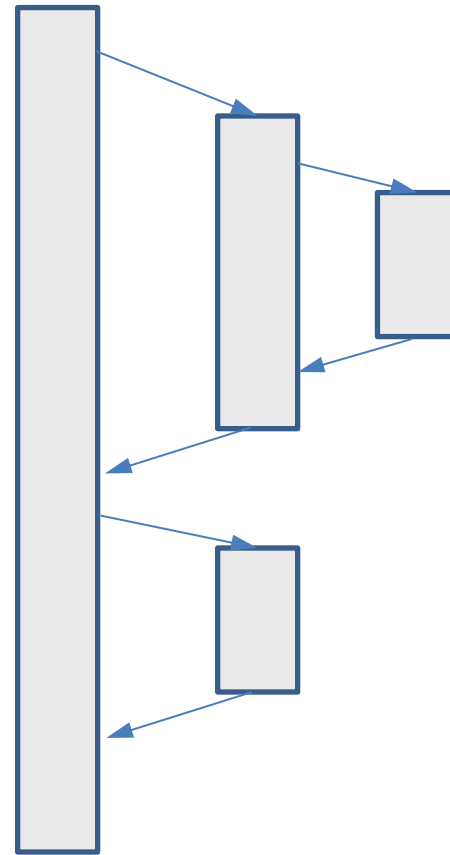
So after the join, we know that any of the operations of t have *completed*.

A special discipline for parallel computation

General synchronization
using locks

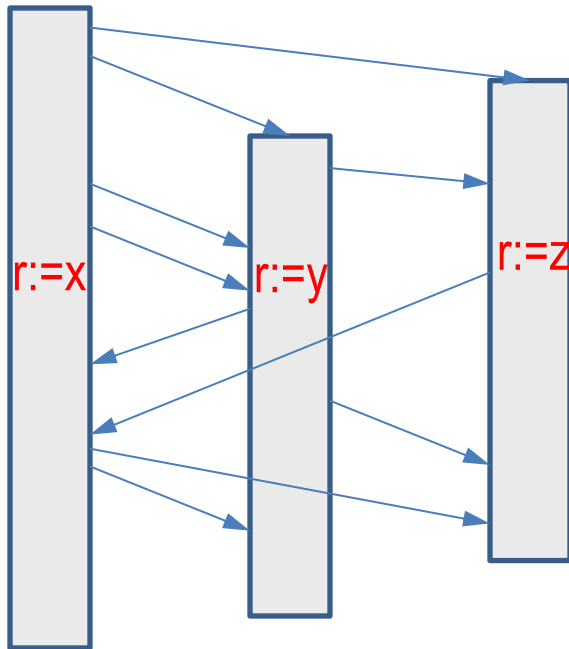


Fork-join parallelism

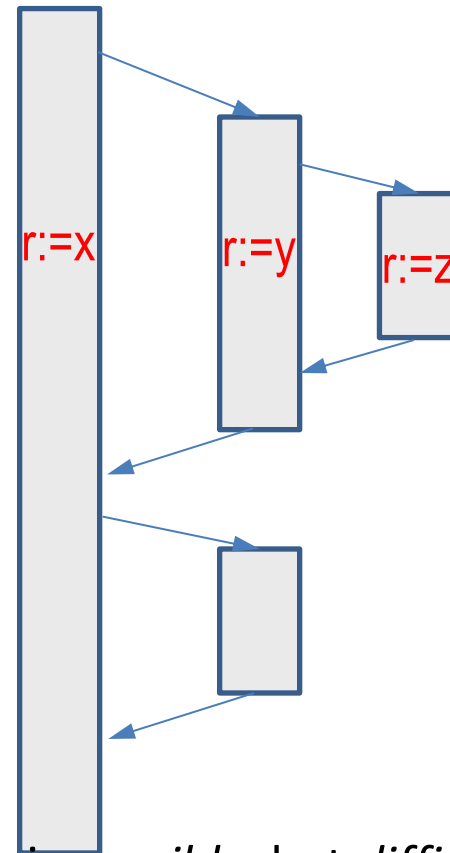


Imperative parallel programming

General synchronization
using locks



Fork-join parallelism



Programming with mutable shared variables is *possible*, but *difficult*, to get right! Can we avoid having to reason about this?



FUTURES: A PARALLEL PROGRAMMING ABSTRACTION



Futures

```
module type FUTURE =  
sig  
  type 'a future  
  
  (* future f x forks a thread to run f(x)  
     and stores the result in a future when complete *)  
  val future : ('a->'b) -> 'a -> 'b future  
  
  (* force f causes us to wait until the  
     thread computing the future value is done  
     and then returns its value. *)  
  val force : 'a future -> 'a  
end
```

Does that interface looks familiar ?



Future Implementation

```
module Future : FUTURE =  
struct  
  type 'a future = {tid    : Thread.t      ;  
                    value  : 'a option ref }
```

end



Future Implementation

```
module Future : FUTURE =  
struct  
  type `a future = {tid    : Thread.t          ;  
                    value : `a option ref }  
  
  let future(f:`a->`b) (x:`a) : `b future =  
    let r = ref None in  
    let t = Thread.create (fun () -> r := Some(f x)) ()  
    in  
    {tid=t ; value=r}  
  
end
```



Future Implementation

```
module Future : FUTURE =  
struct  
  type `a future = {tid    : Thread.t          ;  
                    value  : `a option ref }  
  
  let future(f:`a->`b)(x:`a) : `b future =  
    let r = ref None in  
    let t = Thread.create (fun () -> r := Some(f x)) ()  
    in  
    {tid=t ; value=r}  
  
  let force (f:`a future) : `a =  
    Thread.join f.tid ;  
    match !(f.value) with  
    | Some v -> v  
    | None -> failwith "impossible!"  
  
end
```



Now using Futures

```
let x = future f () in  
let y = g () in  
let v = force x in  
(* compute with v and y *)
```

Back to the Futures

```
module type FUTURE =  
sig  
  type 'a future  
  
  val future : ('a->'b) -> 'a -> 'b future  
  val force : 'a future -> 'a  
end
```

```
val f : unit -> int  
val g : unit -> int
```

with futures library:

```
let x = future f () in  
let y = g () in  
let v = force x in  
y + v
```

without futures library:

```
let r = ref None  
let t = Thread.create  
          (fun _ -> r := Some(f ()))  
          ()  
  
in  
let y = g() in  
Thread.join t ;  
match !r with  
  Some v -> y + v  
| None -> failwith "impossible"
```

Back to the Futures

```
module type FUTURE =  
sig  
  type 'a future  
  
  val future : ('a->'b) -> 'a -> 'b future  
  val force : 'a future -> 'a  
end
```

```
val f : unit -> int  
val g : unit -> int
```

with futures library:

```
let x = future f () in  
let y = g () in  
let v = force x in  
y + v x
```

without futures library:

```
let r = ref None  
let t = Thread.create  
    (fun _ -> r := Some(f ()))  
    ()  
  
in  
let y = g() in  
Thread.join t ;  
match !r with  
  Some v -> y + v  
| None -> failwith "impossible"
```

what happens if
we delete these
lines?

Back to the Futures

```
module type FUTURE =  
sig  
  type 'a future  
  
  val future : ('a->'b) -> 'a -> 'b future  
  val force : 'a future -> 'a  
end
```

```
val f : unit -> int  
val g : unit -> int
```

with futures library:

```
let x = future f () in  
let y = g () in  
let v = force x in  
y + x
```

Moral: Futures + typing ensure entire categories of errors can't happen -- you protect yourself from your own stupidity

without futures library:

```
let r = ref None  
let t = Thread.create  
    (fun _ -> r := Some(f ()))  
    ()  
  
in  
let y = g() in  
Thread.join t ;  
match !r with  
  Some v -> y + v  
| None -> failwith "impossible"
```

Back to the Futures

```
module type FUTURE =  
sig  
  type 'a future  
  
  val future : ('a->'b) -> 'a -> 'b future  
  val force : 'a future -> 'a  
end
```

```
val f : unit -> int  
val g : unit -> int
```

with futures library:

```
let x = future f () in  
let v = force x in  
let y = g () in  
y + v
```

what happens if you
relocate force, join?

without futures library:

```
let r = ref None  
let t = Thread.create  
          (fun _ -> r := Some(f ()))  
          ()  
  
in  
  Thread.join t ;  
  let y = g() in  
  match !r with  
  | Some v -> y + v  
  | None -> failwith "impossible"
```

Back to the Futures

```
module type FUTURE =
sig
  type 'a future

  val future : ('a->'b) -> 'a -> 'b future
  val force : 'a future -> 'a
end
```

```
val f : unit -> int
val g : unit -> int
```

with futures library:

```
let x = future f () in
let v = force x in
let y = g () in
y + x
```

without futures library:

```
let r = ref None
let t = Thread.create
      (fun _ -> r := Some(f ()))
      ()
in
Thread.join t ;
let y = g() in
match !r with
  Some v -> y + v
| None -> failwith "impossible"
```

Moral: Futures are
not a universal savior

An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
    | 0 -> Array.of_list []
    | 1 -> Array.make 1 arr.(start)
    | _ -> let half = len / 2 in
           let a1 = msort start half in
           let a2 = msort (start + half)
                        (len - half) in
           merge a1 a2

and merge (a1:'a array) (a2:'a array) : 'a array =
  ...
```



An Example: Mergesort on Arrays

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) :
    match len with
    | 0 -> Array.of_list []
    | 1 -> Array.make 1 arr.(start)
    | _ -> let half = len / 2 in
           let a1 = msort start half in
           let a2 = msort (start + half)
                       (len - half) in
           merge a1 a2

and merge (a1:'a array) (a2:'a array) : 'a array =
  ...
```

Opportunity for
parallelization



Making Mergesort Parallel

```
let mergesort (cmp:'a->'a->int)
              (arr : 'a array) : 'a array =
  let rec msort (start:int) (len:int) : 'a array =
    match len with
    | 0 -> Array.of_list []
    | 1 -> Array.make 1 arr.(start)
    | _ -> let half = len / 2 in
           let a1_f =
             Future.future (msort start) half in
           let a2 =
             msort (start + half) (len - half) in
           merge (Future.force a1_f) a2

and merge (a1:'a array) (a2:'a array) : 'a array =
```



Divide-and-Conquer

This is an instance of a basic *divide-and-conquer* pattern in parallel programming

- take the problem to be solved and divide it in half
- fork a thread to solve the first half
- simultaneously solve the second half
- synchronize with the thread we forked to get its results
- combine the two solution halves into a solution for the whole problem.

Warning: the fact that we only had to rewrite 2 lines of code for mergesort made the parallelization transformation look deceptively easy

- we also had to verify that any two threads did not touch overlapping portions of the array -- if they did we would have to again worry about scheduling nondeterminism

Caveats

There is some overhead for creating a thread.

- On uniprocessor, parallel code *slower* than sequential code.

Even on a multiprocessor, we do *not always* want to fork.

- when the subarray is small, faster to sort it sequentially than to fork
 - similar to using insertion sort when arrays are small vs. quicksort
- this is known as a *granularity problem*
 - more parallelism than we can effectively take advantage of.



Caveats

Optimizing requires benchmarking and experience

Typically, use parallel divide-and-conquer until:

- we have generated *at least* as many threads as there are processors
 - often *more threads* than processors because different jobs take different amounts of time to complete and we would like to keep all processors busy
- the sub-arrays have gotten small enough that it's not worth forking.

We're not going to worry about these performance-tuning details but rather focus on the distinctions between *parallel* and *sequential algorithms*.



Another Example

```
type 'a tree = Leaf | Node of 'a node
and 'a node = {left   : 'a tree ;
                value  : 'a      ;
                right  : 'a tree }
```



```
let rec fold (f:'a -> 'b -> 'b -> 'b) (u:'b)
            (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
    f n.value (fold f u n.left) (fold f u n.right)
```



```
let sum (t:int tree) = fold (+) 0 t
```

Another Example

```
type 'a tree = Leaf | Node of 'a node
```

```
and 'a node = {left   : 'a tree ;  
                value : 'a      ;  
                right  : 'a tree }
```

```
let rec pfold (f:'a -> 'b -> 'b -> 'b) (u:'b)  
          (t:'a tree) : 'b =
```

```
  match t with
```

```
  | Leaf -> u
```

```
  | Node n ->
```

```
    let l_f = Future.future (pfold f u) n.left in
```

```
    let r = pfold f u n.right in
```

```
    f n.value (Future.force l_f) r
```

```
let sum (t:int tree) = pfold (+) 0 t
```

Note

If the tree is unbalanced, then we're not going to get the same speedup as if it's balanced.

Consider the degenerate case of a list.

- The forked child will terminate without doing any useful work.
- So the parent is going to have to do all that work.
- Pure overhead... 😞

In general, lists are a horrible data structure for parallelism.

- *we can't cut the list in half in constant time*
- for arrays and trees, we can do that (assuming the tree is balanced.)

Side Effects?

```
type 'a tree = Leaf | Node of 'a node
and 'a node = { left   : 'a tree ;
                 value  : 'a      ;
                 right  : 'a tree }
```



```
let rec pfold (f:'a -> 'b -> 'b -> 'b) (u:'b)
            (t:'a tree) : 'b =
  match t with
  | Leaf -> u
  | Node n ->
    let l_f = Future.future (pfold f u) n.left in
    let r = pfold f u n.right in
    f n.value (Future.force l_f) r
```



```
let print (t:int tree) =
  pfold (fun n _ _ -> Printf.print "%d\n" n) ()
```



Huge Point

If code is purely functional, then it never matters in what order it is run.

If $f()$ and $g()$ are pure then all of the following are equivalent:

```
let x = f() in
let y = g() in
e
```

```
let x_f = future f () in
let y    = g ()      in
let x    = force x_f in
e
```

```
let y = g () in
let x = f () in
e
```

```
let y_g = future g () in
let x    = f ()      in
let y    = force y_g in
e
```

As soon as we introduce *side-effects*, the order starts to matter.

- This is why, IMHO, *imperative* languages where even the simplest of program phrases involves a side effect, are doomed.
- Of course, we've been saying this for 40 years!
- See J. Backus's Turing Award lecture, "*Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.*"

<http://www.cs.cmu.edu/~crary/819-f09/Backus78.pdf>



John W. Backus, 1924-2007

A pioneer in programming languages



Designer of FORTRAN,
inventor of Backus-Naur form for context-free grammars
co-designer of ALGOL,
winner of Turing award (1977).

Read his obituary at <https://www.nature.com/articles/446998a>
to see that the mid 20th century was really generous*
with second chances!

*if your family was “wealthy and socially prominent”



SUMMARY



Programming with mutation, threads and locks

Parallel programming can be really tricky!

Much of programming-language design is the art of finding structure that encourages good programs and discourages bad ones.

Functional programming, plus the *future* construct, is easier to reason about* ** ***

```
future : ('a->'b) -> 'a -> 'b future
force  : 'a future -> 'a
```

* in purely functional programs

** the type system helps (though you still have to put your force in the right places)

*** but **future** doesn't help you reason about parallel performance

