Parallel and Reactive Programming
Programmation parallèle et réactive

Albert Cohen, Marc Pouzet, Louis Mandel: cours
Basile Clément: TDs

INRIA and École Normale Supérieure
http://www.di.ens.fr/ParkasTeam.html

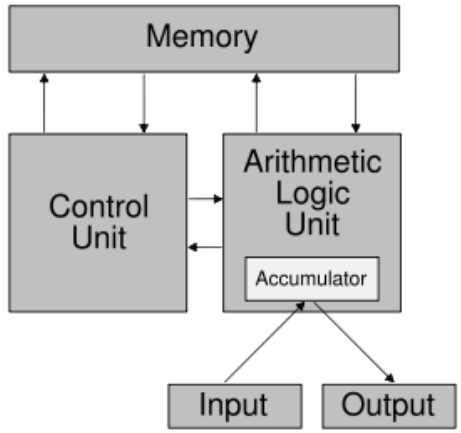
September–November, 2018

Course Material

https://www.di.ens.fr/~pouzet/cours/parallele_et_reactif

1. Parallel Programming: Why Should We Bother?

Stored Program, von Neumann Architecture



- IAS Architecture, John von Neumann

- SSEM "Baby", 1948: Tom Kilburn, Victoria U. of Manchester
  First implementation of the stored program concept in a real machine
- EDVAC, 1949: John Eckert, J. Presper Mauchly and John von Neumann

- Followed by 60 years of exponential growth of processor complexity, driven by Moore's Law

Evolutions of the von Neumann Architecture

What Can We Do With All These Transistors?

- Registers
- Cache (local memory, memory hierarchy)
- Instruction pipeline
- Branch predictor, prefetch, speculative execution
- Superscalar execution, out-of-order execution

- *Multi-thread processor*
- *Multi-processors, multi-core, many-core*
- *Specialization (hardware accelerators)*

Evolutions of the von Neumann Architecture

What Can We Do With All These Transistors?

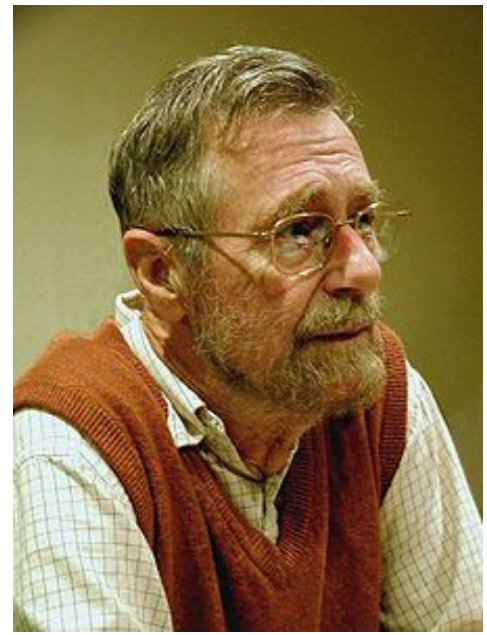
- Registers
- Cache (local memory, memory hierarchy)
- Instruction pipeline
- Branch predictor, prefetch, speculative execution
- Superscalar execution, out-of-order execution

- *Multi-thread processor*
- *Multi-processors, multi-core, many-core*
- *Specialization (hardware accelerators)*

- *Is it the end of the road for the von Neumann architecture?*

Algorithms, Programming Languages, and Multicore Processors?

1. Discrete Mathematics, Computing Science, Computer Science, Computer Engineering?

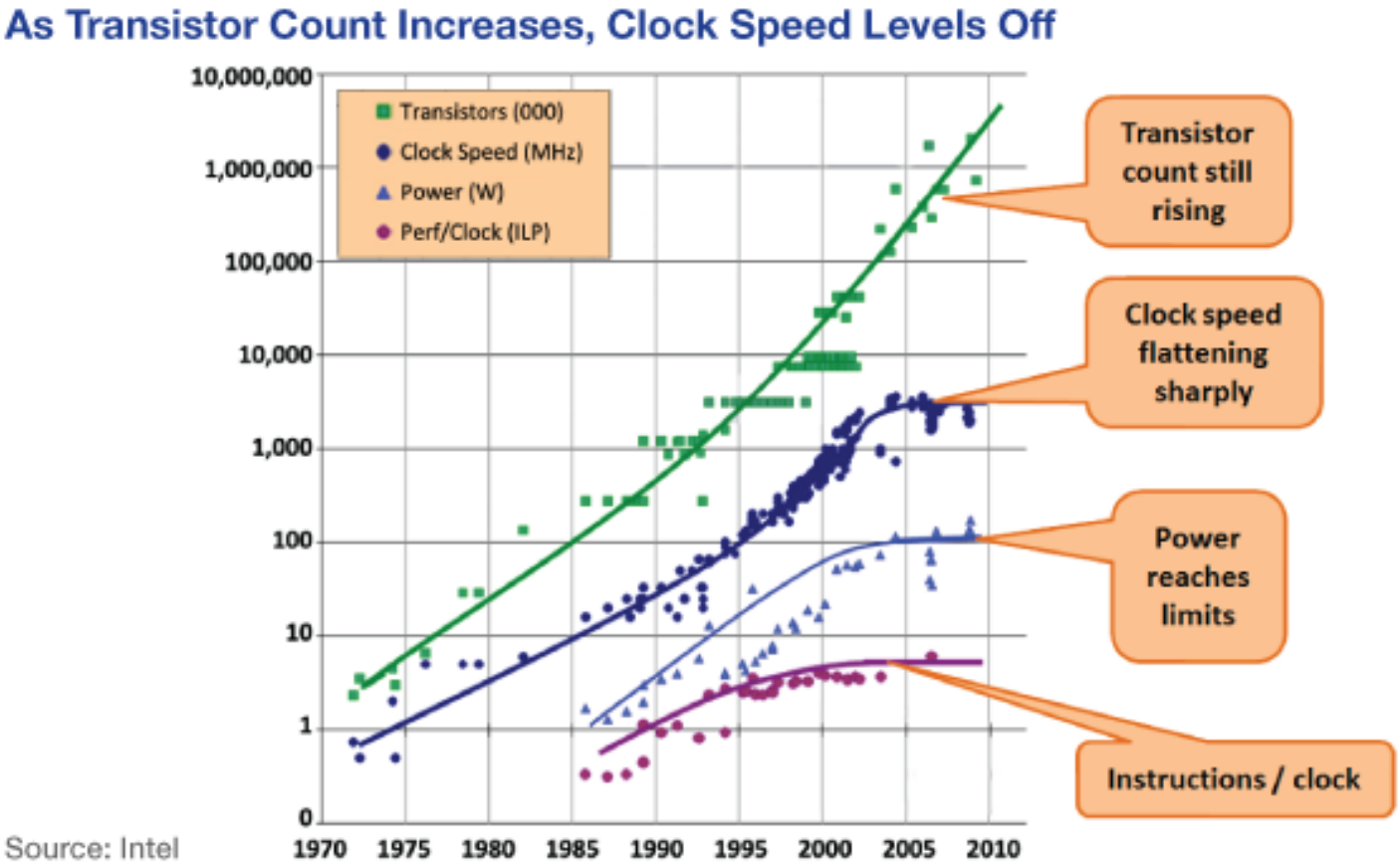


"Computer science is no more about computers than astronomy is about telescopes"

Edsger Dijkstra (1930–2002), Turing Award 1972

2. Let's talk about telescopes and astronomy: telescope builders and astromomers interact a lot, there must be a reason

Moore's Law Is Not Enough: Walls Everywhere



The "Memory Wall"

DDR3-2133 SDRAM

Latency: **10.3** ns

Memory bandwidth: **17.6** GB/s

4-core 2GHz ARM Cortex A15

Compute bandwidth: **2** × **4** threads × **1** NEON unit × **16** bytes × **2** GHz = **1024** GB/s

4-core 3GHz Intel Haswell

Compute bandwidth: **2** × **4** threads × **2** AVX units × **32** bytes × **3** GHz = **1536** GB/s

256-core 400MHz Kalray MPPA

Compute bandwidth: **2** × **256** threads × **2** words × **4** bytes × **400** MHz = **1638.4** GB/s

1536-core 1.006GHz NVIDIA Kepler

Compute bandwidth: **2** × **1536** threads × **1** float × **4** bytes × **1.006** GHz = **12361.6** GB/s
Memory bandwidth: **288** GB/s
PCIe bandwidth: **16** GB/s

Similar walls in the design of data-center and supercomputer networks

Impact on (Parallel) Program Performance

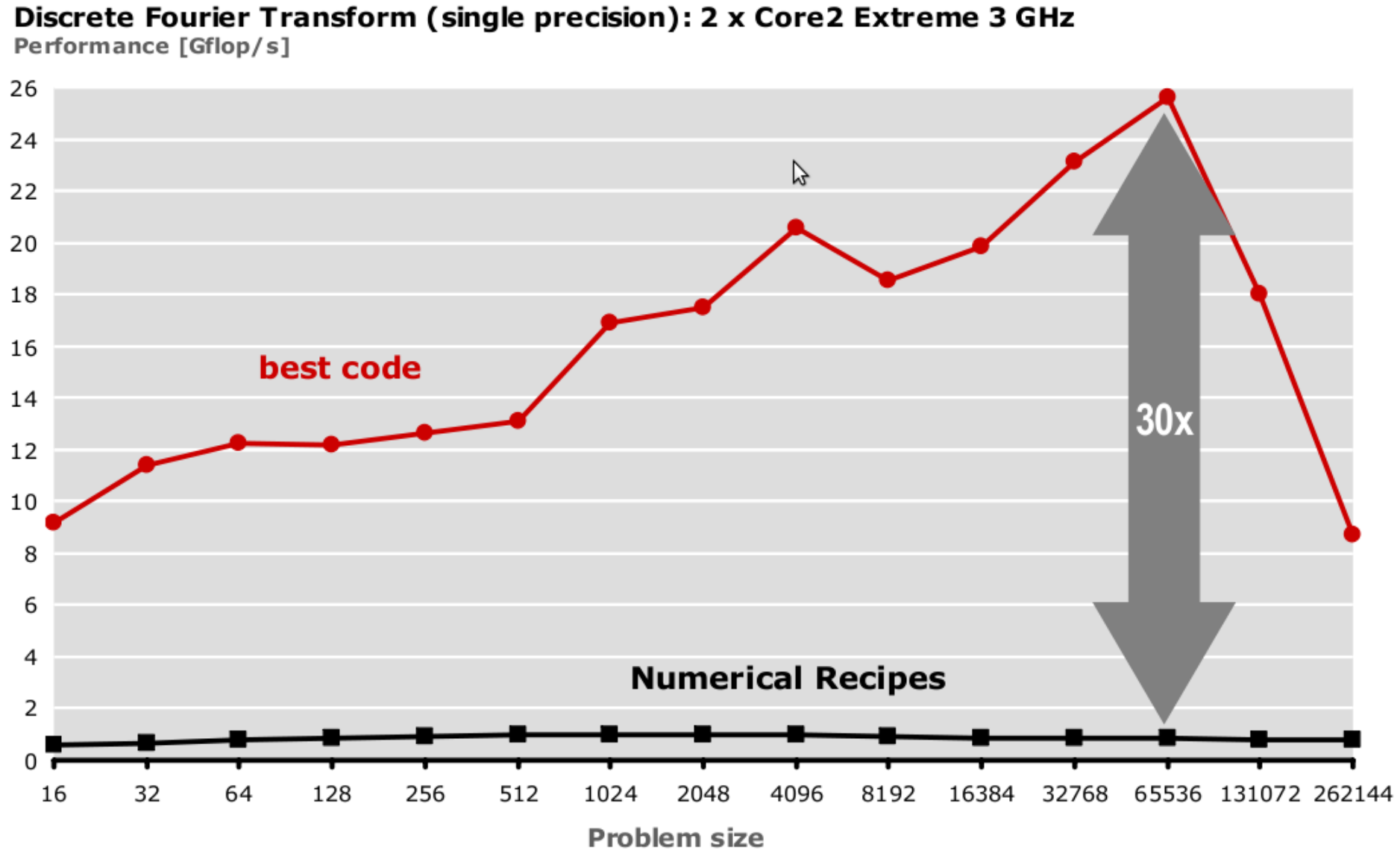
The memory and power walls are here to stay

No performance gains outside parallelism or specialization

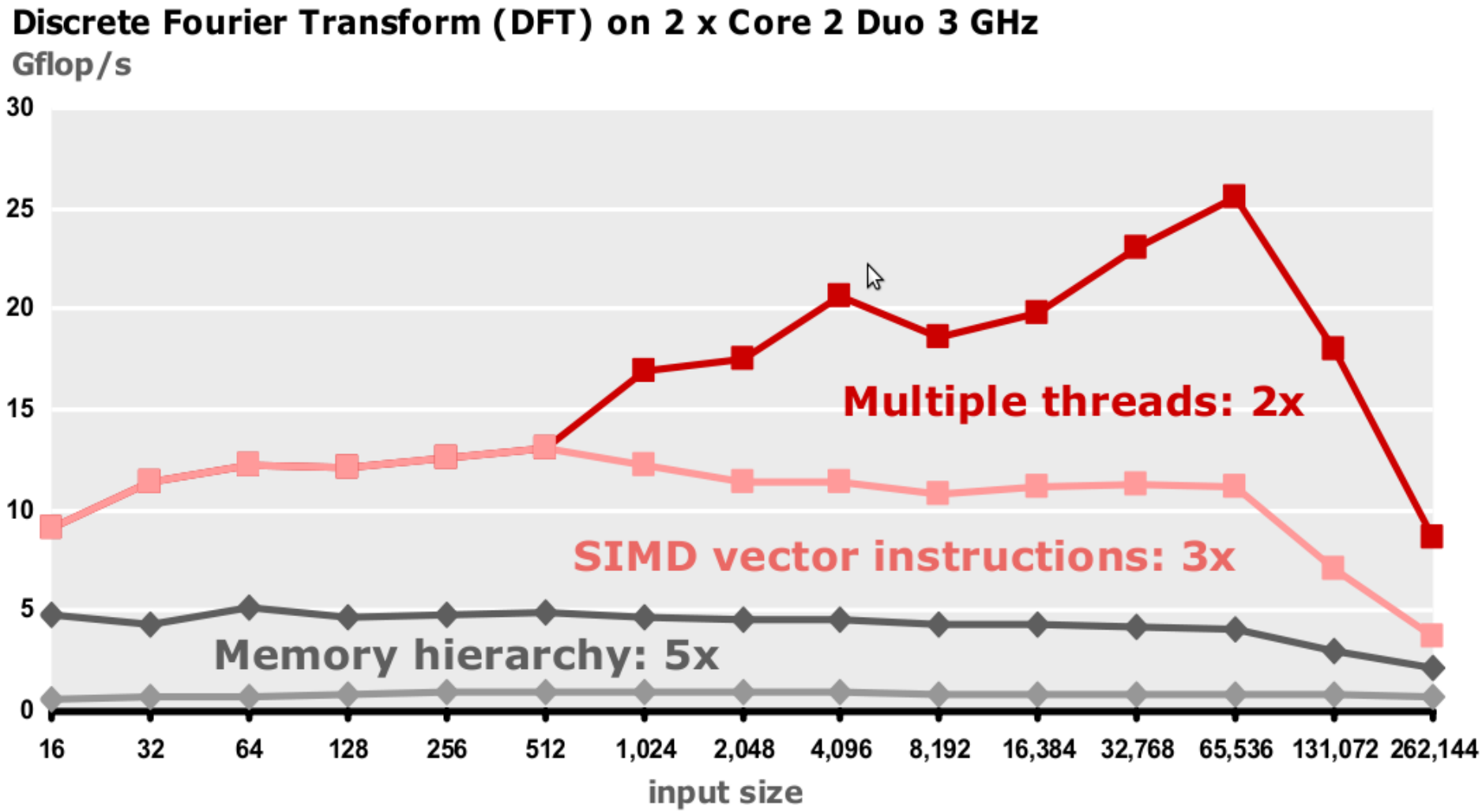
Parallel programming: to scale strongly or weakly?

Hardware and software design: to scale-up or to scale-out?

Example: Discrete Fourier Transform on 2 Dual-Core Processors



Example: Discrete Fourier Transform on 2 Dual-Core Processors



About Algorithms, Programming Languages

What are the essential semantic requirements for source programs?

Which programmers should really care about multicores, the memory wall?

What role for the software stack (compilers, runtime libraries)?

Shared-Memory Concurrency is Not for the Faint of Heart

```
void __lockfunc _##op##_lock(locktype##_t *lock)
{
        for (;;) {
                preempt_disable();
                if (likely(_raw_##op##_trylock(lock)))
                        break;
                preempt_enable();

                if (!(lock)->break_lock)
                        (lock)->break_lock = 1;
                while (!op##_can_lock(lock) && (lock)->break_lock)
                        _raw_##op##_relax(&lock->raw_lock);
        }
        (lock)->break_lock = 0;
}
```

excerpt from Linux spinlock.c

Shared-Memory Concurrency is Not for the Faint of Heart

```
void __lockfunc _##op##_lock(locktype##_t *lock)
{
        for (;;) {
                preempt_disable();
                if (likely(_raw_##op##_trylock(lock)))
                        break;
                preempt_enable();

                if (!(lock)->break_lock)
                        (lock)->break_lock = 1;
                while (!op##_can_lock(lock) && (lock)->break_lock)
                        _raw_##op##_relax(&lock->raw_lock);
        }
        (lock)->break_lock = 0;
}
```

excerpt from Linux spinlock.c

```
int steal(Deque *q) {
  size_t t = load_explicit(&q->top, acquire);
  thread_fence(seq_cst);
  size_t b = load_explicit(&q->bottom, acquire);
  int x = EMPTY;
  if (t < b) {
    /* Non-empty queue. */
    Array *a = load_explicit(&q->array, relaxed);
    x = load_explicit(&a->buffer[t % a->size], relaxed);
    if (!compare_exchange_strong_explicit(&q->top, &t, t + 1, seq_cst, relaxed))
      /* Failed race. */
      return ABORT;
  }
  return x;
}
```

Shared-Memory Concurrency is Not for the Faint of Heart

```
void __lockfunc _##op##_lock(locktype##_t *lock)
{
        for (;;) {
                preempt_disable()
                if (likely(_raw_#
                        break;
                preempt_enable();

                if (!(lock)->brea
                        (lock)->b
                while (!op##_can_
                        _raw_##op
        }
        (lock)->break_lock = 0;
}
```

excerpt from Linux spinlock.c

Lemma 3. *The following properties involving barriers apply:*

(i) $(\mathsf{W}x,\_ \xrightarrow{\text{sync}} \mathsf{W}y,\_ \xrightarrow{\text{pp-sat}} \mathsf{R}z,\_ \vee \mathsf{W}x,\_ \xrightarrow{\text{pp-sat}} \mathsf{R}y,\_ \xrightarrow{\text{sync}} \mathsf{R}z,\_) \implies \mathsf{W}x,\_ \xrightarrow{\text{pp-sat}} \mathsf{R}z,\_$

(ii) $A.\mathsf{W}x,\_ \xrightarrow{\text{rf}} B.\mathsf{R}x,\_ \xrightarrow{\text{sync}} B.\mathsf{W}y,\_ \xrightarrow{\text{pp-sat}} C.\mathsf{R}x,\_ \implies A.\mathsf{W}x,\_ \xrightarrow{\text{pp-sat}} C.\mathsf{R}x,\_$

(iii) *Let* X *stand for* $A.\mathsf{W}x,\_ \xrightarrow{\text{rf}} B.\mathsf{R}x,\_$ *or* $(A \sim B).\mathsf{W}x,\_$ *and* Y *stand for* $C.\mathsf{W}y,\_ \xrightarrow{\text{rf}} D.\mathsf{R}y,\_$ *or* $(C \sim D).\mathsf{W}y,\_$ *then the following holds:* $\neg(X \xrightarrow{\text{sync}} B.\mathsf{R}y,\_ \xrightarrow{\text{fr}} C.\mathsf{W}y,\_ \wedge Y \xrightarrow{\text{sync}} D.\mathsf{R}x,\_ \xrightarrow{\text{fr}} A.\mathsf{W}x,\_)$

```
int steal(Deque *q) {
  size_t t = load_explicit(&q->top, acquire);
  thread_fence(seq_cst);
  size_t b = load_explicit(&q->bottom, acquire);
  int x = EMPTY;
  if (t < b) {
    /* Non-empty queue. */
    Array *a = load_explicit(&q->array, relaxed);
    x = load_explicit(&a->buffer[t % a->size], relaxed);
    if (!compare_exchange_strong_explicit(&q->top, &t, t + 1, seq_cst, relaxed))
      /* Failed race. */
      return ABORT;
  }
  return x;
}
```

Correct and Efficient Parallel Programs?

- Between the hammer of programmability and the anvil of performance

- Productivity: a single source for all purposes
  - an abstract model for verification, formal reasoning
  - a concrete model for testing, simulation
  - the source from which sequential and parallel code can be generated, released, embedded...
  - guarantee strong properties of safety and efficiency at compile-time

- Safety and efficiency: rely on efficient, proven runtime execution primitives
  - safe and efficient memory management
  - lightweight scheduling
  - deterministic concurrency for dependent tasks, concurrent data structures, asynchronous I/O

Lessons for Parallel Programming

The need for parallelism in the machine code does not require the normal programmers to surrender decades of progress in the principles of programming languages and tools!

Runtime systems and compilers are responsible to translate portable, deterministic programming constructs into target-specific, high performance implementations

- 1^{st} **part** Rust programming, data parallelism
- 2^{nd} **part** Deterministic task-parallel programming
- 3^{nd} **part** Well-behaved concurrency (race freedom, deadlock freedom, compositionality)
- 4^{nd} **part** Beyond functions: I/O, distribution, hardware acceleration

2. Parallel Programming With Rust

Online Material: Reference Course on Rust Programming

Excellent course at U. Pennsylvania by David Mally, Terry Sun, Kai Ninomiya

`http://cis198-2016s.github.io`
`http://cis198-2016s.github.io/schedule`

Online Material: Documentation

Cargo	`http://doc.crates.io/guide.html`
Crates	`https://crates.io`
Docs	`http://docs.rs`
RustBook	`https://doc.rust-lang.org/book/second-edition`
RustByExample	`https://doc.rust-lang.org/stable/rust-by-example`
RustLang	`https://www.rust-lang.org`
StdDocs	`http://docs.rs/std`

Online Material: Videos and Documentation

Aaron Turon, Mozilla, overview of Rust (2015)
`https://www.youtube.com/watch?v=O5vzLKg7y-k`

Nicholas Matsakis, Mozilla, overview of Rayon (2017)
`https://www.youtube.com/watch?v=gof_OEv71Aw`
Rayon library
`https://docs.rs/crate/rayon/1.0.2`
and more on his slide deck
`https://speakerdeck.com/nikomatsakis`

Emily Dunham, Mozilla, Rust for developers, the community, environment (2017)
`https://www.youtube.com/watch?v=FMqydRampuo`

3. 1001 Threads

Concurrency vs. Parallelism

Vocabulary

Parallelism refers to the simultaneous or overlapping execution of concurrent operations

- Data parallelism: concurrency between (or parallel execution of) multiple instances of a single operation (on multiple inputs)
- Task parallelism: concurrency between (or parallel execution of) instances of different operations

Concurrency vs. Parallelism

Hardware Point of View

- Hardware Multi-Threading: interleaved or simultaneous, for latency hiding
- Flynn taxonomy: SIMD (vector), MISD (pipelining), MIMD
- NVidia's SIMT (CUDA)
- Levels of parallelism
- Grain of parallelism

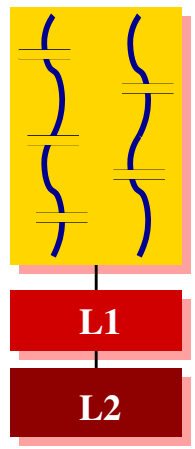
Programming Style

- SPMD work distribution (OpenMP, only possible style with MPI)
- Task parallelism (Cilk, OpenMP)
- Offloading (CUDA, OpenCL)
- Distributed/actors (Erlang)
- Skeletons (MapReduce)

Multi-Threaded Architectures

Simultaneous Multi-Threaded (SMT)

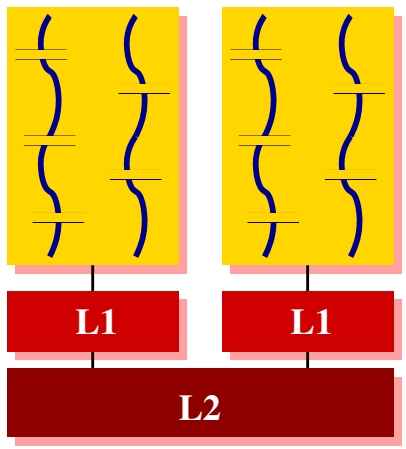
- A.k.a. hyper-threaded (Intel)



Cache-Coherent Multiprocessor Architectures

Chip Multi-Processor (CMP)

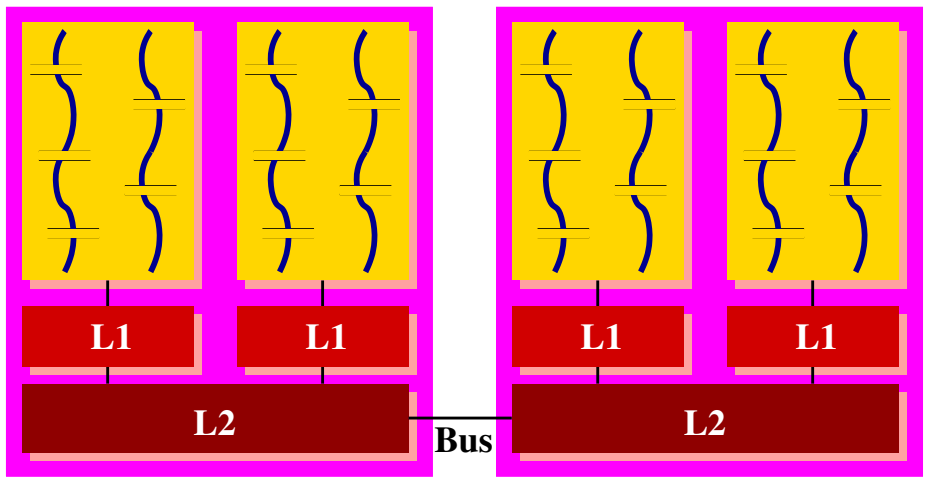
- A.k.a. multicore (Intel)
- Multiple cores interconnected one or more buses
- Snoopy caches: e.g., MESI protocol



Cache-Coherent Multiprocessor Architectures

Symmetric Multi-Processor (SMP)

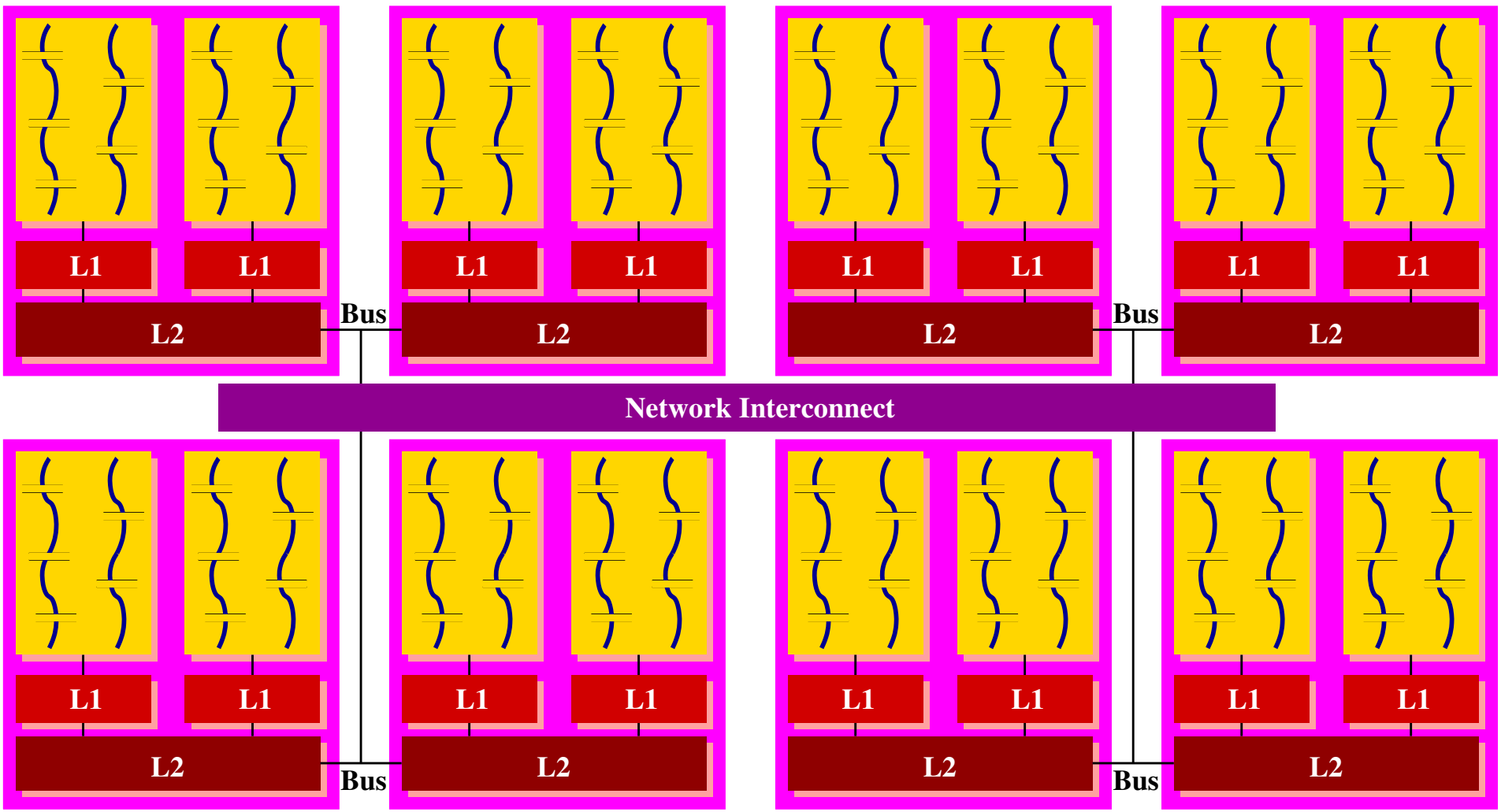
- Multiple chips interconnected one or more buses
- Snoopy caches



Cache-Coherent Multiprocessor Architectures

Non-Uniform Memory Architecture (NUMA)

- Multiple chips interconnected by a network
- Directory-based cache protocol



Logical Threads vs. Hardware Threads

Logical Thread Abstraction

Multiple concurrent execution contexts of the same program, cooperating over a single memory space, called shared address space (i.e., shared data, consistent memory addresses across all threads)

Among the different forms of logical thread abstrations, user-level threads do not need a processor/kernel context-switch to be scheduled

Mapping Logical to Hardware Threads

The hardware threads are generally exposed directly as operating system kernel threads (POSIX threads); these can serve as worker threads on which user-level threads can be mapped

Mapping strategies: one-to-one, many-to-one, many-to-many

Logical Threads vs. Hardware Threads

Thread "Weight"

1. Lightest: run-to-completion coroutines
   $\rightarrow$ indirect function call
2. Light: coroutines, fibers, protothreads, cooperative user-level threads
   $\rightarrow$ register checkpointing, garbage collector, cactus stacks with generalized activation records
3. Heavy: preemptive kernel threads (POSIX threads)
   $\rightarrow$ context switch
4. Heavier: kernel processes
   $\rightarrow$ heavier context switch with page table operations (TLB flush)

4. Mapping Logical Threads to Hardware Threads

Task Pool

General approach to schedule user-level threads

- Single task queue
- Split task queue for scalability and dynamic load balancing

More than one pool may be needed to separate ready threads from waiting/blocked threads

Task Pool: Single Task Queue

Simple and effective for small number of threads

Caveats:

- The single shared queue becomes the point of contention
- The time spent to access the queue may be significant as compared to the computation itself
- Limits the scalability of the parallel application
- Locality is missing all together

Task Pool: Split Task Queue

Work Sharing

Threads with more work push work to threads with less work A centralized scheduler balances the work between the threads

Work Stealing

A thread that runs out of work tries to steal work from some other thread

The Cilk Project

- Language for dynamic multithreaded applications
- Nested parallelism
- C dialect
- Developed since 1994 at MIT in the group of Charles Leiserson

`http://supertech.csail.mit.edu/cilk`
Now part of Intel Parallel Studio (and TBB, ArBB)

Fibonacci in Cilk

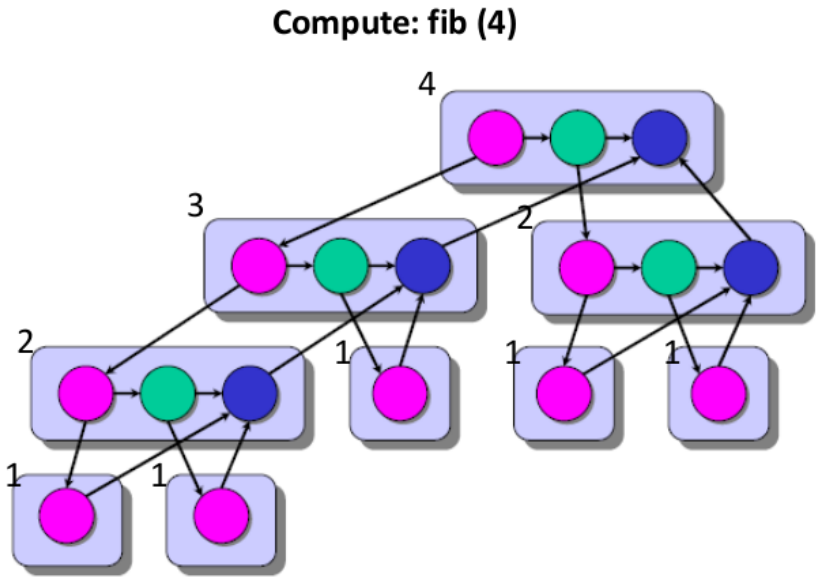
- Tasks are coroutines
- Nested, fork-join parallelism
- Two keywords to implement "parallel pairs"
  - `spawn` *`function`*`()` to indicate that the function call may be executed as a coroutine
  - `sync` to implement a join synchronization, waiting for all child tasks of the current task

```
cilk int fib (int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}
```

Cilk Properties

Cilk programs are canonically sequentialized with the elision of the special keywords
→ Depth-first execution of the task tree by a single-thread

```
cilk fib (int n) {
 if (n < 2) return 1;
   else {
      int rst = 0;
      rst += spawn fib (n-1);
      rst += spawn fib (n-2);
      sync;
      return rst;
   }
}
```



Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

→ As a corollary, all inputs to a task are available at the task creation point
→ A property called strictness (some relation to strictness in functional languages)

Fork-Join Task Parallelism in Rust

- https://docs.rs/rayon/1.0.2/rayon
- https://github.com/nikomatsakis/rayon/blob/master/README.md

Cilk Performance Properties

- Strictness simplifies the runtime support: no need to wait for data availability, it is in the activation record of the task already
  - Implementation as a coroutine
  - `spawn` 3 to 4 times more expensive than an average function call
  - Scheduling can be more expensive: needs a concurrent deque (double-ended queue) and cactus stacks

Fibonacci in Cilk: Generated Code

```
cilk int fib (int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}
```

Fast/sequential clone

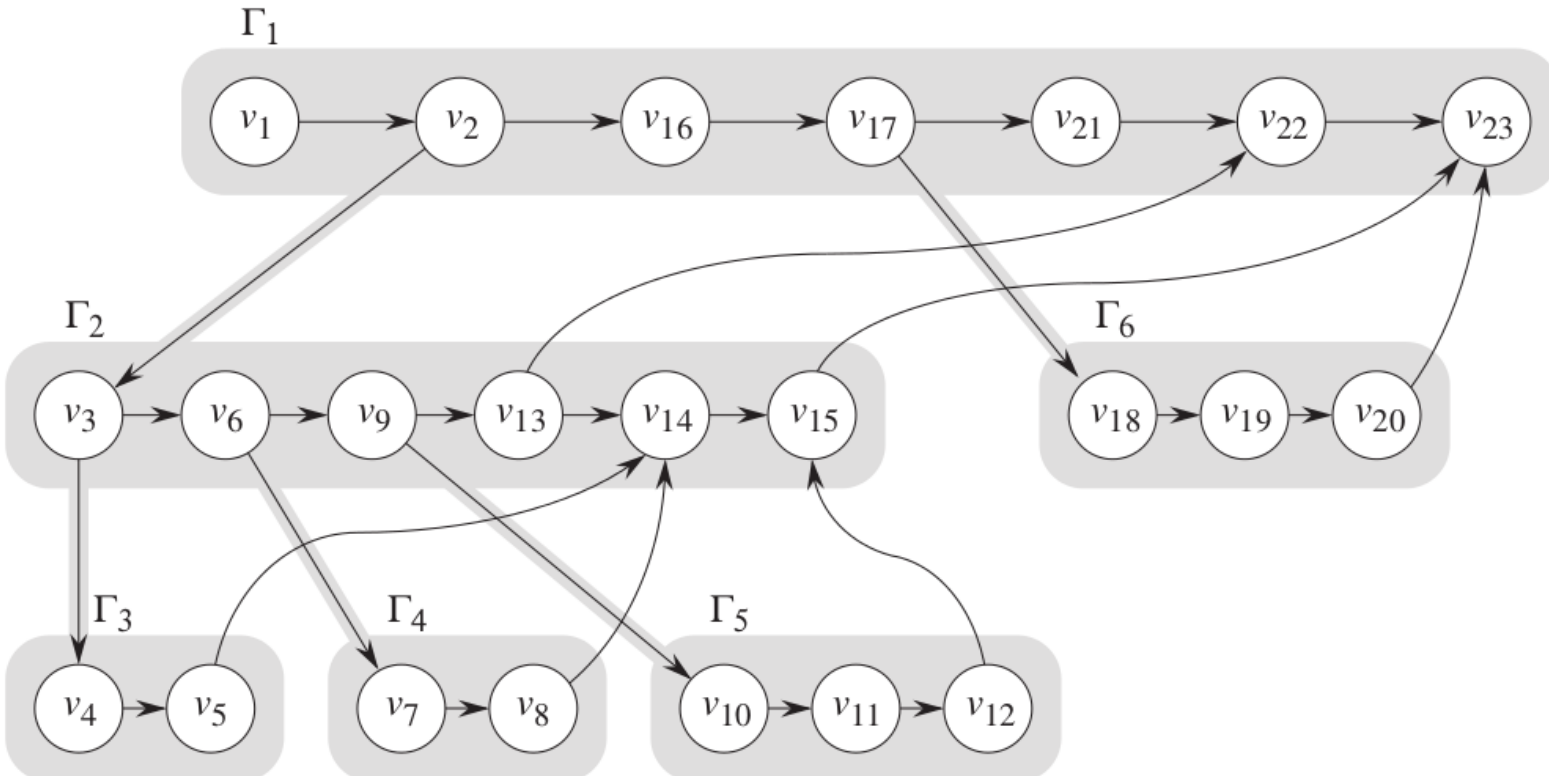
```
 1  int fib (int n)
 2  {
 3      fib_frame *f;                  frame pointer
 4      f = alloc(sizeof(*f));         allocate frame
 5      f->sig = fib_sig;              initialize frame
 6      if (n<2) {
 7          free(f, sizeof(*f));       free frame
 8          return n;
 9      }
10      else {
11          int x, y;
12          f->entry = 1;              save PC
13          f->n = n;                  save live vars
14          *T = f;                    store frame pointer
15          push();                    push frame
16          x = fib (n-1);             do C call
17          if (pop(x) == FAILURE)     pop frame
18              return 0;              frame stolen
19          ...                        second spawn
20          ;                          sync is free!
21          free(f, sizeof(*f));       free frame
22          return (x+y);
23      }
24  }
```

From Frigo et al. 1998

Optimizations: fast/sequential clone vs. slow/concurrent clone, tail-call elimination, specialized memory management for activation frames, concurrency throttling...

Cilk DAG Revisited

Slightly complicated by the optimized fast/slow clone policy



Back to Work-Stealing

- Early implementations are by
  - Burton and Sleep 1981
  - Halstead 1984 (Multi-Lisp)
- Leiserson and Blumofe 1994, randomization and theoretical bounds

Randomized Work-Stealing

- Each processor has a ready-task deque
- For itself, this is operated as a stack
- Others can "steal" from top
- `A` spawns `B`
  Push `A` to bottom, start working on `B`
  (like an eager function call)
- `A` stalls (`sync`) or terminates
  Check own "stack" for ready tasks, else "steal" topmost from other random processor
- Initially, one processor starts with the "root" task, all other work queues are empty

Work-Stealing Implementation With a Lock-Free Deque

State-of-the-art method: David Chase and Yossi Lev 2005:

- Uses an array (automatic, asynchronous growth)
- One compare-and-swap per `take`/`steal` only when the deque has one single element
- Memory consistency: on x86, one store-load fence for each task
- Sophisticated concurrent algorithm and correctness proof

```
int take () {
  long b = bottom - 1;
  item_t *q = deque;
  bottom = b;
  long t = top;
  if (b < t) {
    bottom = t;
    return EMPTY;
  }
  int task = q->buf[b%q->size];
  if (b > t)
    return task;
  if (!atomic_cas (&top, t, t+1))
    return EMPTY;
  bottom = t + 1;
  return task;
}
```

```
void push (int task) {
  long b = bottom;
  long t = top;
  item_t *q = deque;
  if (b - t > q->size - 1)
    expand ();
  q->buf[b%q->size] = task;
  bottom = b + 1;
}
```

```
int steal (item_t *remote_deque) {
  long t = top;
  long b = bottom;
  item_t *q = remote_deque;
  if (t >= b)
    return EMPTY;
  int task = q->buf[t%q->size];
  if (!atomic_cas (&top, t, t+1))
    return ABORT;
  return task;
}
```

Generalized, Distributed Work-Stealing Implementations

Hierarchical, heterogeneous, distributed workstealing, accelerators

KAAPI project in Grenoble, by Thierry Gautier, Jean-Louis Roch et al.
`http://moais.imag.fr/membres/thierry.gautier/TG/home_page.html`

StarPU project in Bordeaux, by Cedric Augonnet (now at NVidia), Samuel Thibault, Raymond Namyst et al., now running the MAGMA and PLASMA linear algebra libraries
`http://runtime.bordeaux.inria.fr/StarPU`

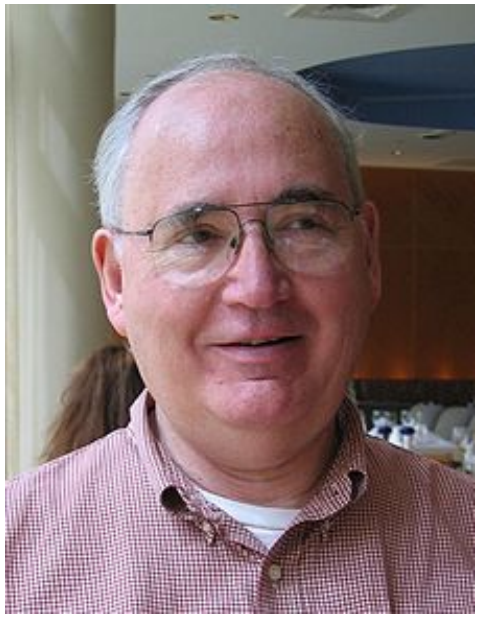
Advanced Course in Shared-Memory Parallel Programming

CMU 15-210 course: "Parallel Computing: Theory and Practice"
Umut A. Acar, with Arthur Charguéraud and Mike Rainey

`http://www.cs.cmu.edu/~15210/pasl.html`
`http://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/tapp.html`

5. Dependent Tasks

Foundations: Scott Topology



Domain Theory

Dana Scott (1932–), Turing Award 1976

Denotational semantics of a system of recursive equations, defined as the least fixpoint of a system of equations over continuous functions

$\rightarrow$ General recursion

Foundations: Kahn Process Networks

Kahn networks, 1974
Gilles Kahn (1946–2006)

Denotational: least fixpoint of a system of equations over continuous functions, for the Scott topology lifted to unbounded streams

$$s \text{ stream prefix of } s' \implies f(s) \text{ stream prefix of } f(s')$$

Operational: communicating processes over FIFO channels with blocking reads

Foundations: Kahn Process Networks

Kahn networks, 1974
Gilles Kahn (1946–2006)

Denotational: least fixpoint of a system of equations over continuous functions, for the Scott topology lifted to unbounded streams

$$s \text{ stream prefix of } s' \implies f(s) \text{ stream prefix of } f(s')$$

Operational: communicating processes over FIFO channels with blocking reads

→ Deterministic by design

→ Dynamic process creation, parallel composition, reactive systems

→ Effective means to distribute computations

- Languages: Lucid, SISAL
  (a KPN can be simulated by a lazy functional language)

See the course's second part and MPRI 2.23.1. Synchronous Systems

Functional Determinism and Dependences

Back to Kahn Networks

- How to build a task graph? At run time?
- How to describe dependences between tasks?
- How to extend a scheduling algorithm to deal with dependent tasks?

Programming Model for Kahn Networks

- Cilk only has join synchronization, and spawned tasks are immediately ready (strictness)
  The schedule is over-constrained, which is detrimental to scalability and load balancing
- Point-to-point dependences allow for more expressive patterns of parallelism
  1. Explicit dependences: futures, channels, tag matching, etc.
  2. Implicit dependences: inferred from the dynamic instrumentation of annotated data accesses: StarSs, dependent tasks in OpenMP 4

Scheduling Dependent Tasks

Three Complementary Strategies

1. Rely on the control sequence only:
   map `A` and `B` to the same worker thread and run `A; B;` if `B` depends on `A`!

2. Partition the task pool into waiting and ready tasks, and check for dependence resolution in the scheduler:
   - control-flow execution model: futures
   - data-driven, or data-flow execution model

3. Use low-level thread synchronization methods:
   - Condition variables with locks
   - FIFO buffers

Futures

- Intuition
  - Lazy evaluation
    (note: can be simulated in an eager, a.k.a. strict language
    `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Lazy.html`)
  - Concurrent execution
    instead of sequential, demand-driven suspension
- Terminology: to fulfill, resolve or bind a future
  `wait()`, `get()`, `bind()`

Futures in C++11

```
#include <future>
#include <iostream>

int calculate_the_answer_to_LtUaE();
void do_stuff();

int main()
{
    std::future<int> the_answer = std::async(calculate_the_answer_to_LtUaE);
    do_stuff();
    std::cout << "The answer to life, the universe and everything is "
              << the_answer.get() << std::endl;
}
```

Digging into the Semantics: Futures in C++11

- Implemented as a library based on the C++11 memory model and thread library
  - Promise (communication and synchronization only): `promise<T>`, `set_value()`, `set_exception()`, `get_future()`
  - Future (promise + thread creation): `std::async`, `get()`
  - Function returning a future value: `packaged_task<T>`
- Memory management: unique vs. shared futures
- Concurrent execution or lazy suspension?
  `std::launch::sync` vs. `std::launch::async` launch policies
- Data-flow or non-deterministic select?
- Optimization for futures:
  some uses of $=$, tuple operations, a few arithmetic operations, futures of constants, etc.

Trolling on futures in C++11:
`http://bartoszmilewski.wordpress.com/2009/03/03/broken-promises-c0x-futures`

Futures in F#

- async combinator and `let!` future-binding keyword
- Async module for work distribution and thread pool management

```
let triple =
    let z = 4.0
    [ async { z * z };
      async { sin z };
      async { log z } ]

let reduced_triple =
    async { let! vs = Async.Parallel triple
            Array.fold_left (fun a b -> a + b) 0.0 vs }

printf "Result = %f\n" (Async.Run reduced_triple)
```

Futures in F# With Asynchronous I/O

```
let TransformImage pixels img =
  // Some image processing

let ProcessImage img =
  async { use inStream = File.OpenRead (sprintf "source%d.jpg" img)
          let! pixels  = inStream.ReadAsync (1024*1024)
          let  pixels' = TransformImage pixels img
          use  outStream = File.OpenWrite (sprintf "result%d.jpg" img)
          do!  outStream.WriteAsync (pixels')
          do   Console.WriteLine "done!" }

let ProcessImages () =
  Async.Run (Async.Parallel
              [ for img in 1 .. numImages -> ProcessImage img ])
```

Each of the load/process/save steps is taking place in a distinct task mapped to the thread pool

Futures: Important Properties

- Very expressive
  - Dynamic dependence graph
- Preserve determinism

- When using futures in a purely functional way (`async` and `get()`):
  - Preserves the strictness property of Cilk, enabling the "compilation of parallelism" through the sequential execution of asynchronous calls
  - No risk of dead-locks
- When separating the promise and the coroutine in futures:
  - Strictness is lost in general
  - Cyclic dependence graphs can be built, leading to deadlocks

Futures: Performance Problems

- Does not interact well with work-stealing
- Dynamic memory management (heap object)
- Need reference counting or garbage collection in general
- Blindness of task scheduler w.r.t. future synchronizations
  - The critical path is hidden
  - Memory consumption of suspended, waiting tasks
- Non-locality of future values w.r.t. their binding (consumer) tasks
- Scheduling overhead of task suspension

Futures in Rust

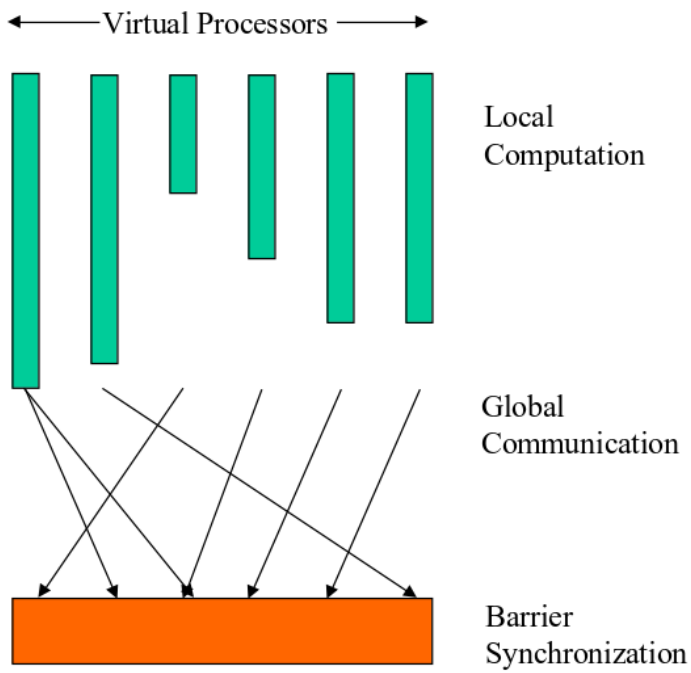
- From Tokio: `https://tokio.rs`
  `https://tokio.rs/docs/getting-started/futures`
  `https://docs.rs/futures/0.1.15/futures`
- Beyond futures: streams and sinks
  `https://tokio.rs/docs/getting-started/streams-and-sinks`
- Wrapping futures for reactive (asynchronous) I/O: event loops
  `https://tokio.rs/docs/getting-started/reactor`

6. Bulk-Synchronous Parallelism

Bulk-Synchronous Parallelism (BSP)

- About BSP: `http://www.bsp-worldwide.org`
- PRAM generalization...
  with point-to-point network and efficient synchronization barriers
- Founded by Leslie Valiant, 2010 Turing Award
  "A Bridging Model for Parallel Computation", CACM, 1990
- Successful for complexity studies

- Vertical Structure
  - Sequential composition of "supersteps"
    - Local computation
    - Process Communication
    - Barrier Synchronization
- Horizontal Structure
  - Concurrency among a fixed number of virtual processors
  - Processes do not have a particular order
  - Historically, locality played no role in the placement of processes on processors... but this can be greatly improved



Bulk-Synchronous Parallelism Reloaded

- BSP-based CPU+GPU programming with locality optimization
  → The H3LMS experiment, with Jean-Yves Vet and Patrick Carribault at CEA

- Data center frameworks:
  → data- and I/O-centric map-(shuffle-)reduce
    ... with transparent distribution and fault tolerance
    ... on key-value pairs

7. Take Home Message

Safe and Efficient Computing

Direction

Reconciling safety and efficiency in parallel programs

State of the Art – Rust and Beyond

- Safe memory management, mostly at compilation time, based on lifetimes and move semantics
- Fork/join task parallelism with the Cilk "parallel pair" and work-stealing
- Bulk-synchronous CPU+GPU acceleration with hierarchical work-stealing
- Dependent tasks with futures
- Asynchronous I/O with futures and on-demand streams
- Explicit reference counting, lock/mutex and low-level atomic operation to manage concurrency at runtime, and to locally break out of functional determinism