8. Parallel computing

Credits to Tucker Taft

AdaCore
The GNAT Pro Company

Program, Processor, Process

- **Program = static piece of text**
  - Instructions + link-time-known data
- **Processor = resource that can execute a program**
  - In a "multi-processor," processors may
    - Share uniformly one common address space for main memory
    - Have non-uniform access to shared memory
    - Have unshared parts of memory
    - Share no memory as in "Shared Nothing" (distributed memory) architectures
- **Process = instance of program + run-time data**
  - Run-time data = registers + stack(s) + heap(s)

Threads, Picothreads, Tasks, Tasklets, etc.

- **No uniform naming of *threads of control* within process**
  - Thread, *Kernel Thread*, OS Thread, Task, Job, Light-Weight Process, Virtual CPU, Virtual Processor, Execution Agent, Executor, *Server Thread*, Worker Thread
  - "Task" generally describes a logical piece of work
  - "Thread" generally describes a virtual CPU, a thread of control within a process
  - "Job" in the context of a real-time system generally describes a single execution consequent to a task release
- **No uniform naming of *user-level lightweight threads***
  - Task, Microthread, *Picothread*, Strand, *Tasklet*, Fiber, Lightweight Thread, *Work Item*
  - Called "user-level" in that scheduling is done by code *outside* of the kernel/operating-system

SIMD – Single Instruction Multiple Data

- **Vector Processing**
  - Single instruction can operate on "vector" register set, producing many adds/multiplies, etc. in parallel
- **Graphical Processing Unit**
  - Broken into independent "warps" consisting of multiple "lanes" all performing same operation at same time
  - Typical GPU might be 32 warps of 32 lanes each ~= 1024 cores
  - Modern GPUs allow individual "lane"s to be conditionally turned on or off, to allow for "if-then-else" kind of programming

How to Support Parallel Programming

- **Library Approach**
  - Provide an API for spawning and waiting for *tasklets*
  - Examples include Intel's TBB, Java Fork/Join, Rust
- **Pragma Approach**
  - No new syntax
  - Everything controlled by pragmas on
    - Declarations
    - Loops
    - Blocks
  - OpenMP is the main example here, OpenACC is similar
- **Syntax Approach**
  - Menu of new constructs
    - Cilk+, Go, CPlex
  - Building Blocks + Syntactic Sugar
    - Ada 202X, ParaSail

What About Safety?

- **Language-provided safety is to some extent orthogonal to the approach to parallel programming support**
  - Harder to provide using Library Approach: Rust does it by having more complex parameter modes
  - Very dependent on amount of "aliasing" in the language
- **Key question is whether compiler**
  - Trusts programmer requests and follows orders
  - Treats programmer requests as hints, only following safe hints
  - Treats programmer requests as checkable claims, complaining if not true
- **If compiler can check claims, compiler can insert safe parallelism automatically**
- **More on this later**

Library Option: TBB, Java Fork/Join, Rust

- **Compiler is removed from the picture completely**
  - Except for Rust, where compiler enforces safety
- **Run-time library controls everything**
  - Focuses on the scheduling problem
  - Need some run-time notion of "tasklet ID" to know what work to do
- **Can be verbose and complex**
  - Feels almost like going back to assembly language
  - No real sense of abstraction from details of solution
  - Can use power of C++ templates to approximate syntax approach

The Rust Language

- **Rust is from Mozilla** *http://rust-lang.org*
  - From "browser" development group
    - Browser has become enormous, complex, multithreaded
  - C-ish syntax, but with more of a "functional" language feel
    - *Trait*-based inheritance and polymorphism; *match* instead of *switch*
  - Safe multithreading using *owned* and *managed* storage
    - *Owned* storage in global heap, but only one pointer at a time (no garbage collection)
      - Similar to C++ "Unique" pointers
    - Originally also provided *Managed* storage in task-local heap, allowing many pointers within task to same object, but since dropped to avoid need for garbage collector
    - *Complex* rules about parameter passing and assignment
      - Copy vs. move semantics
      - Borrowing vs. copying

Pragma Option: OpenMP, OpenACC

- **User provides hints via #pragma**
- **No building blocks – all smartness in the compiler**
- **Not conducive to new ways of thinking about the problem**
  - Historical example: Ada 95 Passive Tasks vs. Protected Types

Ed Schonberg (NYU, AdaCore) on pragmas

- The two best-known language-independent (kind of) models of distribution and parallel computation currently in use, OpenMP and OpenACC, both chose a pragma-like syntax to annotate a program written in the standard syntax of a sequential language (Fortran, C, C++)
  - Those annotations typically carry target-specific information (number of threads, chunk size, etc.)
- This solution eases the inescapably iterative process of program tuning, because it only needs the annotations to be modified

Lesson Learned – Passive Tasks vs. Protected Objects

- **Ada 83 relied completely on task + rendezvous for synchronization**
- **Real-time community familiar with Mutex, Semaphore, Queue, etc.**
- **One solution – Pragma Passive_Task**
  - Required task to be written as loop enclosing a single "select with terminate" statement
  - Passive_Task optimization (Habermann and Nassi described it first) turned "active" task into a "slave" to callers
    - Executed only when task entry was called
    - Reduced overhead for particular idiom
- **Ada 9X Team proposed "Protected Objects"**
  - Provided entries like tasks
  - Also provided protected functions and procedures for simple Mutex functionality

Lesson Learned (cont'd)

- **Major battle**
- **Final result was Protected Objects added to language**
- **Data-Oriented Synchronization Model Widely Embraced**
- **Immediately allowed focus to shift to interesting scheduling and implementation issues**
  - Priority Inheritance
  - Priority Ceiling Protocol
  - Priority Ceiling Emulation
  - "Eggshell" model for servicing of entry queues
  - Use of "convenient" task to execute entry body to reduce context switches
  - Notion of "requeue" to do some processing and then requeue for later steps of processing
- **New way of thinking**
  - Use of Task Rendezvous now quite rare

Syntax Option

- **Menu of new features**
  - Go, Cilk+, CPlex
- **Building Block + Syntactic Sugar approach**
  - Ada 202x
  - ParaSail
- **Asynchronous function call**
  - cilk_spawn C(X)
  - go G(B)
  - _Task _Call F(A)
- **Wait for spawned strand / goroutine / task**
  - cilk_sync;
  - <implicit for Go>
  - _Task _Sync; or end of _Task _Block { … }

Cilk+ from MIT and Intel

- **Keywords – Express task parallelism**
  - *cilk_for* - Parallelize for loops
  - *cilk_spawn* - Specifies that a function can execute in parallel
  - *cilk_sync* – Waits for all spawned calls in a function
- **Reducers**
  - Eliminate contention for shared variables among tasks by automatically creating views of them as needed, and "reducing" them in a lock free manner
  - "tasklet local storage" + reduction monoid (operator + identity)
- **Array Notation**
  - Data parallelism for arrays or sections of arrays
- **SIMD-Enabled Functions**
  - Define functions that can be vectorized when called from within an array notation expression or a #pragma SIMD loop
- **#pragma simd: Specifies that a loop is to be vectorized**

The Go Language

- **Go is from Google** *http://golang.org*
  - Rob Pike from early Bell Labs Unix design team
  - Quite "C" like syntactically, but with some significant differences
    - Object name *precedes* type name in syntax; allows type name to be omitted when can be inferred
      - e.g. "X int;" vs "int X;" ➔ "X := 3;" *// declares // and initializes X*
    - No pointer arithmetic; provides *array slicing* for divide-and-conquer
    - "Goroutines" provide easy *asynchronous function calls*
      - Communicate via *channels* and *select* statements, but no race-condition checking built in
    - *Interfaces* and *method sets* but no classes
    - Fully garbage collected

Building Blocks + Syntactic Sugar

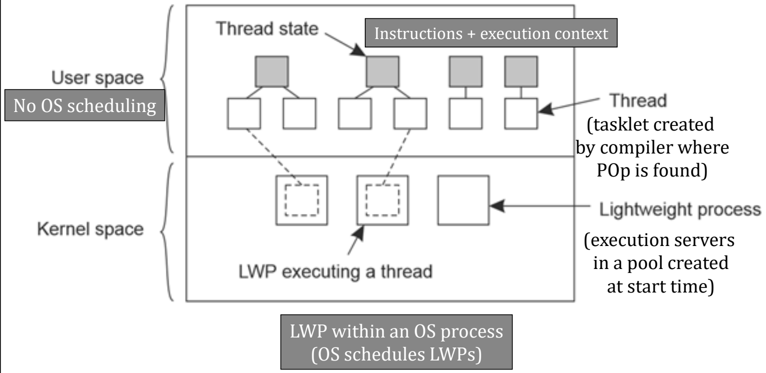
- **Ada 202X, ParaSail**
- **Examples**
  - Operators, Indexing, Literals & Aggregates, Iterators
- **New level of abstraction**
  - Defining vs. calling a function
  - Defining vs. using a private type
  - Implementing vs. using syntactic sugar
- **Minimize built-in-type "magic"**

Parallel Block

```
 parallel
    sequence_of_statements
{and
    sequence_of_statements}
 end parallel;
```

Each alternative is an (explicitly specified) *"parallelism opportunity"* (POp) where the compiler may create a *tasklet,* which can be executed by an *execution server* while still running under the context of the enclosing *task* (same task 'Identity, attributes, etc.)
Compiler will complain if any data races or blocking are possible

POps, Tasklets and Execution Servers



Parallel Loop

```
for I in parallel 1 .. 1_000 loop
   A(I) := B(I) + C(I);
end loop;

for Elem of parallel Arr loop
   Elem := Elem * 2;
end loop;
```

Parallel loop is equivalent to parallel block by unrolling loop

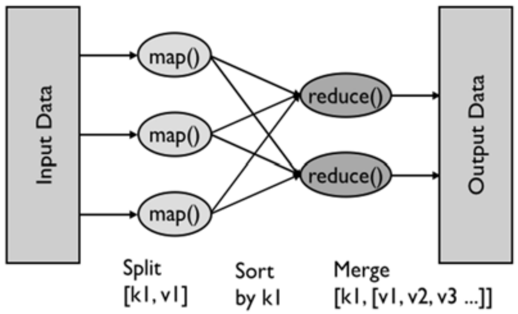
Each iteration as a separate alternative of parallel block

Compiler will complain if iterations are not independent or might block

Simple and Obvious, but What About… ?

- **Exiting the block / loop, or a return statement?**
  - All other tasklets are aborted (need not be preemptive) and awaited, and then, in the case of return with an expression, the expression is evaluated, and finally the exit/return takes place
  - With multiple concurrent exits/returns, one is chosen arbitrarily, and others are aborted
- **With a very big range or array to be looped over, wouldn't that create a huge number of tasklets?**
  - Compiler may choose to "chunk" the loop into sub-loops, each sub-loop becomes a tasklet (sub-loop runs sequentially within tasklet)
- **Iterations are not completely independent, but could become so by creating multiple accumulators?**
  - We provide notion of *parallel array* of such accumulators (next slide)

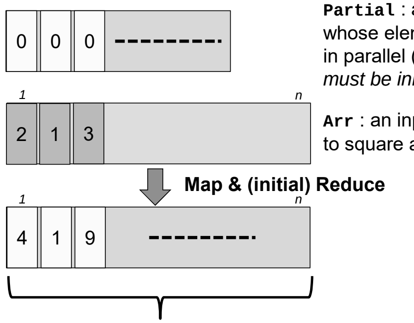
Parallel Arrays of Accumulators: Map / Reduce / 1



Parallel Arrays of Accumulators: Map / Reduce / 2

`Partial` : an array with a dynamic bound whose elements can be worked on in parallel (*the elements of an accumulator must be initialized to 0*)

`Arr` : an input array «mapped» onto **`Partial`** to square all of its elements and sum them up



$\sum_{i=1}^{n} Partial[i]$ (Final) Reduce

Parallel Arrays of Accumulators: Map / Reduce / 3

```
declare
   Partial: array (parallel <>) of Float := (others => 0.0);
   Sum_Of_Squares : Float := 0.0;
begin
   for E of parallel Arr loop -- "Map" and partial reduction
      Partial(<>) := Partial(<>) + E ** 2;
   end loop;

   for I in Partial'Range loop  --  Final reduction step
      Sum_Of_Squares := Sum_Of_Squares + Partial (I);
   end loop;

   Put_Line ("Sum of squares of elements of Arr =" &
     Float'Image (Sum_Of_Squares));
end;
```

Parallel array bounds of <> are set to match number of "chunks" of parallel loop in which they are used with (<>) indices. May be specified explicitly

Map / Reduce Shorthand

- **Final reduction step will often look the same**

```
Total := <identity>;
for I in Partial'Range loop
   Total := <op> (Total, Partial);
end loop;
```

- **Provide an attribute function 'Reduced to do this**

```
Total := Partial'Reduced(Reducer => "+", Identity => 0.0);
```

or

```
Total := Partial'Reduced; -- Reducer and Identity defaulted
```

- **The 'Reduced attribute may be applied to *any* array when Reducer and Identity are specified explicitly**
- **The 'Reduced attribute may be implemented using a tree of parallel reductions**

Parallel Languages Can Simplify Multi / manycore Programming

- **As the number of cores increases, traditional multithreading approaches become unwieldy**
  - Compiler ignoring availability of extra cores would be like a compiler ignoring availability of extra registers in a machine and forcing programmer to use them explicitly
  - Forcing programmer to worry about possible race conditions would be like requiring programmer to handle register allocation, or to worry about memory segmentation
- **Cores should be seen as a resource, like virtual memory or registers**
  - Compiler should be in charge of using cores wisely
  - Algorithm as expressed in programming language should allow compiler maximum freedom in using cores
  - Number of cores available should not affect difficulty of programmer's job or correctness of algorithm

The ParaSail Approach

- **Eliminate global variables**
  - Operation can only access or update variable state via its parameters
- **Eliminate parameter aliasing**
  - Use “hand-off” semantics
- **Eliminate explicit threads, lock / unlock, signal / wait**
  - Concurrent objects synchronized automatically
- **Eliminate run-time exception handling**
  - Compile-time checking and propagation of preconditions
- **Eliminate pointers**
  - Adopt notion of “optional” objects that can grow and shrink
- **Eliminate global heap with no explicit allocate / free of storage and no garbage collector**
  - Replaced by region-based storage management (local heaps)
  - All objects conceptually live in a local stack frame

What ParaSail Retains

- **Pervasive parallelism**
  - Parallel by default; it is *easier* to write in parallel than sequentially
  - *All* ParaSail expressions can be evaluated in parallel
    - In expression like “G(X) + H(Y)”, G(X) and H(Y) can be evaluated in parallel
    - Applies to *recursive* calls as well (as in Word_Count example)
  - Statement executions can be interleaved if no data dependencies unless separated by explicit **then** rather than “;”
  - Loop iterations are *unordered* and possibly concurrent unless explicit **forward** or **reverse** is specified
  - Programmer can express *explicit* parallelism easily using “||” as statement connector, or **concurrent** on loop statement
    - Compiler will complain if any possible data dependencies
- **Full object-oriented programming model**
  - Full class-and-interface-based object-oriented programming
  - All modules are generic, but with fully shared compilation model
  - Convenient region-based automatic storage management
- **Annotations part of the syntax**
  - Pre- and post-conditions
  - Class invariants and value predicates

Why Pointer Free?

- **Consider F(X) + G(Y)**
  - We want to be able to safely evaluate F(X) and G(Y) in parallel *without* looking inside of F or G
  - Presume X and/or Y might be incoming **var** (in-out) parameters to the enclosing operation
  - Clearly, no global variables can help
    - Otherwise F and G might be stepping on same object
  - “No parameter aliasing” is important, so we know X and Y do not refer to the same object
  - What do we do if X and Y are pointers?
    - Without more information, we must presume that from X and Y you could *reach* a common object Z
    - How do parameter modes (in-out vs. in, **var** vs. non-**var**) relate to objects accessible via pointers?
- **Pure *value semantics* for non-concurrent objects**

ParaSail Virtual Machine

- **ParaSail Virtual Machine (PSVM) designed to be output from ParaSail “front end”**
- **PSVM designed to support “pico” threading with parallel block, parallel call, and parallel wait instructions**
- **Heavier-weight “server” threads serve a queue of light-weight pico-threads, each of which represents a sequence of PSVM instructions (parallel block) or a single parallel “call”**
  - Similar to Intel’s Cilk (and TBB) run-time model with *work stealing*
- **While waiting to be served, a pico-thread needs only a handful of words of memory**
- **A single ParaSail program can easily involve 1000’s of pico threads**
- **PSVM instrumented to show degree of parallelism achieved**

A bareboard runtime lib for time-predictable parallelism

Davide Compagnin (2017 PhD candidate), Tullio Vardanega
University of Padova

Moral

- When you seek *sustainable time-composable* parallelism, mind what you abstract away of the (manycore) processor hardware
- Implementation experience suggests that you should hide *much less* than used to be with concurrency

Kalray MPPA-256

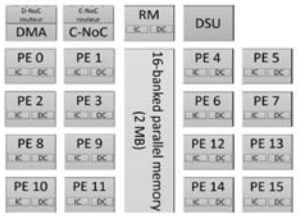


FIGURE 2.1: Kalray's compute cluster

- 288-core single chip
  - 16 17-core compute clusters
  - 4 I/O subsystems (2D torus)
- Each cluster includes 17 cores
  - 16 for general-purpose computing
  - 1 for communication and core scheduling ops
- 2MB RAM per cluster, in 16 128KB-memory banks, grouped pairwise for 8 core pairs
  - Divided in left-side and right-side banks
  - Memory address mapping interleaved or blocked

Our runtime library /1

- An execution model that supports task[let]s, to expose the potential parallelism of applications *efficiently*
- A user-level runtime environment that implements dynamic, load-balanced tasklet scheduling on top of threads
- Applications seen as DAGs

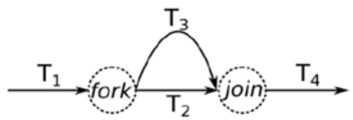


FIGURE 2.2: A fork/join DAG

Runtime architecture

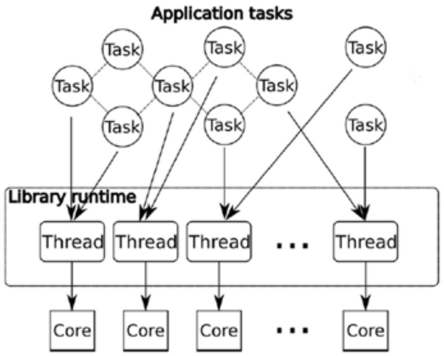


FIGURE 2.3: Execution model

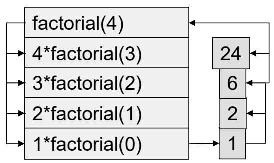
Our runtime library /2

- DAGs model parallel computation
  - Edges denote sequential strands of computation
  - Nodes denote fork/join operations
- Suspension is costly and *should be avoided*
  - Invert control-flow dependencies and convert the program to a *continuation-passing style*
- The computation always makes progress performing a *tail-recursive* function call
  - No return to the caller, but to a “continuation” that represents the remainder of the computation

Escaping linear recursion /1

- Linear recursive functions cost dear stack space as the caller has something to do *after* the callee returns (its stack frame must be kept, and execution must walk back to it)

```
factorial(n) {
   if (n == 0) return 1;
   return n * factorial(n - 1);
 }
```



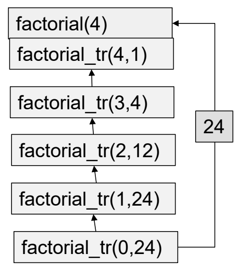
Stack depth = 5

Escaping linear recursion /2

- A tail-recursive function has *nothing* to do after the callee returns (its stack frame can be reclaimed and reused)

```
factorial_tr(n, accumulator) {
   if (n == 0) return accumulator;
   return factorial_tr(n - 1, n * accumulator);
 }

factorial(n) {
   return factorial_tr(n, 1);
 }
```



Stack depth = 2

Continuations /1

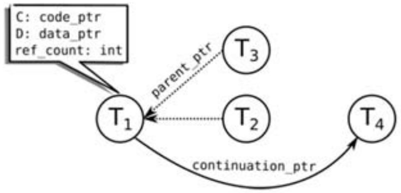


FIGURE 2.5: A task-based implementation of fork/join parallelism

- The completion of T2 **and** T3 triggers the execution of T4 (their continuation)
- The continuation task T4 is seen as part of T1
  - It inherits T1's possible ancestor
  - Children tasks return to their parent effectively by sending return values to the continuation

Continuations /2

- Tasks never suspend
  - Their execution is deferred before starting
- This does away with the nesting of stack frames, and makes the execution of tasks completely *asynchronous*
- This model needs a task pool that stores the tasks that need execution, which neatly allows for *load balancing*

Execution model /1

- Tasks run to completion
  - Hence, there are no blocking, yielding, suspension, or other interfering events
  - Much benefit on temporal and spatial locality
- The runtime is stack-less
  - All tasks that execute within the context of the same executor may share its stack
- The runtime complexity is minimum

Execution model /2

- The schedule loop exits when all tasks have been executed
  - Yet, checking whether the task pool is empty *may not be sufficient*
  - Residual tasks may be still executing with an empty task pool and they can (still) originate a further subtree of tasks
- We check completion of the root of DAG
  - Its completion corresponds to the termination of the computation

Load balancing /1

- *Work-sharing* is work-conserving
  - No worker can be idle as there are ready tasks
- Not very efficient to implement
  - The *push model* feeds one worker at a time
  - The *pull model* requires queue locking, which serializes scheduling decisions and becomes a scalability bottleneck
- It presumes evenly-balanced workloads, which are not frequent

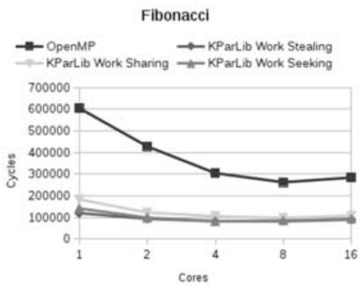
Load balancing /2

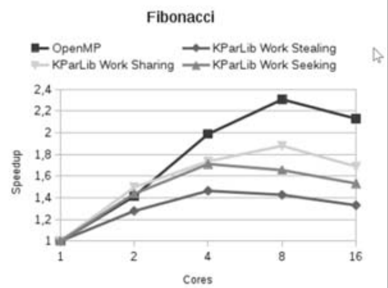
- *Work-seeking* uses cooperative distribution of load between busy and idle workers
  - When a worker empties its local queue, it seeks load from busy workers
- Busy workers regularly check for work-seeking workers and, when they find one, they *synchronously push* a task into their queue
  - If all workers are busy, each will spend time trying to offload (**!**)
- The idle worker suspends on empty (local) queue and resumes as soon as the queue is no longer empty

Load balancing /3

- *Work-stealing* uses double-ended queues
  - One deque per worker
  - Pushing and popping on the tail (serialized LIFO)
- When the local deque becomes empty, the worker steals from a victim
  - Not cooperative: there is no offer from busy worker
  - *Stealing from the head* of the victim's deque (FIFO) to minimize access conflicts with owner
  - Random victim selection propagates work well
- Lesser contention among cores, more data locality, better load balancing

Which is best?





How Do *Iterators* Fit into This Picture?

- ***Computationally-intensive* programs typically Build, Analyze, Search, Summarize, and/or Transform *large data structures* or *large data spaces***
- ***Iterators* encapsulate the process of walking data structures or data spaces**
- **The biggest *speed-up* from parallelism is provided by *spreading* the processing of a large data structure or data space across multiple processing units**
- **High-level iterators that are *amenable* to a *safe*, *parallel interpretation* can be critical to capitalizing on distributed and/or multicore HW**

While Loops and Tail Recursion Issues

+ **While loop – pros**
  – Universal sequential loop construct: semantics defined simply
- **While loop – cons**
  – Necessarily updates a global to advance through iteration
  – Generally doesn't update global until *after* finishing processing current iteration
+ **Tail recursion – pros**
  – No need for global variables: each loop iteration carries its own copy of loop variable(s)
  – Can generalize to walking more complex data structure such as a tree by recurring on multiple subtrees
- **Tail recursion – cons**
  – Next iteration value not specified until making (tail) recursive call
  – Each loop necessarily becomes a separate function

Combine "pros" of Tail Recursion with (Parallel) "for" Loop

- **Parallelism requires each iteration to carry its *own copy* of loop variable(s), like tail recursion**
  – For-loop variable treated as local constant of each loop iteration
- **For loop syntax allows next iteration value to be specified *before* beginning current iteration**
  – Rather than at tail-recursion point or end of loop body
  – Multiple iterations can be initiated in parallel
- **Explicit "continue" statement may be used to handle more complex iteration requirements**
  – Condition can determine loop-variable values for next iteration(s)
- **Explicit "parallel" statement connector allows "continue" statement to be executed in parallel with current iteration**
  – Rather than *after* the current iteration is complete
- **Explicit "exit" or "return" allows easy premature exit**

Safety in a Parallel Program – Data Races

- **Data races**
  – Two simultaneous computations reference same object and at least one is writing to the object
  – Reader may see a partially updated object
  – If two Writers running simultaneously, then result may be a meaningless mixture of two computations
- **Solutions to data races**
  – Dynamic run-time locking to prevent simultaneous use
  – Use atomic hardware instructions such as test-and-set or compare-and-swap
  – Static compile-time checks to prevent simultaneous incompatible references
- **Can support all three**
  – Dyamic: ParaSail "concurrent" objects; Ada "protected" objects
  – Atomic: ParaSail "Atomic" module; Ada pragma "Atomic"
  – Static: ParaSail hand-off semantics plus no globals; SPARK checks

Safety in a Parallel Program – Deadlock

- **Deadlock, also called "Deadly Embrace"**
  - One thread attempts to lock A and then B
  - Second thread attempts to lock B and then A
- **Solutions amenable to language-based approaches**
  - Assign full order to all locks; must acquire locks according to this order
  - Localize locking into "monitor"-like construct and ensure an operation of such a monitor does not call an operation of some other monitor that in turn calls back
    - I.e. disallow cyclic call chain between monitors
- **More general kind of deadlock – waiting forever**
  - One thread waits for an event to occur
  - Event never occurs, or is dependent on some further action of thread waiting for the event
- **No general solution to this general problem**
  - Requires full power of formal proof

Work stealing as the new consensus for scheduling parallel work

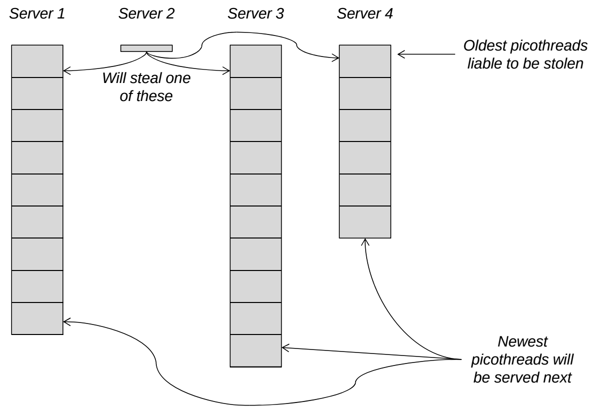
Scheduling All of the Parallel Computing

- **Fully Symmetric Multiprocessor scheduling out of favor**
  - Significant overhead associated with switching processors in the middle of a stream
- **Notion of Processor *Affinity* introduced to limit threads (bouncing) migration across processors**
  - Requires additional specification when creating threads
- **One-to-One mapping of program threads to *kernel* threads falling out of favor**
  - Kernel thread switching is expensive
- **Moving to lightweight threads managed in *user* space**
  - But still need to worry about processor affinity
- ***Work stealing* emerging as consensus solution**
  - Small number of kernel threads (server processes)
  - Large number of lightweight user-space threads
  - Processor affinity managed automatically and adaptively

Work Stealing: Double-ended Queues

- **Approximately one server process per physical core**
- **Each server process has a double-ended queue of very light-weight threads**
  - *"picothreads," "strands," "tasklets," etc.*
- **Server adds new picothreads to end of queue, and serves them in a LIFO manner**
- **When server runs out of picothreads to serve, it *steals* one from some other server – picks the oldest one**
  - Uses FIFO when stealing
  - Picks picothread that has been languishing on some servers queue
- **Provides good combination of features**
  - Automatic load balancing
  - Good locality of reference within a server
  - Good separation between servers
- **Consensus:** *Cilk+, TBB, Java Fork/Join, X10, Fortress, ParaSail, ...*

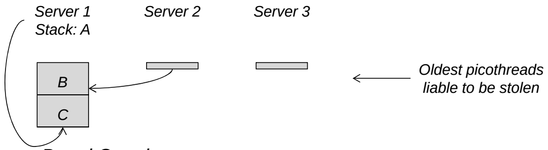
Work Stealing: Double-ended Queues



Work Stealing: Subtleties

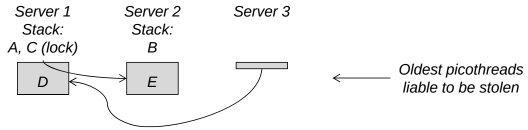
- **Picothreads are very lightweight because they don't need their own stack while waiting to be served**
  - Once started, they piggyback on stack belonging to server
- **Server stack remains occupied (but can start a second picothread) when current executing picothread has to wait**
  - For a sub-picothread to finish
  - For a resource to be released
  - For input to be available
- **Care needed to prevent servers from waiting on each other**
  - May start additional server processes in some cases
- **References on Work Stealing**
  - Blumofe and Leisersen, "Scheduling Multithreaded Computations by Work Stealing," *Journal of the ACM*, Sep 1999, pp 720-748
  - Acar, Blelloch, and Blumofe, "The Data Locality of Work Stealing," *Proceedings of the 12th ACM Symposium on Parallelism in Algorithms and Architectures,* Bar Harbor, ME, July 2000

Work Stealing: Subtleties / 1



- *A spawns B and C and then waits for them;*
- *Server 2 steals B;*
- *Server 1 serves C*

Work Stealing: Subtleties / 2



- *A spawns B and C and then waits for them;*
- *Server 2 steals B;*
- *Server 1 serves C*

- *C acquires a lock and spawns D (on server 1);*
- *B spawns E (on server 2);*
- *Server 3 steals D;*
- *C awaits D (on server 1);*
- *Server 1 steals E;*
- *B awaits E (on server 2)*

Work Stealing: Subtleties / 3

Server 1	Server 2	Server 3
Stack:	Stack:	Stack:
A, C(lock), E	B	D

← *Oldest picothreads liable to be stolen*

- *A spawns B and C and then waits for them;*
- *Server 2 steals B;*
- *Server 1 serves C*
- *C acquires a lock and spawns D (on server 1);*
- *B spawns E (on server 2);*
- *Server 3 steals D;*
- *C awaits D (on server 1);*
- *Server 1 steals E;*
- *B awaits E (on server 2)*
- *D finishes (on server 3);*
- *E (on server 1) tries to acquire same lock already held by C;*
- *E cannot proceed → B cannot complete* ***and*** *C cannot return the lock* ***and*** *A cannot complete*

Work Stealing: Subtleties / 4

Server 1	Server 2	Server 3
Stack:	Stack:	Stack:
A, C(lock), E	B	D

← *Oldest picothreads liable to be stolen*

- *A spawns B and C and then waits for them;*
- *Server 2 steals B;*
- *Server 1 serves C*
- *C acquires a lock and spawns D (on server 1);*
- *B spawns E (on server 2);*
- *Server 3 steals D;*
- *C awaits D (on server 1);*
- *Server 1 steals E;*
- *B awaits E (on server 2)*
- *B waits for E (on server 2);*
- *D finishes (on server 3);*
- *E (on server 1) tries to acquire same lock already held by C;*
- *E is deadlocked!*

Solution*: Server with picothread on its stack holding a lock, may only serve subthreads of that thread → so Server 1 would not steal E, since not a subthread of C*