# 8a. Going parallel

Credits to Tucker Taft AdaCore

**Where we take a bird's eye look at what changes (again!) when jobs become parallel in response to the quest for best use of (massively) parallel hardware**

---

## Terminology / 1

- **Program = static piece of text**
  - Instructions + link-time 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)

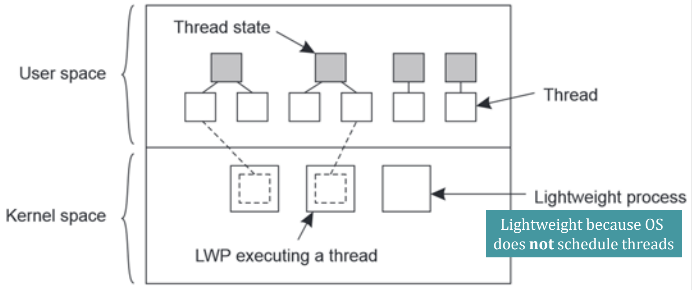
---

## Terminology / 2

- **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
    - Element of a software architecture
  - 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

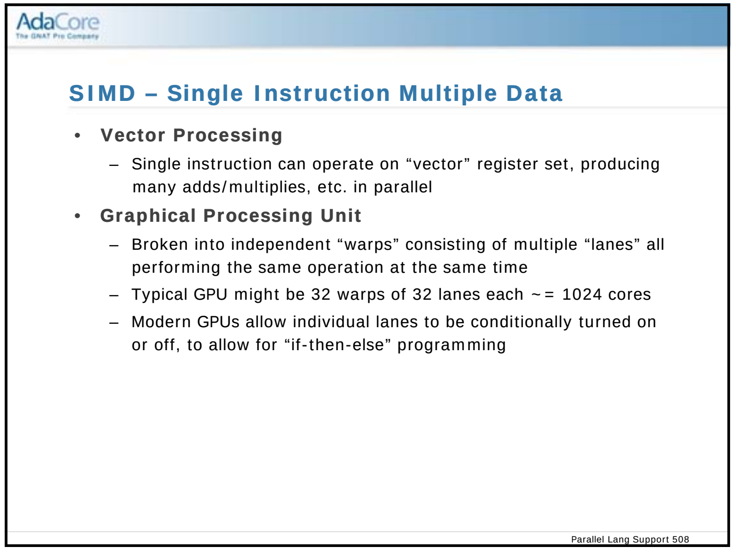
---

## Terms in context: LWT within LWP



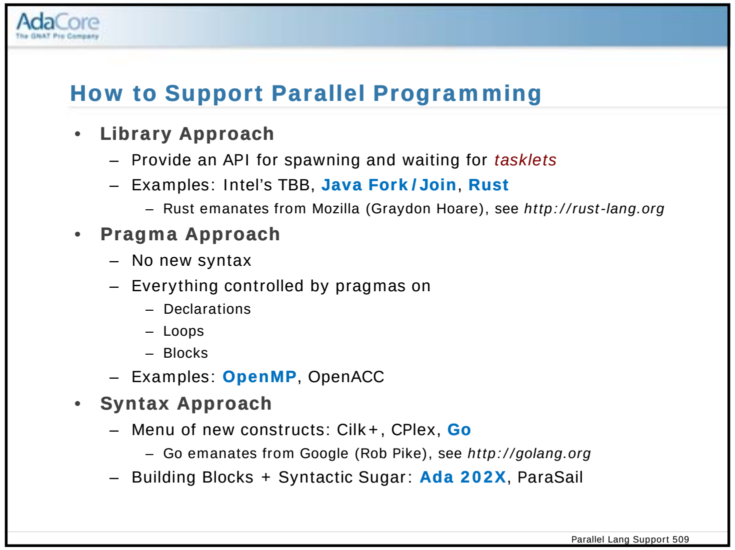
Lightweight because OS does **not** schedule threads

Exactly the same logic applies to user-level lightweight threads (aka tasklets), which do **not** need scheduling



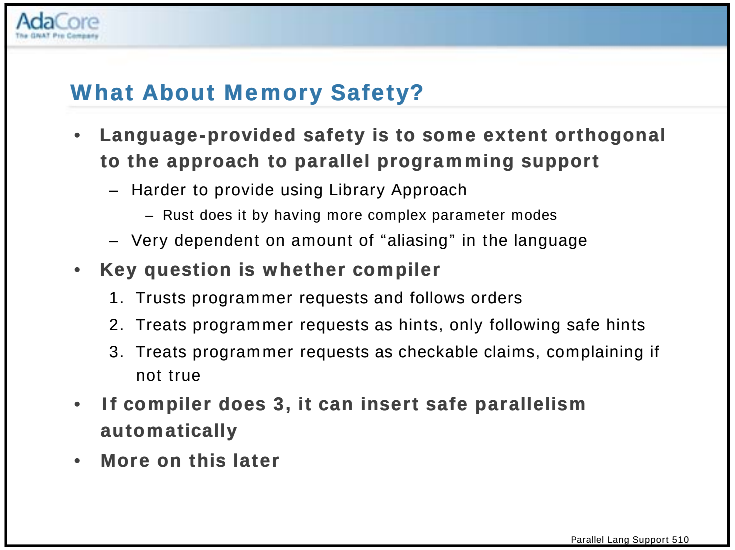
## SIMD – Single Instruction Multiple Data

- **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 the same operation at the same time
  - Typical GPU might be 32 warps of 32 lanes each ~= 1024 cores
  - Modern GPUs allow individual lanes to be conditionally turned on or off, to allow for “if-then-else” programming



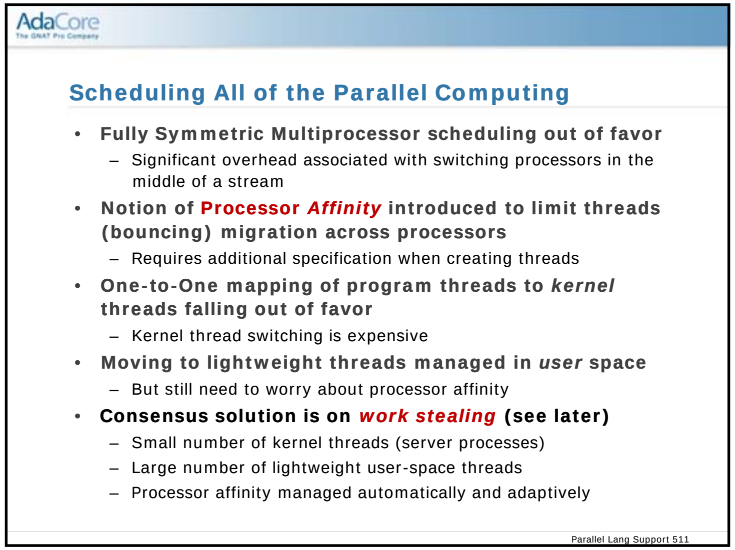
## How to Support Parallel Programming

- **Library Approach**
  - Provide an API for spawning and waiting for *tasklets*
  - Examples: Intel’s TBB, **Java Fork/Join**, **Rust**
    - Rust emanates from Mozilla (Graydon Hoare), see *http://rust-lang.org*
- **Pragma Approach**
  - No new syntax
  - Everything controlled by pragmas on
    - Declarations
    - Loops
    - Blocks
  - Examples: **OpenMP**, OpenACC
- **Syntax Approach**
  - Menu of new constructs: Cilk+, CPlex, **Go**
    - Go emanates from Google (Rob Pike), see *http://golang.org*
  - Building Blocks + Syntactic Sugar: **Ada 202X**, ParaSail



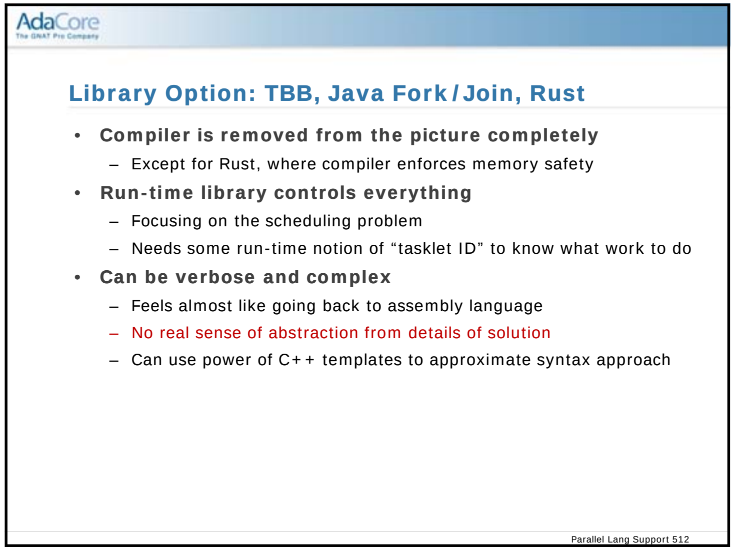
## What About Memory Safety?

- **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**
  1. Trusts programmer requests and follows orders
  2. Treats programmer requests as hints, only following safe hints
  3. Treats programmer requests as checkable claims, complaining if not true
- **If compiler does 3, it can insert safe parallelism automatically**
- **More on this later**



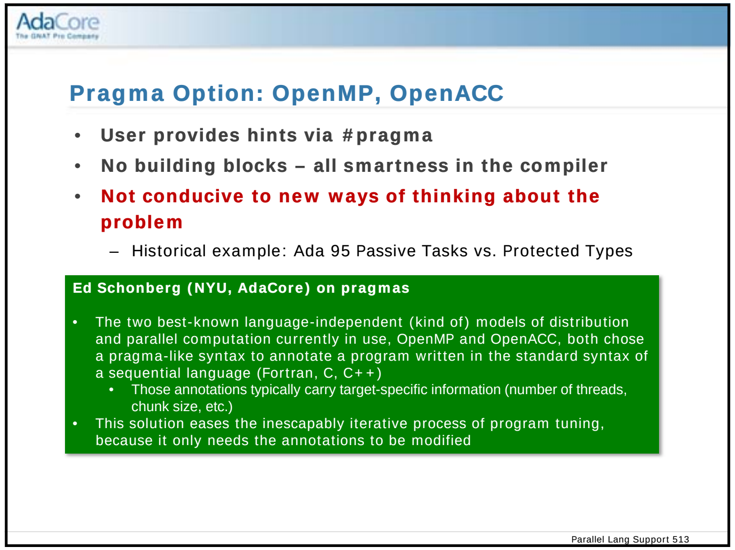
## Scheduling All of the Parallel Computing

- **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
- **Consensus solution is on *work stealing* (see later)**
  - Small number of kernel threads (server processes)
  - Large number of lightweight user-space threads
  - Processor affinity managed automatically and adaptively



## Library Option: TBB, Java Fork / Join, Rust

- **Compiler is removed from the picture completely**
  - Except for Rust, where compiler enforces memory safety
- **Run-time library controls everything**
  - Focusing on the scheduling problem
  - Needs 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

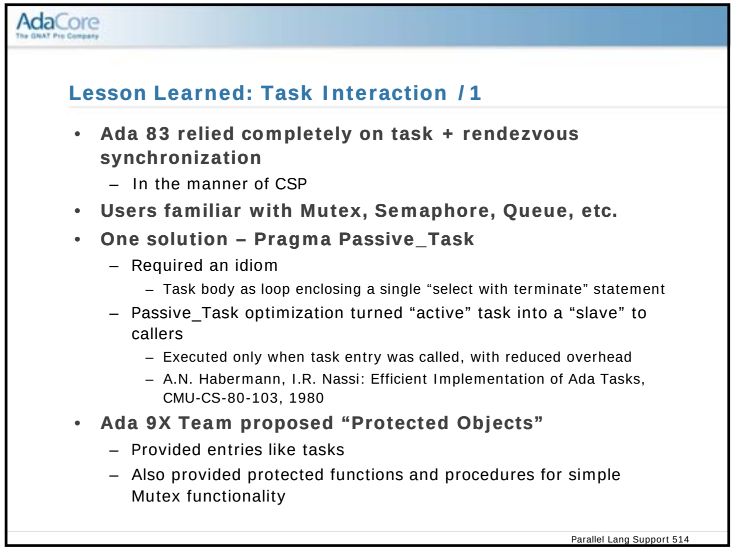


## Pragma Option: OpenMP, OpenACC

- **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

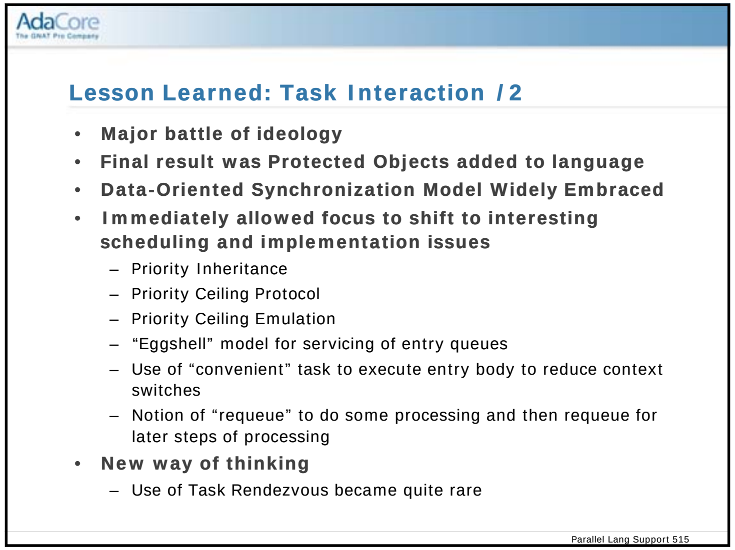
**Ed Schonberg (NYU, AdaCore) on pragmas**

- 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: Task Interaction / 1

- **Ada 83 relied completely on task + rendezvous synchronization**
  - In the manner of CSP
- **Users familiar with Mutex, Semaphore, Queue, etc.**
- **One solution – Pragma Passive_Task**
  - Required an idiom
    - Task body as loop enclosing a single "select with terminate" statement
  - Passive_Task optimization turned "active" task into a "slave" to callers
    - Executed only when task entry was called, with reduced overhead
    - A.N. Habermann, I.R. Nassi: Efficient Implementation of Ada Tasks, CMU-CS-80-103, 1980
- **Ada 9X Team proposed "Protected Objects"**
  - Provided entries like tasks
  - Also provided protected functions and procedures for simple Mutex functionality



## Lesson Learned: Task Interaction / 2

- **Major battle of ideology**
- **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 became quite rare

## Building Blocks + Syntactic Sugar Option

- **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

## Parallel Loop

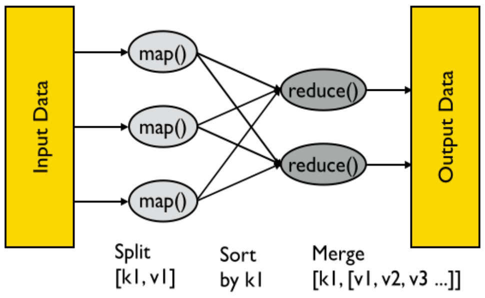
```
for I in parallel 1 .. 1_000 loop
   A(I) := B(I) + C(I);
end loop;

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

- Parallel loop 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

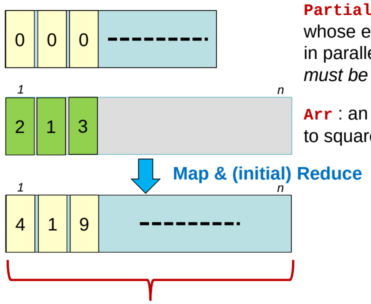
## Simple and Obvious, but Not Quite

- **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 the 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 slides)

## 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**

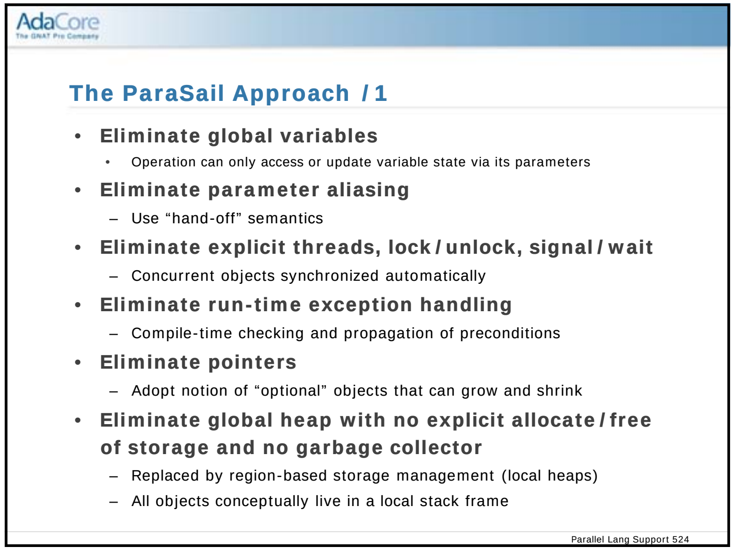
## Parallel Arrays of Accumulators: Map / Reduce / 3

```
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

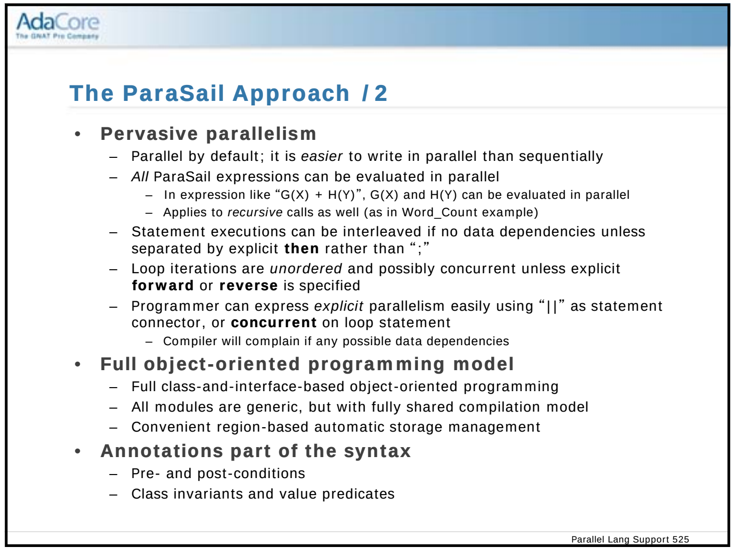
## Parallel Languages Can Simplify Multi- / manycore Programming

- **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



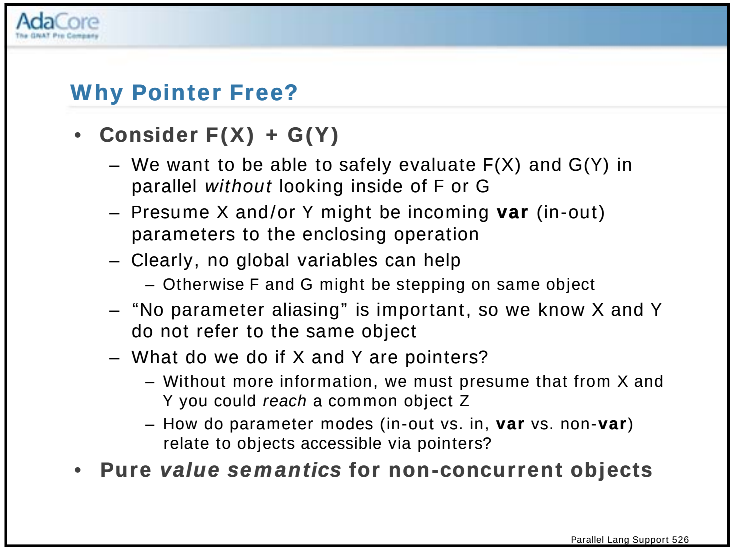
## The ParaSail Approach / 1

- **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



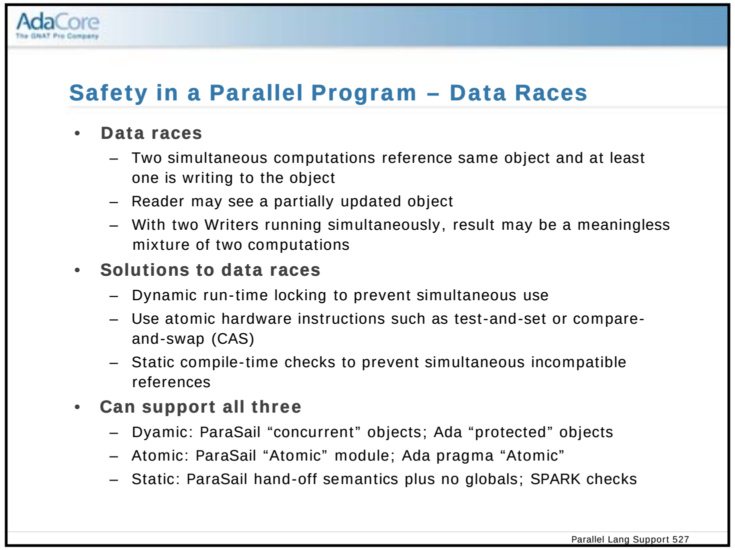
## The ParaSail Approach / 2

- **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**



## 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
  - With two Writers running simultaneously, 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 (CAS)
  - 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

AdaCore
The GNAT Pro Company

## 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

# 8.b A bareboard runtime for time-predictable parallelism

**Credits to D. Compagnin (PhD student, 2017)**

**Where we reflect on the lessons learned from a under-the-hood implementation of a runtime for a parallel-job programming model on a 256-core processor**

## Moral

- When you seek *sustainable time-composable parallelism*, you have to mind what you abstract away of the processor hardware
- Implementation experience suggests that you should hide *much less* than used to be with concurrency
  - Unless you have a powerful language on your side

## Kalray MPPA-256

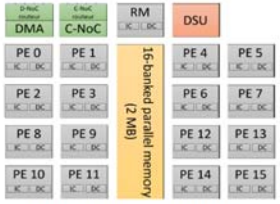


FIGURE 2.1: Kalray’s compute cluster

- 288-core on a 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

- Execution model supports tasklets natively
  - For efficient rendering of the potential parallelism of applications
  - Applications seen as DAGs
    - Edges denote sequential strands of computation
    - Nodes denote fork/join operations

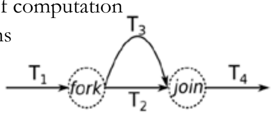


FIGURE 2.2: A fork/join DAG

- User-level runtime implements dynamic, load-balanced tasklet scheduling on top of threads

---

## Our runtime library /2

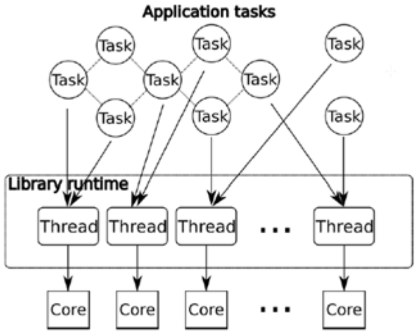


FIGURE 2.3: Execution model

---

## Our runtime library /3

- Suspension is costly because it holds the stack of an executor thread: it 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

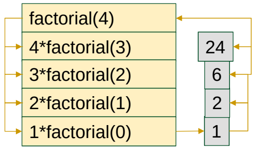
---

## Shedding linear recursion /1

- Linear recursive functions take dear stack space as the call has something to do *after* the last callee returns
  - The call’s stack frame must be kept, and execution must walk back to it

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

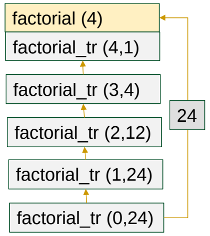
Stack depth = 5



## Shedding linear recursion /2

- A tail-recursive function has *nothing* to do after the last 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

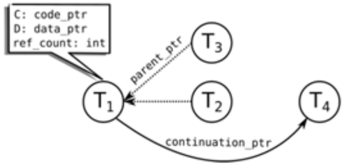


FIGURE 2.5: A task-based implementation of fork/join parallelism

- The completion of $T_2$ and $T_3$ triggers the execution of $T_4$ (their continuation)
- The continuation tasklet $T_4$ is seen as part of $T_1$
  - It inherits $T_1$'s possible ancestor
  - Children tasklets return to their parent effectively by sending return values to the continuation

## Continuations /2

- Tasklets *never* suspend
  - Their execution is simply held until start
  - After that they *run to completion*
- This does away with the nesting of stack frames, and makes the execution of tasklets completely *asynchronous*
- This model needs a tasklet pool that stores the tasklets that need execution
  - That pool neatly caters for *load balancing*

## Execution model /1

- Tasklets run to completion
  - No blocking, yielding, suspension, or other interference
  - Very nice for time and space locality
- The runtime is stack-less
  - All tasklets that execute within the context of the same executor thread may share its stack
- Runtime complexity is minimum

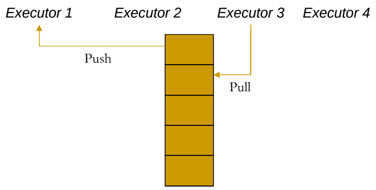
## Execution model /2

- The schedule loop exits when all tasklets have been executed
    - Checking whether tasklet pool is empty *may not be sufficient*
    - Residual tasklets may be still executing with an empty tasklet pool and can (still) spawn further tasklets
- We check completion at the root of the DAG
    - Its completion corresponds to the termination of the computation

## Load balancing /1

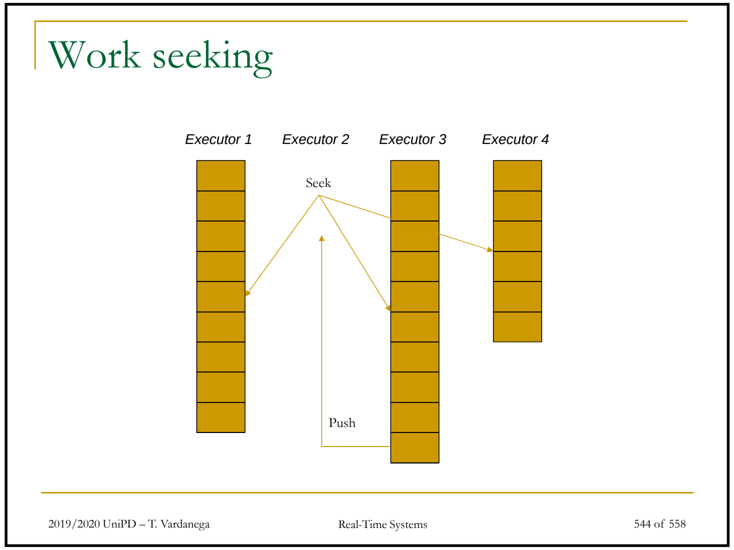
- *Work-sharing* is work-conserving
    - No worker can be idle as there are ready tasklets
- Not very efficient to implement
    - *Push model* feeds one worker at a time
    - *Pull model* needs queue locking, which serializes scheduling and becomes scalability bottleneck
- It presumes evenly-balanced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

## Work sharing



## 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 tasklet into their queue
    - If all workers are busy, each will spend time trying to offload!
- Idle worker suspends on empty local queue and resumes as soon as queue is no longer empty



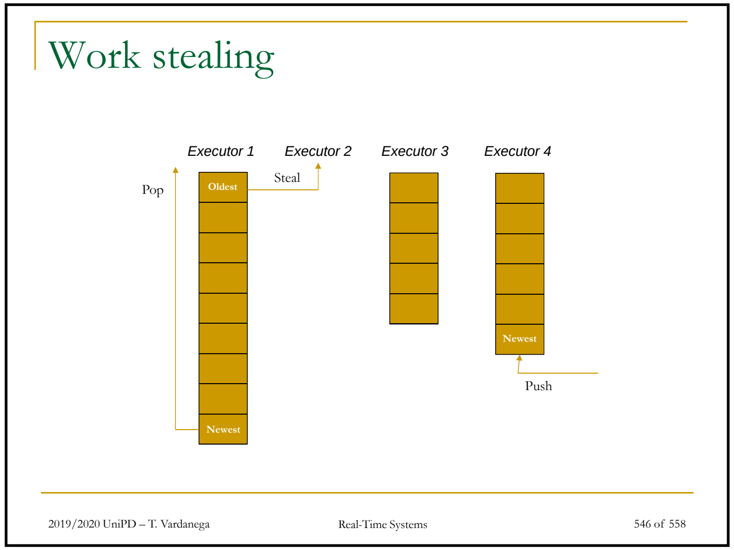
## Work seeking

---

## Load balancing /3

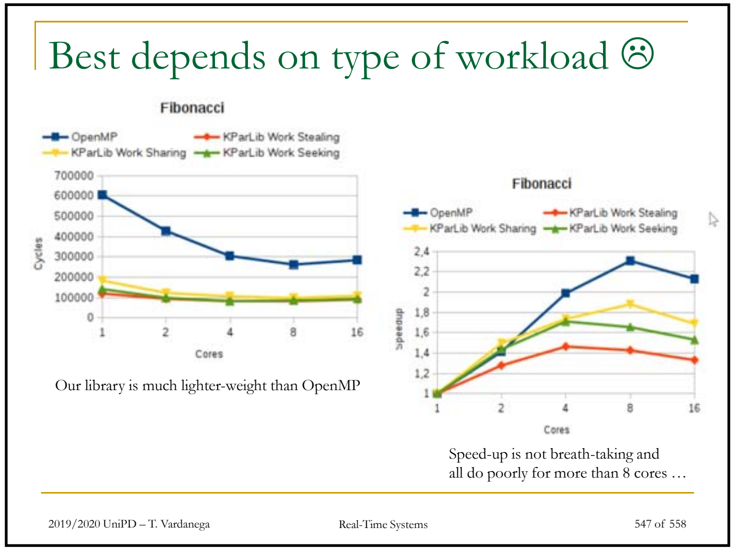
- *Work-stealing* uses **d**ouble-**e**nded queues
  - One deque per worker, one worker per physical core
  - Pushing new on the tail, popping it from there (serialized LIFO)
- When local deque is empty, worker steals from a victim
  - Stealing from the head of victim's deque
    - FIFO style minimizes access conflicts with victim
  - Random victim selection propagates work well
    - Not cooperative: no offer from busy worker
- Nice bag of features
  - Automatic load balancing
  - Good locality of reference within executor,
  - Good separation between executors

---



## Work stealing

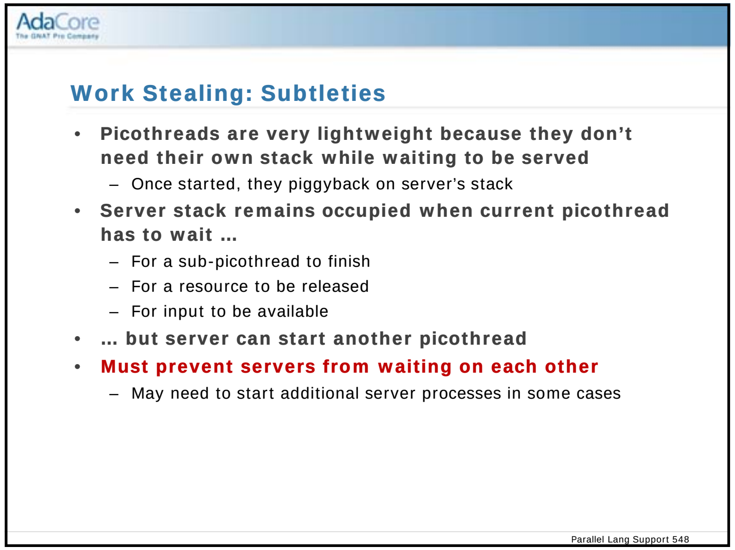
---



## Best depends on type of workload ☹

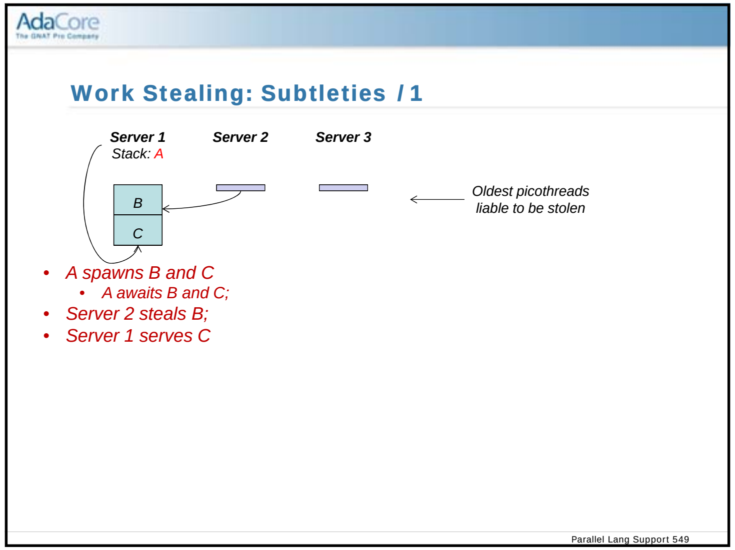
Our library is much lighter-weight than OpenMP

Speed-up is not breath-taking and
all do poorly for more than 8 cores …



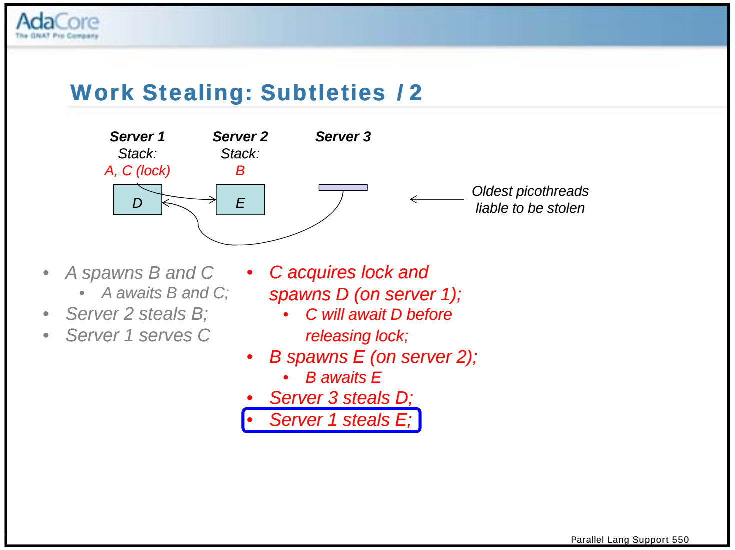
## Work Stealing: Subtleties

- **Picothreads are very lightweight because they don't need their own stack while waiting to be served**
  - Once started, they piggyback on server's stack
- **Server stack remains occupied when current picothread has to wait …**
  - For a sub-picothread to finish
  - For a resource to be released
  - For input to be available
- **… but server can start another picothread**
- **Must prevent servers from waiting on each other**
  - May need to start additional server processes in some cases



## Work Stealing: Subtleties / 1

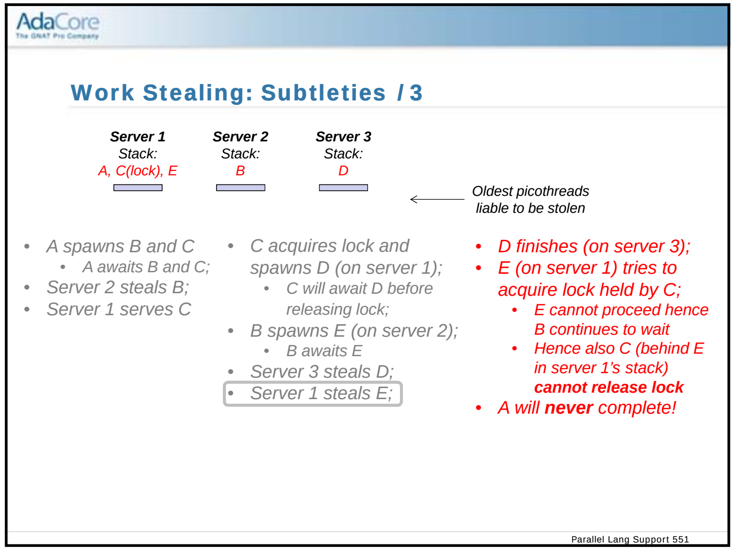
- *A spawns B and C*
  - *A awaits B and C;*
- *Server 2 steals B;*
- *Server 1 serves C*



## Work Stealing: Subtleties / 2

- *A spawns B and C*
  - *A awaits B and C;*
- *Server 2 steals B;*
- *Server 1 serves C*

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

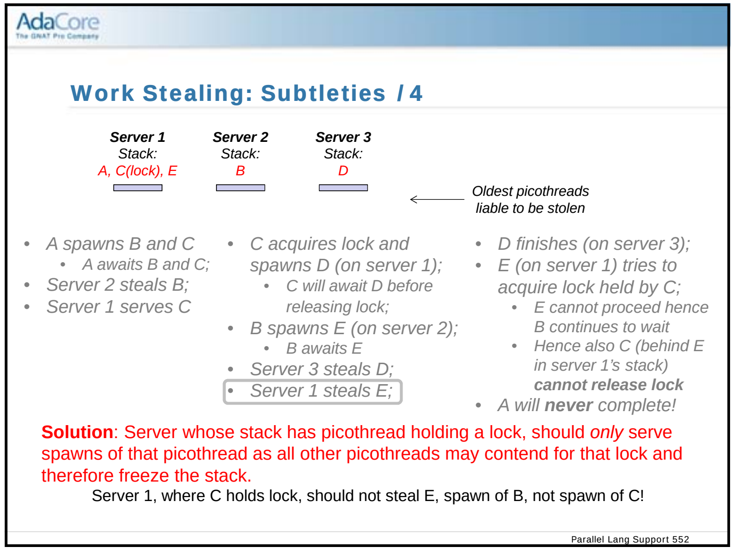


## Work Stealing: Subtleties / 3

- *A spawns B and C*
  - *A awaits B and C;*
- *Server 2 steals B;*
- *Server 1 serves C*

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

- *D finishes (on server 3);*
- *E (on server 1) tries to acquire lock held by C;*
  - *E cannot proceed hence B continues to wait*
  - *Hence also C (behind E in server 1's stack)* ***cannot release lock***
- *A will* ***never*** *complete!*



## Work Stealing: Subtleties / 4

**Server 1** Stack: A, C(lock), E

**Server 2** Stack: B

**Server 3** Stack: D

← *Oldest picothreads liable to be stolen*

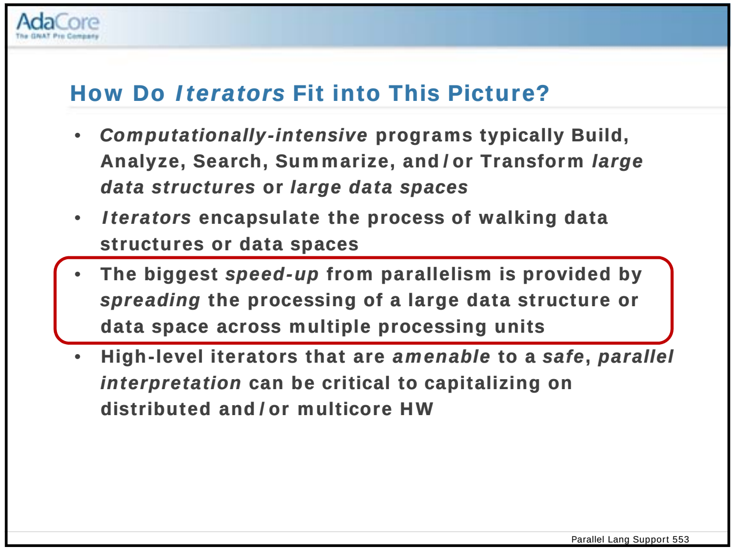
- *A spawns B and C*
  - *A awaits B and C;*
- *Server 2 steals B;*
- *Server 1 serves C*

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

- *D finishes (on server 3);*
- *E (on server 1) tries to acquire lock held by C;*
  - *E cannot proceed hence B continues to wait*
  - *Hence also C (behind E in server 1's stack)* ***cannot release lock***
- *A will* ***never*** *complete!*

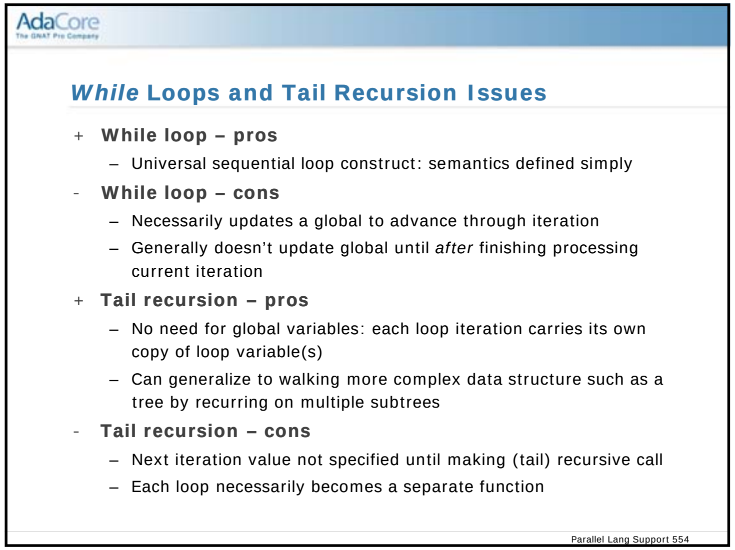
**Solution**: Server whose stack has picothread holding a lock, should *only* serve spawns of that picothread as all other picothreads may contend for that lock and therefore freeze the stack.

Server 1, where C holds lock, should not steal E, spawn of B, not spawn of C!



## 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**



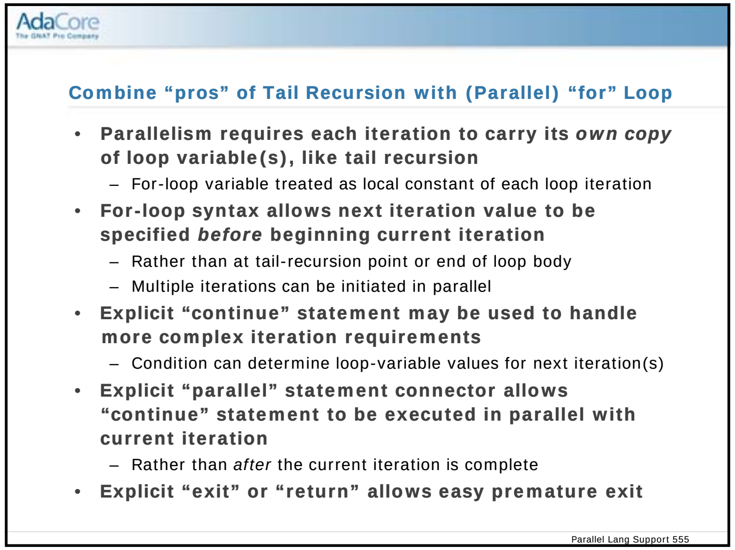
## *While* Loops and Tail Recursion Issues

\+ **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**

## Summary

- When jobs are no longer sequential, a new world opens, whose execution (and scheduling) model differs very much from what we saw before
  - ❑ Sequential jobs in concurrent tasks help organize the application according to collaboration patterns
  - ❑ Parallel jobs are *not* collaborative: they help divide the workload in independent chunks
- Parallel-job programming used to be a non-real-time brute-force *high-performance computing* business only
- The convergence of HPC and embedded real-time causes predictable parallel programming to become a new important dimension of research

## Selected readings

- R.D. Blumofe, C.E. Leisersen (**1999**)
  *Scheduling Multithreaded Computations by Work Stealing*
  DOI: 10.1145/324133.324234
- U.A. Acar, G.E. Blelloch, R.D. Blumofe (**2000**)
  *The Data Locality of Work Stealing*
  DOI: 10.1145/341800.341801