Shared-memory Parallel Programming with Cilk Plus (Parts 2-3)

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Last Thursday

- Why multithreaded programming models?
- Threaded programming models
- Cilk Plus
  - —tasks
  - —algorithmic complexity measures
  - —scheduling
  - —performance and granularity
  - —task parallelism examples
    - – vector addition using divide and conquer
    - – nqueens: exploratory search

Outline for Today

- Cilk Plus
  - —explore speedup and granularity
  - —task parallelism example
    - – cilksort
  - —parallel loops
  - —reducers
- Data race detection with cilkscreen
- Assessing Cilk Plus performance with cilkview

Review: Cilk Plus Parallel Performance Model

$$c_1 = \frac{T_1}{T_s}$$ **work overhead**

$$T_p \le c_1 \frac{T_s}{P} + c_\infty T_\infty$$

"Minimize work overhead (c_1) at the expense of a larger critical path overhead (c_∞), because work overhead has a more direct impact on performance"

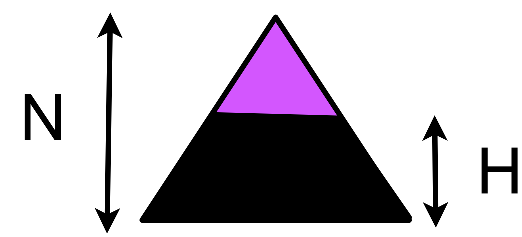
$$T_p \approx c_1 \frac{T_s}{P}$$ **assuming parallel slackness**

Speedup Demo

- **Explore speedup of naive fibonacci program**
  - `cp /projects/comp422/cilkplus/examples/fib ~/fib`
  - `cd ~/fib`
  - examine fib.cpp - a program for computing n^{th} fibonacci #
  - build the examples: `make`
  - experiment with the fibonacci program
    - `make runt W=n` *computes fib(43) with n workers*
    - compute fib(43) for different values of W, $1 \leq W \leq 8$
    - what value of W yields the lowest execution time?
    - what is the speedup vs. the execution time of "`./fib-serial 43`"?
    - how does this compare to the total number of HW threads?

Granularity Demo

- **Explore how changing increasing the granularity of parallel work in fib improves performance (by reducing c_1)**
  - —examine fib-trunc.cpp - a program for computing n^{th} fibonacci #
    - this version differs in that one can execute subtrees of height H sequentially rather than spawning parallel tasks all the way down
  - —build the examples: `make`
  - —experiment with the fibonacci program with truncated parallelism
    - `make runt H=h` *computes fib(43) with lowest H levels serial*
    - compute fib(43) for different values of H, $2 \leq H \leq 44$
    - what value of H yields the lowest execution time
    - what is the speedup vs. the execution time of "`./fib-serial 43`"?
    - how does this compare to the total number of HW threads?



Sorting in Cilk: cilksort

Variant of merge sort

- Divide array into four quarters A,B,C,D of equal size
- Sort each quarter recursively in parallel
- Merge sorted A & B into tA and C & D into tC (in parallel)
- Merge sorted tA and tC into A

High-level sketch

```
void cilksort(low,tmp,size){
  size4 = size/4
  if size <= 1 return input
  cilk_spawn cilksort(A, tA, size4);
  cilk_spawn cilksort(B, tB, size4);
  cilk_spawn cilksort(C, tC, size4);
  cilksort(D, tD, size-3*size4);
  cilk_sync;
  cilk_spawn cilkmerge(tA, tA + size4-1, tB, tB + size4-1, tA);
  cilkmerge(tC, tC + size4-1, tD, tA + size-1, tC);
  cilk_sync;
  cilkmerge(tA, tC-1, tC, tA + size-1, A);
}
```

Merging in Parallel

- How can you incorporate parallelism into a merge operation?
- Assume we are merging two sorted sequences A and B
- Without loss of generality, assume A larger than B

Algorithm Sketch

1. Find median of the elements in A and B (considered together).
2. Do binary search in A and B to find its position. Split A and B at this place to form A_1, A_2, B_1, and B_2
3. In parallel, recursively merge A_1 with B_1 and A_2 with B_2

Optimizing Performance of cilksort

- Recursively subdividing all the way to singletons is <u>expensive</u>
- When size(remaining sequence) to sort or merge is small (2K)
  - —use sequential quicksort
  - —use sequential merge

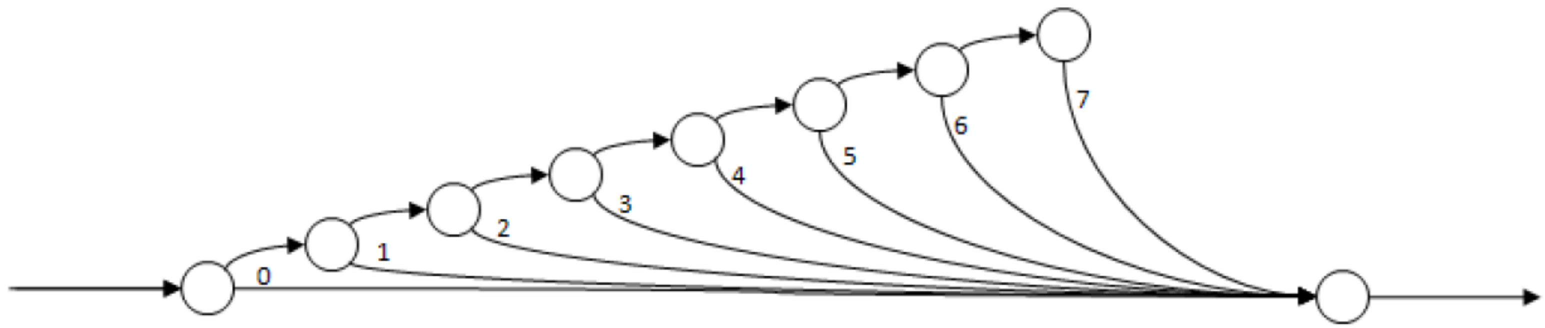
Cilk Plus Parallel Loop: cilk_for

```
cilk_for (T v = begin; v < end; v++) {
    statement_1;
    statement_2;
    ...
}
```

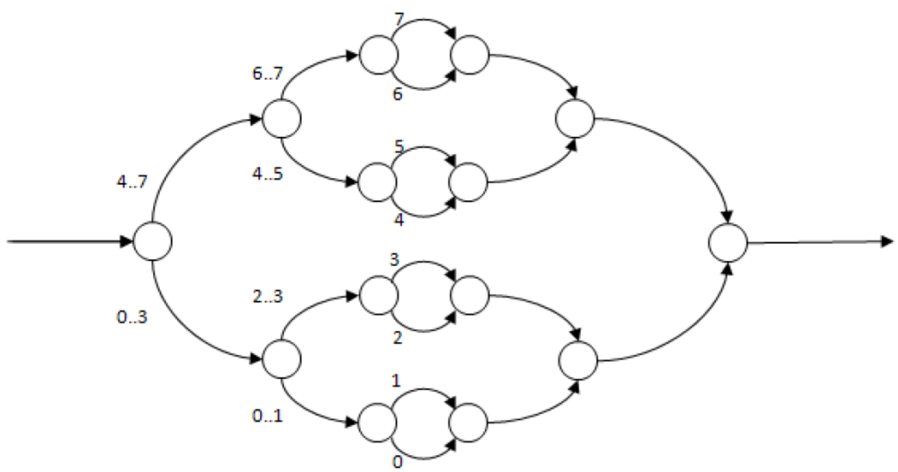
- Loop index v
  - type T can be an integer, ptr, or a *C++ random access iterator*
- Main restrictions
  - runtime must be able to compute total # of iterations on entry to cilk_for
    - must compare v with end value using <, <=, !=, >=, or >
    - loop increment must use ++, --, +=, v = v + incr, or v = v - incr
      - if v is not a signed integer, loop must count up
- Implicit cilk_sync at the end of a cilk_for

Loop with a cilk_spawn vs. cilk_for

- **for (int i = 0; i < 8; i++) { cilk_spawn work(i); } cilk_sync;**



- **cilk_for (int i = 0; i < 8; i++) { work(i);}**



Note: computation on edges

cilk_for uses divide-and-conquer

Figure credits: Intel Cilk++ Programmer's Guide. Document # 322581-001US.

Restrictions for cilk_for

- No early exit
  - —no break or return statement within loop
  - —no goto in loop unless target is within loop body
- Loop induction variable restrictions
  - —cilk_for (unsigned int i, j = 42; j < 1; i++, j++) { ... }
    - – only one loop variable allowed
  - —cilk_for (unsigned int i = 1; i < 16; ++i) i = f();
    - – can't modify loop variable within loop
  - —cilk_for (unsigned int i = 1; i < x; ++i) x = f();
    - – can't modify end within loop
  - —int i; cilk_for (i = 0; i<100; i++) { ... }
    - – loop variable must be declared in loop header

cilk_for Implementation Sketch

- **Recursive bisection used to subdivide iteration space down to chunk size**

```
void run_loop(first, last)
{

    if (last - first) < grainsize)
    {
        for (int i=first; i<last ++i) LOOP_BODY;
    }
    else
    {
        int mid = (last-first)/2;
        cilk_spawn run_loop(first, mid);
        run_loop(mid, last);
    }
}
```

cilk_for Grain Size

- Iterations divided into *chunks* to be executed serially
  - —chunk is sequential collection of one or more iterations
- Maximum size of chunk is called *grain size*
  - —grain size too small: spawn overhead reduces performance
  - —grain size too large: reduces parallelism and load balance
- Default grain size
  - —#pragma cilk grainsize = min(2048, N / (8*p))
- Can override default grain size
  - —#pragma cilk grainsize = expr
    - – expr is any C++ expression that yields an integral type (e.g. int, long)

      e.g. #pragma cilk grainsize = n/(4*__cilkrts_get_nworkers())
  - —pragma must immediately precede cilk_for to which it applies

Parallelizing Vector Addition

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Cilk Plus

```
void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    cilk_spawn vadd (A, B, n/2);
    vadd (A+n/2, B+n/2, n-n/2);
    cilk_sync;
  }
}
```

```
void vadd (real *A, real *B, int n){
  int i; cilk_for (i=0; i<n; i++) A[i]+=B[i];
}
```

The Problem with Non-local Variables

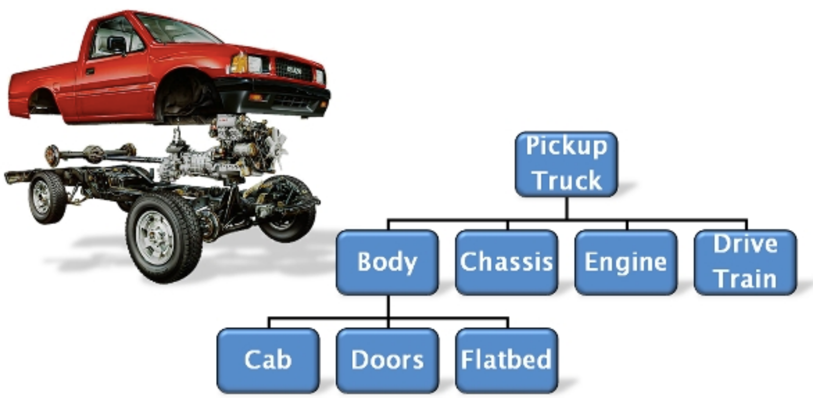
- Nonlocal variables are a common programming construct
  - —global variables = nonlocal variables in outermost scope
  - —nonlocal = declared in a scope outside that where it is used
- Example

```
int sum = 0;
for(int i=1; i<n; i++) {
  sum += i;
}
```

- Rewriting parallel applications to avoid them is painful

Collision Detection

Automaker: hierarchical 3D CAD representation of assemblies



Computing a cutaway view

```
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)  {
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
        output_list.push_back(x);
    break;
  case Node::INTERNAL:
    for (Node::const_iterator
              child = x->begin();
              child != x->end();
              ++child)
        walk(child);
    break;
  }
}
```

Adding Cilk Plus Parallelism

Computing a cutaway view in parallel

```
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)  {
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
        output_list.push_back(x);
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator
              child = x->begin();
              child != x->end();
              ++child)
        walk(child);
    break;
  }
}
```

Global variable causes data races!

Solution 1: Locking

Computing a cutaway view in parallel

```
Node *target;
std::list<Node *> output_list;
mutex m;
...
void walk(Node *x)  {
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
    { m.lock(); output_list.push_back(x); m.unlock(); }
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator
              child = x->begin();
              child != x->end();
              ++child)
        walk(child);
    break;
  }
}
```

- **Add a mutex to coordinate accesses to output_list**
- **Drawback: lock contention can hurt parallelism**

Solution 2: Refactor the Code

```
Node *target;
 std::list<Node *> output_list;
 ...
 void walk(Node *x, std::list<Node *> &o_list) {
   switch (x->kind) {
   case Node::LEAF:
     if (target->collides_with(x))
         o_list.push_back(x);
     break;
   case Node::INTERNAL:
     std::vector<std::list<Node *>>
           child_list(x.num_children);
     cilk_for (Node::const_iterator
               child = x->begin();
               child != x->end();
               ++child)
         walk(child, child_list[child]);
     for (int i=0; i < x.num_children; ++i)
         o_list.splice(o_list.end(), child_list[i]);
     break;
   }
```

- **Have each child accumulate results in a separate list**
- **Splice them all together**
- **Drawback: development time, debugging**

Solution 3: Cilk Plus Reducers

```
Node *target;

cilk::reducer_list_append<Node *> output_list;
 ...
 void walk(Node *x) {
   switch (x->kind) {
   case Node::LEAF:
     if (target->collides_with(x))
         output_list.push_back(x);
     break;
   case Node::INTERNAL:
     cilk_for (Node::const_iterator
               child = x->begin();
               child != x->end();
               ++child)q
         walk(child);
     break;
   }
 }
```

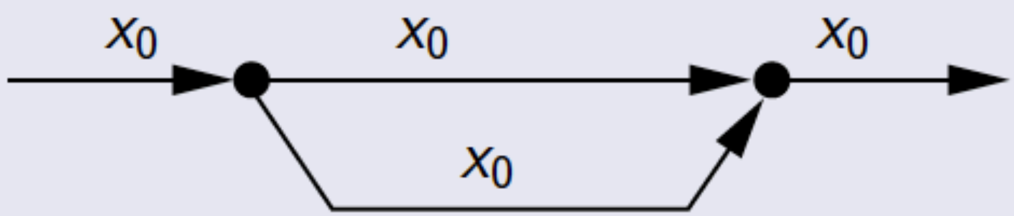
- **Resolve data races without locking or refactoring**
- **Parallel strands may see different views of reducer, but these views are combined into a single consistent view**

Cilk Plus Reducers

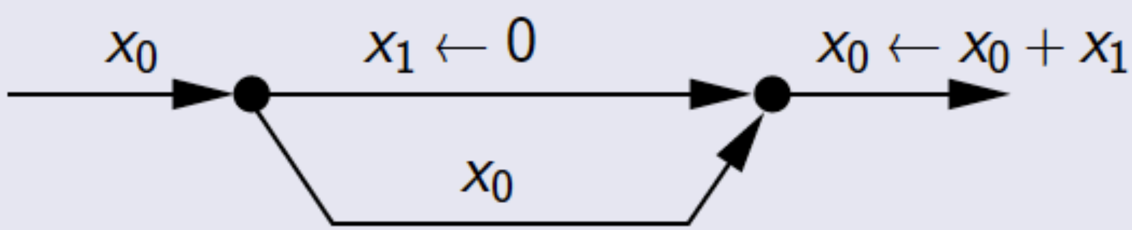
- Reducers support update of nonlocal variables without races
  —deterministic update using associative operations
    – e.g., global sum, list and output stream append, ...
    – result using is same as serial version
      independent of # processors or scheduling
- Can be used without significant code restructuring
- Can be used independently of the program's control structure
  — unlike constructs defined only over loops
- Implemented efficiently with minimal overhead
  —they don’t use locks in their implementation
    – avoids loss of parallelism from enforcing mutual exclusion when updating shared variables

Reducers

Serial execution (depth first):



Parallel execution:



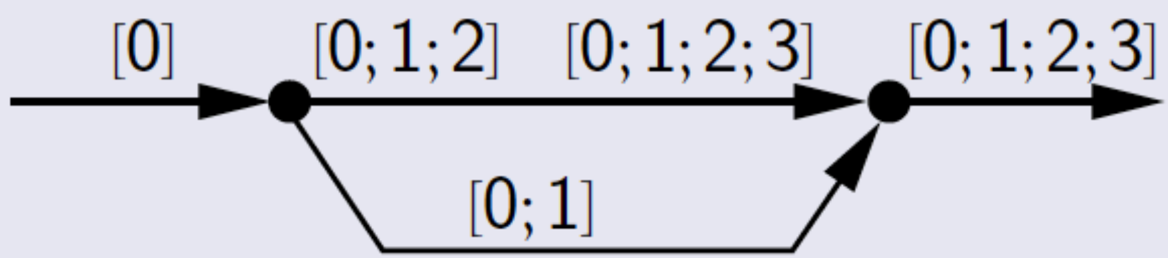
Matteo Frigo, Pablo Halpern, Charles E. Leiserson, Stephen Lewin-Berlin, Reducers and other Cilk++ hyperobjects. Slides for *SPAA'09,* August 11–13, 2009, Calgary, Alberta, Canada.

Reducing Over List Concatenation

Program:

```
x.append(0);
cilk_spawn x.append(1);
x.append(2);
x.append(3);
cilk_sync;
```

Serial execution:



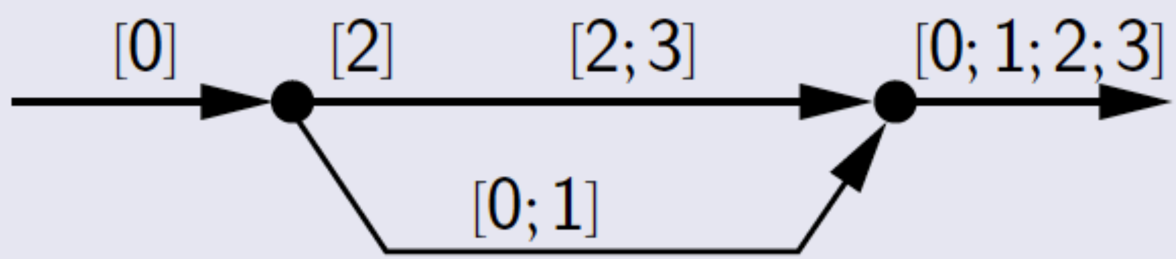
Matteo Frigo, Pablo Halpern, Charles E. Leiserson, Stephen Lewin-Berlin, Reducers and other Cilk++ hyperobjects. Slides for *SPAA'09,* August 11–13, 2009, Calgary, Alberta, Canada.

Reducing Over List Concatenation

Program:

```
x.append(0);
cilk_spawn x.append(1);
x.append(2);
x.append(3);
cilk_sync;
```

Parallel execution:



Matteo Frigo, Pablo Halpern, Charles E. Leiserson, Stephen Lewin-Berlin, Reducers and other Cilk++ hyperobjects. Slides for *SPAA'09,* August 11–13, 2009, Calgary, Alberta, Canada.

Using Cilk Plus Reducers

- Include the appropriate Cilk Plus reducer header file

  `reducer_opadd.h`, `reducer_min.h`, `reducer_max.h`, `reducer_opor.h`, `reducer_opand.h`, `reducer_opxor`, `reducer_list.h`, `reducer_ostream.h`

- Declare a variable as a reducer rather than a standard type
  - global sum
    - `cilk::reducer_opadd<unsigned long> sum`
  - list reducer
    - instead of "`std::list<int> sequence`", use `cilk::reducer_list_append<int> sequence`

- Use reducers in the midst of work that includes parallelism created with `cilk_spawn` or `cilk_for`

- Retrieve the reducer's terminal value with `var.get_value()` after the parallel updates to the reducer are complete

reducer Demo

- **Explore benefits of reducers**
  - `cp /projects/comp422/cilkplus/examples/sum ~/sum`
  - `cd ~/sum`
  - inspect the programs
    - race.cpp - a cilk_for summation with a race
    - lock.cpp - a safe cilk_for summation using a lock
    - reducer.cpp - a safe cilk_for summation using a reducer
  - `make run`

- On STIC, see /projects/comp422/cilkplus/examples/features-tutorial
- In that directory, there is subdirectory “reducers” that contains more reducer examples

Cilk++ Reducers Under the Hood

- **If no steal occurs, a reducer behaves like a normal variable**
- **If a steal occurs**
  - **the continuation receives a view with an identity value**
  - **the child receives the reducer as it was prior to the spawn**
  - **at the corresponding cilk_sync**
    - **the value in the continuation is merged into the reducer held by the child using the reducer's `reduce` operation**
    - **the new view is destroyed**
    - **the original (updated) object survives**

Concurrency Cautions

- **Only limited guarantees between descendants or ancestors**
  - **—DAG precedence order maintained and nothing more**
  - **—don’t assume atomicity between different procedures!**

Race Conditions

- **Data race**
  - —two parallel strands access the same data
  - —at least one access is a write
  - —no locks held in common
- **General determinacy race**
  - —two parallel strands access the same data
  - —at least one access is a write
  - —a common lock protects both accesses

A Data Race Example

- **Example**

```
int sum = 0;
cilk_for(int i=1; i<n; i++) {
  sum += i;
}
```

- **What can go wrong?**

—concurrent reads and writes can interleave in unpredictable ways

time ↓

read sum
read sum
write sum + i_j
write sum + i_k

legend
thread n
thread m

— the update by thread m is lost!

Cilkscreen

- Detects and reports data races when program terminates
  - —finds all data races even those by third-party or system libraries
- Does not report determinacy races
  - —e.g. two concurrent strands use a lock to access a queue
    - – enqueue & dequeue operations could occur in different order

      potentially leads to different result

Race Detection Strategies in Cilkscreen

- **Lock covers**
  - —two conflicting accesses to a variable don't race if some lock L is held while each of the accesses is performed by a strand
- **Access precedence**
  - —two conflicting accesses do not race if one must precede the other
    - access A is by a strand X, which precedes the cilk_spawn of strand Y which performs access B
    - access A is performed by strand X, which precedes a cilk_sync that is an ancestor of strand Y

Cilkscreen Race Example

```
#include <stdio.h>
#include "mutex.h"

long sum = 0;
mutex m;

#ifdef SYNCH
#define LOCK m.lock()
#define UNLOCK m.unlock()
#else
#define LOCK
#define UNLOCK
#endif
```

```
void do_accum(int l, int u)
{
        if (u == l) { LOCK; sum += l; UNLOCK; }
        else {
          int mid = (u+l)/2;
          cilk_spawn do_accum(l, mid);
          do_accum(mid+1, u);
        }
}
int main()
{
        do_accum(0, 1000);
        printf("sum = %d\n", sum);

        long ssum = 0;
        for (int i = 0; i <= 1000; i++) ssum +=i;
        printf("serial sum = %d\n", ssum);
}
```

note: mutex class coded using pthread_mutex lock primitives

Cilkscreen Limitations

- **Only detects races between Cilk Plus strands**
  - **—depends upon their strict fork/join paradigm**
- **Only detects races that occur given the input provided**
  - **—does not prove the absence of races for other inputs**
  - **—choose your testing inputs carefully!**
- **Runs serially, 15-30x slower**
- **Increases the memory footprint of an application**
  - **—could cause an error if memory demand is too large**
- **If you build your program with debug information (compile with -g), cilkscreen will associate races with source line numbers**

Cilkscreen Output

```
Cilkscreen Race Detector V2.0.0, Build 3229
summing integers from 0 to 20000

Race condition on location 0x6016f0
  write access at 0x400b7f: (/home/johnmc/examples/races/sum2.c:22, do_accum+0x169)
  read access at 0x400b78: (/home/johnmc/examples/races/sum2.c:22, do_accum+0x162)
    called by 0x400ca9: (/home/johnmc/examples/races/sum2.c:26, do_accum+0x293)
    called by 0x400c8f: (/home/johnmc/examples/races/sum2.c:25, do_accum+0x279)
    called by 0x400c8f: (/home/johnmc/examples/races/sum2.c:25, do_accum+0x279)
    ...
    called by 0x400c8f: (/home/johnmc/examples/races/sum2.c:25, do_accum+0x279)
    called by 0x400c8f: (/home/johnmc/examples/races/sum2.c:25, do_accum+0x279)
    called by 0x400e47: (/home/johnmc/examples/races/sum2.c:37, main+0x85)

Race condition on location 0x6016f0
  write access at 0x400b7f: (/home/johnmc/examples/races/sum2.c:22, do_accum+0x169)
  write access at 0x400b7f: (/home/johnmc/examples/races/sum2.c:22, do_accum+0x169)
    called by 0x400ca9: (/home/johnmc/examples/races/sum2.c:26, do_accum+0x293)
    called by 0x400c8f: (/home/johnmc/examples/races/sum2.c:25, do_accum+0x279)
    called by 0x400c8f: (/home/johnmc/examples/races/sum2.c:25, do_accum+0x279)
    ...
    called by 0x400c8f: (/home/johnmc/examples/races/sum2.c:25, do_accum+0x279)
    called by 0x400c8f: (/home/johnmc/examples/races/sum2.c:25, do_accum+0x279)
    called by 0x400e47: (/home/johnmc/examples/races/sum2.c:37, main+0x85)

sum = 200010000
serial sum = 200010000
2 errors found by Cilkscreen
Cilkscreen suppressed 119998 duplicate error messages
```

cilkscreen Demo

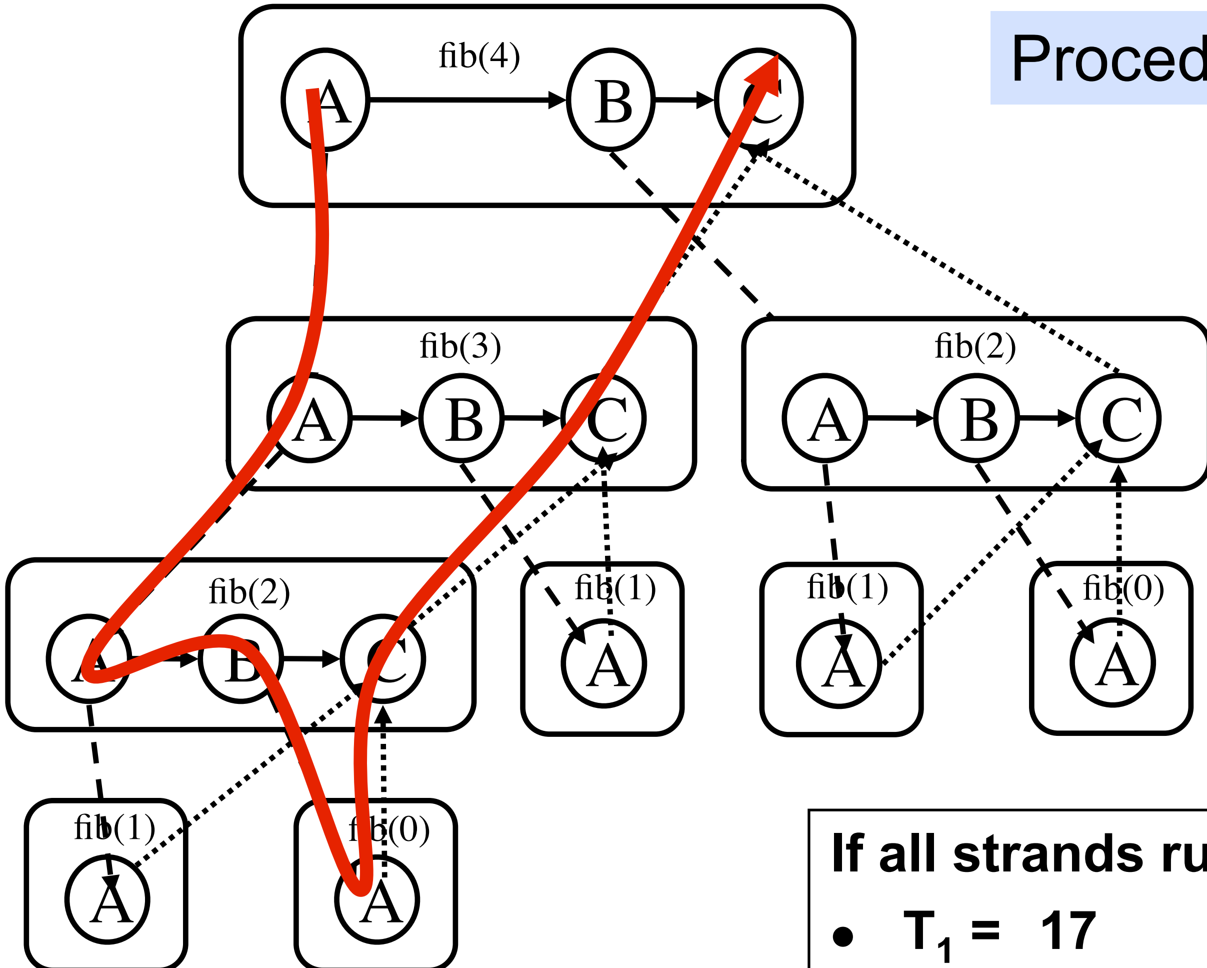
- Explore cilkscreen race detection
  - `cp /projects/comp422/cilkplus/examples/races ~/races`
  - `cd ~/races`
  - inspect the programs
    - `sum1.c - a cilk_for summation with a race`
    - `sum2.c - a task parallel summation w/ optional mutex`
    - `nocover.c - program using locks with races`
    - `cover.c - lock cover prevents races`
  - `make run`

Performance Measures

- T_s = serial execution time
- T_1 = execution time on 1 processor (total work), $T_1 \geq T_s$
- T_p = execution time on P processors
- T_∞ = execution time on infinite number of processors
  - —longest path in DAG
    - – length reflects the cost of computation at nodes along the path
  - —known as "critical path length"

Work and Critical Path Example

Procedure oriented view

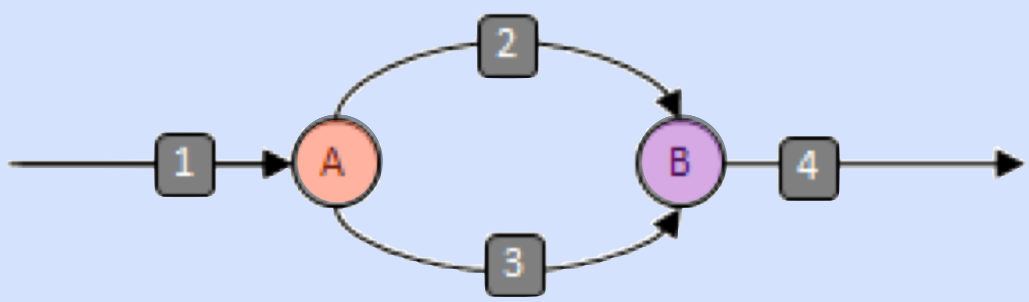


If all strands run in unit time

- T_1 = 17
- T_∞ = 8 (critical path length)

Execution DAG View

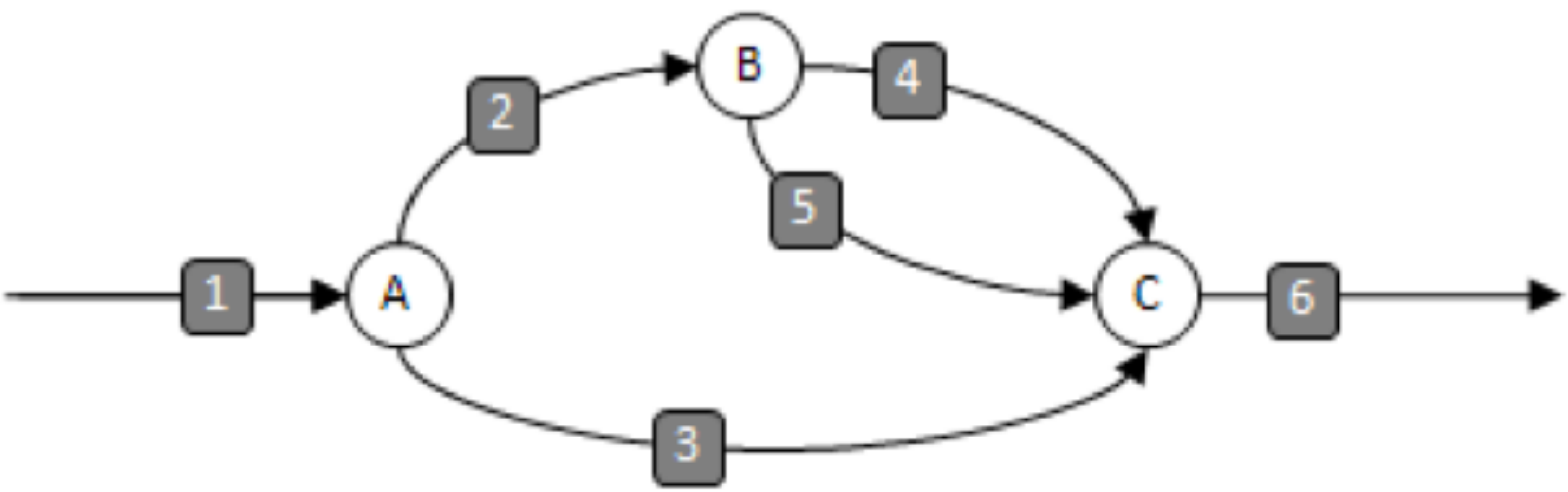
- Cilk Plus uses the word "strand" for a serial section of the program
- A "knot" is a point where three or more strands meet
- Two kinds of knots
  - —spawn knots: one input strand, two output strands
  - —sync knots: two or more input strands, one output strand




```
...
do_stuff1();
cilk_spawn func3();
do_stuff2();
cilk_sync;
do_stuff4();
...
```

Another Execution DAG

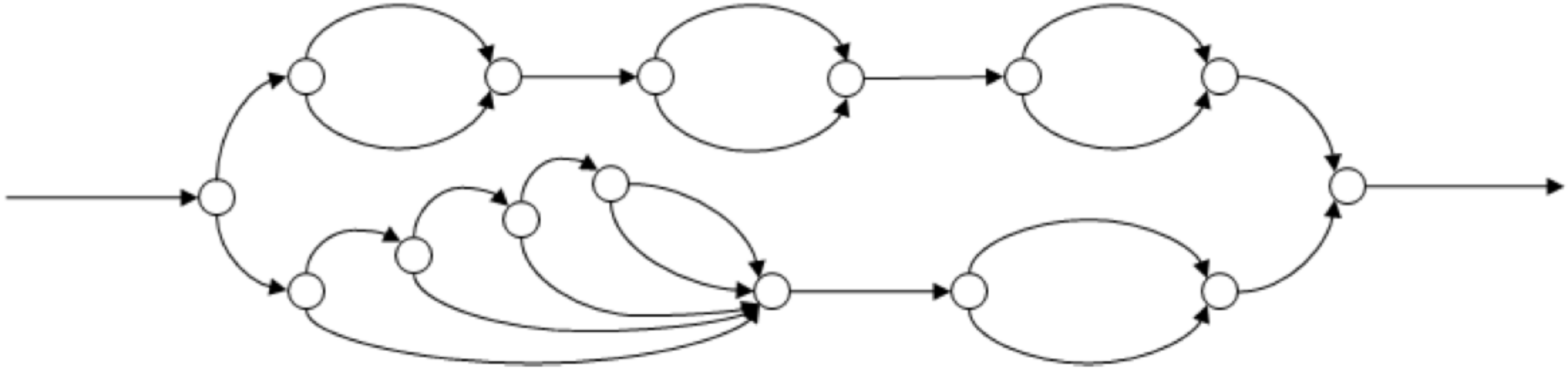
- **DAG represents the series-parallel structure of the execution of a Cilk Plus program**
- **Example:**
  - —two spawns (A) & (B)
  - —one sync (C)



Note: computation on edges

Work and Span

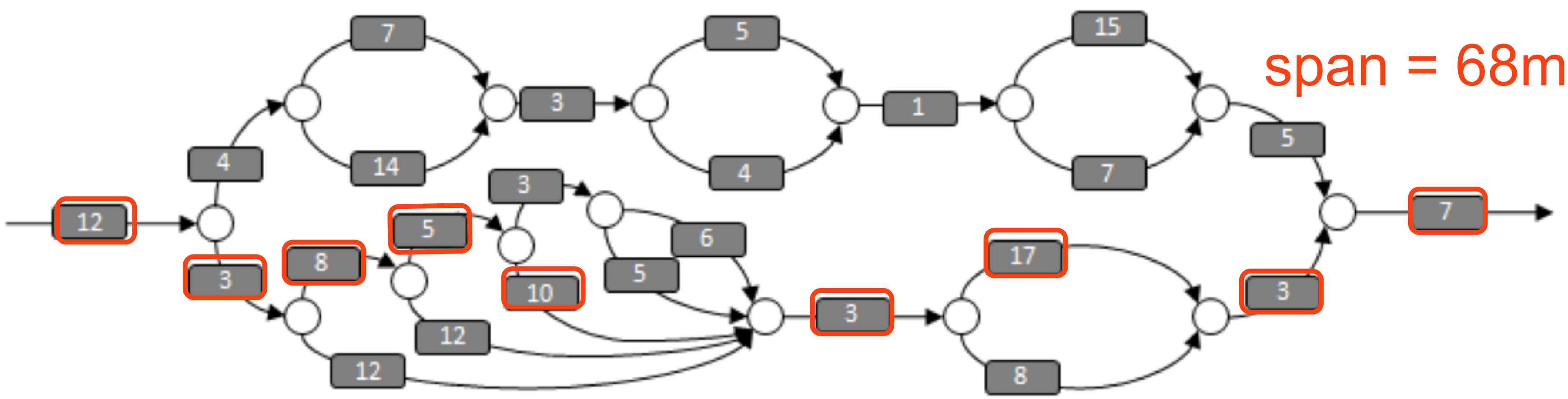
- **Edges represent serial computation (work)**



- **Span: most expensive path from beginning to end**
  - **—also known as critical path length**

work = 181ms

span = 68ms



Note: computation on edges

cilkview

- **Rewrites executable to measure execution in terms of work and span**
  - **—measures**
    - **work - total # instructions executed, w/o parallel ovhd**
    - **span - # instructions executed on the critical path (w/o ovhd)**
    - **burdened span - # instructions executed on critical path (incl ovhd)**
    - **parallelism - work/span (max speedup on infinite cores, w/o ovhd)**
    - **burdened parallelism - work/(burdened span)**
    - **number of spawns/syncs**
    - **average instructions per strand - work/strands**
    - **strands along span - # strands in the critical path**
    - **average instructions / strand on span = work/(strands along span)**
    - **total number of atomic instructions - e.g., used for locks**
    - **frame count**
- **Predicts speedup on various numbers of processors based on work and span**

cilkview Demo

- **Explore cilkview for performance analysis**
  - `cp /projects/comp422/cilkplus/examples/fib ~/fib`
  - `cd ~/fib`
  - `cilkview ./fib 20`
  - `cilkview ./fib 30`
  - `cilkview ./fib 35`
  - `cilkview ./fib-trunc 35 10`

Remaining Cilk Plus Features

- **Introduce Cilk Plus array notation**
- **Gain more experience with Cilk Plus reducers**
  - `cp /projects/comp422/cilkplus/examples/features-tutorial .`
    - **array_notations: vector notation in Cilk Plus**
    - **reducers: more reducer examples**
  - **each directory contains a Makefile that can build and run all examples**

Cilk Plus Array Notation

- **Elementwise arithmetic**

```
c[:] = a[:] + 5;
```

- **Set even rows in a 2D array**

```
b[0:5:2][:] = 12;
```

- **Vector conditionals**

```
// Check and report each element containing 5 w/ Array Notation
if (5 == a[:]) an_results[:] = "Matched";
else an_results[:] = "Not Matched";
```

- **Vector conditionals**

```
// Call a fn on each element of a vector using Array Notation
fn(a[:]);
```

– On STIC,
  see `/projects/comp422/cilkplus/examples/features-tutorial`

Recall: Task Scheduling in Cilk

Strategies

- **Work-stealing: processor looks for work when it becomes idle**
- **Lazy parallelism: don't realize parallelism until necessary**
  - **—benefits:**
    - **executes with precisely as much parallelism as needed**
    - **minimizes the number of threads that must be set up**
    - **runs with same efficiency as serial program on uniprocessor**

Compilation Strategy

`MIT Cilk` generates two copies of each procedure

- Fast clone: for optimized execution on a single processor
  - spawned threads are fast
- Slow clone: triggered by work stealing, full parallel support
  - used to handle execution of "stolen procedure frames"
  - supports Cilk's work-stealing scheduler
  - few steals when enough parallel slackness exists
    - speed of slow copy is not critical for performance
- "Work-first" principle: minimize cost in fast clone

Two Schedulers

- Nanoscheduler: compiled into cilk program
  - —execute cilk function and spawns in exactly the same order as C
  - —on one PE: when no microscheduling needed, same order as C
  - —efficient coordination with microscheduler
- Microscheduler
  - —schedule procedures across a fixed set of processors
  - —implementation: randomized work-stealing scheduler
    - – when a processor runs out of work, it becomes a thief
    - – steals from victim processor chosen uniformly at random

Nanscheduler Sketch

- Upon entering a cilk function
  - —allocate a frame in the heap
  - —initialize frame to hold function's state
  - —push the frame on the bottom of a deque
    - – frame on stack ↔ frame in deque
- At a spawn
  - —save function state into the frame
    - – only live, dirty variables
  - —save the entry number into the frame
  - —call spawned procedure as a function
- After each spawn
  - —check to see if if parent has been stolen
    - – if frame is still in the deque, it has not
  - —if so, clean up C stack
- Each sync becomes a no-op
- When the procedure returns
  - —pop the frame off the deque
  - —resume caller after the spawn

Fast clone

```
int fib (int n)
{
    fib_frame *f;                   frame pointer
    f = alloc(sizeof(*f));          allocate frame
    f->sig = fib_sig;               initialize frame
    if (n<2) {
        free(f, sizeof(*f));        free frame
        return n;
    }
    else {
        int x, y;
        f->entry = 1;               save PC
        f->n = n;                   save live vars
        *T = f;                     store frame pointer
        push();                     push frame
        x = fib (n-1);              do C call
        if (pop(x) == FAILURE)      pop frame
            return 0;               frame stolen
        ...                         second spawn
        ;                           sync is free!
        free(f, sizeof(*f));        free frame
        return (x+y);
    }
}
```

Fast Clone and Nanoscheduler

- Fast clone is never stolen
  - —converted to slow when steal occurs
  - —enables optimizations
- No sync needed in fast clone
  - —no children have been spawned
- Frame saves state:
  - —PC (entry number)
  - —live, dirty variables
- Push and pop must be fast

Nanoscheduler Overheads

Basis for comparison: serial C

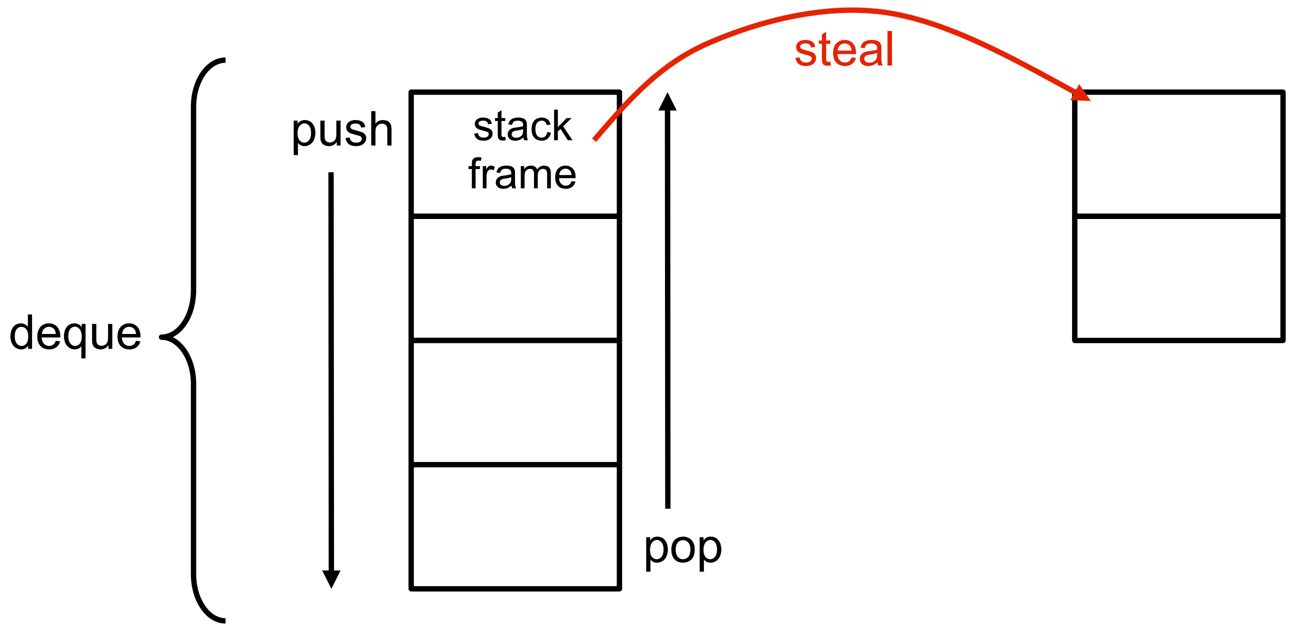
- **Allocation and initialization of frame, push onto ‘stack’**
  - —a few assembly instructions
- **Procedure’s state needs to be saved before each spawn**
  - —entry number, live variables
- **Check whether frame is stolen after each spawn**
  - —two reads, compare, branch
- **On return, free frame - a few instructions**
- **One extra variable to hold frame pointer**

Runtime Support for Scheduling

Each processor has a ready deque (doubly ended queue)

- Tail: worker adds or removes procedures (like C call stack)
- Head: thief steals from head of a victim's deque

Deque for a Process



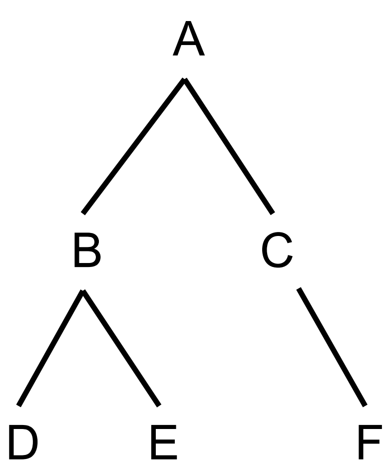
- *Deque* grows downward
- *Stack frame* contains local variables for a procedure invocation
- Procedure call → new frame is pushed onto the bottom of the deque
- Procedure return → bottom frame is popped from the deque
- Deque maintains order (synchronizes) between caller and callee

Cilk's Cactus Stacks

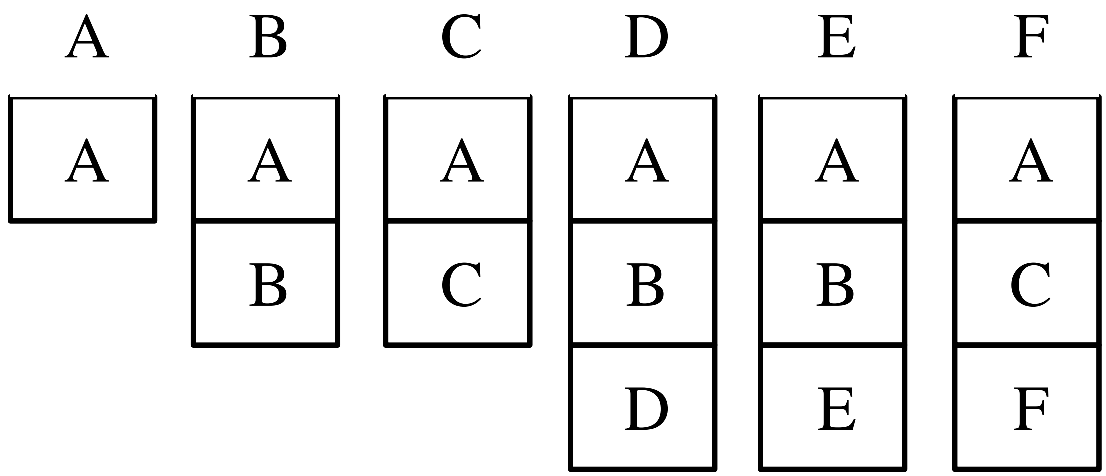
A cactus stack enables sharing of a C function's local variables

```
void A() { B(); C(); }
void B() { D(); E(); }
void C() { F(); }
void D() {}
void E() {}
void F() {}
```

call tree



each procedure's view of stack



Rules

- —**pointers can be passed down call chain**
- —**only pass pointers up if they point to heap**
  - – **functions <u>cannot</u> return ptrs to local variables**

Microscheduler

Schedule procedures across a fixed set of processors

- **When a processor runs out of work, it becomes a thief**
  - —steals from victim processor chosen uniformly at random
- **When it finds victim with frames in its deque**
  - —takes the topmost frame (least recently pushed)
  - —places frame into its own deque
  - —gives the corresponding procedure to its own nanoscheduler
- **Microscheduler executes slow clone**
  - —receives only pointer to frame as argument
    - – real args and local state in frame
  - —restores pgm counter to proper place using switch stmt (Duff's device)
  - —at a sync, must wait for children
  - —before the procedure returns, place return value into frame

Coordinating Thief and Worker

Options

- **Always use a lock to manipulate each worker's deque**
- **Use protocol that only relies on atomicity of read and write**
  - **—based on ideas from a locking protocol by Dijkstra**

Simplified THE Protocol (Without the 'E')

- **Shared memory deque**
  - T: first unused
  - H: head
  - E: exception
- **Work-first**
  - move costs from worker to thief
- **One worker per deque**
- **One thief at a time**
  - enforced by lock

Worker/Victim

```
push() {
  T++;
}

pop() {
  T--;
  if (H > T) {
    T++;
    lock(L);
    T--;
    if (H > T) {
      T++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Thief

```
steal() {
  lock(L);
  H++;
  if (H > T) {
    H--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

- actions on tail contribute to work overhead
- actions on head contribute only to critical path overhead

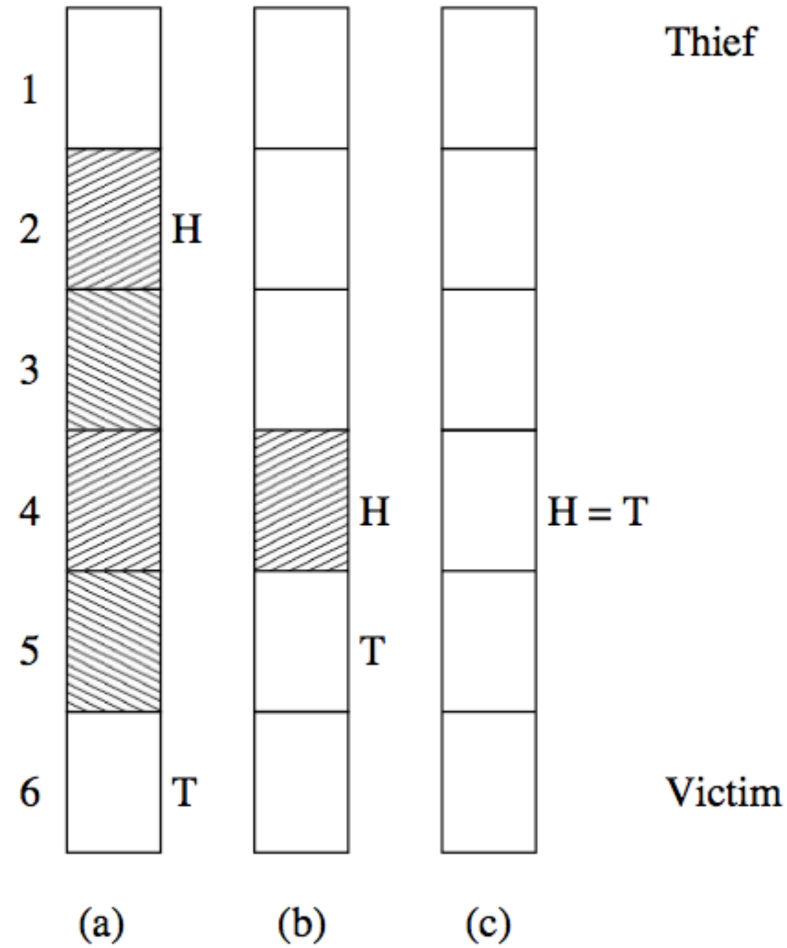
Deque Pop

Three cases

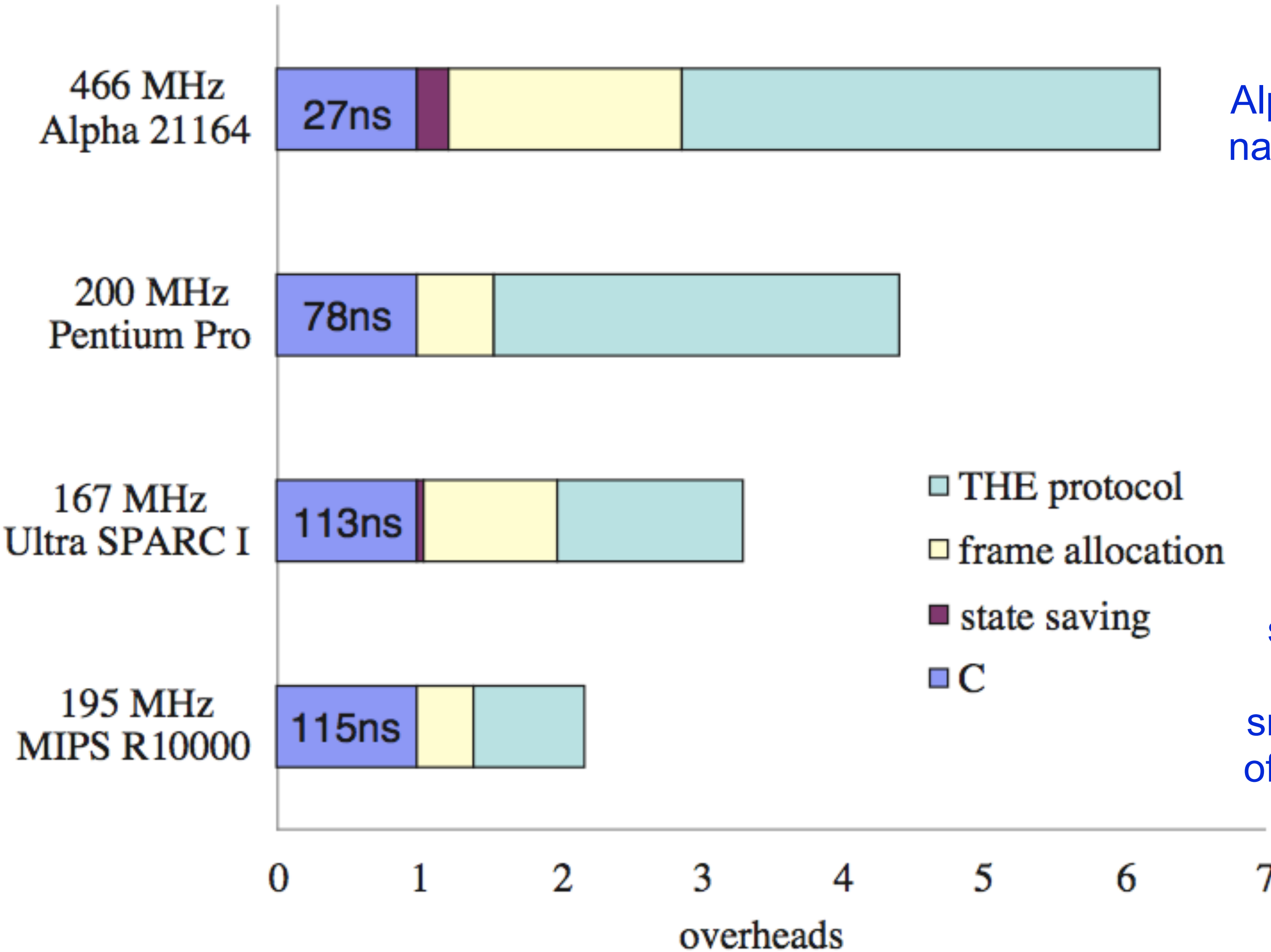
(a) no conflict

(b) At least one (thief or victim) finds (H > T) and backs up; other succeeds

(c) Deque is empty, both threads return



Work Overhead for `fib`



Alpha has fast native function calls

state saving overhead small because of write buffers

References

- Charles E. Leiserson. Cilk LECTURE 1. Supercomputing Technologies Research Group. Computer Science and Artificial Intelligence Laboratory. http://bit.ly/mit-cilk-lec1
- Charles Leiserson, Bradley Kuzmaul, Michael Bender, and Hua-wen Jing. MIT 6.895 lecture notes - Theory of Parallel Systems. http://bit.ly/mit-6895-fall03
- Matteo Frigo, Charles Leiserson, and Keith Randall. The implementation of the Cilk-5 multithreaded language. In PLDI (Montreal, Quebec, Canada, June 17 - 19, 1998), 212-223.
- Intel Cilk++ Programmer’s Guide. Document # 322581-001US.
- Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. SPAA '09, 79-90. Talk Slides. April 11, 2009. http://bit.ly/reducers