# Efficient Work Stealing for Fine-Grained Parallelism

Karl-Filip Faxén

Swedish Institute of Computer Science

November 26, 2009

# Task parallel fib in Wool

```
TASK_1( int, fib, int, n )
{
  if( n<2 ) {
    return n;
  } else {
    int a,b;
    SPAWN( fib, n-2 );
    a = CALL( fib, n-1 );
    b = SYNC( fib );
    return a+b;
  }
}
```

# Two kinds of fine-grainedness

Task granularity How often are tasks spawned?

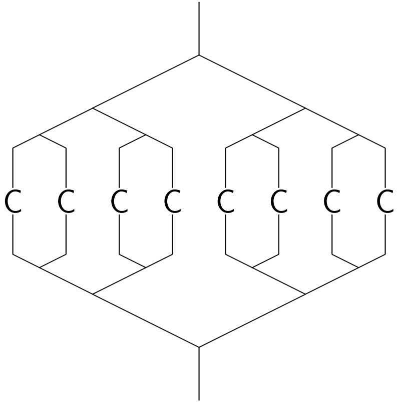
$$G_T = T_S / N_T$$

Load balancing granularity How often must load balancing (migration, stealing) be done?

$$G_L = T_S / N_M$$

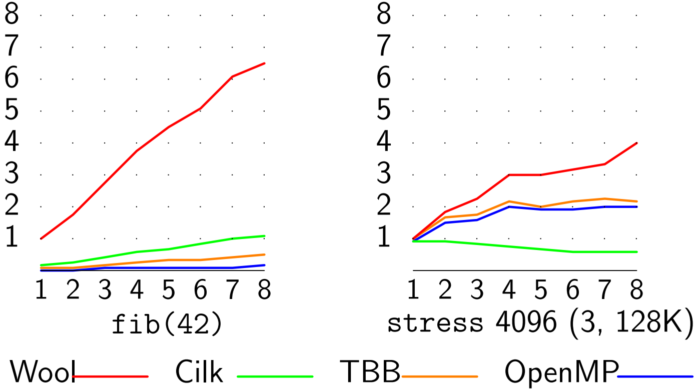
- $T_S$ is serial run-time with no parallelism overhead
- $N_T$ is number of tasks spawned
- $N_M$ is number of migrations (steals in a work stealing implementation)

# The stress program




- Repeat $r$ times (figure shows one repetition):
  - spawn a tree of depth $d$ of tasks ($d = 3$ in figure);
    - the leaves do empty loop C for $n$ iterations ($2n$ cycles)

# Fine-grain tasks and fine-grain load balancing



# Basic structures

- The tasks are scheduled on top of *worker threads*, one per core
- Each worker has a *worker descriptor* containing
  - A *task pool* with ready tasks for other workers to steal
  - A lock protecting the task pool
- Each task is represented by a *task descriptor* with
  - A pointer to the code to run
  - Arguments for the code
  - Space for return value
  - A pointer to the thief, if stolen

# Designing for fast inlinined tasks

The taskpool

- is a *stack* managed by a `top` pointer in task descriptor
  - push on SPAWN
  - pop on SYNC

  while thieves use a `bot` pointer, also in task descriptor,
- contains task descriptors, *not pointers*
  - simple memory management

Most of the design follows from this.

# Optimizing inlined tasks: Synchronize on task

- SYNC (join) needs to synchronize with thiefs, so takes lock in the baseline
- Avoid taking lock on every SYNC
  - Writes to worker descriptor (makes subsequent thief accesses miss)
  - Slow operation
- Synchronize thief and victim with atomic swap on flag in task descriptor
  - Thiefs still take lock in worker descriptor

# Optimizing inlined tasks: Task specific join

- Generate specialized SYNC for each task (rather than generic SYNC in RTS)
  - Knows which task to call when inling, so can use a direct call, not via pointer in task descriptor
  - Knows type of return value, so can pass that in standard way rather than updating via pointer
- When inlining, this optimization replaces three calls
  - Application to SYNC (an RTS function)
  - RTS to wrapper function (indirect call)
  - Wrapper function to task function

  with two

  - Application to specialized SYNC (inlinable, defined in header)
  - Specialized SYNC to task (within the same file)

# Optimizing inlined tasks: Private tasks

- Avoid atomic swap on each SYNC by making some tasks private
  - A private task can not be stolen, so no synchronization is needed
  - Private tasks can become public (the task descriptor is still built) at the discretion of the owner
  - Owner must check for the need for more public tasks
  - Thiefs notify owner when only *n* public tasks remain

# Results for inlining optimizations

| Version | Time (s) | Overhead (cyc) |
|---|---|---|
| Base | 18.9 | 77 |
| Synchronize on task | 7.8 | 29 |
| Task specific join | 5.9 | 19 |
| Private tasks (no private) | 6.0 | 19 |
| Private tasks (all private) | 3.0 | 3 |
| Seq | 2.4 | 0 |

- Measured by timing parallel version of `fib(42)` on a single processor.
- Overhead calculated as $(T_1 - T_S)/N_T$, that is: time difference divided by number of SPAWNs
  - Measures the marginal overhead over procedure call

# Optimizing steals: peek

- Before trying to lock a victim, check if it has work
- If victim has no work, thief does no write
  - Several thiefs can cache the relevant info in a worker in a cache coherent machine
  - Hence spin locally
  - Important when work is hard to find (low parallelism)
  - When a worker spawns, the write notifies the thieves by means of the coherence protocol

# Optimizing steals: trylock

- When a thief finds a victim with work, it uses `pthread_mutex_trylock` rather than `pthread_mutex_lock`
- If lock is not free, try another victim
  - Contention is expensive
  - Other workers might also have work

# Optimizing steals: nolock

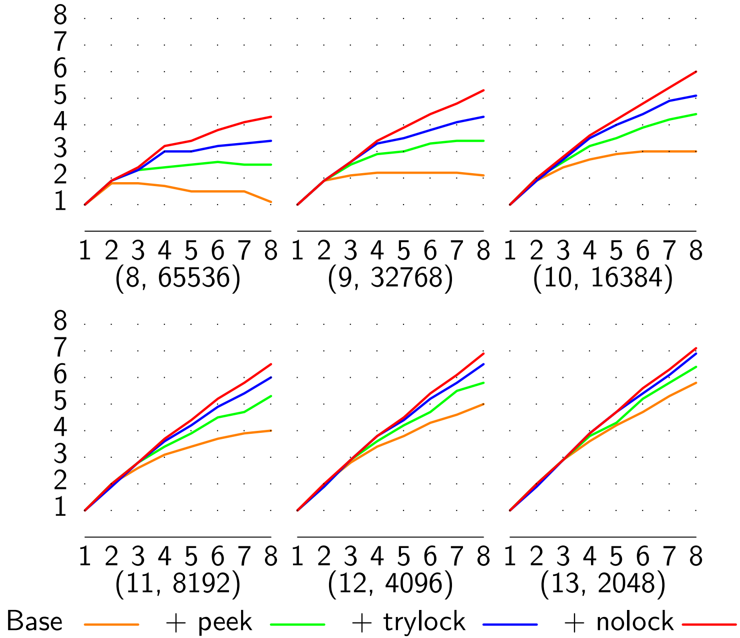
- Get rid of the lock on the worker descripor altogether
- We have mutual exclusion between thieves and owner by the atomic swap on the task descriptor
- This almost gives mutex on worker descriptor (`bot`) since
  - only the task that `bot` points to can be stolen
  - `bot` is only updated upon successful steal

# Optimizing steals: nolock

- However, long delay is possible between read of `bot` and atomic swap (scheduling, interrupts,...)
  - Thief 1 and 2 both read `bot` = 3
  - Thief 1 steals task 3, then finishes it
  - Owner joins with task 3, then with 2 and 1
  - Owner spawns several tasks
  - Thief 2 steals task 3

  Now tasks are stolen out of order; if thief 2 updates `bot`, tasks 1 and 2 becomes invisible until joined with
- Solution: Only update `bot` when it still points at the stolen task

# Optimizing steals: stress tests

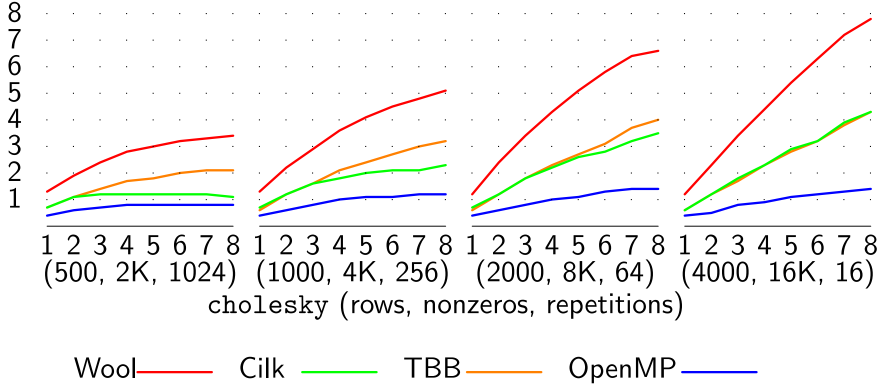


# Comparing Wool with Cilk++, TBB, and OpenMP

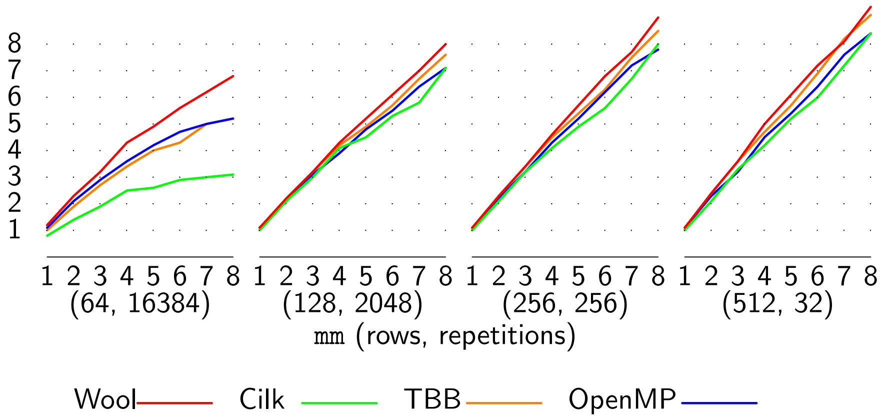
| System | Inlined | 2 | 4 | 8 |
|---|---|---|---|---|
| Wool | 3–19 | 2 200 | 5 600 | 10 400 |
| Cilk++ | 134 | 31 050 | 73 600 | 110 400 |
| TBB | 323 | 5 800 | 14 000 | 30 000 |
| OpenMP | 878 | 4 830 | 9 200 | 20 240 |

- Column labeled Inlined gives cost of inlined tasks computed using `fib`
- Columns labelled 2,4 and 8 give per repetition overhead of `stress` for
    - a tree of depth 1,2 and 3 on 2, 4 and 8 processors (respectively), over
    - a tree of depth 0 on one processor (with same number *n* of leaf loop iterations)

# More measurements: Cholesky



# More measurements: Matrix multiply



# More measurements: Sub String Finder