Distributed Computing and Systems

Chalmers university of technology

HÖGSKOLAN I BORÅS

VETENSKAP FÖR PROFESSION

Brushing the Locks out of the Fur:

A Lock-Free Work Stealing Library Based on Wool

Håkan Sundell

University College of Borås

Parallel Scalable Solutions AB

Philippas Tsigas

Chalmers University of Technology

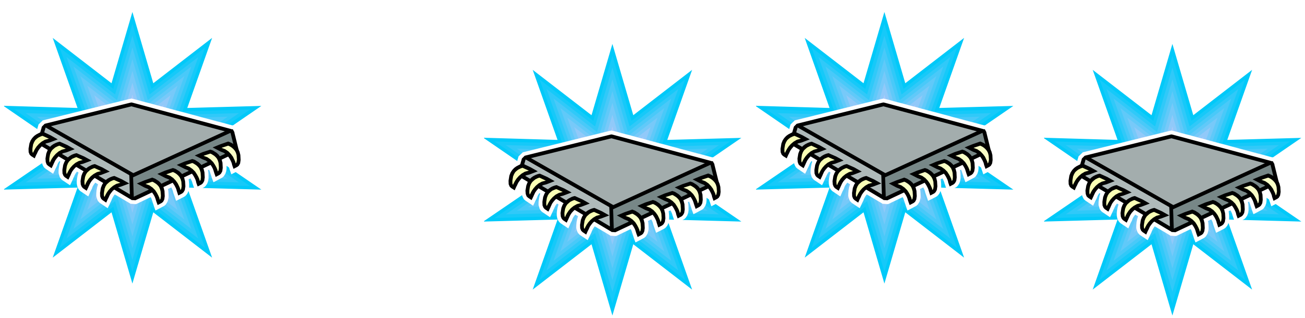
Outline

- Synchronization of Shared Data
- Task Parallelism Library
  - Light-Weight
  - Previous Work
- The Wool Library
  - Architecture
  - Synchronization
- Wool with Lock-Free Synchronization
- Experiments
- Conclusions

Parallel (e.g. Multi-threaded) Software

- Programs consist of many tasks (threads)

- That execute on one or more (logical) processors



Critical Sections

- Problem: operations on shared variables in programming languages are not atomic.

counter=counter+1; = Read + Write

Task 1:	Read *1*		+	Write *2*	
Task 2:		Read *1*	+		Write *2*

! counter=2, but should be 3!

- Straightforward solution: Apply mutual exclusion

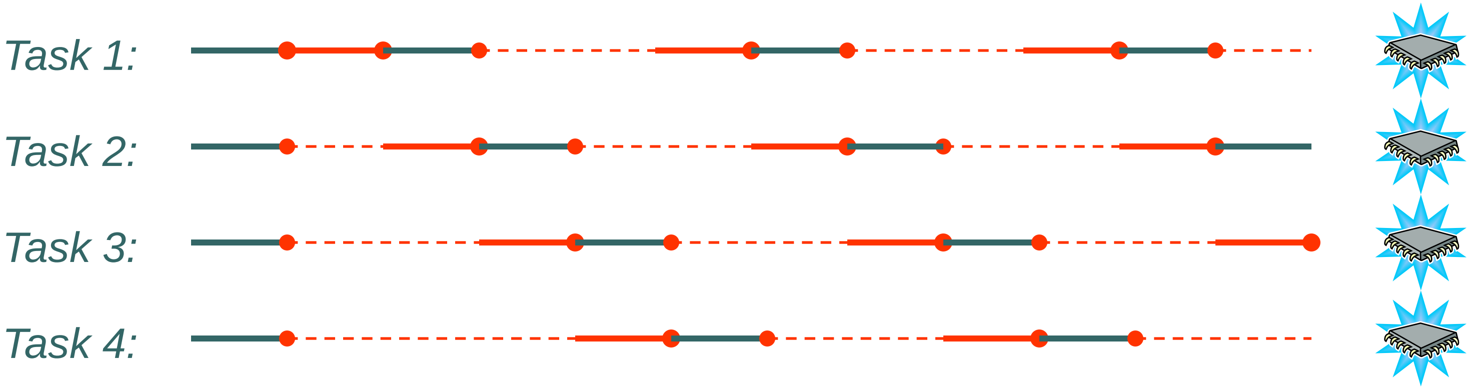
Task 1:	LOCK	Read *1*	+	Write *2*				
Task 2:					LOCK	Read *2*	+	Write *3*

Critical Sections + Scheduling

- **Blocking**. More advanced and pessimistic schedulability analysis.
- **Deadlocks**. Reduced fault-tolerance, if one task fails, other (even all) might also fail.
- **Priority Inversion**. Tasks might not execute with the proper priority even though it was set. Deadlines might be missed.

Critical Sections + Multiprocessors

- **Reduced Parallelism**. Several tasks with overlapping critical sections will cause waiting processors to go idle.



Avoid Critical Sections!

- **Avoid Blocking**. Easier and more optimistic analysis, i.e. less hardware needed.
- **Avoid Deadlocks**. Increased fault-tolerance as failed tasks can not affect others to fail.
- **Avoid Priority Inversion**. Easier and more reliable analysis, and avoids complex and high-overhead solutions.
- **Increased Parallelism**. Increased overall performance, more optimistic analysis, i.e. less hardware needed.

Non-Blocking Synchronization

- The key lies in how mutual exclusion (i.e. mutex, semaphore) is implemented in actual hardware (i.e. processors).
  - Atomic primitives in hardware can atomically update one memory word.
- Sophisticated solutions can exploit the same atomic primitives to support access to shared resources without locks, i.e. non-blocking.

Non-Blocking Algorithms

- **Obstruction-Free.**
  - Guarantees progress in absence of contention.
  - Need extra module for contention management.
- **Lock-Free**.
  - Guarantees that always one operation is making progress.
  - Combined with scheduling information, schedulability analysis can be done.
- **Wait-Free**.
  - Guarantees that any operation will finish in a finite time.
  - Schedulability analysis can be done directly.

Task Parallelism Library

- Fine-grained parallelism is desired for achieving maximal speed-up.
- Spawning threads is expensive.
- Task-based approach:
  - Dynamically (recursively) spawn tasks.
  - Each *Task* contains a relatively small work-load.
    - Usually just a function call.
    - Side-effects are (usually) allowed.
- A *Task Parallelism Library* is usually a multi-threaded program (run-time system) together with a programming framework.

Fibonacci Example (Wool)

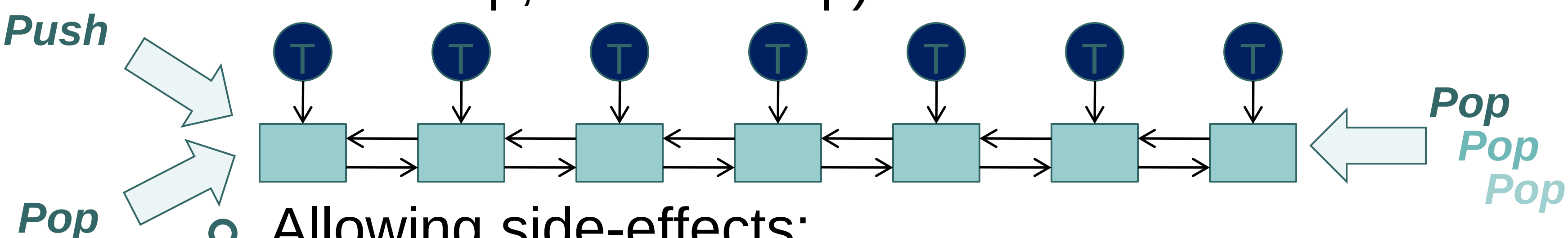
```
#include <stdio.h>
#include <stdlib.h>
#include "wool.h"

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;
        }
}

TASK_2 ( int , main, int , argc,
char **, argv )
{
        printf( "%d \n", CALL ( fib, atoi( argv[1] ) ) );
}
```

Light-Weight Task Management Libraries

- If considering a large number of tasks, handling costs becomes a bottleneck for efficiency (e.g. speed-up).
- Core issues:
  - Data structure in which the tasks are stored.
  - Strategy for load balancing between workers (i.e. threads)
  - Synchronization for moving tasks between workers and corresponding data structures in order to realize load balancing strategy.

Work-Stealing ”Deques”

- Task objects stored in a “deque” (local Push/Pop, thieves Pop) data structure.



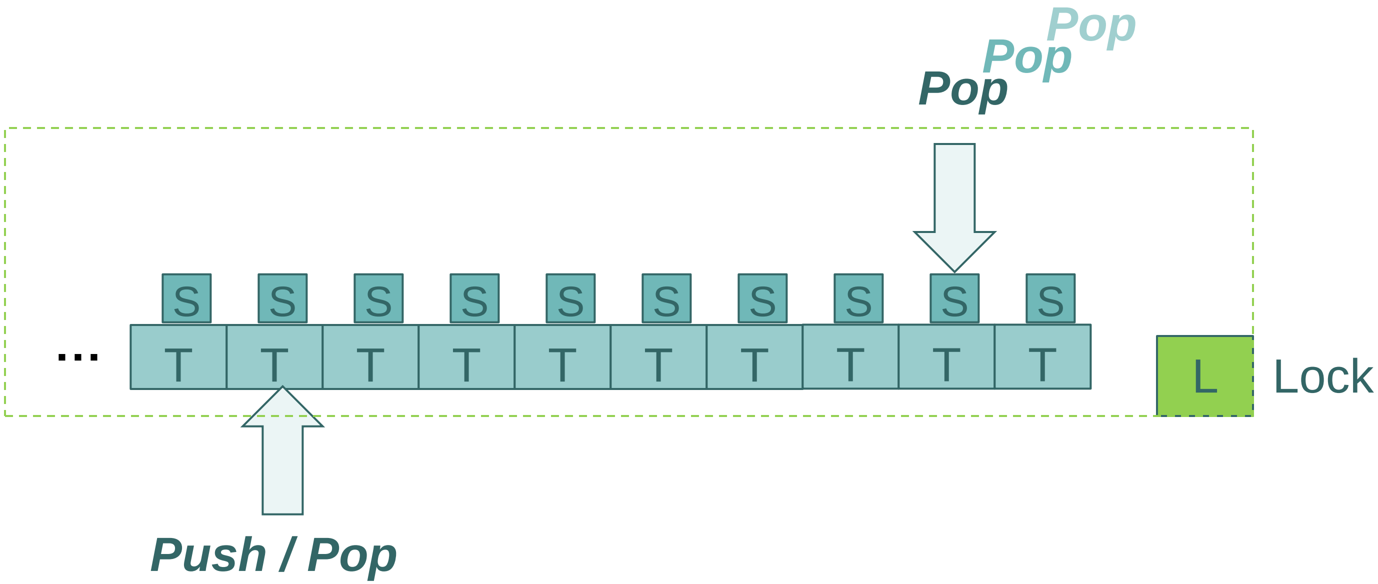
- Allowing side-effects:
  - Arora et al. “*Thread scheduling for multiprogrammed multiprocessors*”. 1998.
  - …
  - Chase and Lev. “*Dynamic circular work-stealing deque*”. 2005.
- Disallowing side-effects:
  - Michael et al. ”*Idempotent work stealing*”. 2009.

The Wool Library (v.0.1.1)

- Karl-Filip Faxén, ”*Wool – A work stealing library*”, MCC 2008.
- Really light-weight.
  - Simplified framework.
- Efficient synchronization
  - Tasks and ”deque” data structure is the same (”collapsed layers”).
  - Un-even synchronization
    - Optimizes for the average case.

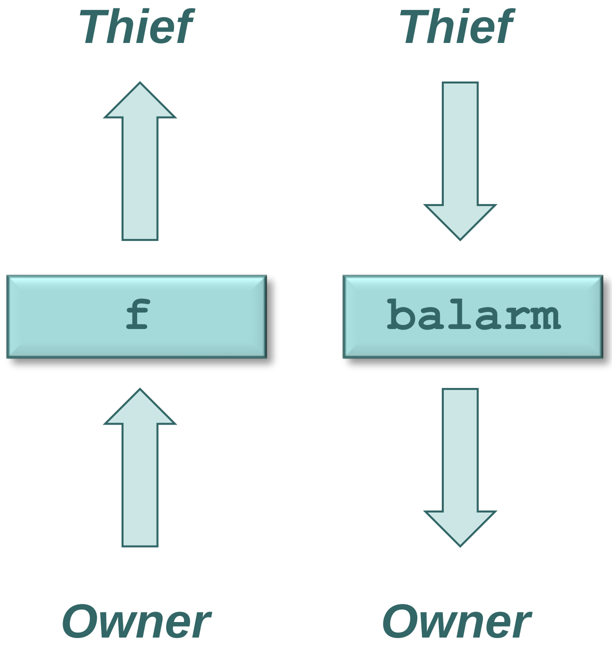
Wool: Architecture

- Each worker has a large array of Tasks.
- Each Task includes stealing/availability status.



Modified Task data structure

- Thieves synchronize through lock.
- Thief and owner synchronize through both `f` and `balarm`.



Wool: Stealers

```
bool steal( Worker *victim )
{
   lock( victim->lck );
   Task *t = victim->bot;
   t->balarm = STOLEN;
   memory_barrier();
   if( t->f == INLINED ) {
      unlock( victim->lck );
      t->balarm = READY;
      return false;
   } else {
      victim->bot++;
      unlock( victim->lck )
      ... // Run the task
      memory_barrier();
      t->balarm = DONE;
      return true;
   }
}
```

Wool: Task owners

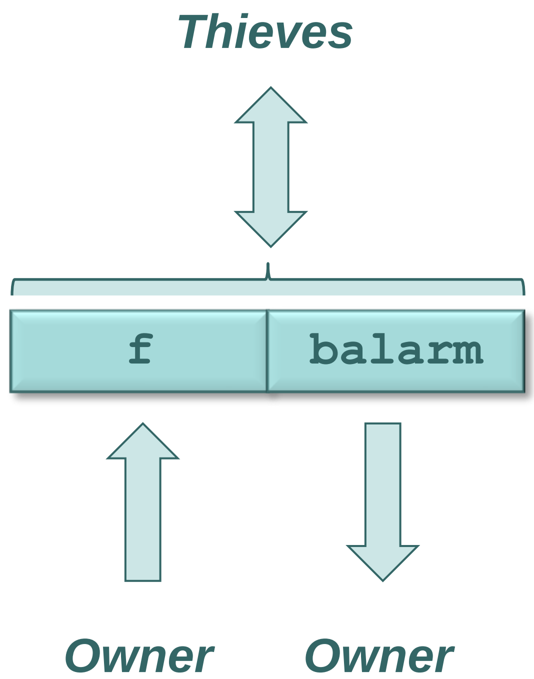
```
void sync( Task *t )
{
   t->f = INLINED;
   memory_barrier();
   if( t->balarm != READY ) {
      // Wait for thief to fully decide
      lock( self->lck );
      if( t->balarm == READY ) {
         unlock( self->lck );
         ... // Run the task
      } else {
         unlock( self->lck );
         ... // Wait for thief to finish
         self->bot--;
      }
   }
}
```

Lock-Free Approach: Atomic Primitives

```
void FAA( int volatile *address, int number ) atomically do {
   *address = *address + number;
}
//
bool CAS( int volatile *address, int oldvalue, int newvalue ) atomically do {
   if( *address == oldvalue ) {
      *address = newvalue;
      return true;
   }
   else return false;
}
//
bool DWCAS( int volatile *address, int oldvalue1, int oldvalue2, int newvalue1, int newvalue2) atomically do {
   if( address[0] == oldvalue1 && address[1] == oldvalue2 ) {
      address[0] = newvalue1;
      address[1] = newvalue2;
      return true;
   }
   else return false;
}
```

Modified Task data structure

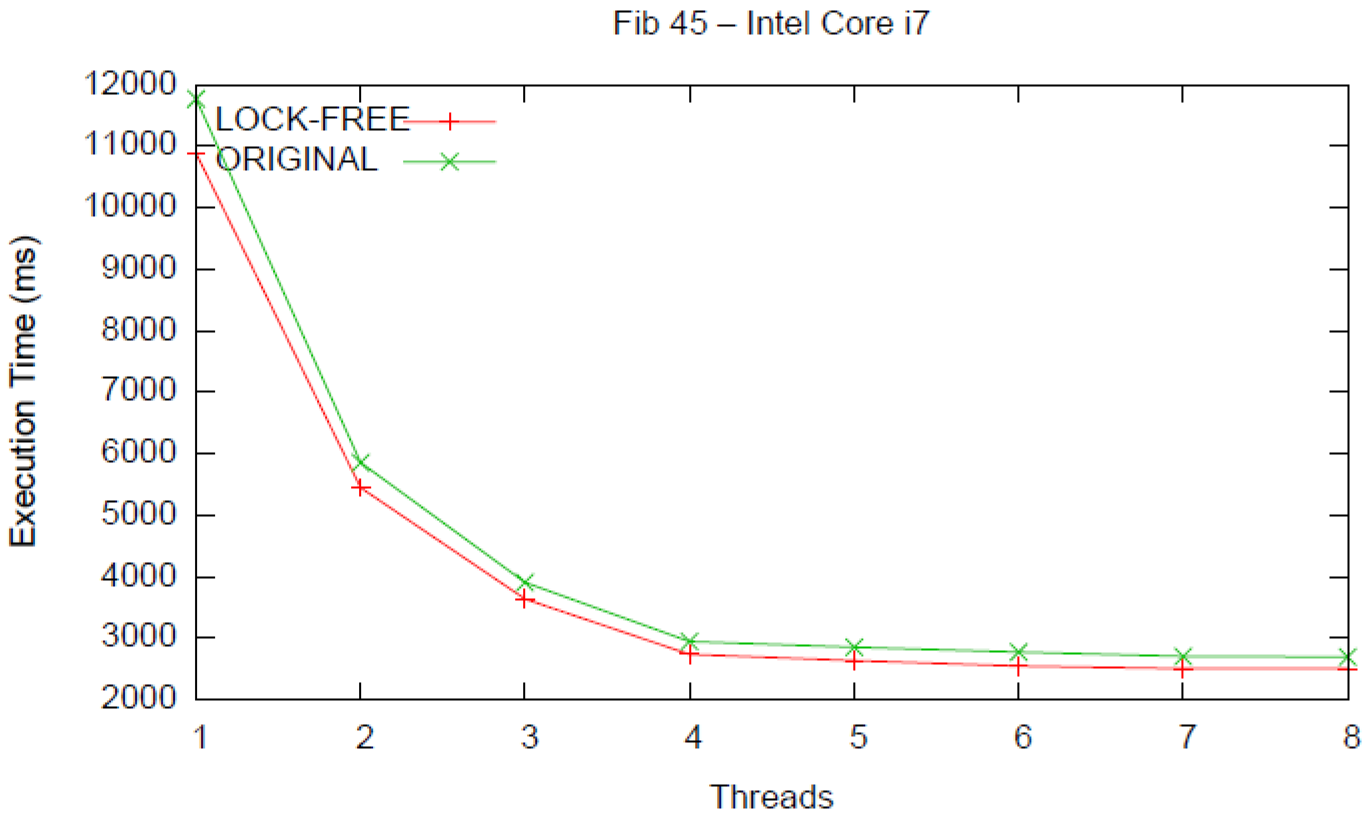
- Place both `f` and `balarm` into same double-word.



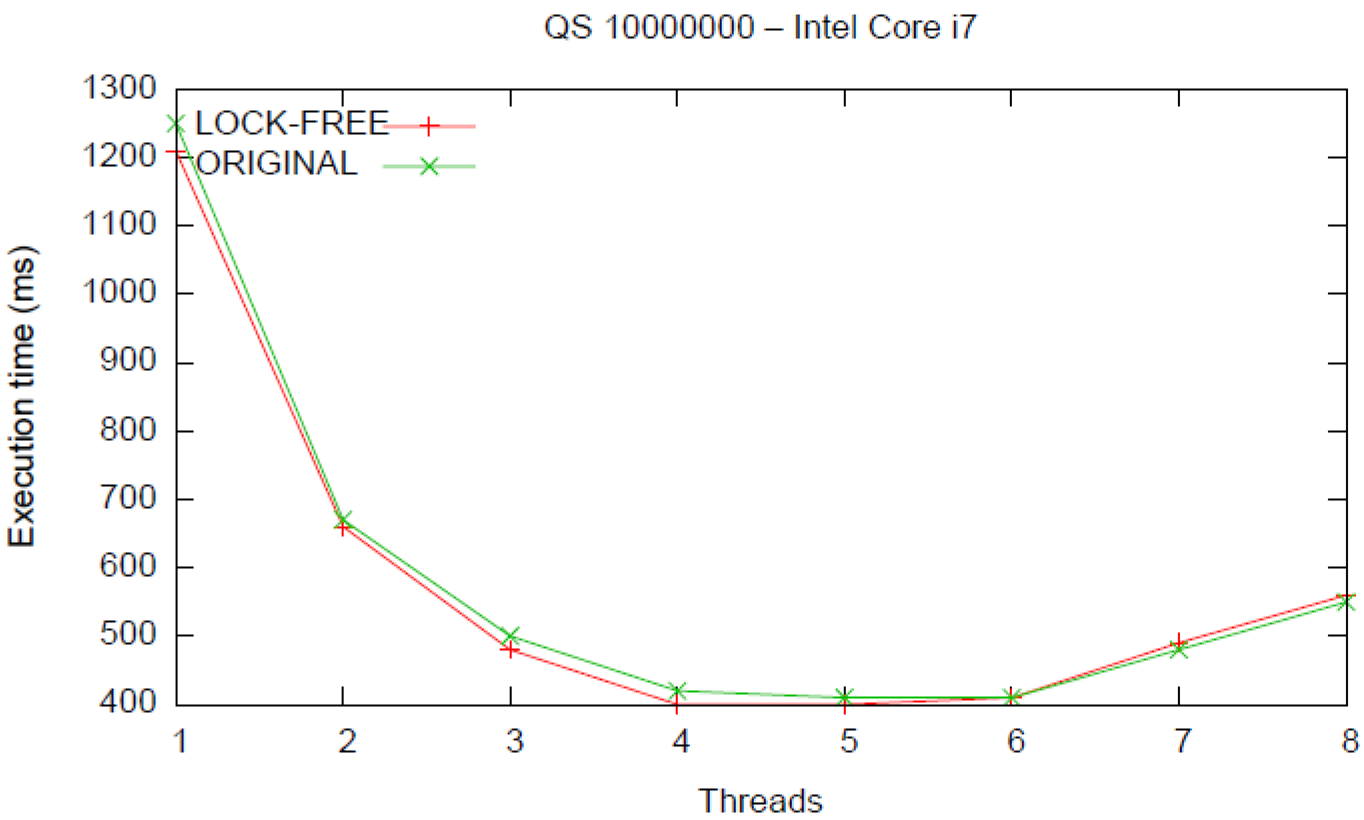
Lock-Free Wool

```
bool steal( Worker *victim )
{
   Task *t = victim->bot;
   f = t->f;
   if( f != INLINED && DWCAS( &t->f, f, READY, f, STOLEN ) ) {
      FAA( &victim->bot, 1 );
      ... // Run the task
      memory_barrier();
      t->balarm = DONE;
      return true;
   }
   else return false;
}

void sync( Task *t )
{
   t->f = INLINED;
   memory_barrier();
   if( t->balarm == READY ) {
      ... // Run the task
   }
   else {
      ... // Wait for thief to finish
      FAA( &self->bot, -1 );
   }
}
```

Experiments (Intel core i7):
Fibonacci, fully expanded spawn-tree

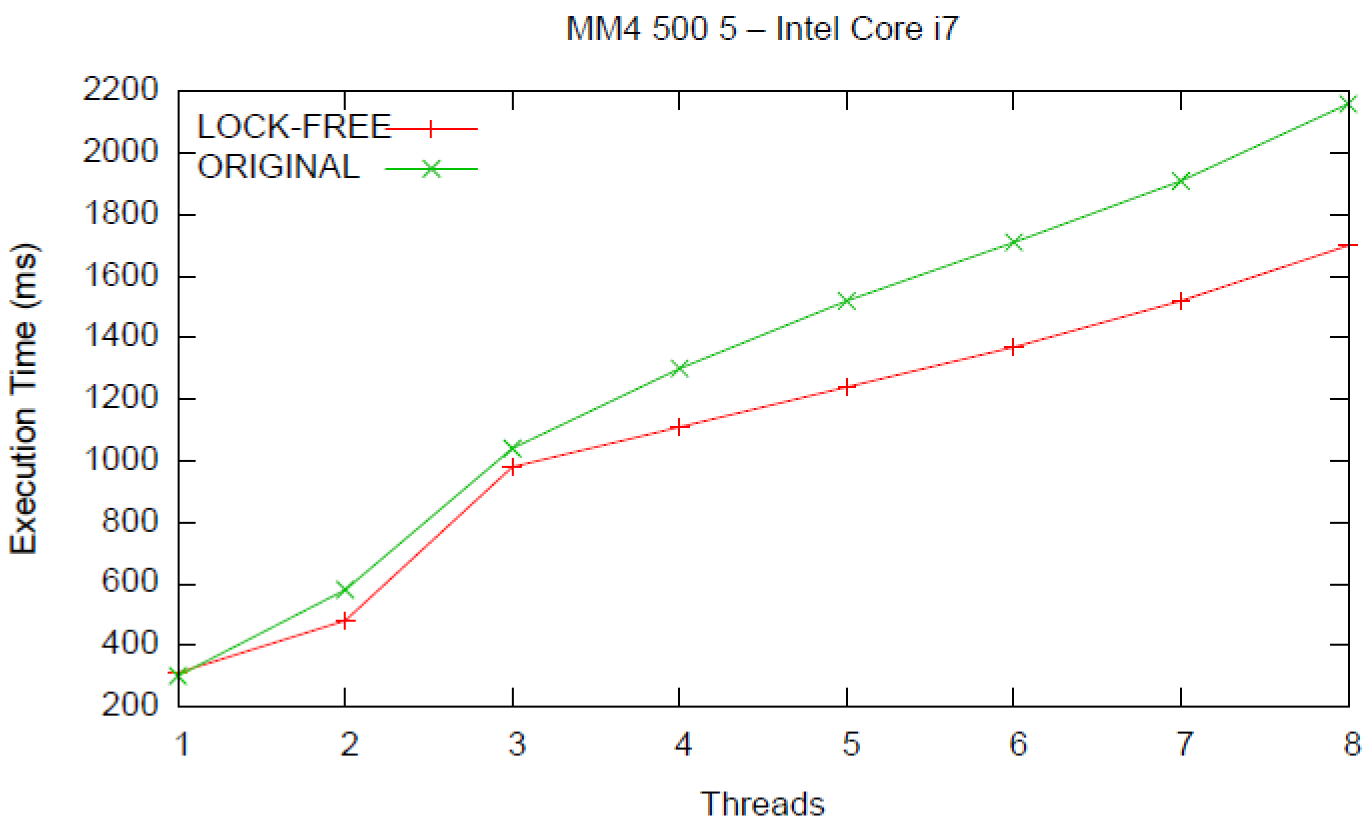


Experiments:
Quicksort using shared memory



Experiments:

Matrix multiplication using ”parallel for”



Conclusions

- Although Wool was highly optimized, adding Lock-Free synchronization could improve (absolute) performance.
- ”Un-even” synchronization is an interesting technique for optimizing the average case.
- Task ”size” is significant for performance.
  - ”parallel for” is especially sensitive for task size, due to relatively high overhead.

Questions?

Thank You for listening!

www.pss-ab.com

www.adm.hb.se/~hsu

www.cse.chalmers.se/~tsigas