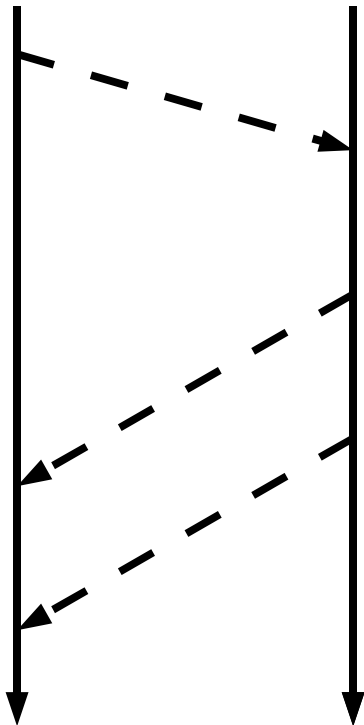


Wool-a work stealing library

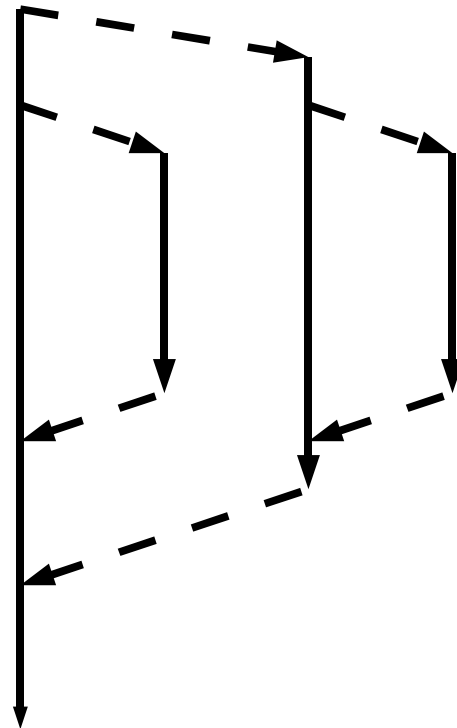
Karl-Filip Faxén
Swedish Institute of Computer Science
Ronneby 28/11 2008

Threads and tasks

Threads: synchronize everywhere



Tasks: synchronize at creation and termination



Example Wool code

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

TASK_1( int, fib, int, n ) // 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] ) ) );
}
```

Example Wool code

```
#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 );          // do fib( n-1 ) in parallel
        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] ) ) );
}
```

Example Wool code

```
#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 );    // ensure that SPAWNed task done
                           // do it otherwise

        return a+b;
    }
}

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

Example Wool code

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

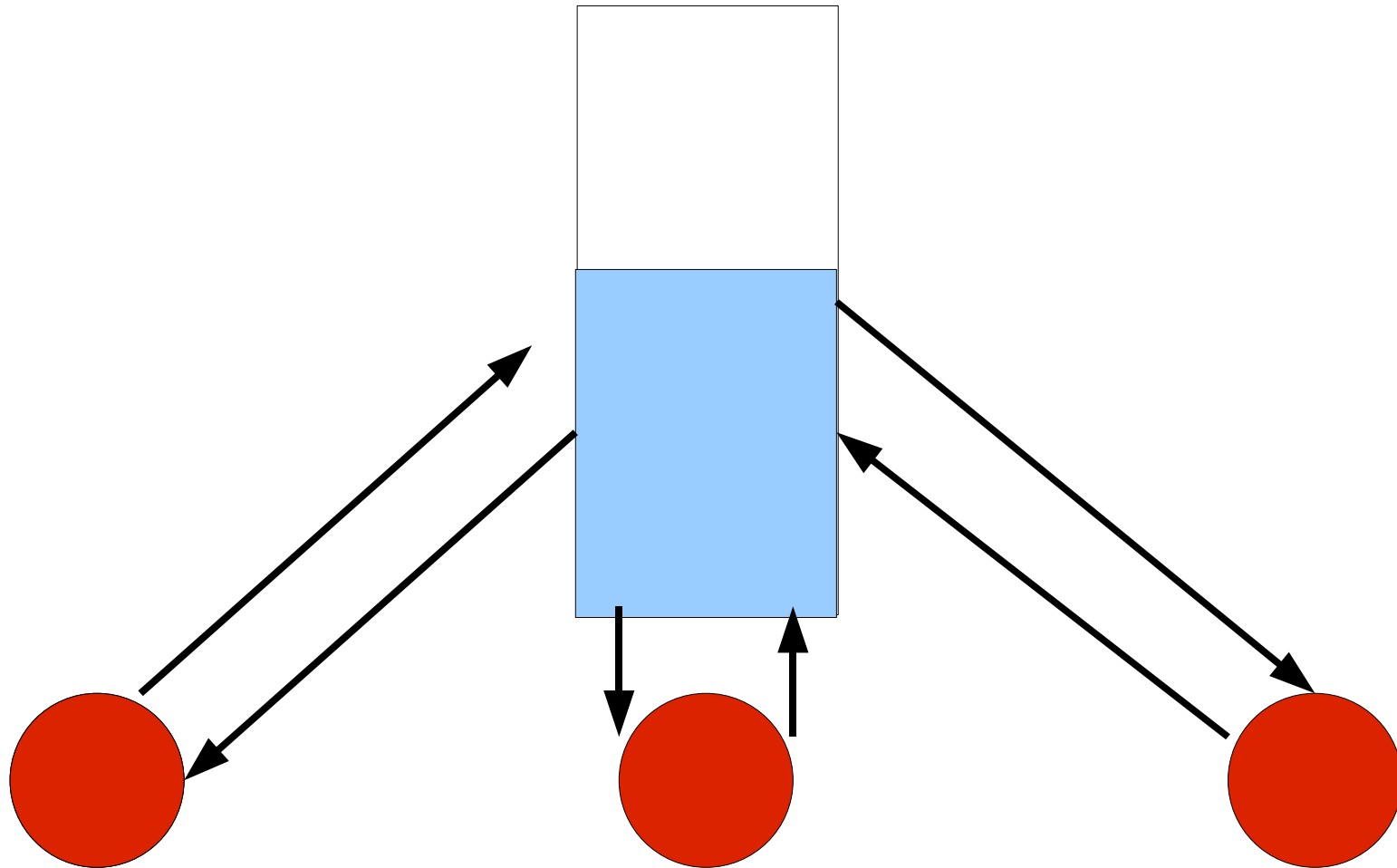
TASK_1( int, fib, int, n )
{
    if( n<2 ) return n;
    else {
        int a,b;

        SPAWN( fib, n-2 );
        a = CALL( fib, n-1 ); // optimization of SPAWN+SYNC
        b = SYNC( fib );

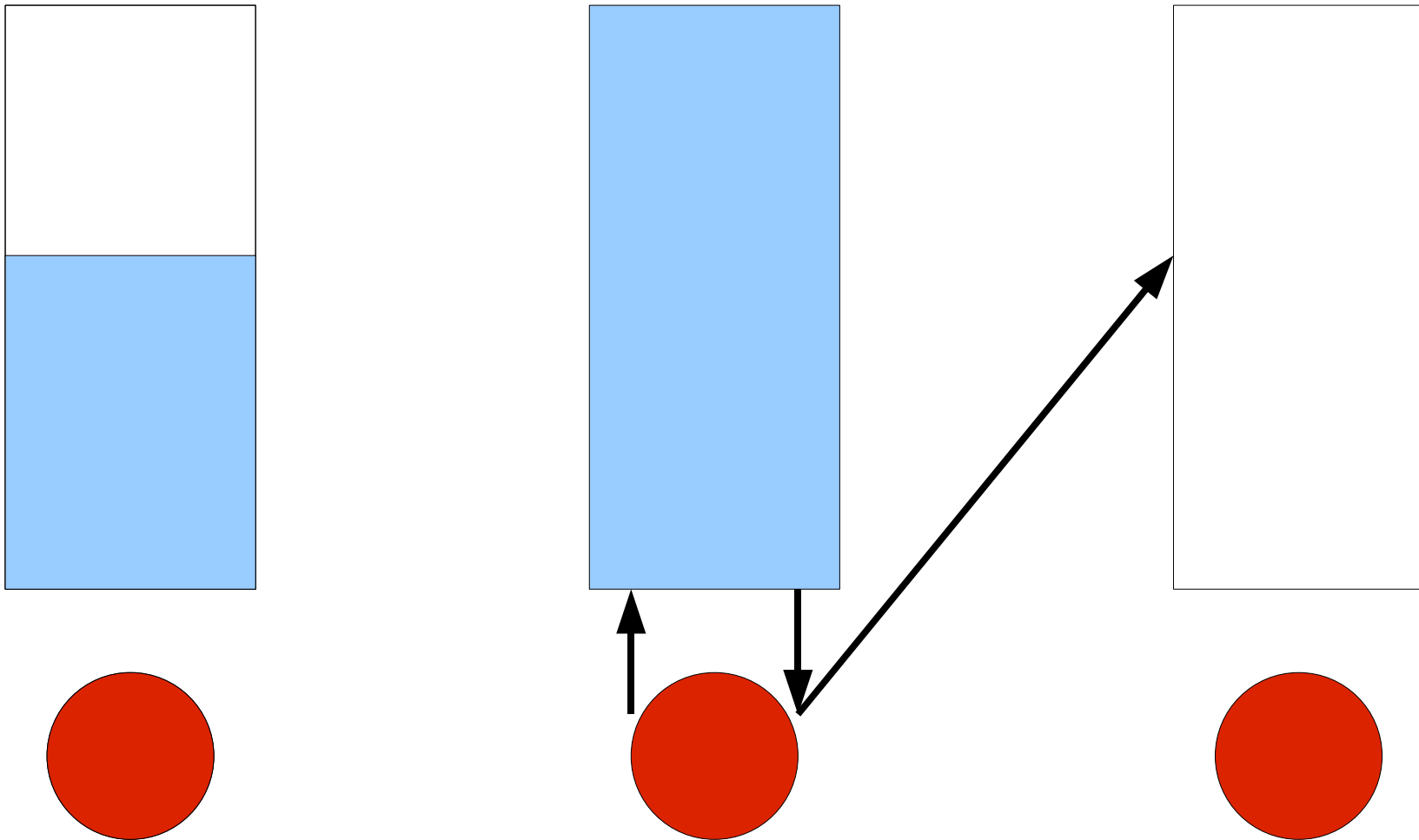
        return a+b;
    }
}

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

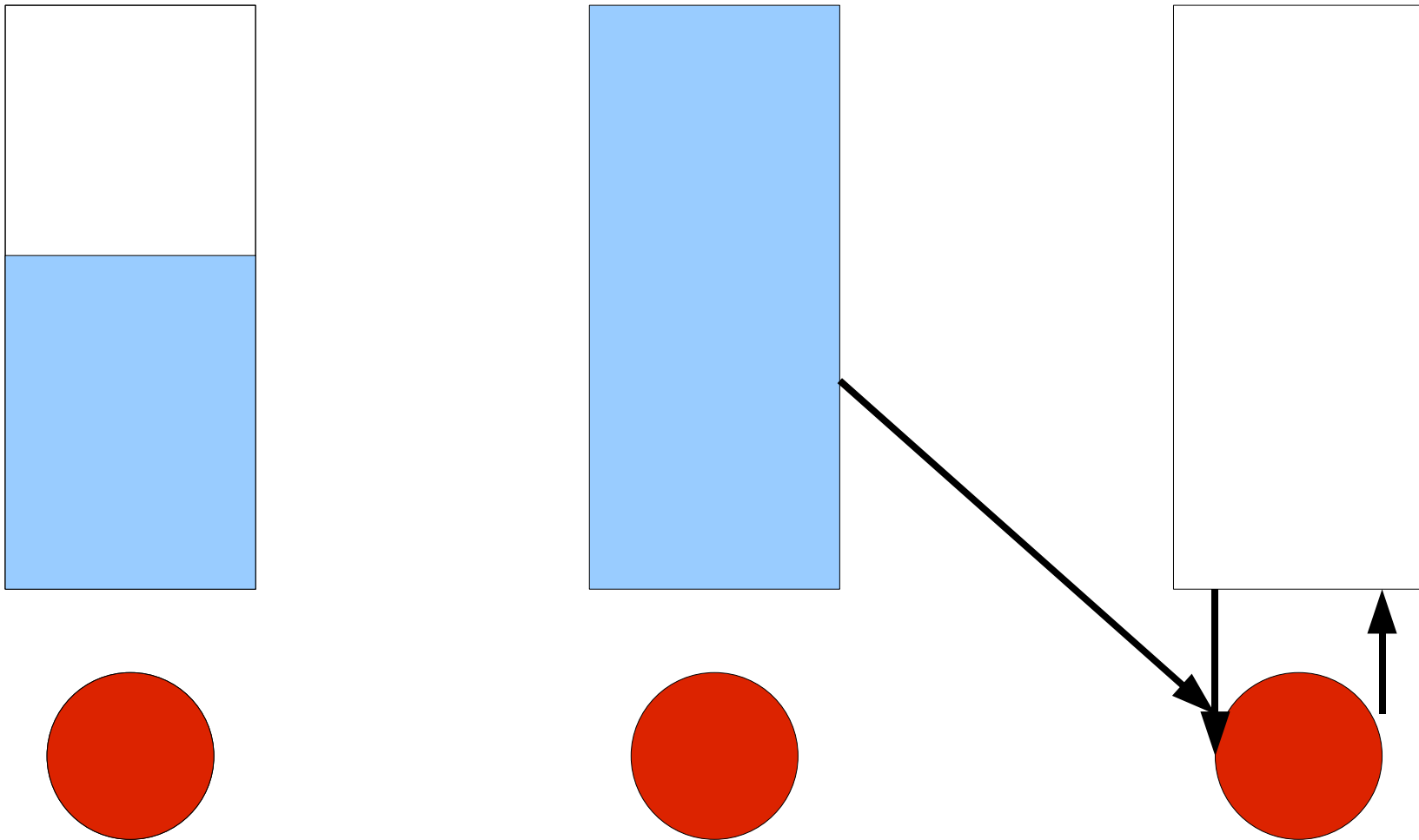
Scheduling: shared run queue



Scheduling: Push tasks



Work stealing



The waiting problem

- A sync may find the task
 - not stolen: run it!
 - stolen and completed: return!
 - stolen and in progress: wait :-(
- What to do while waiting?
 - Nothing
 - Have more workers than cores
 - Steal some other work

Having more workers

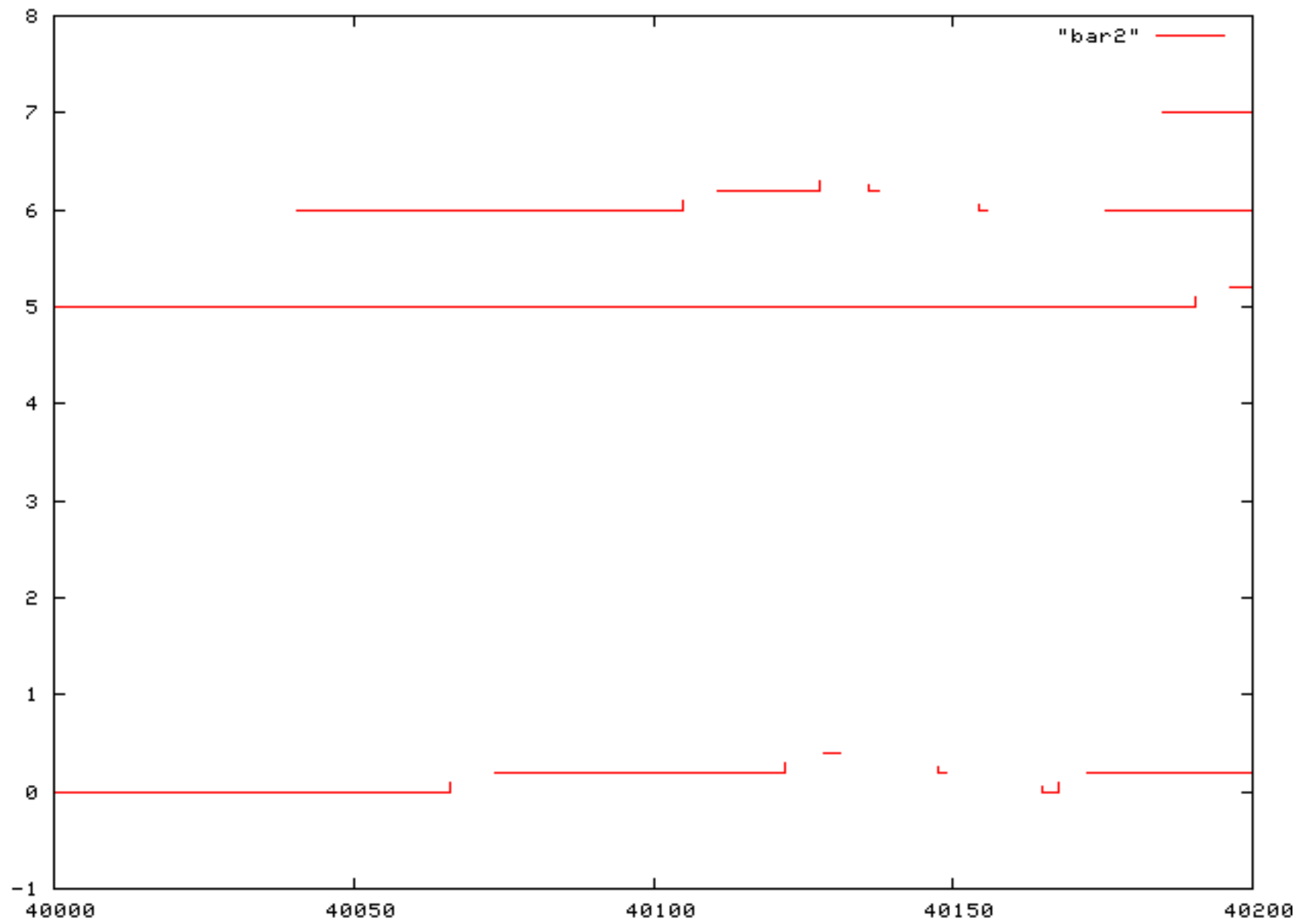
- Worse locality and overhead by involving OS
- Park excess workers
 - Activate parked worker when waiting
 - Park yourself instead of stealing if too many active

Steal to have something to do

- Problem!
 - Worker A finds task stolen by B
 - Then steals from C
 - Then B finishes, so A could continue
 - But A is working on task from C
- So there is work that can not be run by scheduler

Leapfrogging

- Idea: Steal only from the thief ($B=C$)
- A is not occupied when B finishes, so can continue



Test program for waits: stress

```
VOID_TASK_2( tree, int, d, int, n )
{
    if( d>0 ) {
        SPAWN( tree, d-1, n );
        CALL( tree, d-1, n);
        SYNC( tree );
    } else {
        loop( n );
    }
}

TASK_2( int, main, int, argc, char **, argv )
{
    int i, d, n, m;

    n = atoi( argv[1] );
    d = atoi( argv[2] );
    m = atoi( argv[3] );

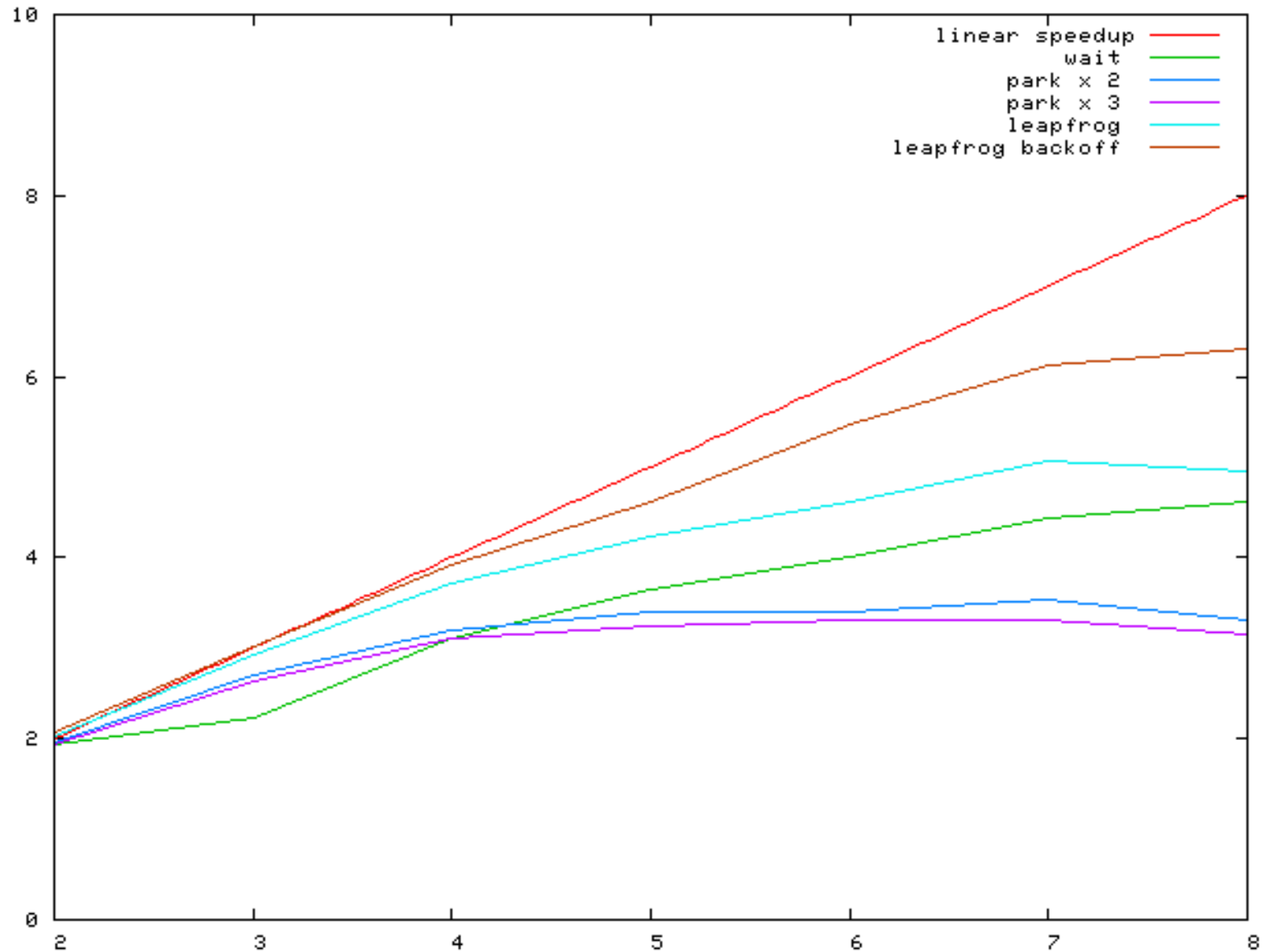
    for( i=0; i<m; i++) {
        CALL( tree, d, n );
    }
    printf( "DONE\n" );

    return 0;
}
```

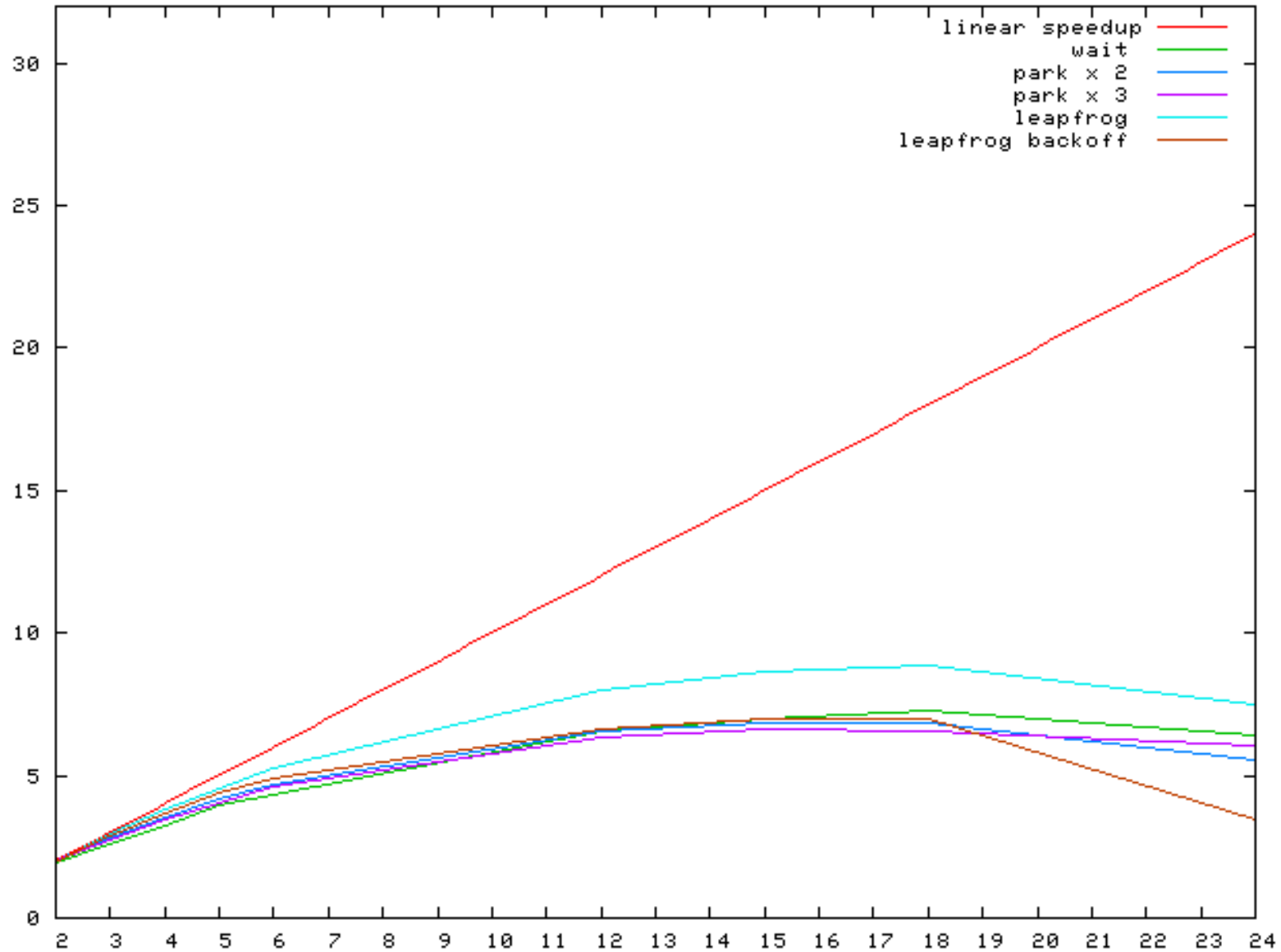
Experiments on wait handling

- Machines used
 - Scheutz: 8 UltraSPARC II, 248 Mhz, Solaris
 - Millennium: 6 core UltraSPARC T1, 1 Ghz, Solaris
 - Small: 2 Core 2 Quad (8 cores), 2.8 Ghz, MacOS
- Alternatives tested
 - Just wait
 - Parking with 2 or 3 times as many workers as cores
 - Leapfrog
 - Leapfrog with ~800 cycles backoff between steals and leaps

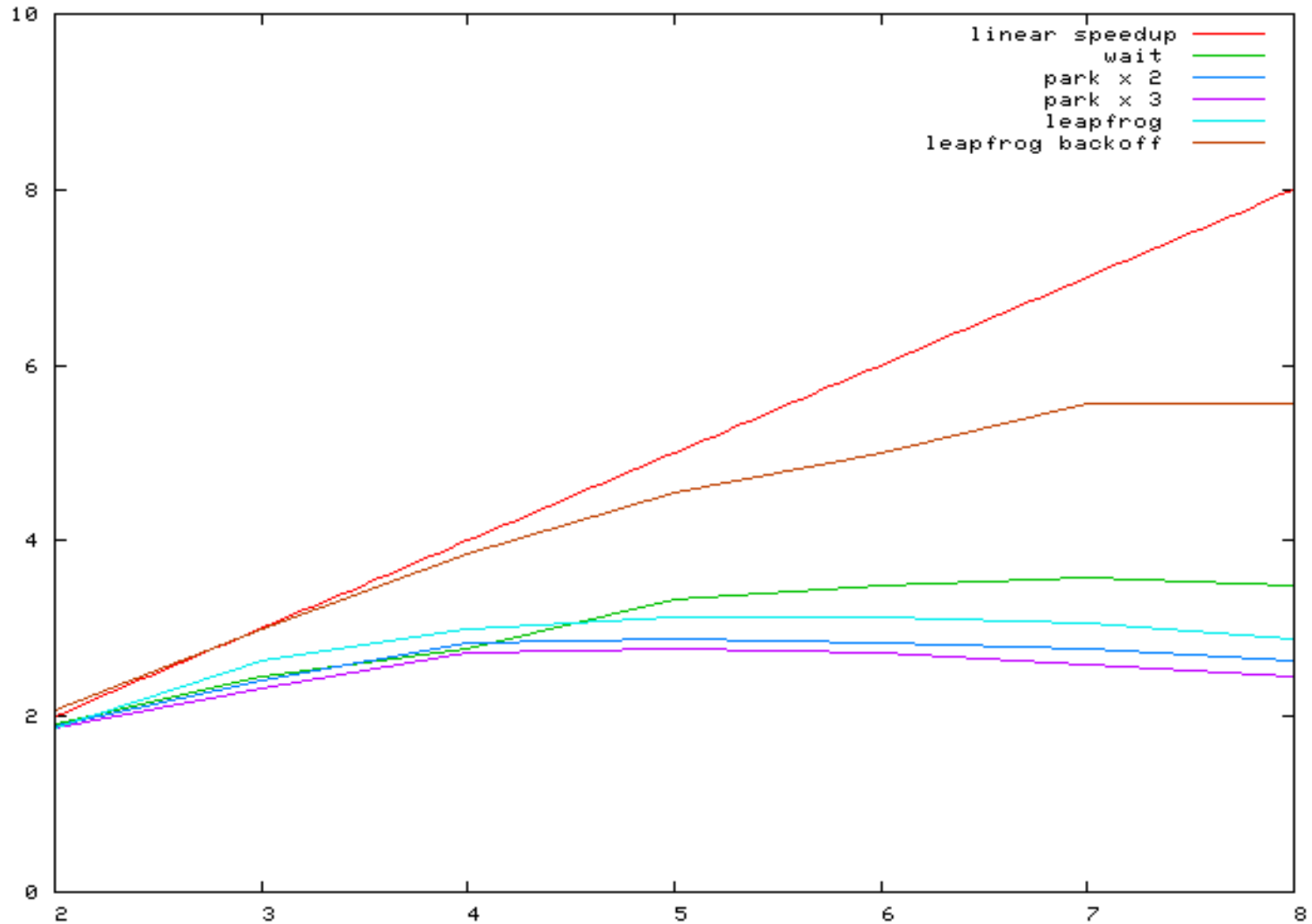
Wait handling: schez



Wait handling: millennium



Wait handling: small



How to wait?

- Leapfrogging rules, parking sucks!
- OS scheduling is slooooooow
- Backoff seems to be a good idea

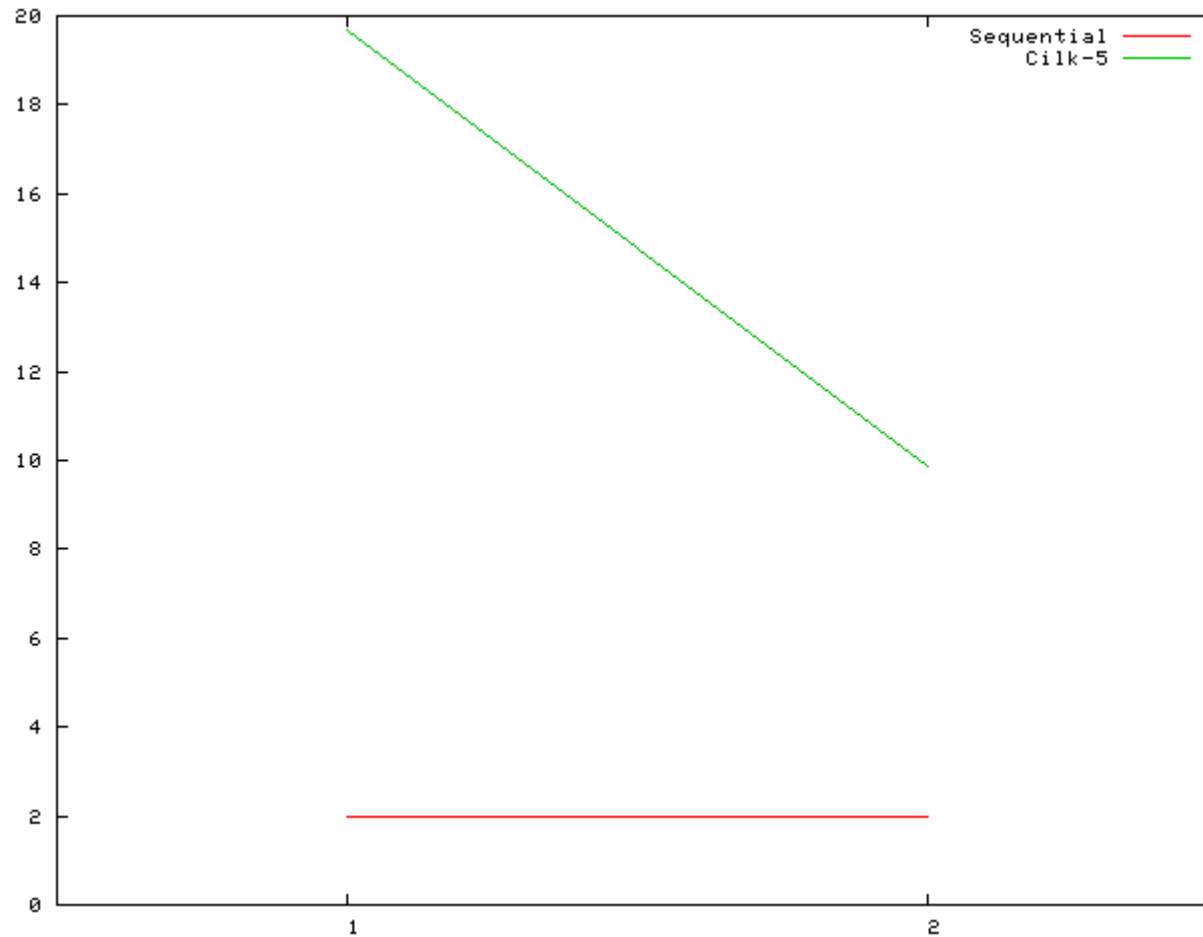
More tuning...

- We all know that locking is bad ...
- ... but each steal locks the victim!
- Now try to avoid locking
 - Check if there is any work, otherwise steal from somebody else
 - Use `pthread_mutex_trylock` instead of `lock`; returns rather than waits when lock is already held

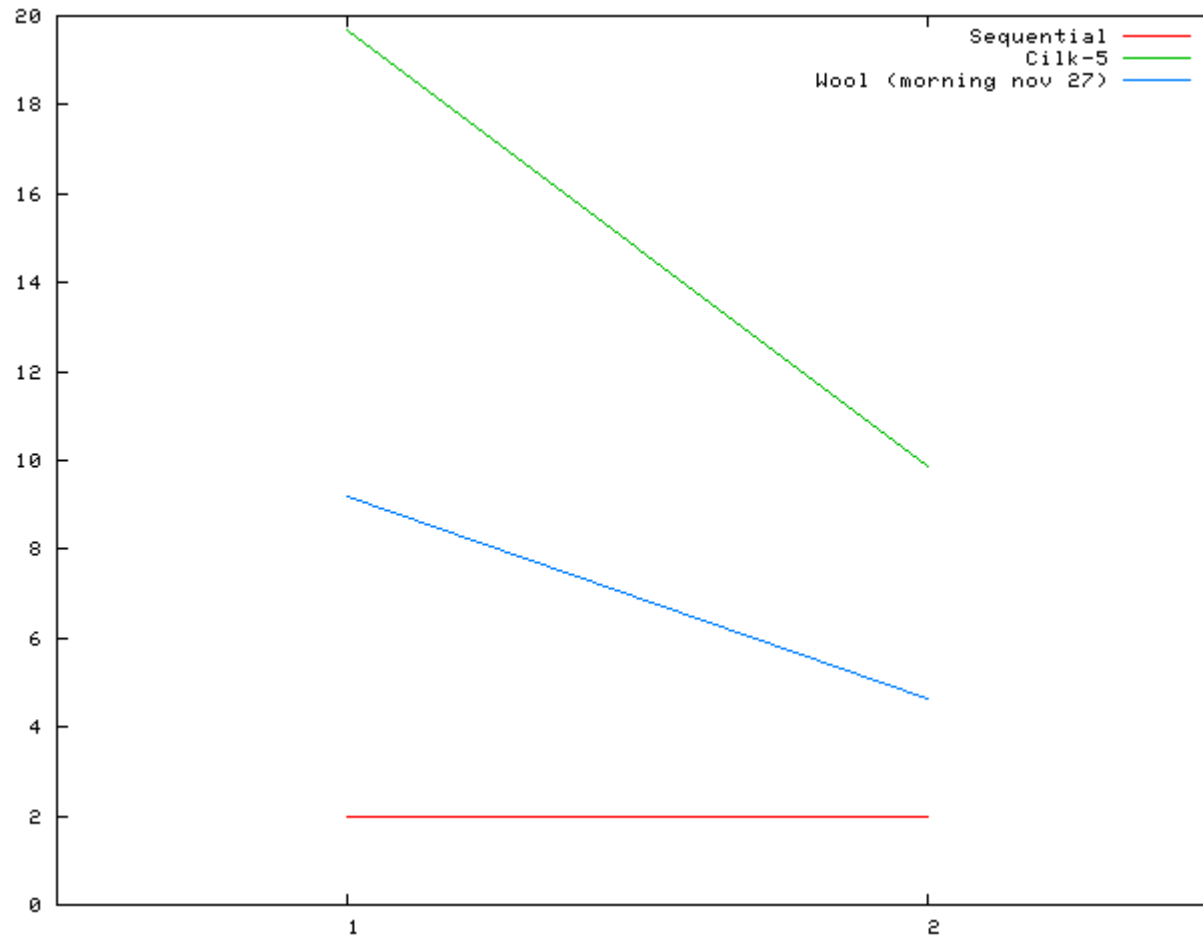
Performance of spawn and sync

- Some programs do not steal often
- Overhead is dominated by spawn and sync on the same worker
 - typically motivates a cutoff
 - sequential algorithm close to the leafs
- What is the performance without cutoff?
- Comparison to MIT Cilk-5 on dual core laptop
 - (I had trouble running Cilk on the other machines)
 - fib(41)

Performance of fib on Core 2 Duo



Performance of fib on Core 2 Duo



Optimize spawn and sync

- We do not want to do memory barriers on every spawn and sync if the task is not stolen anyway
- So we only do it in the base of the task pool
 - Unsynchronized tasks can not be stolen
 - We get fewer stealable tasks
 - But less overhead
- Three levels more than log of number of workers is enough
 - Gives each worker eight tasks

Performance of fib on Core 2 Duo

