# Cilk

supertech.lcs.mit.edu/cilk

algorithmic multithreaded language

programmer specifies parallelism and exploits data locality

runtime system schedules computation to a parallel platform
→ load balancing, paging, communication

extension of C

fork-join execution style

older version Cilk-3: dag consistency [Blumofe et al.'96]

latest version: Cilk-5.3 [Frigo/Leiserson/Randall'98]
typ. overhead for spawning on SMP ca. 4× time for subroutine call



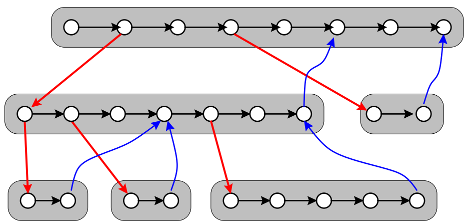
# Example

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
    }
}
```

Omitting the Cilk keywords yields a legal C program with same behavior.



# DAG model for multithreaded computations in Cilk



Each thread is a sequence of unit-time tasks (→ continue edges)
activation frame for local values, parameters etc.

Threads may spawn other threads (→ spawn edges)
similar to subroutine call, but control continues
activation tree hierarchy

A thread dies if its last task is executed.

Results produced are consumed by parent (→ data dependence edges)
in-degree, out-degree per task bounded by a constant



# Scheduling Cilk DAGs

Execution schedule:
maps tasks to processors × time steps (Gantt chart)
follows the DAG precedence constraints
each processor executes at most one task per time step

A thread remains alive until all its children die.

Activation frame remains allocated during a thread's lifetime.

A task is ready for scheduling if all data dependencies are saturated.

DAG depth($t$) = length of longest path to task $t$

DAG depth $T_\infty$ of entire computation = DAG depth of last task
= critical path length = # time steps with ∞ processors

## Greedy scheduling of DAGs

Parallel time $T_p \geq T_\infty$ for any fixed $p$.
Parallel work = $T_1$ = # tasks. $T_p \geq T_1/p$

Brent's theorem: $p$-processor schedule with time $T_p \leq T_1/p + T_\infty$ exists

Greedy scheduling: At each time step issue all (max. $p$) ready tasks.

Greedy-scheduling theorem:

For any multithreaded computation with work $T_1$ and DAG depth $T_\infty$
and for any number $p$ of processors,
any greedy execution schedule achieves $T_p \leq T_1/p + T_\infty$

Speedup linear if $T_p = O(T_1/p)$
for greedy schedule: if average available parallelism $T_1/T_\infty = \Omega(p)$



## Busy-leaves algorithm

Keep a central thread pool $Q$.

Whenever a processor $i$ has no thread to work on,
it removes any ready thread $A$ from $Q$ and begins to execute $A$.
(a) If $A$ spawns a child $B$, return $A$ to $Q$ and execute $B$.
(b) If $A$ waits for data, return $A$ to $Q$ and fetch new work from $Q$.
(c) If $A$ dies:
If $A$'s parent thread $C$ has no live children and no other processor works on $C$
then remove $C$ from $Q$ and begin executing it.
Otherwise, take any ready thread from $Q$ and execute it.

Theorem [Blumofe/Leiserson'94]
The busy-leaves scheduling algorithm computes a $p$-processor schedule with execution time $T_p \leq T_1/p + T_\infty$ and maximum activation depth $S_p \leq S_1 \cdot p$.



## Randomized work stealing algorithm

Modification of busy-leaves algorithm:

Each processor $i$ keeps a ready deque (doubly ended queue) $R_i$
new threads inserted on bottom end
threads can be removed from either bottom or top end.

Whenever processor $i$ runs out of work,
it removes thread $A$ from $R_i$.bottom and begins to execute $A$.
(a) If $A$ enables a stalled thread $B$ (parent), place $B$ in $R_i$.bottom.
(b) If $A$ spawns a child $B$, place $A$ in $R_i$.bottom and start $B$.
(c) If $A$ dies or stalls:
If $R_i$ nonempty: remove $R_i$.bottom and begin executing it.
If $R_i$ empty, steal topmost thread from $R_j$.top
of a randomly chosen processor $j \neq i$, and begin executing it.
If $R_j$ was empty, repeat stealing with another randomly chosen $j$.



## Randomized work-stealing algorithm (cont.)

Theorem [Blumofe/Leiserson'94]

The *expected* running time of the schedule computed by the randomized work stealing algorithm, including scheduling overhead, is $O(T_1/p + T_\infty)$.

For any $\varepsilon > 0$, with probability at least $1 - \varepsilon$,
the execution time is $T_p = O(T_1/p + T_\infty + \log p + \log(1/e))$

Proof (7 pages) see [Blumofe/Leiserson'94]