Cilk

Cilk is an algorithmic multithreaded language that allows the programmer to specify parallelism and exploit data locality. The runtime system schedules computation to a parallel platform with load balancing, paging, and communication.

Cilk supports the following features:
- Extension of the C fork-join execution style
- Older version Cilk-3: dag consistency
- Latest version Cilk-5.3: load balancing, scheduling

Example

```c
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);
    }
}
```

Cilk provides the following benefits:
- In-dagge, out-dagge per task bounded by a constant
- Results produced are consumed by parent (data dependence edges)
- A thread dies if its last task is executed
- Each thread is a sequence of unit-time tasks (continue edges)

Cilk models parallelism as a DAG model for multithreaded computations.

DAG model for multithreaded computations in Cilk

The DAG model consists of:
- Each thread is a sequence of unit-time tasks (continue edges)
- A thread may spawn other threads (spawn edges)
- Activation frame for local values, parameters etc.
- Each thread is a sequence of unit-time tasks (continue edges)
- Results produced are consumed by parent (data dependence edges)

Execution Schedule:

Scheduling Cilk DAGs

The Cilk keywords yield a legal C program with the same behavior as:

```c
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);
    }
}
```

Each thread is a sequence of unit-time tasks (continue edges)

DAG model for multithreaded computations in Cilk
Greedy scheduling of DAGs

Parallel time $T_p$ for any fixed $p$.

Parallel work $= T_1 = \# $ tasks.

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

Speedup linear if $T_p / T_1 = O / (T_1 / = p /)$.

The 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 $A$.
  1. If $A$ spawns a child $B$, return $A$ to $Q$ and execute $B$.
  2. If $A$ waits for data, return $A$ to $Q$ and fetch new work from $Q$.
  3. If $A$ dies:
     a. If $A$’s parent thread $C$ has no live children and no other processor works on $C$’s parent thread $C$ has no live child and no other processor removes $C$ from $Q$ and begins executing $C$.
     b. If $A$ has a child $B$, return $A$ to $Q$ and execute $B$.
     c. If $A$ removes any ready thread $A$ from $Q$ and begins executing $A$.
     d. Whenever a processor $\emptyset$ has no thread to work on, keep a central thread pool $Q$.

Randomized work stealing algorithm:
- Each processor $i$ keeps a ready deque (doubly ended queue) $R_i$.
- Whenever processor $i$ runs out of work, it removes thread $A$ from $R_i$ and begins to execute $A$.
  1. If $A$ enables a stalled thread $B$ (parent), place $B$ in $R_i$.
  2. If $A$ spawns a child $B$, place $A$ in $R_i$ and start $B$.
  3. If $A$ removes thread $A$’s ready thread $A$ from $R_i$ and begins to execute $A$.
- Whenever processor $i$ runs out of work, new threads can be removed from either bottom or top end.
- Each processor $i$ keeps a ready deque (doubly ended queue) $Q_i$.

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

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