Cilk: An Efficient Multithreaded Runtime System

What is Cilk? It is a C-based runtime system that is used for multithreaded parallel programming. Cilk applies a work-stealing scheduler which allows the programmer to focus on reducing the work and critical path computation insulated from load balancing and other runtime scheduling issues. The applications of Cilk includes complex tasks such as protein folding, graphic rendering and chess engines.

Cilk provides a programming model in which the execution time is dependent of *work* and *critical path*. Work and critical path are observable quantities which guarantees performance as a function of these. Work is defined as the execution time of the program if we only have 1 processor (T_1). Critical path is defined as the execution time with an infinite number of processors (T_∞). Thus, the total execution time with P processors will never be less than T_1/P or less than T_∞.

Multithreading in Cilk is done by creating, synchronizing and scheduling of threads. To visualize the multithreading in Cilk, one can view the program as a spawn tree - which is a DAG (directed acyclic graph). In the spawn tree, a program (which consists of a collection of procedures), is broken into a sequence of threads, which all form edges in the DAG. If one thread spawns a child procedure, we have a downward edge. If one thread is returning a value to another thread, we have a data dependency among two threads. In Cilk, this is sorted out by having the thread spawning a child also spawning a successor thread that waits for the returned values. This is necessary because each thread is non-blocking, which means that it can run to completion without waiting.

Since Cilk is an extension of C, a thread in Cilk is defined similarly to a thread in C, where the Cilk preprocessor translates T into a void C function of one argument. This argument points to a *closure* data structure. A closure holds a pointer to the C function, a slot for each arguments, and a *join counter* (indicates the number of missing arguments needed before the thread can run). A closure is ready to be used as the argument when it has obtained all of its arguments. A *continuation* [cont] data type is global reference to an empty argument slot of a closure. A thread might *spawn* [spawn] a child thread by creating a closure, and it might also spawn successor threads [spawn_next] (semantically identical, but informs scheduler that the new closure is a successor). For message passing (called *explicit continuation passing*) between threads one uses [send_argument] (for sending values from one closure to another).

Cilk's scheduler applies a work-stealing algorithm for processors (thieves) who runs out of work to steal work from other processors (victims). The thieves always steals the shallowest ready thread of a randomly selected victim's spawn tree. Each processor has a local *ready queue* for holding ready closures, and each closure has a *level* which corresponds to the number of spawns on the path from the root of the spawn tree. The ready queue is a linked list with the i'th element containing all ready closures at the i'th level. The work-stealing scheduling starts out by initializing all ready queues to empty placing the root thread into the level-0 list of processor_0's queue. Then it starts a scheduling loop on each processor, in which each processor first checks if the ready queue is empty and if it is, the it begins with work stealing. The reason for stealing threads from the shallowest level of the ready queue is that we would like to steal as much work as possible, and shallow closures would most likely execute for longer than deep closures. Stealing bigger closures allows for less communication cost. Stealing at the shallowest level makes processors steal to make progress along the program's critical path.

When applying Cilk to a few well-known parallelizable problems, the article shows that Cilk programs achieve near perfect linear speedup, but when the average parallelism is small, the speedup is less.