Cilk: An Efficient Multithreaded Runtime System

This article is about Cilk, a multithreaded runtime system based on C for parallel programming. A multithreaded system enables the developer to create, synchronize and schedule different threads as needed. Cilk provides a scheduler that can predict the runtime of a system by using an algorithm based on the “critical path” and “work” measures. This can be visualized by a directed acyclic graph. A Cilk program consists of Cilk procedures. A Cilk procedure consists of threads and their respective successors. Threads can spawn children, and if a certain thread requires a return value, it must spawn a successor to retrieve the return value from the children. This is because threads cannot be blocked. Each thread receives a data dependency value, and threads need to wait for these values in order to be able to continue to run.

A Cilk program measures the execution time based on two parameters; *work* T1 and *critical path* P. The work is the sum of the execution time of all threads on a single-processor execution. The critical path is the total time required on an infinite-processor execution. The execution time can therefore never be less than T1/P or less than T∞. “Work stealing” is an algorithm used to achieve execution time close to these measurement sums.

To simplify the development of Cilk programs, the creators have created a C language extension. In Cilk, the definition of a function is similar to C, and the Cilk preprocessor translates Cilk functions into C functions. A thread in Cilk is defined as such: *thread T (arg-decl..) { stmts }*, where the argument is a pointer to a closure dataset that holds arguments for T. The closure consists of a pointer to a C function for *T* for each argument, and join counter to make sure *T* has all its arguments before it can run. The Cilk scheduler invokes the thread as a procedure by using the closure itself to run a ready closure. The arguments are copied into local variables, and can be allocated from a runtime heap when created, and returned to the heap when the thread terminates. Cilk uses a data type called *continuation*. This is a global reference to an empty argument slot of a closure which contains pointers to an offset and closure. This is used for communication among threads. This is just some of many functions and datatypes Cilk provides. Other examples include *Spawn T (args)*, which creates a child closure, *Spawn_next T (args...)* which creates a successor thread, and *Send_argument (k, value)* which is used to send values between closures.

The article provides an example on how to implement the Fibonacci sequence by using these functions. It uses *send_argument* to return n specified by *continuation* k if the boundary case is reached. If it is not reached, it spawns a successor thread to hold the sum with two subchildren for calculating each with a continuation saying which argument in the sum thread they should return their value to. These values are in turn summed up by the sum thread, and returned to the designated slot determined by *k*. The Cilk system is further simplified by dividing the procedures into non-blocking threads. It also saves on context switching, by allowing spawning of several children from a non-blocking thread. Cilk also provides further control over runtime performance; if a thread’s last action is to spawn a ready thread, it can use *call* instead of *spawn*, so the scheduler does not need to be invoked.

The Cilk work stealing scheduler uses a technique that involves a processor stealing a thread from another processor, if it has nothing to do. The processor being stolen from is randomly picked. Each processor maintains a ready queue to hold closure with an associated level, which corresponds to the number of spawns. This is represented as an array, with the *L*th element holding a linked list with ready closures at level *L*. At the beginning of the execution of the program, the ready queues are initialized empty. The root thread in the level 0-list gets placed in the processors 0 queue. When a scheduled loop occurs, the processor checks its ready queue first; if its empty it will perform “work stealing”. The communication between the worker and the stealer is handled by a request-reply communication protocol.

For the purpose of benchmarking Cilk, several applications have been used. Examples of these are fib, queens, pfold, ray, knary(k,n,r) and socrates. The results implied that Cilk induces little overhead when dealing with long threads, considering the relationship between efficiency and thread length. When the number of processors increase, more work is done. Cilk programs achieve almost perfect linear speedup; when looking at the average parallelism, $T1/T\infty$ is big in contrast to the amount of processors. However, when the average parallelism is lower, the speedup is not as significant.

The Cilk scheduler uses a near-optimal approach to schedule applications. To show that the running time of an application can be modeled as $Tp = T1/P + c\infty T\infty$ where $c\infty = 1.5$ on P processors, they use the *knary* synthetic benchmark. The *knary* benchmark is used for evaluation of the Cilk scheduler.

For the purpose of proving that the Cilk scheduler is efficient, the authors have used an algorithmic analysis technique. This is in the case of “fully strict” Cilk programs. For space, time and communication they prove three bounds. For space, they use the “busy-leaves” property. For execution time and communication, they use different proofs for generalization of the result.

This article clearly explain why programmers should think about work and critical paths in their designs. It should be possible to characterize performance without knowing anything about the machine configurations. Cilk programs can guarantee performance based on observing work and critical path.