6.895 Theory of Parallel Systems Lecture 8

Cilk Scheduler

Lecturer: Michael Bender *Scribe: Kevin Matulef and Barbara Mack*

Lecture Summary

1. *"Ideal" Scheduling*
   This section outlines the properties of an "'ideal"' global scheduler: it is a greedy scheduler and it employs a Busy Leaves scheduling model.
2. *Cilk Computation*
   We review the model of Cilk Computation and the terminology involved.
3. *Main Data Structure*
   This section describes the ready deque.
4. *Work Stealing*
   This section describes how work stealing is done with the Cilk scheduler.
5. *Cilk Scheduling Algorithms*
   This section details the Cilk scheduling algorithm.
6. *Structural Lemma*
   Our first lemma about the scheduler.

1 "Ideal" Scheduling

Goal: Develop a distributed scheduler that approximates the performance of an "ideal" global scheduler. Note that a global scheduler knows the entire system state (the state of all processors), while a distributed scheduler knows only the local state.

What sort of properties would we want in an "ideal" global scheduler?

1. **The ideal global scheduler is a *greedy* scheduler**.

   Recalling the work of Graham and Brent, we know that with a greedy sheduler,

   $$T_p \leq \frac{T_1}{P} + T\infty \tag{1}$$

   That is to say, the time it takes for P processors to run the computation (T_P) is less than or equal to the total work (T_1) divided by the number of processors, plus the critical path length ($T\infty$).

   In most practical instances, $T\infty$ is much less than $\frac{T_1}{P}$. Thus, if we have a choice, we would like for any overhead (e.g. the cost for steals) to be a multiple of T_∞, rather than $\frac{T_1}{P}$.

2. **The ideal global scheduler is a *Busy Leaves* scheduler**, with, at most, P active leaves at any time.

   In Professor Leiserson's lecture on the Busy Leaves Scheduler, we saw that it reduces the space used by a parallel algorithm compared to the space used by a sequential algorithm:

   $$S_p \leq P * S_1 \tag{2}$$

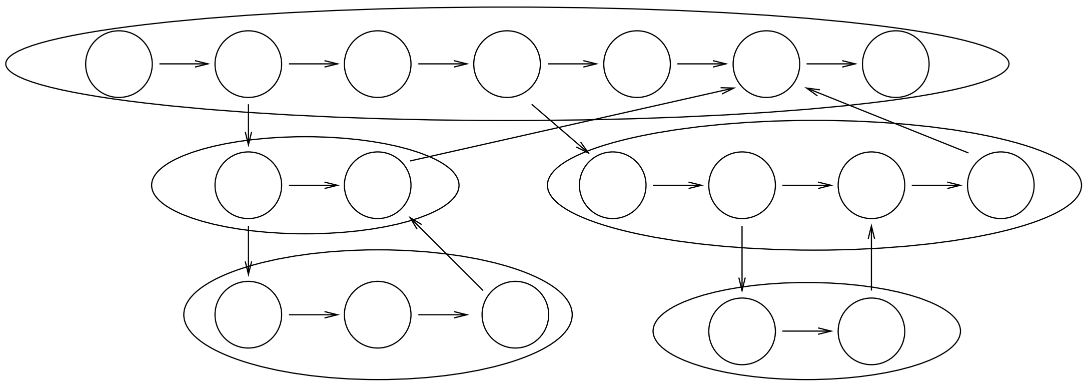

Figure 1: Example of the Cilk DAG.

where S_p is the space on p processors.

This principle applies to stack allocation, where all memory allocation occurs at the beginning of procedures.

3. **The ideal global scheduler contains a small number of migrations/steals.**

   In Cilk, when a processor doesn't have any work to do, it steals work from another processor. This produces more efficient work flow, but there is a cost associated with the stealing effort. Fewer steals will produce a better overall result. If possible, we would like to achieve the following bound on the number of steals:

$$\#\text{steals} \le O(PT_\infty) \tag{3}$$

   If we can develop a greedy scheduler that satisfies that bound, then the cost of steals, averaged over the number of processors, is $O(T_\infty)$. Then we would at least have the following bound: $T_p \le \frac{T_1}{P} + O(T_\infty)$.

2 Cilk Computation

The Cilk DAG is made up of subroutines, which themselves are composed of threads. Each thread has either 0,1, or 2 immediate successors, taken in series parallel fashion (S-P).

In our model, illustrated in Figure 1, each oval is a subroutine and each node is a thread. Without loss of generality, we will assume that all threads take the same amount of time to execute (if one thread took longer, we could just write it as multiple threads connected in series).

3 Main Data Structure

Each processor maintains a data structure called a "ready deque," which is a double ended queue as indicated in Figure 2.

The ready deque stores threads of subroutines that are ready to execute. Threads are inserted at the bottom of the deque, but they may be deleted from either end. Thus, the deque resembles a queue at the top, and a stack at the bottom.

4 Work Stealing

When the processor begins work stealing, it operates as follows:

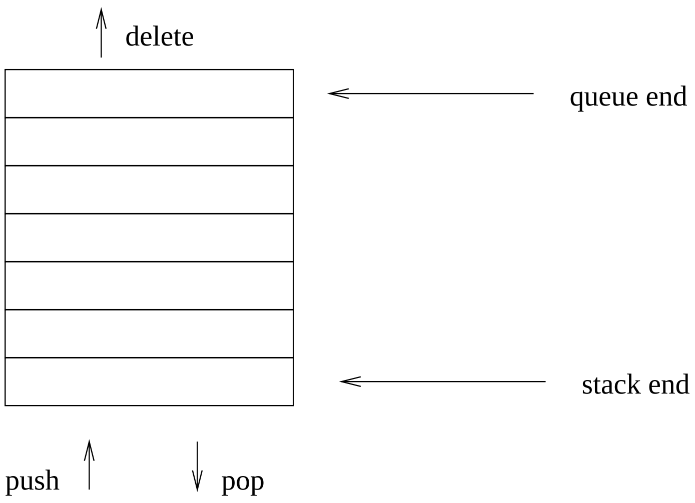

Figure 2: The Ready Deque

- The processor chooses a victim uniformly at random.
- If the victim's deque is empty, the processor tries again.
- Otherwise, the processor steals the top (oldest) thread of the victim and begins to work on it.

5 Scheduling Algorithm

In the Cilk scheduling algorithm, a processor works on subroutine α until:

1. α *spawns* subroutine β
   - In this case, the processor pushes α to the bottom of the ready deque, and starts work on subroutine β.
2. α *returns*
   - If the deque is nonempty, the processor pops the bottom subroutine and begins working on it.
   - If the deque is empty, first the processor tries to execute α's parent.
   - If α's parent is busy, the processor steals work at random.
3. α *synchs* with another subroutine
   - If there exists outstanding children and the computation cannot proceed, then the processor worksteals. Note that the deque must be empty in this case.

It is worth pointing out what this algorithm does on a single processor system. In a one processor system, the program would behave like a normal C program would: if it spawns, then it suspends the parent and goes to work on the child. This is just the usual depth-first execution of the series-parallel DAG.

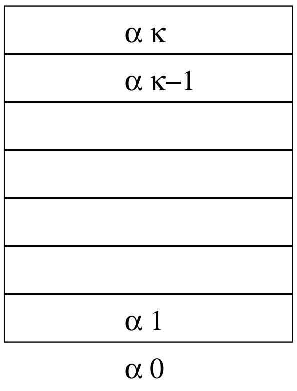

Figure 3: The Ready Deque for Processor p at time t

6 Structural Lemma

Consider any processor p at any time t. Let α_0 be the procedure that processor p is working on. Let $\alpha_1, \ldots, \alpha_k$ be the procedures in p's deque from bottom to top, as shown in Figure 3

.

Lemma 1 *At time t_i:*

1. - α_i is a child of α_{i+1}, for $i = 0, \ldots, k-1$
   - α_i is an *only* child of α_{i+1}, for $i = 0, \ldots, k-2$
   - α_{k-1} may be one of many children of α_k

   Figure 4 illustrates these properties by showing the spawn tree at time t.

2. The number of actives leaves is at most P, the number of processors. Recall that a leaf is defined as a subroutine with no current outstanding children.

.

Proof To prove this lemma, we do an induction on actions. That is, we examine what happens during each type of action (spawn, return, steal, sync) and show that all properties outlined in the lemma are preserved.

- First we examine what happens in the case of a spawn. Let us label the child of α_0 as α_{child}. Then the spawn tree is modified as shown in Figure 5, and α_{child} is pushed onto the deque.

  Note that the properties stated in the lemma are preserved. If α_0 was not at the top of the deque, then it could not have been stolen. It could not have spawned other children in the past, so α_{child} must be it's only child. If α_0 *was* at the top of the deque, then α_{child} might not be α_o's only child, but that is consistent with the lemma.

  Likewise, let us look at the number of leaves. α_0 was a leaf, but now it is not, and instead α_{child} is a leaf. Thus the number of leaves is preserved.

- Now we examine what happens in the case of a steal, as illustrated by Figure 6

  Assuming the inductive hypothesis that α_k is the only process with multipe children, then just after a steal, *all* the remaining processes α_{k-1} through α_0 left in the deque must be only-children. Moreover, the number of leaves is clearly preserved.

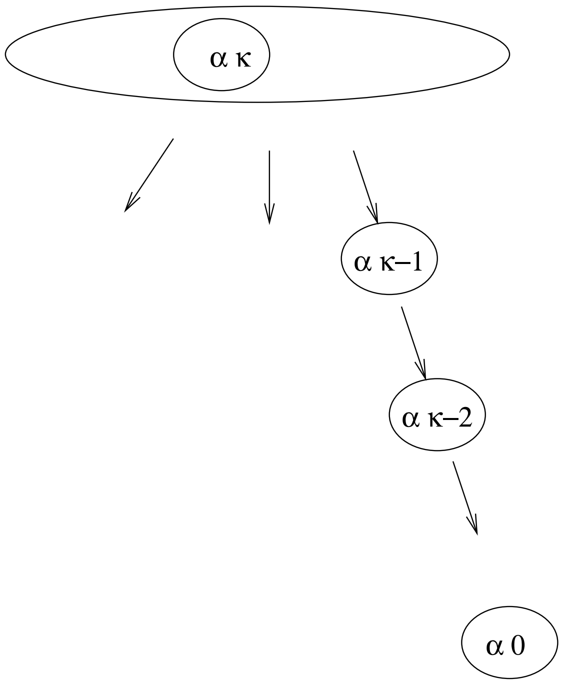

Figure 4: Spawn tree at time t

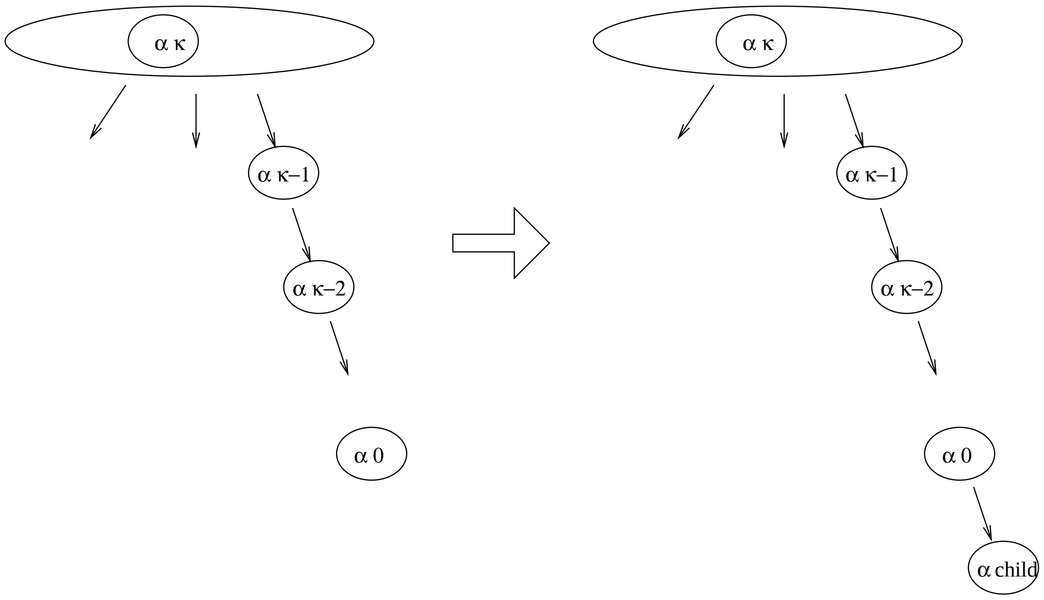

Figure 5: What happens during a spawn

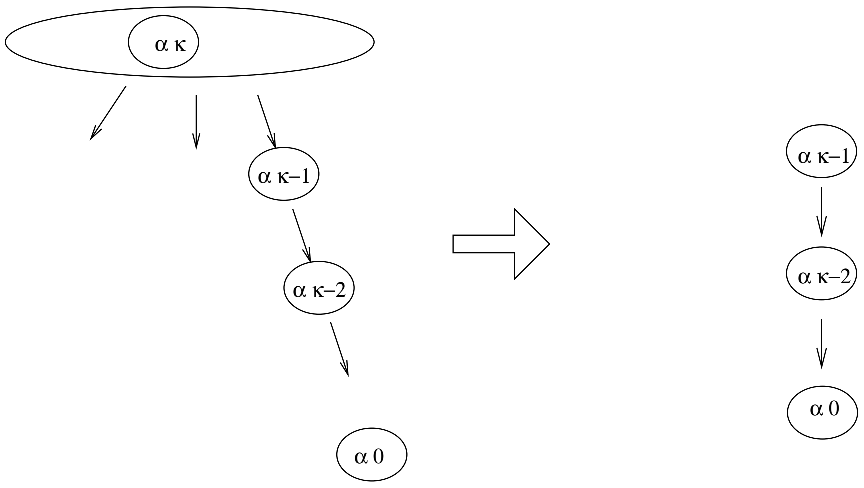

Figure 6: What happens during a steal

- Now we examine what happens in the case of a return. There are two possibilities. If α_0 is not at the top of the deque when it returns, then the processor simply begins working on α_1 and the spawn tree changes in the opposite direction from what occurs in figure Figure 5. On the other hand, if α_0 is the only thing in the deque, then the processor either puts α_0's parent at the top of its deque, or it worksteals the top process off of some other deque. Either way, the number of leaves is maintained.

$\square$

7 The Ideal Thread to Steal

When the Cilk scheduler worksteals, it does so at random. But is this the best method? Which thread should a processor p steal in an ideal schedule?

We'd like to acquire the thread that is closest to the beginning of the DAG, thereby making progress on the critical path. However, it is impossible to know exactly which thread this is without knowing the structure of the remainder of the DAG- that is, without being able to see into the future. So which thread should we steal?

Some designers might be tempted to steal the root. However, we know that the root may be stalled at a synch and is not the most productive thread at that time. Instead, processor p should steal a thread near the beginning of the DAG, ensuring that it is not stalled at a synch and that it will assist with making progress on the critical path. Roughly speaking, the threads at the top of each deque are those that are closest to the beginning of the DAG, so stealing one thread from the top of each deque should reduce the critical path by one.