Outline of this lecture:

1. The time bound that we want to prove
2. The categories of the operations of a worker
3. The structured lemma and its proof

1 Cilk’s scheduler

The strategy used by Cilk’s scheduler to assign jobs to workers and handle load balance has been stated in the previous lecture.

Assume the program has $T_1$ work and $T_\infty$ span. This strategy has the following property in the case where there are $P$ workers: (to be proved later)

- Time bound: $T_P \leq T_1/P + O(T_\infty)$
  This bound guarantees near-perfect linear speedup when the applications has ample parallelism, i.e., $P \ll \frac{T_1}{T_\infty}$.

- Stack space bound: $S_P \leq S_1 \cdot P$

2 Workers’ operations

We can group a worker’s possible operations on its deque into the following four categories:

1. If function A spawns B: push A onto the deque.
2. If function B returns to A:
   - Pop A off the deque and resume A.
   - If the deque is empty, try to steal A back before resuming A.
   - If A can’t be resumed, jump to Operation 4 (randomly steal work).
3. If function A executes a sync, we decide what to do based on whether or not the sync was successful:
   - Successful: continue the execution of A.
   - Failed: suspend A and jump to Operation 4 (randomly steal work).
4. Work steal: randomly choose a victim and steal the work from the top of its deque.
3 Structural Lemma

Definition 1 (Leaf) A subroutine with no outstanding children.

Lemma 1 (Structural Lemma) For deques maintained by the Cilk scheduler, the following properties will always hold:

1. \( \alpha_i \) is a child of \( \alpha_i \) for \( i = 1, \ldots, k \)

2. \( \alpha_i \) is the ONLY child of \( \alpha_i \) for \( i = 1, \ldots, k - 1 \)

3. \( \alpha_k \) may be one of MANY children of \( \alpha_k \)

4. (Busy-leaves property) The number of active leaves is at most \( P \), the number of workers.

Proof. This can be proven by induction on the action performed by a worker on its deque. We will assume that at time step \( t \), these properties hold, and a worker performs either a spawn, a return, a sync, or steal. Then we show that whichever action the worker chooses to take, these properties still hold at time step \( t + 1 \).

Say a worker \( w \) is executing \( \alpha_0 \) and performs:

1. A spawn: say \( \alpha_0 \) spawns \( \alpha'_0 \). Because \( \alpha_0 \) spawned \( \alpha'_0 \), \( \alpha'_0 \) is a child a \( \alpha_0 \), so Property 1 still holds. For other properties, there are two possibilities:

   - The frame \( \alpha_0 \) is now at the top of the deque (i.e., the deque was empty with \( \alpha_0 \) being worked on). In which case, by the inductive hypothesis, \( \alpha_0 \) can have many children, and \( \alpha'_0 \) is one of the many children of \( \alpha_0 \) has, so the Property 3 holds. Also, if \( \alpha_0 \) had other outstanding child at time \( t \), \( \alpha_0 \) was not a leaf, and by Property 1, there were at most \( P - 1 \) leaves at time \( t \), so even though we have a new leaf, i.e., \( \alpha'_0 \), Property 4 is still preserved.

   - The frame \( \alpha_0 \) is not at top of the deque. In which case, \( \alpha_0 \) cannot have been stolen ever, so \( \alpha_0 \) cannot have any outstanding children. Once \( \alpha_0 \) spawns \( \alpha'_0 \), \( \alpha_0 \) has one child, and Property 2 holds. Also, \( \alpha_0 \) was a leaf but not anymore, and \( \alpha'_0 \) becomes the new leaf. Thus, Property 4 is preserved.

2. A return: say \( \alpha_0 \) returns. Again there are two possibilities:
• The deque was empty, so \( w \) either go work steal, or is able to resume \( \alpha_0 \)’s parent (say \( \alpha_1 \)) and pushes \( \alpha_1 \) onto the top of its deque. In either case, there will be at most one frame in the deque, and Properties 1–3 trivially hold. The number of leaves (Property 4) is preserved, by similar arguement as in the case of spawn when \( \alpha_0 \) is at the top of the deque.

• The deque was not empty, and by inductive hypothesis, \( \alpha_1 \) is \( \alpha_0 \)’s parent, so \( w \) pops \( \alpha_1 \) off the deque and resuems it. Since Properties 1–3 hold at time \( t \) and now the deque has one less frame but remain the same otherwise, they still hold. Also, \( \alpha_0 \) was a leaf, but it’s finished, and \( \alpha_1 \) becomes the new leaf (since \( \alpha_0 \) was its only child), so the number of leaves (Property 4) is preserved.

3. A sync: say \( \alpha_0 \) executes a sync. Again there are two possibilities:

• The deque was empty, and sync fails. In which case, \( w \) suspends \( \alpha_0 \) and goes to work steal. Properties 1–3 trivially hold since the deque is empty. The number of leaves (Property 4) holds still, because \( \alpha_0 \) wasn’t a leaf and \( w \) didn’t get any new leaf.

• Execution of sync succeeds. In this case, the deque is unchanged, so all properties are preserved.

4. A steal: say \( w \) is executing a steal, stealing from \( w' \).

• \( w' \)’s deque now has one less item, but otherwise the same, so all properties are still preserved.

• \( w \)’s deque was empty and now has one stolen frame. Property 1 trivially holds. Since this stolen frame was on top of \( w' \)’s deque, it can have many outstanding children at time \( t \), and thus Property 3 holds still. Finally, the number of leaves (Property 4) is preserved, since \( w \) the stolen frame does not add to the number of active leaves — it was on top of \( w' \)’s deque so it couldn’t have been a leaf.

Note that the space bound follows from the busy-leaves property (Property 4). Since there can be at most \( P \) busy leaves at any given time point, and the stack space used by a busy leaf and all its ancestor is bounded by \( S_1 \), hence \( S_P \) is bounded by \( S_1 \cdot P \).

4 Time bound of Cilk scheduler

Consider the execution of a Cilk computation with \( T_1 \) work and \( T_\infty \) span, using Cilk’s work stealing algorithm on \( P \) processors. For any \( \epsilon > 0 \), the execution time on \( P \) processors is

\[
T_P \leq T_1/P + O(T_\infty + \lg(1/\epsilon))
\]

with probability at least \( 1 - \epsilon \).

We will show this bound using an accounting method. Assume we have two buckets, a work bucket, and a steal bucket. At every time step,
• a worker puts $1 into the work bucket if it executes something in the user computation;
• a worker puts $1 into the steal bucket if it tries to steal.

Then at every time step, collectively $P$ get put into these buckets. At the end of the execution, the total time steps taken can be bounded by

$$T_p \leq \frac{\$\text{work bucket} + \$\text{steal bucket}}{p}.$$ 

It’s easy to see that the number of dollar in the work bucket must be less than $T_1$, because there is only $T_1$ amount of work in the computation. In the next lecture, we will see how we bound the dollars in the steal bucket.