Beyond “Nested Parallelism”

**Futures** are a way to pipeline computations while still having “deterministic” parallel programs (i.e. ones that don’t depend on the schedule).

Available in Java concurrency library.
Futures

future e : generate a “future” cell and a parallel task to evaluate e. Return the cell immediately. When e is done, it places its value in the “future” cell.

? e : This operation (called “touch”) gets the value out of the cell. If the value is not ready, it waits until it is and then returns the value.
fun produce(0) = nil
   | produce(n) = f(n):: future produce(n-1);

fun consume(nil, state) = state
   | consume(h::t, state) =
      consume(?t, g(h, state));

fun map(nil) = nil
   | map(h::t) = h(t):: future map(?t);

The producer and consumer can work in parallel in a pipelined fashion.
The diagram illustrates the process of producing and consuming data over time. It shows two separate processes:

1. **Produce-consume cycle**: This cycle involves two processes, P1 and P2, where P1 produces data that is consumed by P2. This cycle repeats over time as indicated by the arrow labeled "Time".

2. **Produce-map-consume cycle**: This cycle involves P1 producing data, which is mapped by P2 and then consumed by P3. This cycle also repeats over time.

The arrows indicate the flow of data between the processes, with P1 and P2 on the left side of the diagram and P1, P2, and P3 on the right side.
Consumer Producer

fun ints(0) = 0
  | ints(n) = n :: future ints(n-1)

fun sum(nil, accum) = accum
  | sum(h::t, accum) = sum(?t, h+accum)

sum(ints(1000))

When there are no race conditions futures have a sequential semantics (they always return the same thing as if they were ignored)
fun filter f nil = nil
    | filter f h::t =
      if f(h)
      then h::(future (filter f ?t))
      else filter f ?t;

fun qsort(nil,rest)  = rest
    | qsort(h::t,rest) =
      let val L1 = filter (fn b => b < h) ?t
        val L2 = filter (fn b => b >= h) ?t
        in qsort(L1,h::future qsort(L2,rest)) end;

fun quicksort(L) = qsort(L,nil);
Qsort Complexity

Sequential Partition
Parallel calls

Depth = $O(n)$

Work = $O(n \log n)$
QuickSort with Futures

Depth = $O(n)$

Work = $O(n \log n)$

Still not a very good parallel algorithm
Merging

\[ \text{Merge}(A, B) = \begin{align*}
&\text{let} \\
&\text{Node}(A_L, m, A_R) = A \\
&(B_L, B_R) = \text{split}(B, m) \\
&\text{in} \\
&\text{Node}(\text{Merge}(A_L, B_L), m, \text{Merge}(A_R, B_R))
\end{align*} \]

Depth = \(O(\log^2 n)\)
Work = \(O(n)\)

Merge in parallel

Futures
Merging with Futures

\[
\text{Merge}(A, B) = \\
\text{let} \\
\quad \text{Node}(A_L, m, A_R) = A \\
\quad (B_L, B_R) = \text{futureSplit}(B, m) \\
\text{in} \\
\quad \text{Node}(\text{Merge}(A_L, B_L), m, \text{Merge}(A_R, B_R))
\]

Depth = $O(\log n)$
Work = $O(n)$