Blellock 5

How to memoize:

w/ side effects – careful in parallel world

assume Table implementation (hash table, etc)

- find
- insert

} on tuples of integers

O(1) insertion, find
↳ in expectation (hashes, etc.)

- new → creates empty table

a = arg type

b = result type

memoize : ((a → b) → (a → b)) → (a → b)

How to use:

```
val fib = memoize (
    fn fib' n => if (n ≤ 1) then 1
                 else fib'(n-1) + fib'(n-2) )
```

added to usual impl of fib.

memoize - create memoized version of the function and pass it in

<u>Sequential version</u>

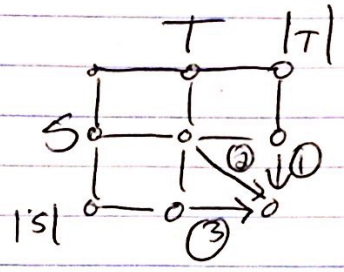
```
fun memoize f = let
    val cache = ref (Table.new())
    fun mem arg =
        case Table.find (!cache) arg of
          Some r => r
        | None   => let val ~result~ r = f mem arg
                        val _ = cache :=
                                Table.insert
                                (!cache, (arg, r))
                    in r
                    end
in mem end
```

for parallel, want:

```
fn fib' n => if n ≤ 1 then 1
             else let (a,b) = fib(n-1) || fib(n-2)
                  in a+b
                  end
```

we looked at most sharing 2 positions (3)

Med $\overbrace{(i,j)}^{2}$



call location possibilities

Parallel

#1) Make table safe for concurrency

"linearizable"

operations act as if <u>atomic</u>

atomic interleaving of instructions

a b c

insert find insert

(non-deterministic which happens first,

but always acts as if linear ops)

ex: a b c
a c b

whether slow depends on <u>contention</u>

Problem:

wastes work

(2 ~~to~~ can realize value
is not evaluated
and start evaluating
duplicating work)

Starts looking like laziness
(first forces, second notices
value is being evaluated)

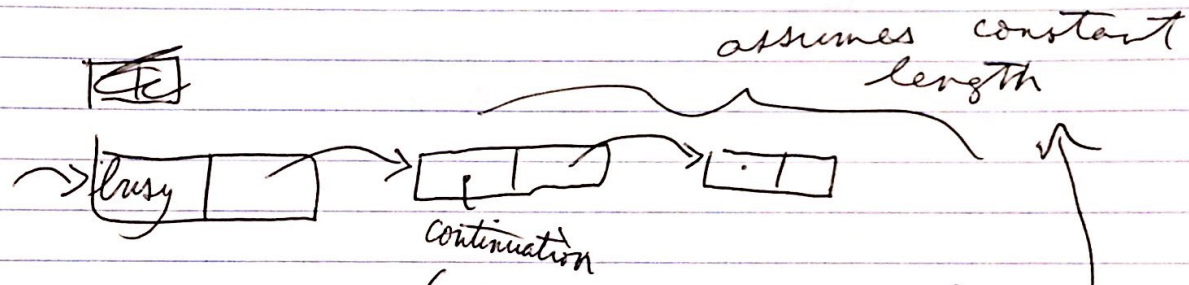
first one coming to work
"lays claim" to this location/computation
other ones gets a response that work
is being done

3 reponses to find:

1) empty
2) busy
3) full w/ value

What to do when table.find returns busy?

need to (associate,) set of continuations need to <u>suspend</u>



(thunk returning unit)

if busy, then add self to busy computation

use: CAS (compare & swap)

(will fail constant number of times due to —)

(this is combined w/ work-stealing queues)

when computation finishes,

1) change from busy to full

2) wake up continuations

push those onto work-stealing stack

(better when constant continuations)

(identical to futures
but w/ extra hashtable)

future
fork job
return handle to job
return immediately

when access handle to job,
will wait till done

3 parts

1) busy

2) push on value assoc w/ busy indicator
& suspend (i.e. in hash table)

3) when full, wake up suspended jobs

If not constant in call paths
then would be problems with:

- contention
- long span due to many continuations

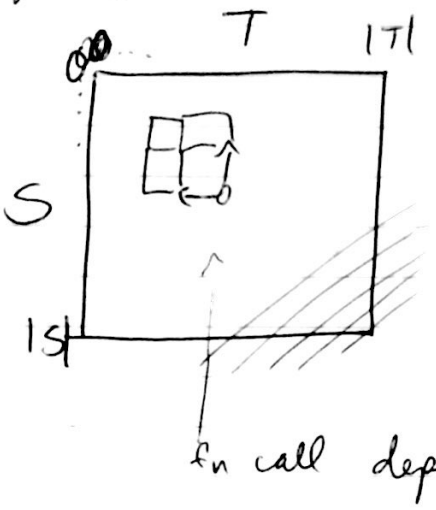
Q: memory consumption

bounds in work stealing do not hold here

This is An implementation which:

- doesnt waste work
- doesnt add to span

A <u>cache-efficient</u> way to do <u>dynamic</u> programs



MED

DAG looks like

looks like the above (dep on left & above)

"n^2" work (eval every cell

could fill up table w/ diagonal

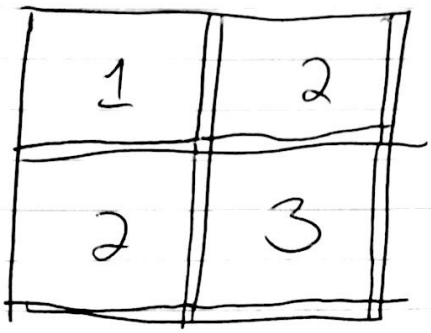
(another way to see linear span)

diagonals $= |S| + |T|$

Problem: diagonals are not cache-friendly

Alternative

idea: div & conq



divide in 4 quadrants

can generate 1 w/o other quadrants

can do both 2's in parallel

finally do 3

need to store one col + one row of quadrant

don't need memoization magic

input: part S, part T, row, cols at bottom / right

assume $|S|+|T|=n$

recurrence: $W(n) = 4\left(\frac{n}{2}\right) + O(n)$ → appending cols & rows

$= O(n^2)$

$$\text{Span} = S(n) = 3\,S\left(\frac{n}{2}\right) + O(1)$$

$$= 3^{\log_2 n} = n^{\log_2 3} = n^{1.4}$$

lost span vs memoized

parallelism

$$= \frac{n^2}{n^{\log_2 3}} \approx n^{.6}$$

advantage = lots of cache locality

common method of dyn prog now

program synthesis ⇝ convert to div & conq.

to gen parallel version