WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Chapter 20

Parallelizing Computations

- Parallel map
- MapReduce
- Idea 1: take advantage of side-effect-free computation
- Idea 2 (again): embody the parallel implementation details in a higher-order function (or module) so the hard stuff is only done once

Map

- Recall the common code pattern, for some function f1, that from one list builds another list consisting of f1 applied to each of the elements.

```
f([]) -> [];
f([H|T) -> [f1(H) | f(T)].
```

map f1,

map (f1, []) -> [];

map (f1, [H|T]) -> [f1(H) | map(f1, T)].

- We can rewrite this using the higher-order function map

```
f(L) -> map(f1, L).
```

Map is an opportunity for parallelizing

- map definition

```
map(_, []) -> [];
map(F1, [H|T]) -> [F1(H)|map(F1,T)].
```

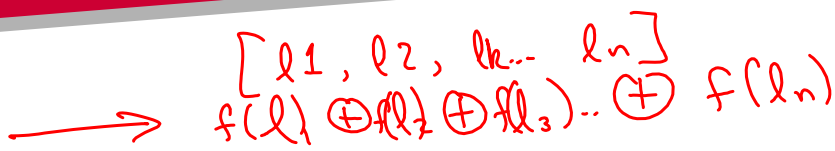
- `F1(H)` doesn't depend on other list elements

```
pmap(F, L) ->
    S = self(),
    Ref = erlang:make_ref(),
    Pids = map(fun(I) ->
       spawn(fun() -> do_f(S, Ref, F, I) end)
       end, L),
    map(fun (Pid) ->
       receive {Pid, Ref, Ret} -> Ret end
       end, Pids)
% what is the type of Pids?
```

What is the overhead of pmap?

- Cost of spawning all the processes
- Cost of constructing the Pids list (== cost of constructing the result list)
- Cost of sending and receiving each of the Ret values from the spawned processes
- Worthwhile:
  - Enough processors
  - Enough work in each process

} Amdahl's Law



reduce in functional programming

- **Also known as "fold" it captures the second main recurring pattern in fp**

```
count([]) = 0;
count([H|T]) = 1 + count(T).
```

- **Reduce definition**

accumulator

```
reduce(_, Acc0, []) = Acc0;
reduce(F, Acc, [H|T]) =
    reduce(F, F(Acc, H), T).
count(L) = reduce(fun(A,H) -> A+1 end, 0, L).
sum(L) = reduce(fun(A,H) -> A+H end, 0, L).
prod(L) = reduce(fun(A,H) -> A*H end, 1, L).
```

MapReduce

- Invented by Jeffrey Dean and Sanjay Ghemawat at Google
- Idea: mapreduce(F1, F2, L, Acc0)
  - call F1 on each element of L. F1 produces as its result a list of pairs [{key, val}]
  - mapreduce gathers all the vals produced for a given key and
  - calls F2(key, [vals], Acc0) for each key, producing the final result
- Poor choice of name(?)
  - They call F1 "map" and pass it a pair that they call {key, val}. Using a pair for this argument is not essential
  - "reduce" is an ok name

What's cool about MapReduce

- It encapsulates the pmap idea and the distribution of the spawned processes to different processors
- It encapsulates gathering all of the intermediate results with each key into one list (this is tricky to do well on a cluster so doing it once is good)
- It encapsulates the distribution of the reduce actions across the cluster
- It encapsulates error handling and recovery (processor failures are not rare in large clusters!)

Use of MapReduce

```
listwords(Filename) ->
    % foreach word, w, in the file emit
    % {w, 1}.
reduceCounts(W, L, _Acc0) = {w, length(L)}.
wordCount(Filenames) -> mapreduce(listwords,
  reduceCounts, 0, Filenames).
```

- The Google paper is linked on the website.
- Note the use of FP ideas in a system implemented in C++; and note that it is a little clunky around the edges but the key ideas are clear.

Another Use – Word index for set of documents

- Want: a list of all the documents that contain each word

```
listwords(Filename) ->
    % foreach word, W, in the file emit
    % {W, Filename}.
ident(W, L, _Acc0) -> {W, L}.
createIndex(Filenames) ->
    mapreduce(listwords, ident, 0,
 Filenames).
```

Another Use – Word index for set of documents

- Want: a list of all the documents that contain each word

```
listwords(Filename) ->
    % foreach word, W, in the file emit
    % {W, Filename}.
ident(W, L, _Acc0) -> {W, L}.
createIndex(Filenames) ->
    mapreduce(listwords, ident, 0,
 Filenames).
```