S4.3: Expression Evaluation: Parallel Evaluation

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

Parallelism vs Concurrency

These terms, "parallelism" and "concurrency" are often confused and mistakenly thought to mean the same thing.

What do you think they mean?

Questions about components

We often think of breaking programs into their component pieces, so what are some relevant questions?

- What are the dependencies between components? Does one use information produced by another?
- Do the components have side-effects on global memory, the file system, the network?
- Is the computation deterministic? Respecting the dependencies, does the scheduling of the component affect the result computed? If so, the computation is not deterministic.

Concurrency

A concurrent program has multiple threads of control that have effects which are visible to other threads.

In a concurrent program, these threads are scheduled to execute in an arbitrary way.

These programs may be non-deterministic when the order of thread execution affects the result of the overall computation.

The need for concurrency is a property of the application.

Application of concurrency

Many problems can be nicely organized as concurrent programs:

- A web browser: the user interface code and the web page rendering code are typically separate communicating threads.
  Pulling down a menu does not have to stop the page from rendering.
- Any GUI for that matter
- Multi-player games
- Modeling and simulations, *e.g.* simulating automobile traffic flow.

Concurrent Programming

Controlling the possible non-determinism in a concurrent program can be quite difficult and is the main challenge in concurrent programming.

Often low-level language constructs such as threads and locks are used and it is difficult to use these correctly.

In this approach, a thread should acquire the lock for a portion of shared memory before it is written to.

When multiple locks are needed, acquiring them in the right order is difficult and it is easy to forget to acquire the lock before writing to memory.

Locks

Assume `lock_x` is the lock for `x`.

A. `acquire ( lock_x )`	C. `acquire ( lock_x )`
1. `y = x;`	4. `z = x;`
2. `y = y + 1;`	5. `z = z + 1;`
3. `x = y;`	6. `x = z;`
B. `release ( lock_x )`	D. `release ( lock_x )`

One of the processes will "get the lock" first and thus prevent the other from proceeding to access and and change `x` until the first has released the lock.

There are higher level constructs for concurrency than these low-level locks.

Thus, sequence 1, 4, 2, 5, 3, 6 is not possible.

Parallelism

A parallel program is one that uses multiple processors or cores with the goal of executing faster than one using only one processor.

Parallelism is a property of the machine or the translation of a program.

Some algorithms may be able to better utilize multiple processors than others, and certain formulations may better expose opportunities to execute components at the same time on different processors.

Parallelism is an effort to make deterministic programs run faster.

An example

We could possibly use multiple processors to evaluate the following expression more quickly than using just one processor.

```
let result = map fact xs
```

A parallel implementation of `map` might split this computation over multiple processors.

Of course, we can use concurrent programming models and languages to write programs with the aim of speeding them up on parallel computers.

This is why you'll hear people say "parallel programming is hard" when they should be saying that "concurrent programming, with threads and locks is hard."

Using low-level techniques to control the concurrency of threads that have effects on a global memory is difficult.

Consider a global mutable variable g

...with an initial value of `0` in this pseudo-code.

```
let f x =
   let y = g
   in y := y + x;
      g := y;
      return y

let result = map f [1;2;3;4;5;6;7;8;9;10]
```

Using `map` here might not be deterministic.

The reason for this the function `f` is not pure, that it, it has side-effects.

Thus parallel programming in a purely functional language is much simpler than in using concurrent programming languages or features.

Said another way ...

The reason that we can easily and correctly think of `map` as safely (deterministically) working in parallel is because the sub-computations do not interfere or interact with one another.

This is the same reason why call by name and call by value semantics evaluate expressions to the same value (when both result in a value).

The order of evaluation of subexpressions did not matter because they were pure.

Now, instead of considering the order in which sub expressions evaluate, we consider evaluating them at the same time.

Parallelism in pure functional programs

We've said before the reasoning about pure functional programs is to some extent easier than reasoning about imperative programs.

We saw how proving properties of pure functions we, perhaps, more straightforward than for imperative loops.

This also has benefits in parallel programming, without side effects parallel programming may be easier because we don't have to worry about the order in which independent, pure computations are performed.

Independent computations

A very simple example of an opportunity to execute sub-computations in parallel is in binary operators.

In `e1 + e2`, we could evaluate `e1` and `e2` in parallel since they are pure.

But parallelizing a program in this way is usually not worth the trouble.

Parallel Map

The simplest example that does have some performance benefit is a parallel map: `map f data`

If `data` = | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

then `map f data` =

f 1	f 2	f 3	f 4	f 5	f 6	f 7

And we can execute all applications of `f` at the same time, if we have enough processors.

Parallel Fold

Consider `foldl (+) 0 data` where
`data` = | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

This becomes
`(((((((0 + 1) + 2) + 3) + 4) + 5) + 6) + 7)`

Is this a parallel operation?

No. But `0 + 1 + 2 + 3 + 4 + 5 + 6 + 7` might be performed differently.

For example
`(((0 + 1) + (2 + 3)) + ((4 + 5) + (6 + 7)))`
might be done in parallel.

What property of `+` makes this possible?

Associative operators

An associative operator $\oplus$ is one for which

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

So, $+$ and $\times$ are associative. But not all operations are.

Folds using associative operations can be done in parallel.

But is `foldl (+) 0 data` "as parallel as" `map f data`?

How can we precisely answer this question?

Guy Blelloch defines two concepts: **work** and **depth**.

Work and depth

- work as, roughly, the total amount of computation that must be performed.

  This corresponds to the time it would take to execute the program on a single processor.

- depth as, roughly, the longest chain of dependent computations – where the input of one is the output of the previous one.

  This corresponds to the time it would take to execute the program on an unbounded number of processor.

In `map f data`, n is the size of `data`, and to simplify things, assume `f` is a constant time function.

What is the work of this computation? $O(n)$

What is the depth of this computation? $O(1)$

In `foldl (+) 0 data`, n is again the size of `data`.

What is the work of this computation? $O(n)$

What is the depth of this computation? $O(log_2 n)$

Recall...

`map f data` =

`f 1`	`f 2`	`f 3`	`f 4`	`f 5`	`f 6`	`f 7`

Scheduling the evaluations of `f` for different values is something we'd like the language's compiler figure out for us.

We can write the program to expose the parallelism, but let the compiler efficiently `exploit` that parallelism.

This is what now happens with register allocation. People used to want to write assembly code so they could control these register resources explicitly. Now, a good compiler does a better job than people can usually do. So no one does that anymore.

Perhaps, eventually parallel programming, at least in some settings, will be the same way.

Structured vs Flat Data

We've been glossing over an important point so far.

Lists are not random access data structures, they are accessed from the front only.

To access the 11th element we must traverse over the first 10.

Flat data, like arrays, are random access.

We can access the 11th element in constant time without accessing the first 10.

Most parallel languages work on flat data, not structured data.

MapReduce

MapReduce is a well-known framework for distributing large parallel computations over a large cluster of workstations.

It was developed by Jeffrey Dean and Sanjay Ghemawat at Google.

See the paper at
`http://research.google.com/archive/mapreduce.html`

Jeff Dean is an alum of the CS department who took 1901 and learned about higher order functions.

Hadoop is a common open source implementation of MapReduce and is widely used. Spark is another.

MapReduce

- Input is a set of key/value pairs - `('k1 * 'v1) list`
- A user-provided "mapper" function is applied to each pair, returning a list of new key/value pairs.
  Its type: `('k1 * 'v1) -> (('k2 * 'v2) list)`
- These results are grouped by the new key type resulting in data of this type:
  `('k2 * ('v2 list)) list`
- Each of these is given to a user-defined "reducer" function with the type
  `('k2 * ('v2 list)) -> 'v2 option`
- Each output, if there is one, associated with its input key and the final output has the type `('k2 * 'v2) list`.

MapReduce

- We could thus define `mapReduce` to have the following type:

```
mapReduce : (('k1 * 'v1) -> (('k2 * 'v2) list))
         -> ('k2 * ('v2 list)) -> 'v2 option
         -> ('k1 * 'v1) list
         -> ('k2 * 'v2) list
```

  The inputs are the mapper function, the reducer function, and the input data.
- We used OCaml types to better explain this, but these types don't exist in the MapReduce paper.

MapReduce and Types

- An interesting paper by Ralf Lämmel studies MapReduce from a type-oriented perspective and derives Haskell types for MapReduce.
- It shows how thinking in terms of types makes things precise and exposes weaknesses in the original paper. It also suggests other ways to think about MapReduce.
- You can find that paper here:
  `http://userpages.uni-koblenz.de/~laemmel/MapReduce/`
- The OCaml types above are derived from that paper.

Exploiting parallelism

- If `map` and `fold` provide opportunities for parallel evaluation, how can we exploit that?
- What language do we translate `map` and `fold` into in order to realize the parallelism in the computation?

Concurrency constructs

What capabilities do we need to in concurrent programs?

- a way to create threads
- a way to control their execution

Cilk

Cilk is an extension to C with two primary additions.

- `spawn`
  executes a function and returns immediately
- `sync`
  this waits for all spawned functions (from that thread) to complete

Consider `fib.cilk`.

Non-determinism

- In Cilk, synchronization of tasks is handled explicitly.
- Leaving out a `sync` statement may result in incorrect results.
- Consider this in the `fib.cilk`.

Map and fold?

As eluded to before, we'll consider `map` and `fold` over flat arrays instead of structured lists.

We'll have

- `read` - read an array from a file
- `mkArray : (int -> 'a) -> int -> 'a array`
- `map`
- `fold`

OCaml to Cilk

OCaml and Cilk are somewhat different.

What challenges might we face in translating the OCaml constructs to Cilk?

Some simplifications

Let's not consider all of OCaml and limit how functions and arrays are used.

Also, recall that OCaml programs are a sequence of declarations.

- No lambda-expressions. All functions are declared "at the top level".
  A process called "lambda lifting" can perform this transformation.
- The last function is called `main` and has type `unit -> unit`.

Some more simplifications

- Simple array expressions.
  `fold plus (map square (mkArray id 100))`
  must be written

```
let a1 = mkArray id 100 in
let a2 = map square a1  in
let v = fold plus a2   in
v
```

  Each array computation is done in sequence.
  This looks like a sequence of assignments.

Thus ...

```
fold (fun x y -> x + y)
  (map (fun x -> x * x) (mkArray (fun x -> x) 100))
```

may become

```
let plus x y = x + y
let square x = x * x
let id x = x
let main () =
  let a1 = mkArray id 100 in
  let a2 = map square a1 in
  let v = fold plus a2 in
  print_endline v
```

Simplified OCaml to Cilk

- Arrays:
  - in OCaml: `read`, `mkArray`, `map`, and `fold`.
    but only as pseudo-statements in `let` expressions.
  - in Cilk: C arrays, for-loops and array access using `[ ]`

Exploiting parallelism

- Even with our simplifying assumptions, efficiently exploiting parallelism is difficult.
- Each `spawn` has some overhead.
- When functions have are small, this overhead can dominate execution time.
- When functions have variable execution times, it is difficult to schedule them statically.
- Lack of good solutions has hindered a good parallel implementation of OCaml.