Lecture 27: Parallel programming and functional languages

- Why parallel programming is hard
- Why functional programming helps
- Two case studies
  - Google’s MapReduce
  - F#’s asynchronous workflows

Why parallel programming is hard

- Dependencies
- Race conditions
- Deadlock

Granularity of parallelism

- Instruction-level parallelism
- Higher-level parallelism

Approaches to parallel programming

- Automatic parallelization, i.e. parallelizing compilers
- Manual parallelization – low-level
  - MPI, OpenMP
- Manual parallelization – high-level
  - Languages incorporate abstract models of parallelism
  - Libraries implement models of parallelism

Why functional languages help

- Reduce number of dependencies – makes both automatic and manual methods easier
- E.g. in application of map function, applications of function to each element are usually independent.

Why functional languages help

“Due to the absence of side-effects in a purely functional program, it is relatively easy to partition programs so that sub-programs can be executed in parallel: any computation which is needed to produce the result of the program may be run as a separate task. …

“Higher-order functions (functions which act on functions) can also introduce program-specific control structures, which may be exploited by suitable parallel implementations.”

- *Kevin Hammond, www-fp.dcs.st-and.ac.uk/~kh/papers/pasco94/pasco94.html*

Why functional languages help

- Consider imperative and functional implementations of quicksort

Imperative:

```
qsort(a, lo, hi):
    p = choose pivot, move to a[lo]
    partition (a, lo+1, hi, pivot)
    qsort(a, lo+1, (lo+hi)/2)
    qsort(a, (lo+hi)/2+1, hi)
```

Functional:

```
qsort(lis):
    p = choose pivot, remove from lis
    (l, u) = partition(lis, p)
    l’ = qsort(l)
    u’ = qsort(u)
    l’ @ [p] @ u’
```

Two case studies

- Google's MapReduce
  - Parallelism in processing large amounts of data from multiple processors in a data center
  - Library-based model of parallelism
- Microsoft's F# w/ asynchronous workflows
  - Programming model for parallelism in functional language

Google’s MapReduce

- Used to access data from Google’s data centers.
- Inspired by map and reduce (fold) operations:
  - Divide calculation into two parts:
    - map – apply function to data independently on a set of processors
    - reduce – combine results of map operations
- Available to public in “hadoop” implementation
- More info: Dean & Ghemawat, “MapReduce: Simplified data processing in large clusters”

Google's MapReduce

- User defines (usually in C++) functions map and reduce:

  map: string*string -> (string * string) list

  reduce: string*(string list) -> string list

- **map** is executed on a collection of processors, producing a list of (key,value) pairs on each
- The underlying MapReduce library combines these pairs, groups and sorts by key, then calls **reduce** for each key, giving all the values associated with that key. It returns the combined list of all values returned from these calls.

Word-counting

- map (string docname, string doccontents):

  ```
  for each word w in doccontents:
      emit (w, “1”)
  ```

- reduce (string word, list<string> counts):

  ```
  int result = 0
  for each n in counts:
      result := parseInt(n)
  emit([“”+result])
  ```

- User also supplies mapreduce specification object telling system how to get started (e.g. document names to apply map to)

F#’s asynchronous workflows

- F# a .NET implementation of (a variant of) OCaml.
- “Asynchronous workflows” is a way to turn ordinary programs into parallel programs.
  - Based on language feature called “computation expressions”
  - Underlying implementation uses “Task Parallel Library”
- [show video - http://channel9.msdn.com/pdc2008/TL11/]

How asynchronous workflows work

- “Computation expressions,” are an F# feature, inspired by the Haskell “monad” feature, which allows for a kind of reflection.
- Computation expressions allow certain language constructs to be re-interpreted using user-supplied semantics. The Async library is a workflow.

Computation expressions

- seq { … yield e … } executes "… yield e …" and gathers the values of e into a list.
- Within "…", can use limited number of constructs:
    - use var=expr in expr
    - let var=expr in expr
    - expr; expr
    - yield expr, …
- "seq" is not a keyword, but the name of an object that says how to interpret these language constructs.

Computation expressions

- General form of computation expression:

  name { … expression as above … }

- name must be bound to an object of a class that implements these operations:
  - Bind: α comp * ($\alpha \rightarrow \beta$ comp) $\rightarrow \beta$ comp
  - Delay: (unit $\rightarrow \alpha$ comp) $\rightarrow \alpha$ comp
  - Let: α comp * ($\alpha \rightarrow \alpha$ comp) $\rightarrow \alpha$ comp
  - Return: $\alpha \rightarrow \alpha$ comp

where comp is any type constructor you want (e.g. list).

Computation expressions (cont.)

The definitions of the above operators are used the interpret the syntax within the computation expression. E.g.

```
c { let n1 = f in1
      let n2 = g in2
      let sum = n1+n2
      yield sum }
```

would translate (statically) to

```
c.Delay(fun () ->
   c.Bind(f in1, (fun n1 ->
        c.Bind(f in2, (fun n2 ->
             c.Let(n1+n2, (fun sum -> c.Return sum)))))))
```

Asynchronous workflows

Asynchronous workflows are an application of computation expressions.

The Async module implements these operations (among others) using the Async type constructor:

Bind: α Async * ($\alpha \rightarrow \beta$ Async) $\rightarrow$ β Async

Return: $\alpha \rightarrow \alpha$ Async

plus these methods:

Run: α Async *int * bool $\rightarrow$ α

Parallel: (α Async) list $\rightarrow$ (α list) Async

Spawn: unit Async $\rightarrow$ unit