Ruby Threads – Thread Creation

- Create thread using `Thread.new`
  - **New** method takes code block argument
    
    ```ruby
t = Thread.new { …body of thread… }
t = Thread.new (arg) { | arg | …body of thread… }
    ```
  - **Join** method waits for thread to complete
    ```ruby
t.join
    ```

- **Example**
  ```ruby
  myThread = Thread.new {
    sleep 1           # sleep for 1 second
    puts "New thread awake!"
    $stdout.flush     # flush makes sure output is seen
  }
  ```
Ruby Threads – Locks

- **Monitor, Mutex**
  - Object intended to be used by multiple threads
  - Methods are executed with mutual exclusion
    - As if all methods are synchronized
  - Monitor is reentrant, Mutex is not

- **Create lock using Monitor.new**
  - Synchronize method takes code block argument
    ```ruby
    require 'monitor.rb'
    myLock = Monitor.new
    myLock.synchronize {
      # myLock held during this code block
    }
    ```

Ruby Threads – Condition

- **Condition derived from Monitor**
  - Create condition from lock using `new_cond`
  - Sleep while waiting using `wait_while`, `wait_until`
  - Wake up waiting threads using `broadcast`

- **Example**
  ```ruby
  myLock = Monitor.new # new lock
  myCondition = myLock.new_cond # new condition
  myLock.synchronize {
    myCondition.wait_while { y > 0 } # wait as long as y > 0
    myCondition.wait_until { x == 0 } # wait as long as x == 0
  }
  myLock.synchronize {
    myCondition.broadcast # wake up all waiting threads
  }
  ```
Parking Lot Example

```ruby
require "monitor.rb"
class ParkingLot
  def initialize  # initialize synchronization
    @numCars = 0
    @myLock = Monitor.new
    @myCondition = @myLock.new_cond
  end
  def addCar
    ...  
  end
  def removeCar
    ... 
  end
end
```

Parking Lot Example

```ruby
def addCar # do work not requiring synchronization
  @myLock.synchronize {
    @myCondition.wait_until { @numCars < MaxCars }
    @numCars = @numCars + 1
    @myCondition.broadcast
  }
end

def removeCar # do work not requiring synchronization
  @myLock.synchronize {
    @myCondition.wait_until { @numCars > 0 }
    @numCars = @numCars - 1
    @myCondition.broadcast
  }
end
```
Parking Lot Example

```ruby
garage = ParkingLot.new
valet1 = Thread.new { # valet 1 drives cars into parking lot
  while …
    # do work not requiring synchronization
    garage.addCar
  end
}
valet2 = Thread.new { # valet 2 drives car out of parking lot
  while …
    # do work not requiring synchronization
    garage.removeCar
  end
}
valet1.join # returns when valet 1 exits
valet2.join # returns when valet 2 exits
```

Ruby Threads – Difference from Java

- Ruby thread can access all variables in scope when thread is created, including local variables
  - Java threads can only access object fields, or final local variables
- Exiting
  - All threads exit when main Ruby thread exits
  - Java continues until all non-daemon threads exit
- When thread throws exception
  - Ruby only aborts current thread (by default)
  - Ruby can also abort all threads (better for debugging)
    - Set Thread.abort_on_exception = true
Ocaml Threads – Thread Creation

Create thread using Thread.create
- method takes closure as its argument
  let t = Thread.create (fun x -> ...body...) arg;;
- Join method waits for thread to complete
  Thread.join t

Example
let myThread = Thread.create (fun _ ->
  Unix.sleep 1; (* sleep for 1 second *)
  print_string "New thread awake!";
  flush Pervasives.stdout (* flush ensures output is seen *)
);;

Ocaml Threads – Locks & Conditions

Mutex module
- Not reentrant
- Has lock, unlock functions
  let myLock = Mutex.create ();;
  Mutex.lock myLock;
  (* myLock held here *)
  Mutex.unlock myLock

Condition module
- Create condition directly
- Sleep while waiting using wait (takes mutex arg)
- Wake up waiting threads using broadcast
Multithreading (Java threads, pthreads)

+ Portable, high degree of control
- Low-level and unstructured
  • Thread management, synchronization via locks and signals essentially manual
    • Blocking synchronization is not compositional, which inhibits nested parallelism
  • Easy to get wrong, hard to debug
    • Data races, deadlocks all too common

Parallel Language Extensions

- MPI – expressive, portable, but
  • Hard to partition data and get good performance
    ➢ Temptation is to hardcode data locations, number of processors
  • Hard to write the program correctly
    ➢ Little relation to the sequential algorithm
- OpenMP, HPF – parallelizes certain code patterns (e.g., loops), but
  • Limited to built-in types (e.g., arrays)
  • Code patterns, scheduling policies brittle
Two Directions To A Solution

- Start with clean, but limited, languages/abstractions and generalize
  - MapReduce (Google)
  - StreamIt (MIT)
  - Cilk (MIT)

- Start with full-featured languages and add cleanliness
  - Software transactional memory
  - Static analyzers (Locksmith, Chord, …)
  - Threaded Building Blocks (Intel)

Space of Solutions

Cleanliness

Flexibility
Kinds of Parallelism

- **Data parallelism**
  - Can divide parts of the data between different tasks and perform the same action on each part in parallel

- **Task parallelism**
  - Different tasks running on the same data

- **Hybrid data/task parallelism**
  - A parallel pipeline of tasks, each of which might be data parallel

- **Unstructured**
  - Ad hoc combination of threads with no obvious top-level structure

MapReduce: Programming the Pipeline

- Pattern inspired by Lisp, ML, etc.
  - Many problems can be phrased this way

- Results in clean code
  - Easy to program / debug / maintain
    - Simple programming model
    - Nice retry / failure semantics
  - Efficient and portable
    - Easy to distribute across nodes

Thanks to Google, Inc. for some of the slides that follow
Map & Reduce in Lisp / Scheme

- `(map f list)`
- `(map square '(1 2 3 4))`
  - `(1 4 9 16)`
- `(reduce + '(1 4 9 16) 0)`
  - `(+ 1 (+ 4 (+ 9 (+ 16 0) ) ) )`
  - `30`
- `(reduce + (map square '(1 2 3 4)) 0)`

MapReduce a la Google

- `map(key, val)` is run on each item in set
  - emits new-key / new-val pairs
- `reduce(key, vals)` is run for each unique key emitted by `map()`
  - emits final output
Count Words In Documents

- Input consists of (url, contents) pairs

  - map(key=url, val=contents):
    - For each word \( w \) in contents, emit \( (w, "1") \)

  - reduce(key=word, values=uniq_counts):
    - Sum all "1"s in values list
    - Emit result "(word, sum)"

Count, Illustrated

map(key=url, val=contents):
  - For each word \( w \) in contents, emit \( (w, "1") \)
reduce(key=word, values=uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>see</td>
<td>1</td>
<td>bob</td>
<td>1</td>
</tr>
<tr>
<td>bob</td>
<td>1</td>
<td>run</td>
<td>1</td>
</tr>
<tr>
<td>run</td>
<td>1</td>
<td>see</td>
<td>2</td>
</tr>
<tr>
<td>see</td>
<td>1</td>
<td>spot</td>
<td>1</td>
</tr>
<tr>
<td>spot</td>
<td>1</td>
<td>throw</td>
<td>1</td>
</tr>
<tr>
<td>throw</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Execution**

Input

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

Intermediate

<table>
<thead>
<tr>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1:v</td>
<td>k1:v</td>
<td>k2:v</td>
<td>k1:v</td>
<td>k3:v</td>
</tr>
<tr>
<td>k4:v</td>
<td>k5:v</td>
<td>k4:v</td>
<td>k5:v</td>
<td></td>
</tr>
</tbody>
</table>

Group by Key

Grouped


Output

| | | |

**Parallel Execution**

Map Task 1

Map Task 2

Map Task 3

Sort and Group

| k2:v | k4:v,v:v | k5:v |

Reduce Task 1

Reduce Task 2

Key: no implicit dependencies between map or reduce tasks
Model Is Widely Applicable
MapReduce Programs In Google Source Tree 2004

Example uses:
- distributed grep
- term-vector / host
- document clustering
- clustering for Google News
- distributed sort
- web access log stats
- machine learning
- popularity for Google Trends
- web link-graph reversal
- inverted index construction
- statistical machine translation for Google Translate

The Programming Model Is Key

- Simple control makes dependencies evident
  - Can automate scheduling of tasks and optimization
    - Map, reduce for different keys, embarrassingly parallel
    - Pipeline between mappers, reducers evident
  - map and reduce are pure functions
    - Can rerun them to get the same answer
      - In the case of failure, or
      - To use idle resources toward faster completion
    - No worry about data races, deadlocks, etc. since there is no shared state
Compare to Dedicated Supercomputers

According to Wikipedia, in 2009 Google uses
- 450,000 servers (2006), mostly commodity Intel boxes
- 2TB drive per server, at least
- 16GB memory per machine
- More recent details are kept secret by Google

More computing power than even the most powerful supercomputer