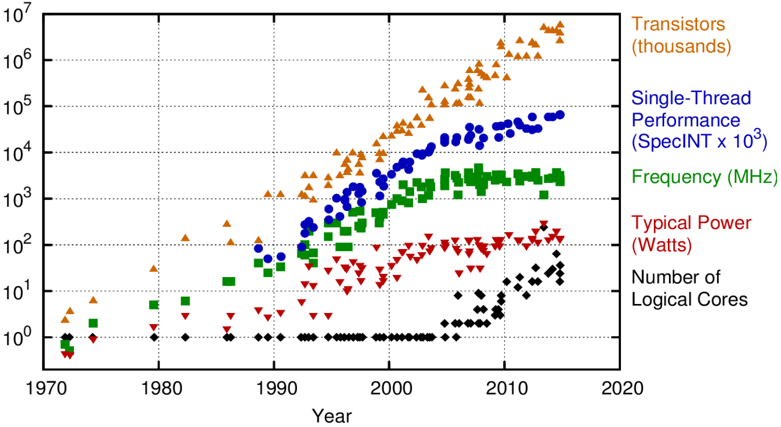
Programming in the Multi-Core Era

Emil Sekerinski

Department of Computing and Software
McMaster University

June 2018

40+ Years of Microprocessor Trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

We have reached instruction level parallelism wall and power wall

Still increasing number of transistors leads to more processor cores

Communication and Synchronization in Multi-Programs

Message Passing:

- Asynchronous: Unix pipes & filters, Erlang processes
- Synchronous: Go goroutines

Shared Variables:

- Semaphores: Linux, C, Python
- Monitors: Java, C#, Python
- Transactional memory: C++, Haskell
- Remote procedure call: Java, Python
- Distributed shared memory

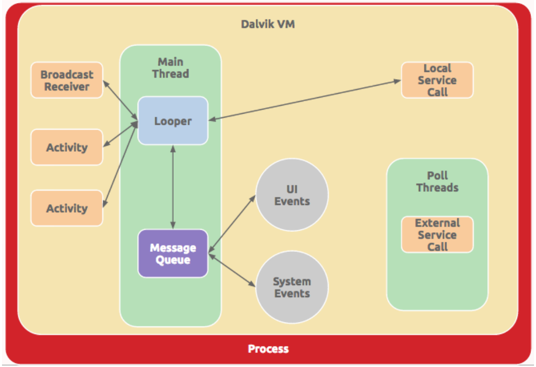
Introduced for resource sharing, interactive programs, distribution

Suitable for multi-core processors?

Threads in Programs

Threads have to be explicitly created to make use of multiple cores:

- Too few threads: not all cores will be used
  → turn independent tasks into threads

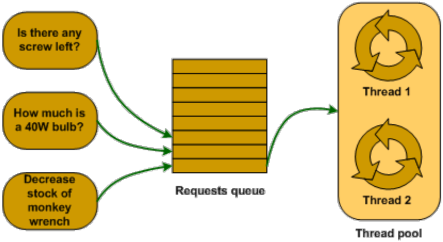


[Android Guides]

Threads in Programs

Threads have to be explicitly created to make use of multiple cores:

- Too few threads: not all cores will be used
  → turn independent tasks into threads
- Too many threads: slowdown due to overhead of cores switching between threads, e.g. multiplying $n \times n$ matrices
  → use thread pools to reduce number of threads



[Thread Pool in the .NET Framework]

Threads in Programs

Threads have to be explicitly created to make use of multiple cores:

- Too few threads: not all cores will be used
  → turn independent tasks into threads
- Too many threads: slowdown due to overhead of cores switching between threads, e.g. multiplying $n \times n$ matrices
  → use thread pools to reduce number of threads

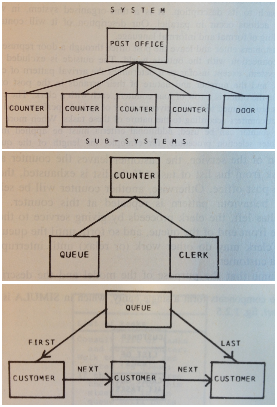
In addition to the static structure of modules, programs have a dynamic structure of threads.

(State is distributed over global variables and thread-local variables)

Inspiration: Simula-67 and Smalltalk-80

Simula-67

- objects in programs mimic objects of the "real world"
- objects are cooperatively scheduled
  → coroutines



[Birtwhistle, Dahl, Myhrhaug, Nygaard. Simula Begin, 1979.]

Inspiration: Simula-67 and Smalltalk-80

Simula-67

- objects in programs mimic objects of the “real world”
- objects are cooperatively scheduled
  → coroutines

Smalltalk-80	Java, C#, Python, . . .
sender	caller
receiver	callee
message	method

“Part of the problem description becomes part of the solution”

Inspiration: Simula-67 and Smalltalk-80

Simula-67

- objects in programs mimic objects of the "real world"
- objects are cooperatively scheduled
  → coroutines

Smalltalk-80	Java, C#, Python, . . .
sender	caller
receiver	callee
message	method

"Part of the problem description becomes part of the solution"

Objects should be thought of as having "independent lives", even if the implementation is sequential

Inspiration: Simula-67 and Smalltalk-80

Simula-67

- objects in programs mimic objects of the "real world"
- objects are cooperatively scheduled → coroutines

"Part of the problem description becomes part of the solution"

Smalltalk-80	Java, C#, Python, . . .
sender	caller
receiver	callee
message	method

Objects should be thought of as having "independent lives", even if the implementation is sequential

How to make objects "truly concurrent"?

Ada uses rendezvous, Erlang and Akka use actors

Guarded Commands: A Model for Concurrency

```
do
    $x_1 > x_2 \rightarrow$ swap $x_1$ and $x_2$
    $x_2 > x_3 \rightarrow$ swap $x_2$ and $x_4$
    $x_3 > x_4 \rightarrow$ swap $x_3$ and $x_4$
```

if $x_1 > x_2$ then swap x_1 and x_2,

. . .

if both $x_1 > x_2$ and $x_3 > x_4$, swap concurrently

Guarded Commands: A Model for Concurrency

```
**do**
    $x_1 > x_2 \to$ swap $x_1$ and $x_2$
    $x_2 > x_3 \to$ swap $x_2$ and $x_4$
    $x_3 > x_4 \to$ swap $x_3$ and $x_4$
```

Used for verification tools, TLA (Amazon), Event B (railways), SPIN (NASA), Microsoft Device Driver Verifier, . . . , for hardware description languages, but not programming languages:

Widespread belief in the 80's that guarded commands cannot be implemented efficiently

Lime: a Concurrent Object-Oriented Language

```
class DelayedDoubler
  var y: int; d: boolean
  init()
    d := true
  method store(u: int)
    y := u; d := false
  method retrieve(): int
    d → return y
  action double
     not d → y := 2 * y; d := true
```

- Actions execute when enabled
- All objects are concurrent
- Objects synchronize through method calls
- Method blocks when guard is false
- Execution is atomic up to method calls
- Guards only over local fields

Lime: a Concurrent Object-Oriented Language

```
class DelayedDoubler
  var y: int; d: boolean
  init()
    d := true
  method store(u: int)
    y := u; d := false
  method retrieve(): int
    d → return y
  action double
     not d → y := 2 * y; d := true
```

A correct implementation of:

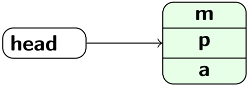
```
class Doubler
  var x: int
  method store(u: int)
    x := 2 * u
  method retrieve(): int
    return x
```

Representative for writing to a file, sending over network, . . .

Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ . . .

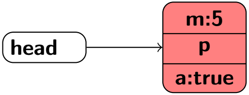
```
class PriorityQueue
 var l : PriorityQueue
 . . .
 method add(e: int)
   when not a and not r do
     if l = nil then
        m := e ; l := new PriorityQueue
     else p := e ; a := true
 action doAdd
   when a do
     if m < p then l.add(p)
     else l.add(m); m := p
     a := false
```



Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ . . .

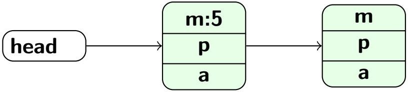
```
class PriorityQueue
  var l : PriorityQueue
  . . .
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```



Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ ...

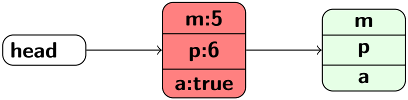
```
class PriorityQueue
  var l : PriorityQueue
  ...
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```



Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ ...

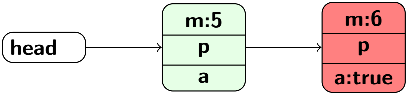
```
class PriorityQueue
  var l : PriorityQueue
  ...
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```



Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ . . .

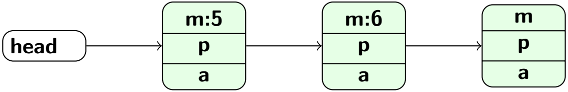
```
class PriorityQueue
  var l : PriorityQueue
  . . .
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```



Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ . . .

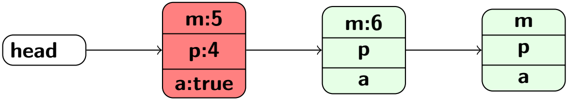
```
class PriorityQueue
  var l : PriorityQueue
  . . .
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```



Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ . . .

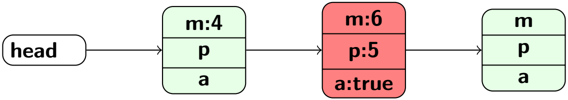
```
class PriorityQueue
  var l : PriorityQueue
  . . .
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```



Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ ...

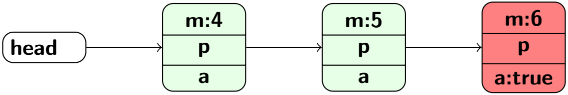
```
class PriorityQueue
  var l : PriorityQueue
  ...
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```



Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ . . .

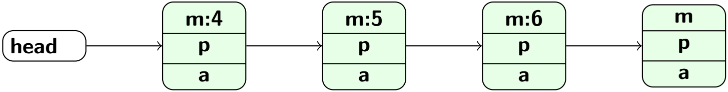
```
class PriorityQueue
  var l : PriorityQueue
  ...
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```



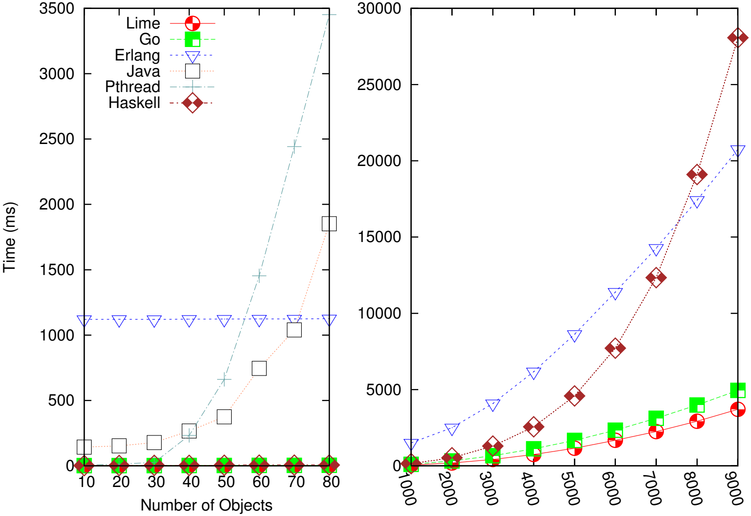
Priority Queue

- Add an integer to the queue
- Remove the smallest integer
- Remove executes in $O(1)$ time
- Initialize; add $5, 6, 4$ ...

```
class PriorityQueue
  var l : PriorityQueue
  ...
  method add(e: int)
    when not a and not r do
      if l = nil then
         m := e ; l := new PriorityQueue
      else p := e ; a := true
  action doAdd
    when a do
      if m < p then l.add(p)
      else l.add(m); m := p
      a := false
```

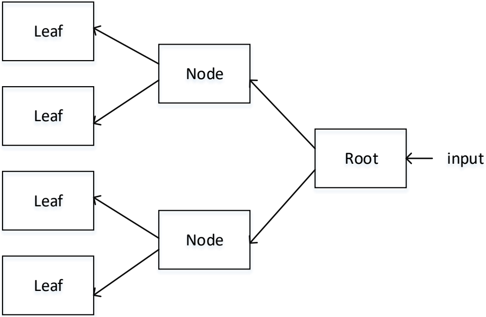


Results of Priority Queue

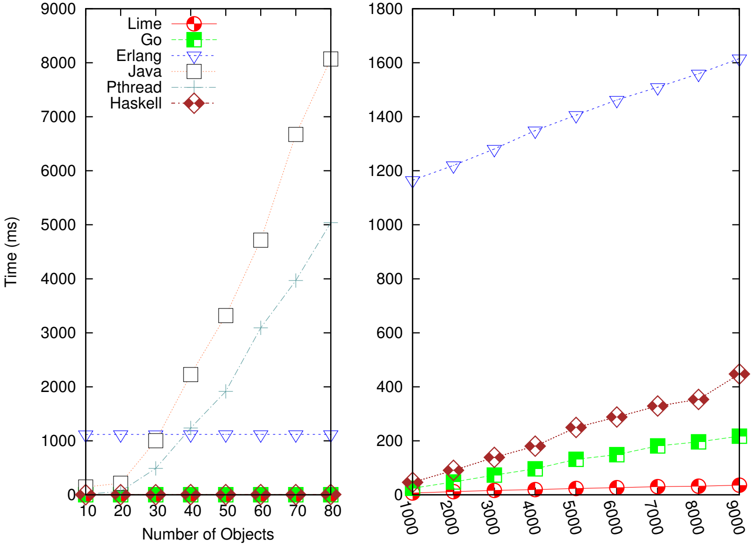


Leaf-oriented Tree

- Internal nodes contain only guides
- The elements are stored in the leaves

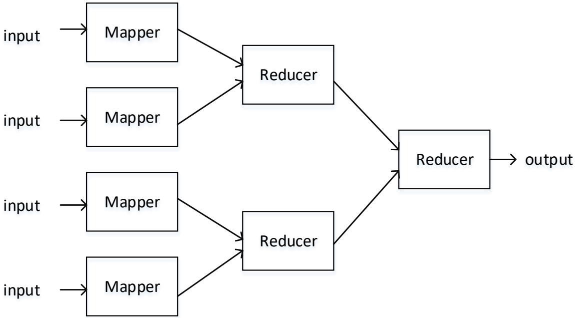


Results of Leaf-oriented Tree

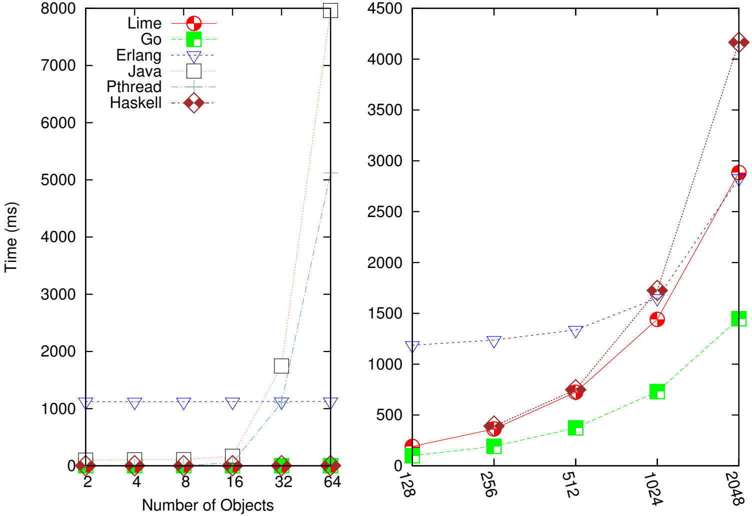


MapReduce

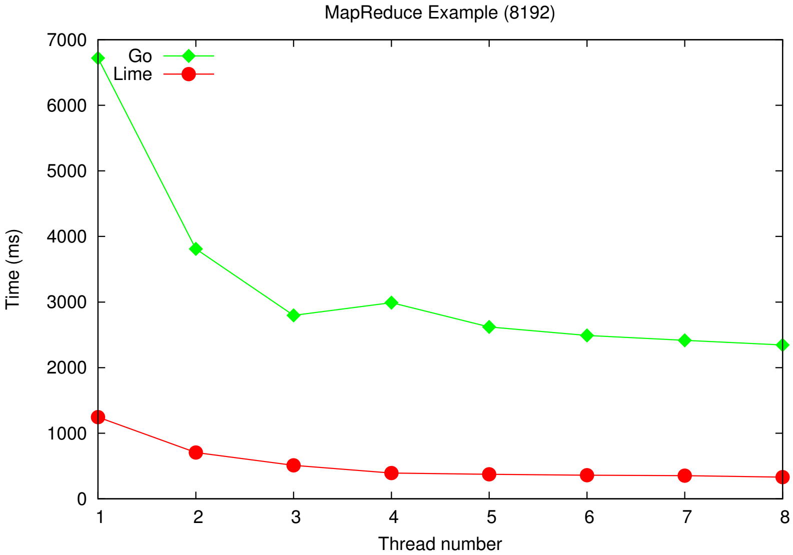
- MapReduce computes the sum of squares from 1 to n
- The mapper: $map(x) = x^2$
- The reducer: $reduce(x, y) = x + y$.



Results of MapReduce



MapReduce with Varying Number of Threads



Santa Claus Problem

Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on a tropical island, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don't want to leave the tropics, and therefore they stay there until the last possible moment.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh. [Trono, 1994]

Includes priority, multi-party synchronization, barriers, and batch processing.

A number of flawed solutions have been published.

Results of Santa Claus

Repetitions of Santa	Lime (guards)	C (semaphores)	Go (channels)	Java (monitors)
10,000	0.03 / 0.03 / 0.00	0.87 / 0.26 / 1.18	0.08 / 0.12 / 0.01	6.38 / 2.48 / 5.30
100,000	0.21 / 0.21 / 0.00	8.82 / 2.50 / 12.0	0.77 / 1.18 / 0.06	60.3 / 21.6 / 52.0
1,000,000	2.03 / 2.03 / 0.01	93.0 / 24.8 / 123	7.51 / 11.6 / 0.55	≈534 / 159 / 509

Execution time in sec on AMD 16 core (32 threads) processor: real / user / system

GO and Lime

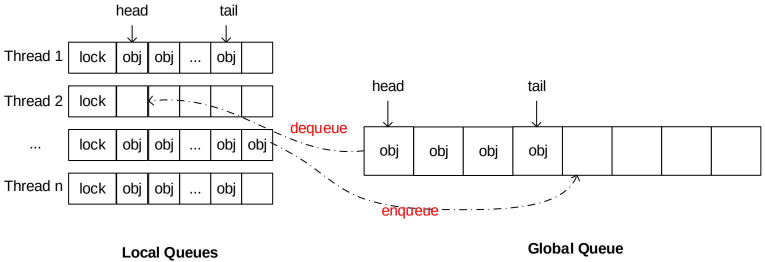
- GO threads (goroutines) use synchronous communication (CSP)
- Object-oriented structure deemphasized, thread structure emphasized

GO and Lime make use of goroutines resp. actions so cheap that concurrency can be used whenever natural;

The runtime will make use of all available cores.

How is that achieved?

Lime Runtime System



- Number of worker threads with local queue ≈ number of cores
- Local queue is full / empty: enqueue / dequeue to / from global queue
- Local and global queues are empty: steal from the other threads.
- Lock-free implementation of queues

Each action is a coroutine: compiler inserts transfers in code, i.e. schedules cooperatively → lightweight threads

Conclusions

Efficient implementation of concurrency with guarded commands is possible, when local to objects

Based on the Ph.D. thesis of Shucai Yao (2018) and

- Joshua Moore-Oliva, M.Sc, (2010)
- Xiao-lei Cui, M.Sc. (2009)
- Upasana Pujari, M.Sc. (2009)
- Kevin Lou, M.Sc. (2004)
- Jie Liang, M.Sc. (2004)

Ongoing work on inheritance, ownership, exceptions

http://www.software-pioneers.com (2001)