CMSC 330: Organization of Programming Languages

Threads, con't.

Parallelizable Applications of Interest

- Knowledge discovery: mine and analyze massive amounts of distributed data
  – Discovering social networks
  – Real-time, highly-accurate common operating picture, on small, power-constrained devices
- Simulations (games?)
- Data processing
  – NLP, vision, rendering, in real-time
- Commodity applications
  – Parallel testing, compilation, typesetting, …

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

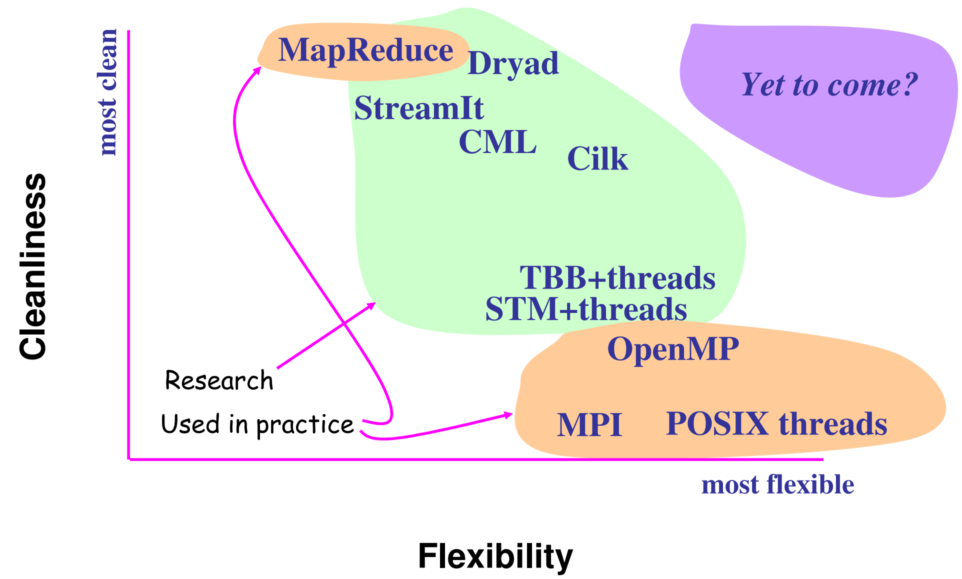
Parallel Language Extensions

- 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



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

Unary operator

- (map ***f list***)
- (map square '(1 2 3 4))
  (1 4 9 16)

Binary operator

- (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")

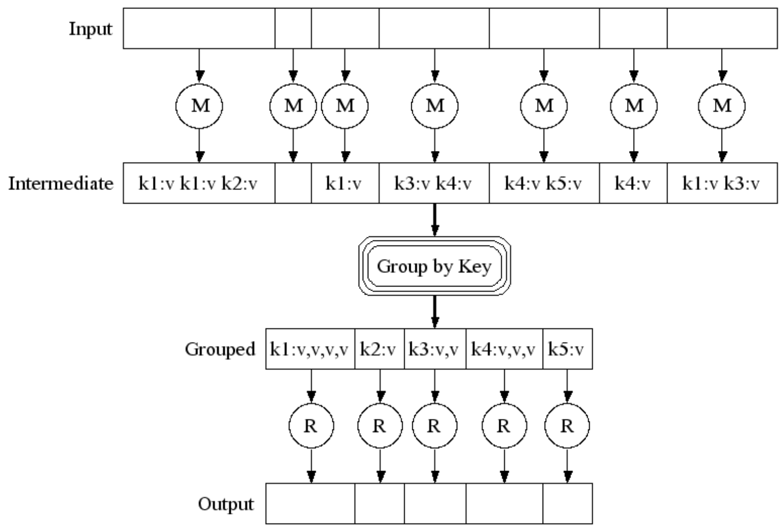
reduce(key=word, values=uniq_counts):
Sum all "1"s in values list
Emit result "(word, sum)"

see bob throw
see spot run

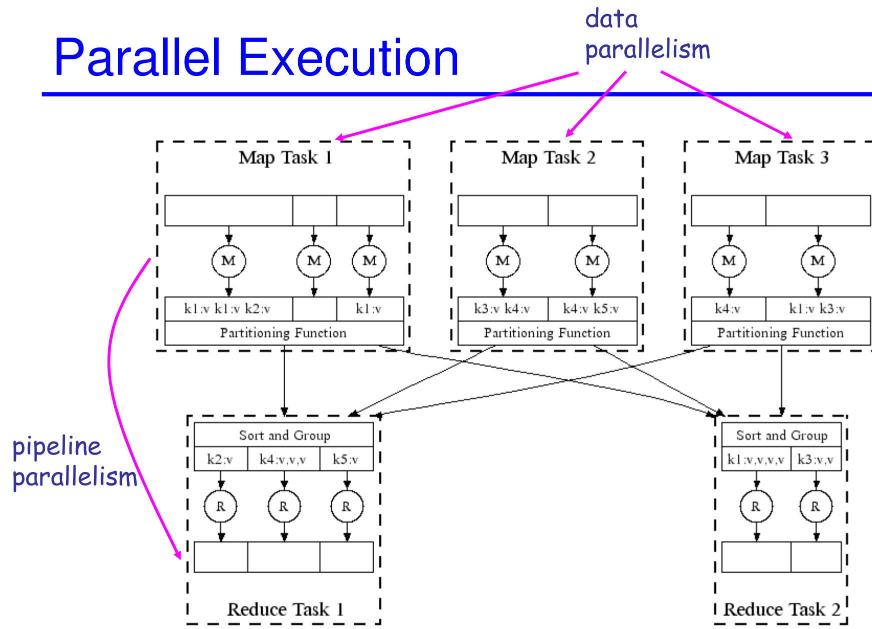
see	1
bob	1
run	1
see	1
spot	1
throw	1

bob	1
run	1
see	2
spot	1
throw	1

Execution

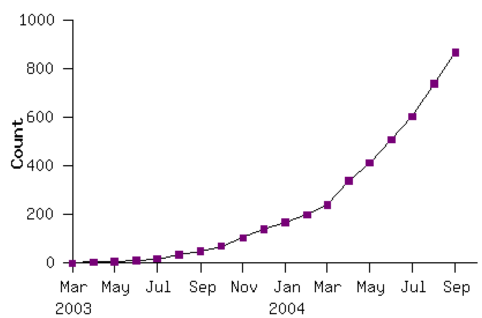


Parallel Execution



Key: no implicit dependencies between map or reduce tasks

Model is Widely Applicable
MapReduce Programs In Google Source Tree 2004



Example uses:

distributed grep	distributed sort	web link-graph reversal
term-vector / host	web access log stats	inverted index construction
document clustering	machine learning	statistical machine translation
...	...	...

The Programming Model Is Key

- Simple control makes dependencies evident
  - Can automate scheduling of tasks and optimization
    - Map, reduce for different keys, embarassingly 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 race conditions, deadlocks, etc. since there is no shared state

Compare to Dedicated Supercomputers

- According to Wikipedia, in 2006 Google uses
  - 450,000 servers from 533 MHz Celeron to dual 1.4GHz Pentium III
  - 80GB drive per server, at least
  - 2-4GB memory per machine
  - Jobs processing 100 terabytes of distributed data
- More computing power than even the most powerful supercomputer