Parallel Languages and Platforms

Chris Kauffman

CS 499: Spring 2016 GMU

Logistics

Schedule

4/19	Tue	PThreads
	Thu	Java Threads
4/26	Tue	Parallel Languages Mini-Exam 4
	Thu	Parallel Platforms
5/3	Tue	Review HW 4 Due
5/5	Thu	Final Exam 12:00-2:00pm

Today

- Mini-exam 4 back
- Parallel Languages and Platforms

An Upcoming Event: Thursday 4/28 9-11am, DC

High-performance computing (HPC)—which leverages "supercomputers" and massively parallel processing techniques to solve complex computational problems through computer modeling, simulation, and data analysis techniques—represents a strategic, game-changing technology with tremendous economic competitiveness, science leadership, and national security implications for the United States. But as competitor countries dramatically increase their investments in HPC systems and technologies, future U.S. leadership in HPC is no longer assured. Join ITIF as it releases an important new report explaining why HPC is increasingly vital to America's economic competitiveness and details the intensifying global competition for HPC systems leadership, in particular the race to exascale computing. The event is free and open to the public

– *The Vital Importance of High-Performance Computing to U.S. Competitiveness and National Security,*

HW 4: Password Cracker

- Parallelize breaking single passwords (single mode)
- Simple approach: try multiple words from given dictionaries
- Do so with
  - OpenMP: `omp_passcrack.c`
  - PThreads: `pthread_passcrack.c`
- Don't touch original serial code
- Add code in `parallel_funcs.c`
- Modify `main()` funcs in appropriate C files
- Tricks: Unroll recursion of `try_crack()` function one layer
- Pay attention to "canceling" other threads when a solution is found
- Use timing script to get a sense of timing

Menagerie of Parallel Languages and Platforms

Distributed Memory Only

Erlang, Map+Reduce / Hadoop, Job Schedulers

Shared Memory Only

Cilk, Clojure

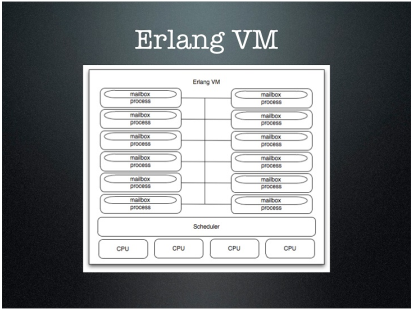
Distributed + Shared

Unified Parallel C, Chapel (later)

Device Concurrency / GPUs

CUDA / OpenCL (later)

Erlang



Source

- Developed for distributed computation, telephony systems
- Virtual machine which mirrors many OS functions
- Process `spawn` to create *lightweight* procs
- `send/receive` clauses to share information among processes
- Facilities to contact a remote Erlang VM and talk to its processes

Erlang Sample straight from Wikipedia

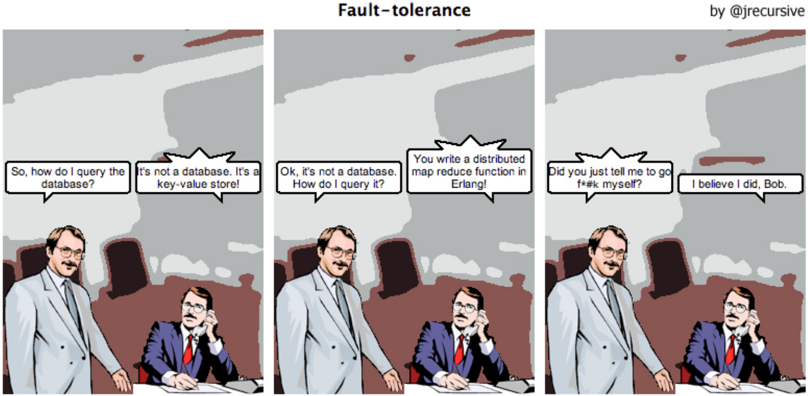
```
% Create a process on this machine and invoke the function
%   web:start_server(Port,MaxConnections)

ServerProcess = spawn(web, start_server, [Port, MaxConnections]),

% Create a remote process and invoke the function
%   web:start_server(Port, MaxConnections)
% on machine RemoteNode

RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnections]),

% Send a message to ServerProcess (asynchronously). The message
% consists of a tuple with the atom "pause" and the number "10".

ServerProcess ! {pause, 10},

% Receive messages sent to this process
receive
    a_message -> do_something;
    {data, DataContent} -> handle(DataContent);
    {hello, Text} -> io:format("Got hello message: ~s", [Text]);
    {goodbye, Text} -> io:format("Got goodbye message: ~s", [Text])
end.
```

Erlang's Nature and Target

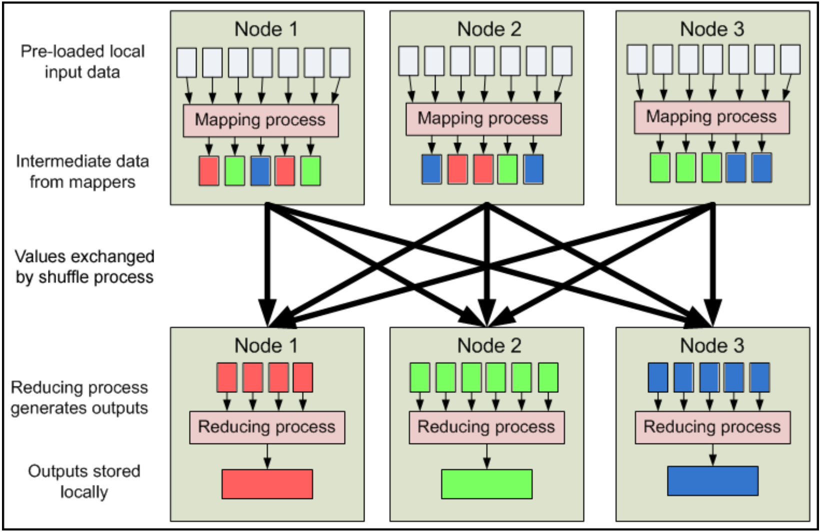
- Syntax and semantics are somewhat odd/archaic but can be "gotten used to"
- Targeted at client server architectures, computation distributed across many nodes
- Well known for robustness of the VM, fault-tolerance features to keep application going if participating nodes go down
- Not targeted at high-performance computation / scientific problems, more towards business, IT, web services

MapReduce (or more properly Map, Shuffle, Reduce)

- A style of programming, inspired by functional programming
  `(def doub-sum (reduce + 0 (map double '(1 2 3 4 5))))`
- Targeted at big data: large distributed stores of data
  - Map: Transform / filter data in some way
  - Shuffle: Move data with same properties to same node
  - Reduce: Combine results on individual nodes



Basic Architecture of MapReduce



Source: Yahoo Developer Tutorial on MapReduce

Shameless Wikipedia Example: Document Word Counts

Pseudocode

```
function map(String name,
             String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word,
                Iterator partialCounts):
  // word: a word to count
  // partialCounts: list of
                    partial counts
  sum = 0
  for each pc in partialCounts:
    sum += pc
  emit (word, sum)
```

- Goal: produce frequency of each word in a document
- Nodes are each fed the document
- During `reduce()` *emit* pairs like `("apple",1)` and `("Dell",1)`
- System automatically sends pairs with key apple to the same nodes (redistribute)
- Nodes run `reduce()` to count apple occurrences, may redistribute further

Variety of Languages for MapReduce Framework

```
public static class MapClass extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable>
{
  private final static
    IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void
  map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException
  {
    String line = value.toString();
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      output.collect(word, one);
    }
  }
}
public static class Reduce extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable>
{
  public void
  reduce(Text key, Iterator<IntWritable> values,
         OutputCollector<Text, IntWritable> output,
         Reporter reporter) throws IOException
  {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

Problem: Word Frequencies in a Document

← Java (Source)

Pig Latin ↓ (Source)

```
A = load './input.txt';
B = foreach A generate
      flatten(TOKENIZE((chararray)$0))
      as word;
C = group B by word;
D = foreach C generate COUNT(B), group;
store D into './wordcount';
```

MapReduce Framework Notes

- Primary contribution of implementations is distributing load across many machines efficiently
- When machines both store some data and participate in MapReduce, gain locality for speed
- Alternative to large database processing, may open up opportunities for parallelism to avoid read/write locks in traditional DBs
- Most frequently referenced implementation is Apache Hadoop

  But other implementations exist, some proprietary
- All implementation implement a MapReduce server/scheduler to which jobs are submitted

```
> javac MRWordCount.java
> java MRWordCount &
> bin/hadoop job -list
1 jobs currently running
JobId    State          StartTime  UserName
job_0001     1      1218506470390  kauffman
```

Job Schedulers

- Has long been the need for many parallel jobs to be run on individual systems/clusters
- Job schedulers offer frameworks for this: submit many programs to run, scheduler assigns resources
- `slurm`: scheduler on `medusa` cluster which you used to schedule jobs running
- Generic form of easy concurrency for variety of different programs, resources etc.

Cilk and CilkPlus (supported in gcc/g++)

- Tutorial Here
- Extensions to C/C++ which enable easy spawning of threads (*strands*) to run functions concurrently
- Primary Additions are keywords to easily spawn functions

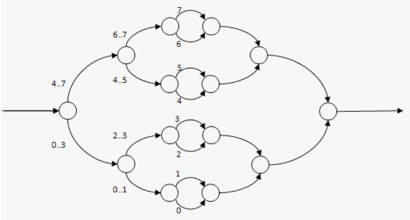
```
// Run func concurrently - separate thread
int x = cilk_spawn func(n);
...

// Wait for all running functions to finish
cilk_sync;



// Compile with cilk features enabled
> gcc -fcilkplus cilk-fib.c
```

- Examine: `cilk-fib.c`
- Contrast with PThread startup

cilk_for loop, Reducers

- Similar to OpenMP, Cilk provides loop parallelization and reduction
- See cilk-for-picalc.cpp

```
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>

// C++ class for reductions
cilk::reducer_opadd<int> total_hits;

// Appropriate for parallel contexts
total_hits++;

// Retrive final value
int th = total_hits.get_value();

// Automatic loop parallelization
cilk_for (int i = 0; i < npoints; i++) {
   ...;
}
```



Array-based Operations

- Cilk provides some convenient syntax for array operations
- Vectorized operations automatically created
- Compile can sometimes do this but special notation helps hint

```
// standard element-wise vector multiply
void axpy(int n, double alpha, const double *x, double *y)
{
  for (int i = 0; i < n; i++) {
    y[i] += alpha * x[i];
  }
}

// Cilk Plus abbreviated syntax
void axpy_cilk(int n, double alpha, const double *x, double *y)
{
  y[0:n] += alpha * x[0:n];
}
```

- Matlab/Octave are well known for this style
- See `cilk-array-syntax.c` for uses

Clojure and Software Transactional Memory

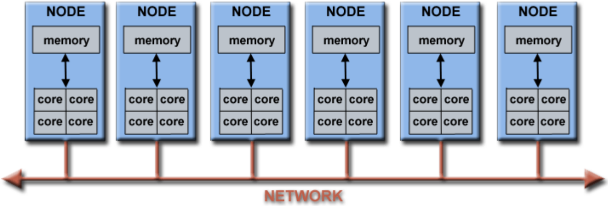
- A lisp which runs on the Java Virtual Machine
- Design Goal: allow for shared memory parallelism to be exploited
- Realization:
  - Each data element has well-defined local/shared semantics
  - Data is immutable by default
  - Provides `atom` data types for atomic alterations
  - Other alterations to shared resources occur in `dosync` blocks
  - Software Transactional Memory (STM) system: try changing a shared area, if it changes, try again with the new current value
- Runs as an executable JAR

```
> java -jar clojure-1.8.0.jar
Clojure 1.8.0
user=> (def x (atom 0))
#'user/x
user=> x
#object[clojure.lang.Atom 0x6c372fe6 {:status :ready, :val 0}]
user=> @x
0
```

Picalc in Clojure

```
;; Serial version with atomic updates
(defn calc-pi-atoms [iterations]
  (let [hits (atom 0)]
    (dotimes [i iterations]
      (let [x (rand) y (rand)]
        (if (<= (+ (* x x) (* y y)) 1)
          (swap! hits inc))))
    (double (* (/ @hits iterations) 4))))

;; Parallel version with atomic updates
(defn calc-pi-atoms [iterations nthreads]
  (let [hits (atom 0)]
    (doall (pmap ;; parallel map, force evaluation
      (fn [x]
        (dotimes [i (/ iterations nthreads)]
          (let [x (rand) y (rand)]
            (if (<= (+ (* x x) (* y y)) 1)
              (swap! hits inc)))))
      (range nthreads))) ;; map onto number of threads
    (double (* (/ @hits iterations) 4))))
```

Common HPC Parallel Platform



- Similar config to `medusa` cluster we used
- Cluster of machines, each with multiple cores
- Options to program:
  - Serial execution on each core/machine
  - Parallel shared memory execution on each machine
  - Parallel distributed memory execution on each core
  - Mixed: Distributed/Shared parallel execution

Mixing MPI and OpenMP

Unified Parallel C (requires special compiler)

- Extensions to the C language
- Aimed at BOTH shared memory and distributed memory
- Automatic `THREADS` and id `MYTHREAD` variables
- Thread is more generalized: might be same machine (shared) or different machine (distributed)
- Shared memory blocks

  `shared int all_hits[THREADS];`

  Access like `int x = all_hits[1];` will work locally (shared) or via MPI-style message passing if remote (distributed)
- Compiler/runtime automatically sets up sharing
- Standard locks
- Automatic loop parallelization via `upc_forall(..)` with some *affinity* control: which thread executes which iteration
- Control over layout of shared blocks of memory: which thread gets what section
- Examine `upc-shared-picalc.c`