A Bit More Parallelism

COS 326
David Walker
Princeton University

Last Time: Parallel Collections

The parallel sequence abstraction is powerful:

- tabulate
- nth
- length
- map
- split
- treeview
- scan
  - used to implement prefix-sum
  - clever 2-phase implementation
  - used to implement filters
- sorting

PARALLEL COLLECTIONS IN THE "REAL WORLD"

Big Data

If Google wants to index all the web pages (or images or gmails or google docs or ...) in the world, they have a lot of work to do

- Same with Facebook for all the facebook pages/entries
- Same with Twitter
- Same with Amazon
- Same with ...

Many of these tasks come down to map, filter, fold, reduce, scan hadoop

Scala

Parallel Collections with Scala

Jul 6' 2012 > Vikas Hazrati > vikas@knoldus.com > @vhazrati

knól

The Bloom Programming Language

MapReduce

DRYAD

LINQ

Microsoft® .NET

FLUME

Google Map-Reduce

Google MapReduce (2004): a fault tolerant, massively parallel functional programming paradigm

- based on our friends "map" and "reduce"
- Hadoop is the open-source variant
- Database people complain that they have been doing it for a while
  - ... but it was hard to define

Fun stats circa 2012:

- Big clusters are ~4000 nodes
- Facebook had 100 PB in Hadoop
- TritonSort (UCSD) sorts 900GB/minute on a 52-node, 800-disk hadoop cluster

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

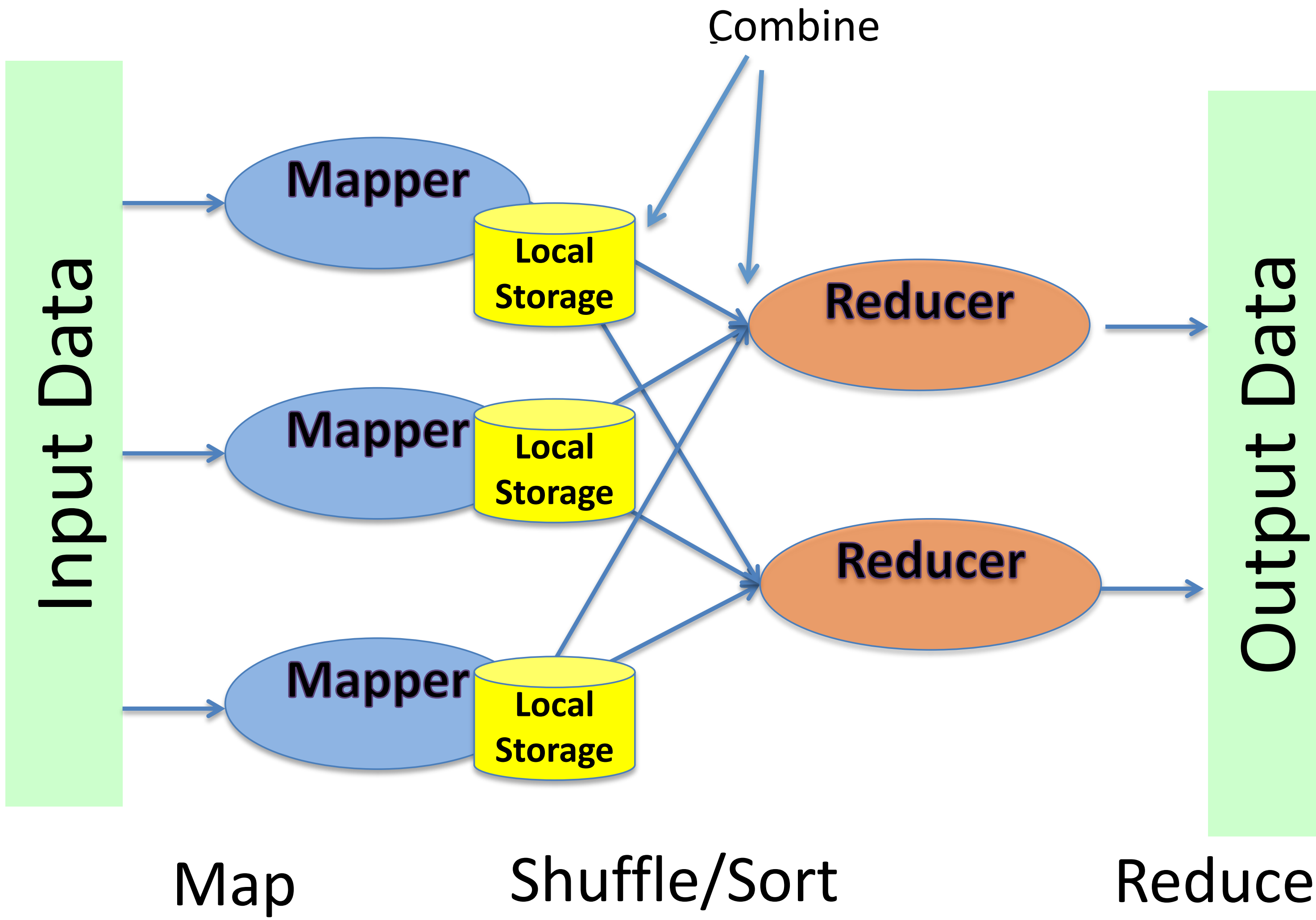
Data Model & Operations

- Map-reduce operates over collections of key-value pairs
  - millions of files (eg: web pages) drawn from the file system and parsed in parallel by many machines
- The map-reduce engine is parameterized by 3 functions, which roughly speaking do this:

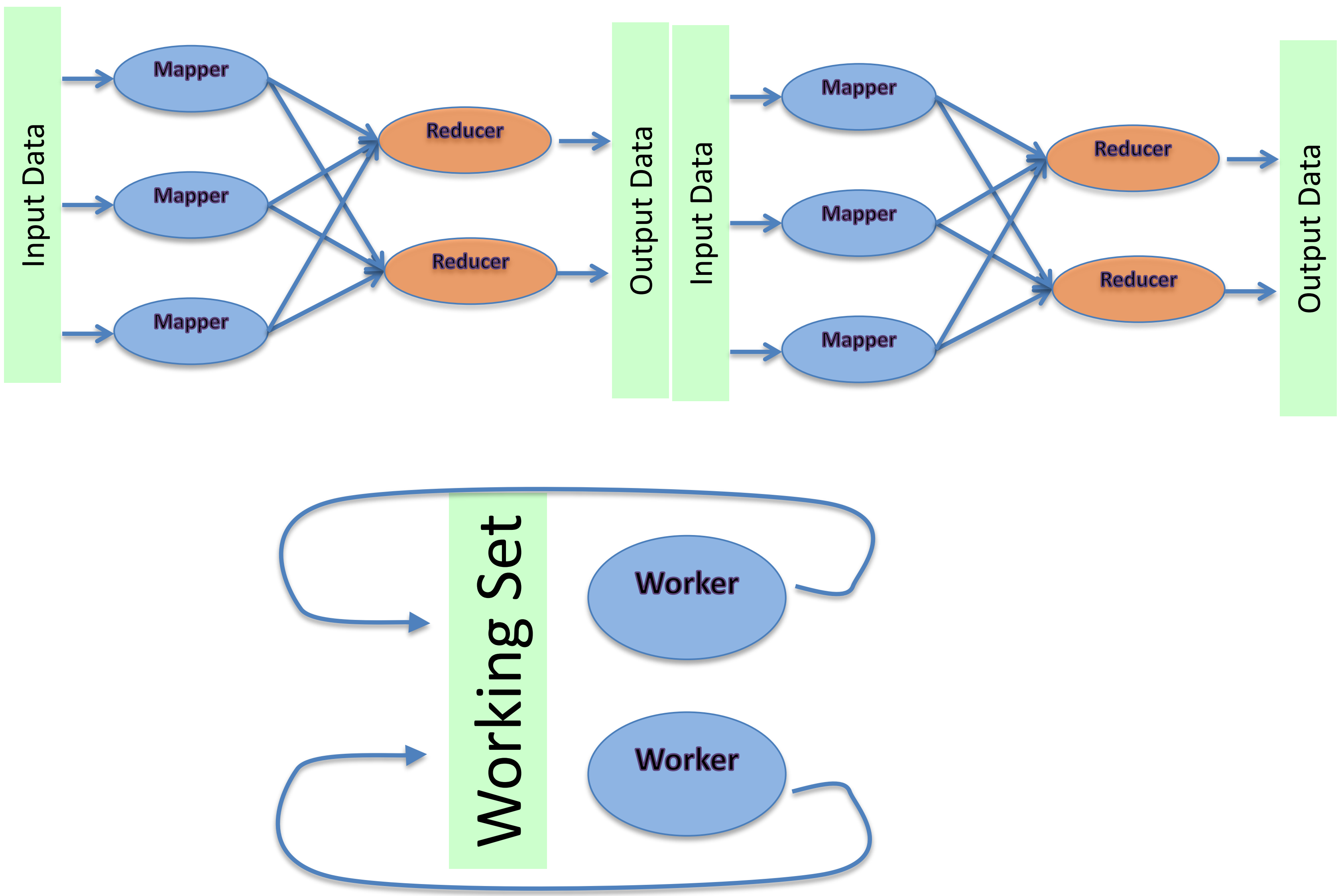
```
map     : key1 * value1          -> (key2 * value2) list
combine : key2 * (value2 list) -> value2 option
reduce  : key2 * (value2 list) -> key3 * (value3 list)
```

optional – often used to compress data before transfer from a mapper machine to a reducer machine

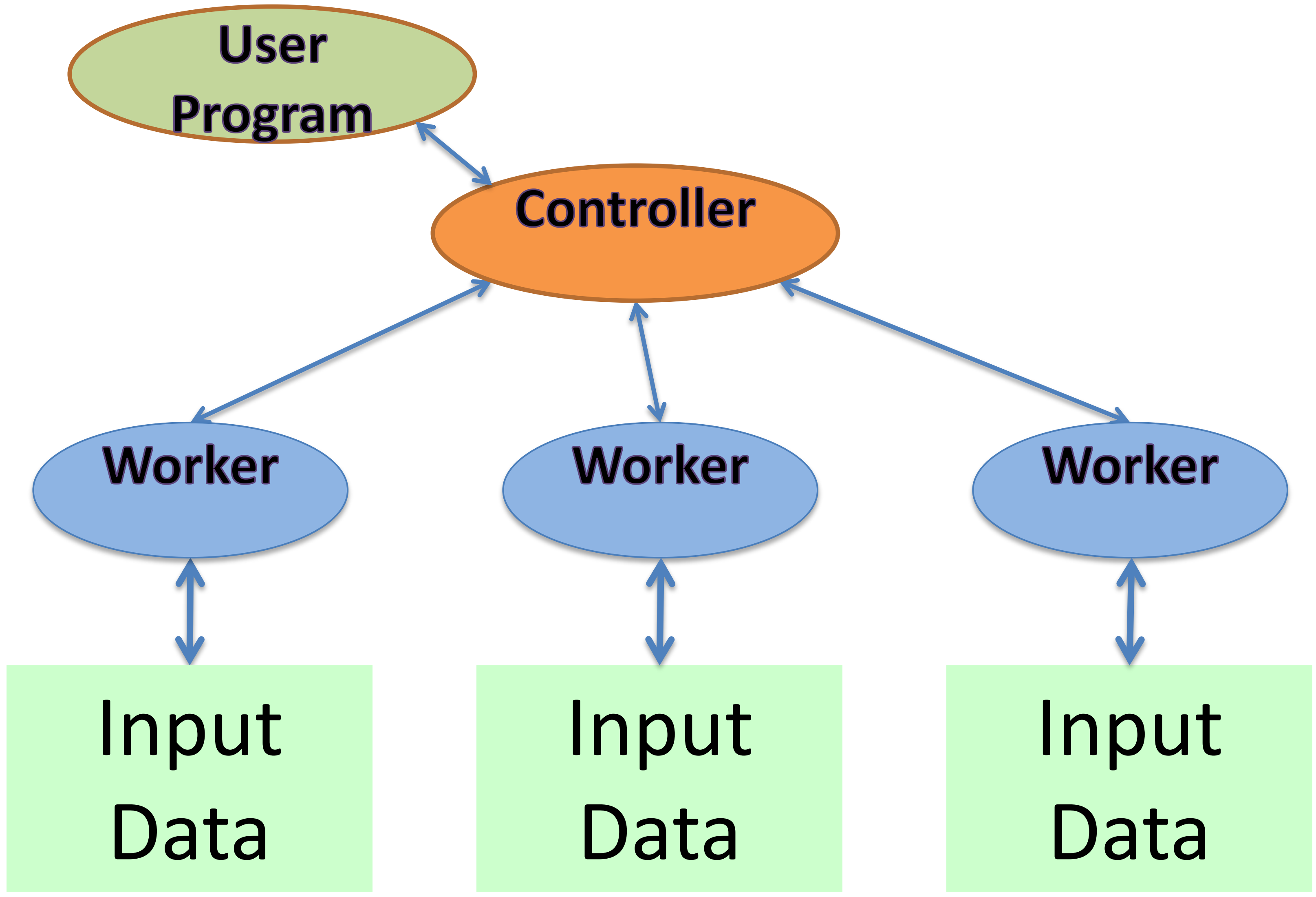
Architecture



Iterative Jobs are Common



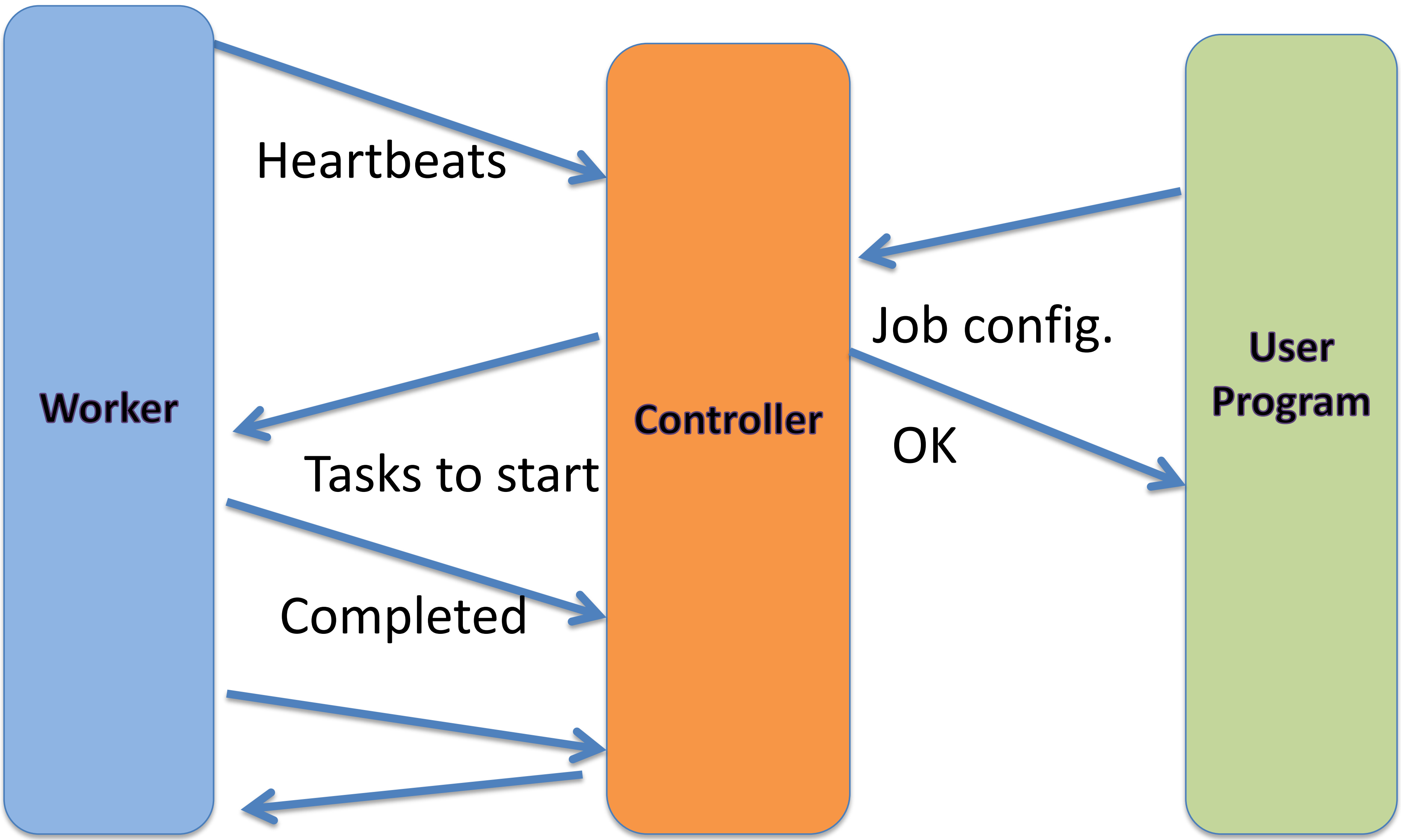
The Control Plane



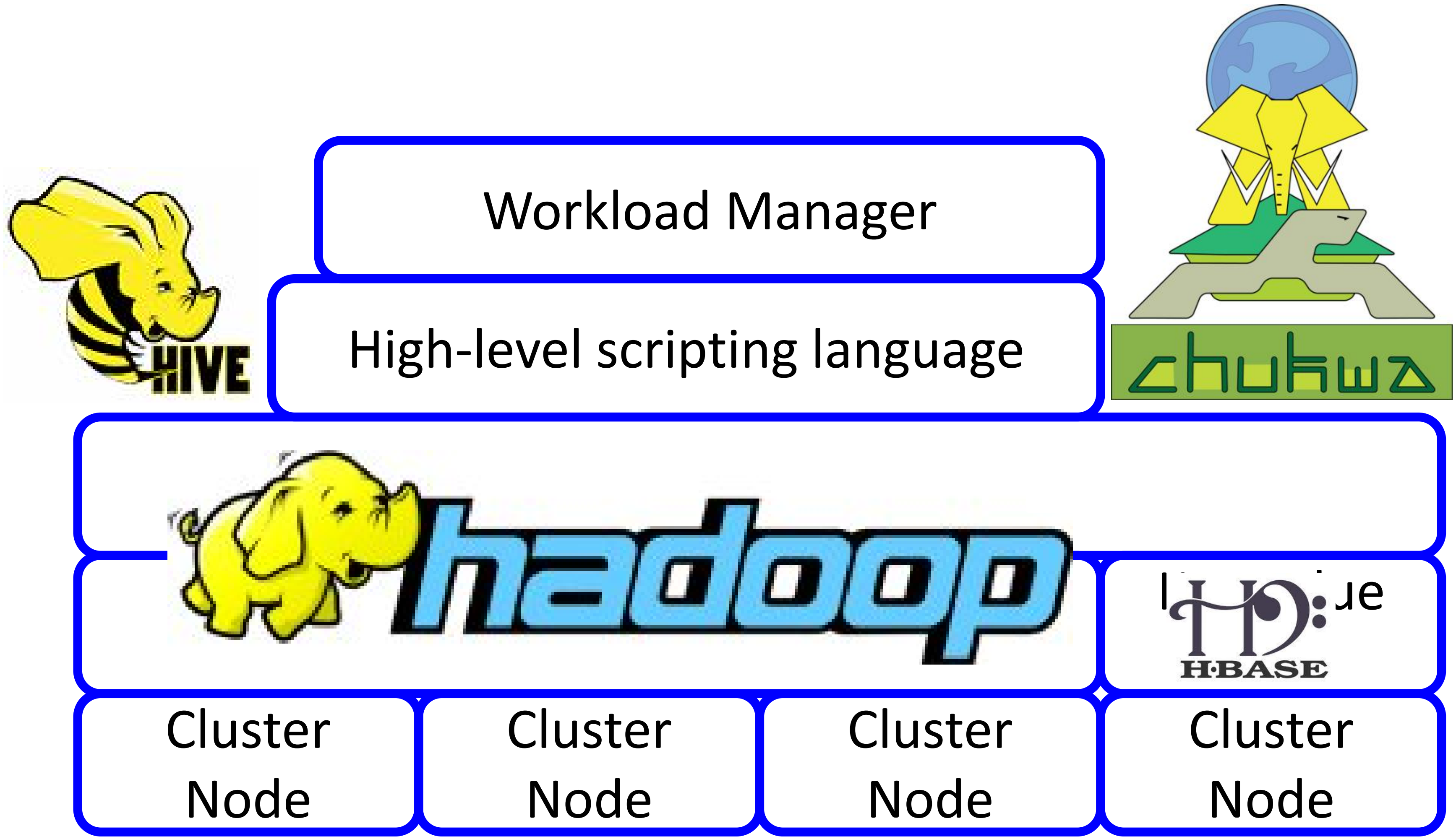
Jobs, Tasks and Attempts

- A single *job* is split in to many *tasks*
- Each *task* may include many calls to map and reduce
- *Workers* are long-running processes that are assigned many tasks
- Multiple workers may *attempt* the same task
  - each invocation of the same task is called an attempt
  - the first worker to finish "wins"
- Why have multiple machines attempt the same task?
  - machines will fail
    - approximately speaking: 5% of high-end disks fail/year
    - if you have 1000 machines: 1 failure per week
    - *repeated failures become the common case*
  - machines can partially fail or be slow for some reason
    - reducers can't start until *all* mappers complete

Flow of Information



A Modern Software Stack



Sort-of Functional Programming in Java

Hadoop interfaces:

```
interface Mapper<K1,V1,K2,V2> {
  public void map (K1 key,
                   V1 value,
                   OutputCollector<K2,V2> output)
   ...
}
```

```
interface Reducer<K2,V2,K3,V3> {
  public void reduce (K2 key,
                      Iterator<V2> values,
                      OutputCollector<K3,V3> output)
   ...
}
```

Word Count in Java

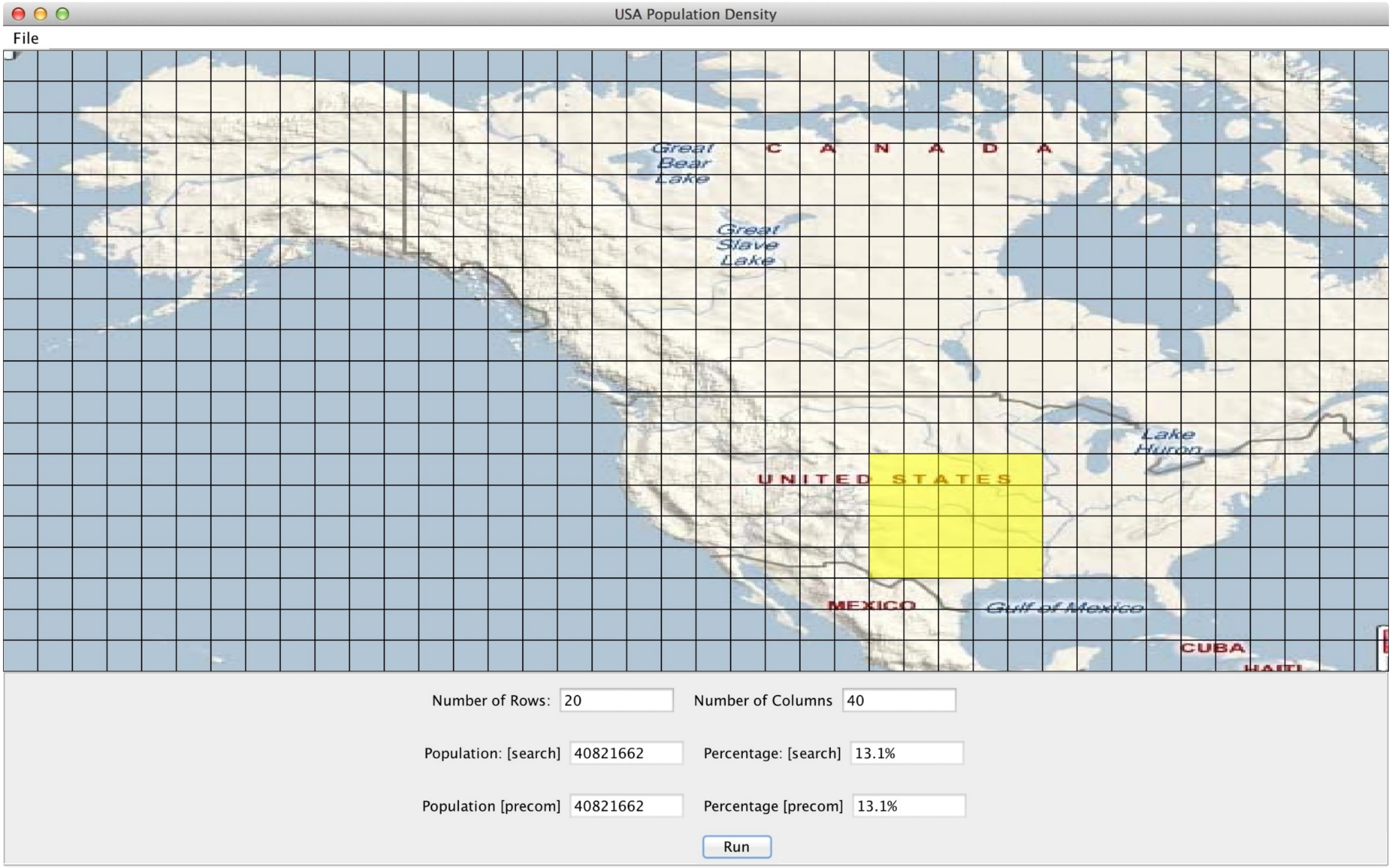
```
class WordCountMap implements Map {
  public void map(DocID key
                  List<String> values,
                  OutputCollector<String,Integer> output)
  {
     for (String s : values)
       output.collect(s,1);
  }
}
```

```
class WordCountReduce {
  public void reduce(String key,
                     Iterator<Integer> values,
                     OutputCollector<String,Integer> output)
  {
    int count = 0;
    for (int v : values)
      count += 1;
    output.collect(key, count)
  }
```

PLEASE RELAX
AND FOR THE SAKE OF HYGIENE,
WIPE THE
JAVA CODE OFF YOUR BRAIN

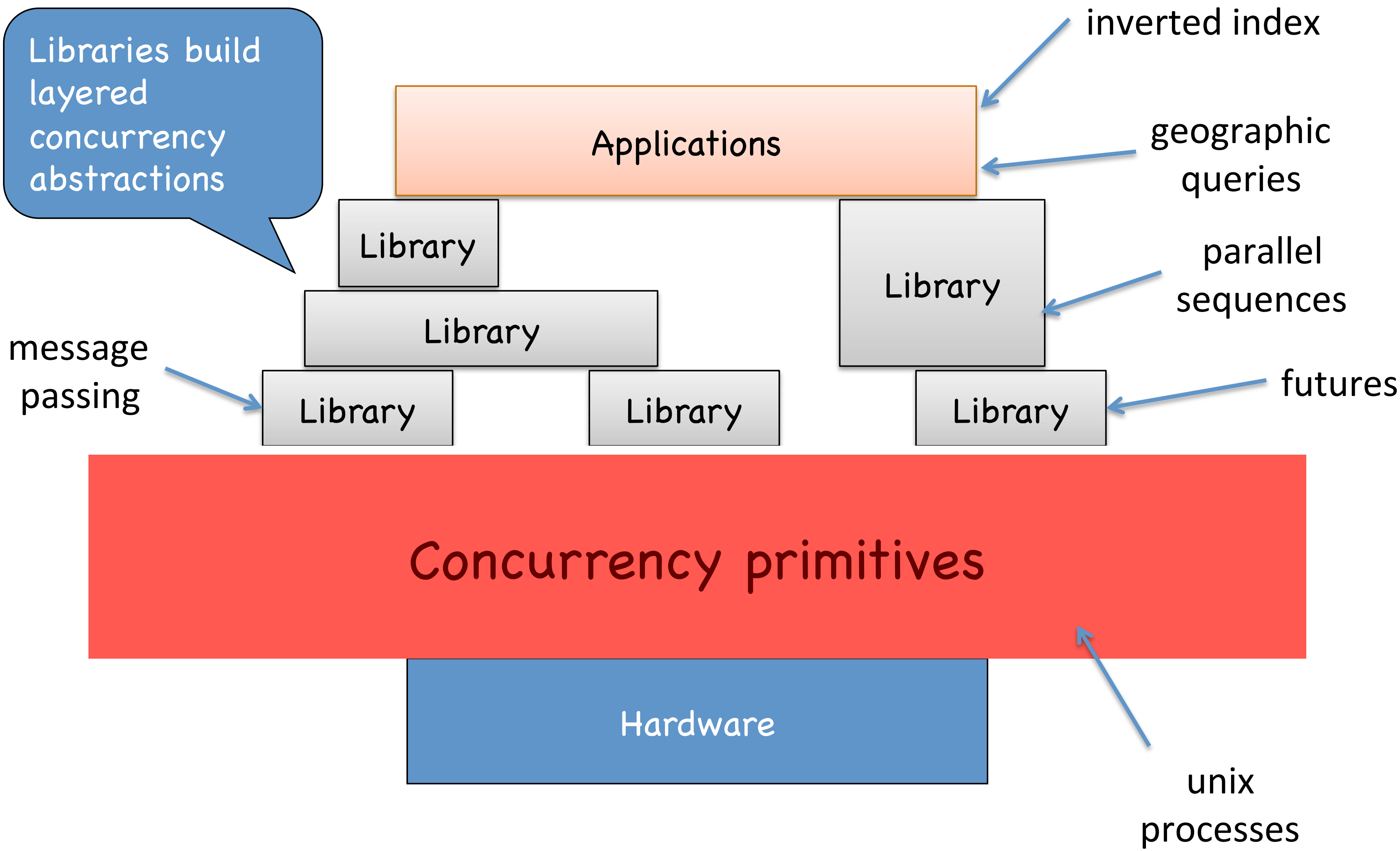
ASSIGNMENT #7: IMPLEMENTING AND USING PARALLEL COLLECTIONS

US Census Queries



End goal: develop a system for efficiently computing US population queries by geographic region

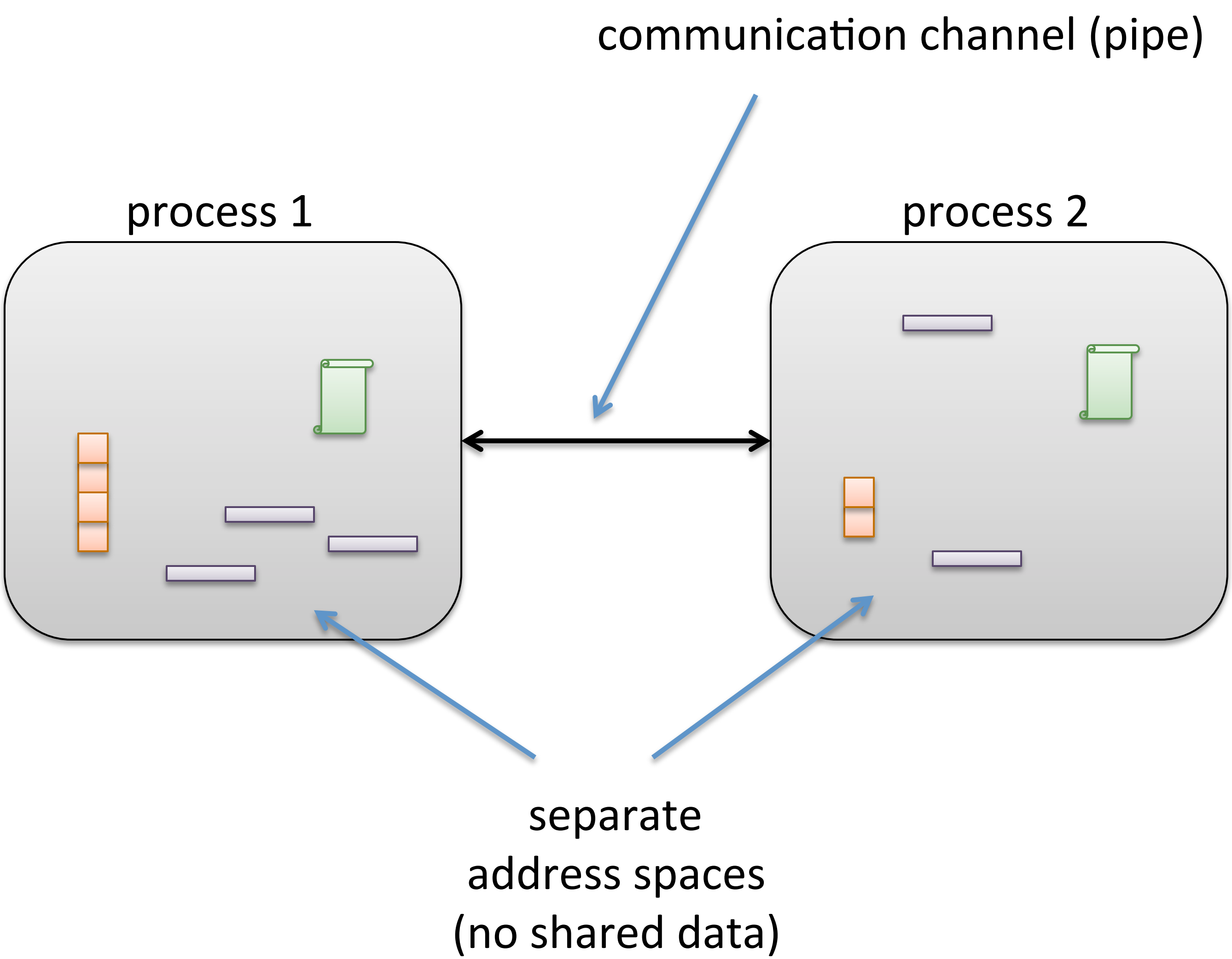
Assignment 7



map-reduce API for Assignment 7

tabulate (f: int->'a) (n: int) : 'a seq	Create seq of length n, element i holds f(i)	Parallel
seq_of_array: 'a array -> 'a seq	Create a sequence from an array	Constant time
array_of_seq: 'a seq -> 'a array	Create an array from a sequence	Constant time
iter (f: 'a -> unit): 'a seq -> unit	Applying f on each element in order. Useful for debugging.	Sequential
length: 'a seq -> int	Return the length of the sequence	Constant time
empty: unit -> 'a seq	Return the empty sequence	Constant time
cons: 'a -> 'a seq -> 'a seq	(nondestructively) cons a new element on the beginning	Sequential
singleton: 'a -> 'a seq	Return the sequence with a single element	Constant time
append: 'a seq -> 'a seq -> 'a seq	(nondestructively) concatenate two sequences	Sequential
nth: 'a seq -> int -> 'a	Get the nth value in the sequence. Indexing is zero-based.	Constant time
map (f: 'a -> 'b) -> 'a seq -> 'a seq	Map the function f over a sequence	Parallel
reduce (f: 'a -> 'a -> 'a) (base: 'a): 'a seq -> 'a	Fold a function f over the sequence. f must be associative, and base must be the unit for f.	Parallel
mapreduce: ('a->'b)->('b->'b->'b)-> 'b -> 'a seq -> 'b	Combine the map and reduce functions.	Parallel
flatten: 'a seq seq -> 'a seq	flatten [[a0;a1]; [a2;a3]] = [a0;a1;a2;a3]	Sequential
repeat (x: 'a) (n: int) : 'a seq	repeat x 4 = [x;x;x;x]	Sequential
zip: ('a seq * 'b seq) -> ('a * 'b) seq	zip [a0;a1] [b0;b1;b2] = [(a0,b0);(a1,b1)]	Sequential
split: 'a seq -> int -> 'a seq * 'a seq	split [a0;a1;a2;a3] 1= ([a0],[a1;a2;a3])	Sequential
scan: ('a->'a->'a) -> 'a -> 'a seq -> 'a seq	scan f b [a0;a1;a2;...] = [f b a0; f (f b a0) a1; f (f (f b a0) a1) a2; ...]	Parallel

Processes


communication channel (pipe)
process 1
process 2
separate
address spaces
(no shared data)

Need-to-know Info

- Processes are managed by your operating system
- Share time executing on available cores
- Processes have separate address spaces so communication occurs by:
  - serializing data (converting complex data to a sequence of bits)
  - writing data to a buffer
  - reading data out of the buffer on the other side
  - deserializing the data
- Cost is relative to the amount of data transferred
  - minimizing data transfers is an important performance consideration

Unix (Linux)   pipe(), fork(), exec()

(Standard Unix, C-language calling sequences)

int pipe(int fd[2]);

(now can read from file-descriptor fd[0], write to fd[1])

int fork(void)

(creates a new OS process;

in child, returns 0; in parent, returns process id of child.)

int execve(char *filename, char *argv[], char *envp[])

(overwrite this process with a new execution of filename(argv);

if execve returns at all, then it must have failed)

Typical use of pipe, fork, exec

What you write at the shell prompt

cat foo | grep abc

One learns this in COS 217

What the shell does (simplified)

fd 0 – standard in
fd 1 – standard out

```
int fd[2];  int pid1, pid2;
pipe (fd);
pid1 = fork();
if (pid1) { /* in the parent */
    close(fd[0]); close(1); dup2(fd[1],1); close(fd[1]);
    exec("/bin/cat","foo");
} else { /* in the child */
    close(fd[1]); close(0); dup2(fd[0],0); close(fd[0]);
    exec("/bin/grep", "abc")
}
```

Typical use of pipe, fork, exec

What you write at the shell prompt

cat foo | grep abc

What the sh s (simplified)

```
int fd[2];  int pid1, pi
pipe (fd);
pid1 = fork();
if (pid1) { /* in the parent */
    close(fd[0]); close(1);  dup2(
    exec("/bin/cat","foo");
} else { /* in the child */
    close(fd[1]); close(0); dup
    exec("/bin/grep", "abc")
}
```

One learns this
in COS 217

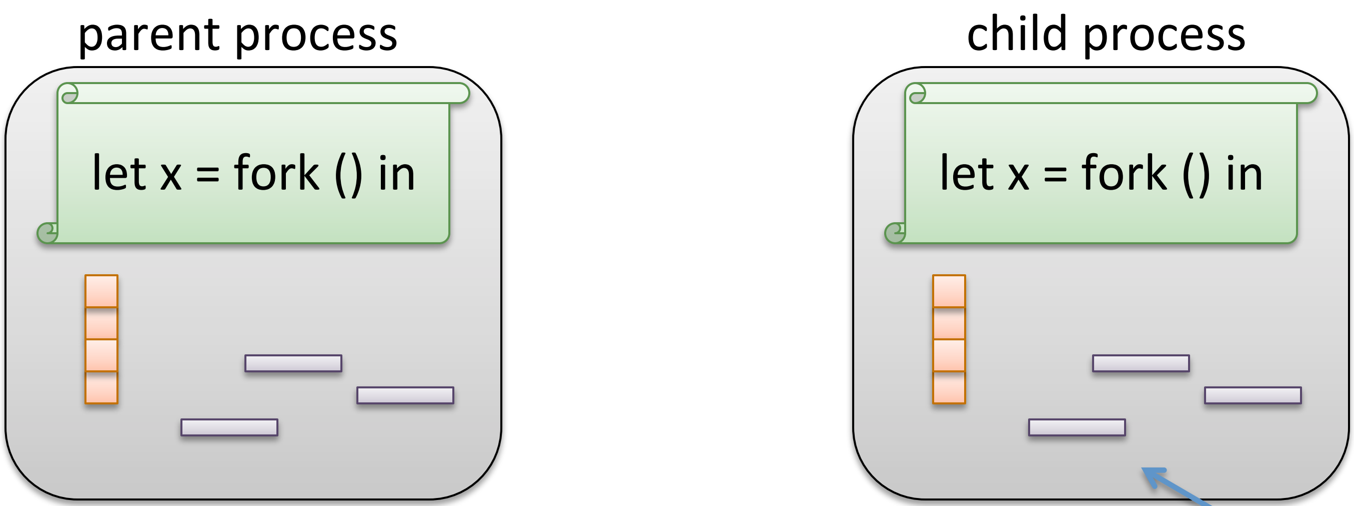
pipe is a beautiful functional abstraction, isn't it?

It hides all this garbage so I don't have to think about it!!

Processes in OCaml

create a child process using fork : unit -> int

– creates two processes; identical except for the return value of fork()



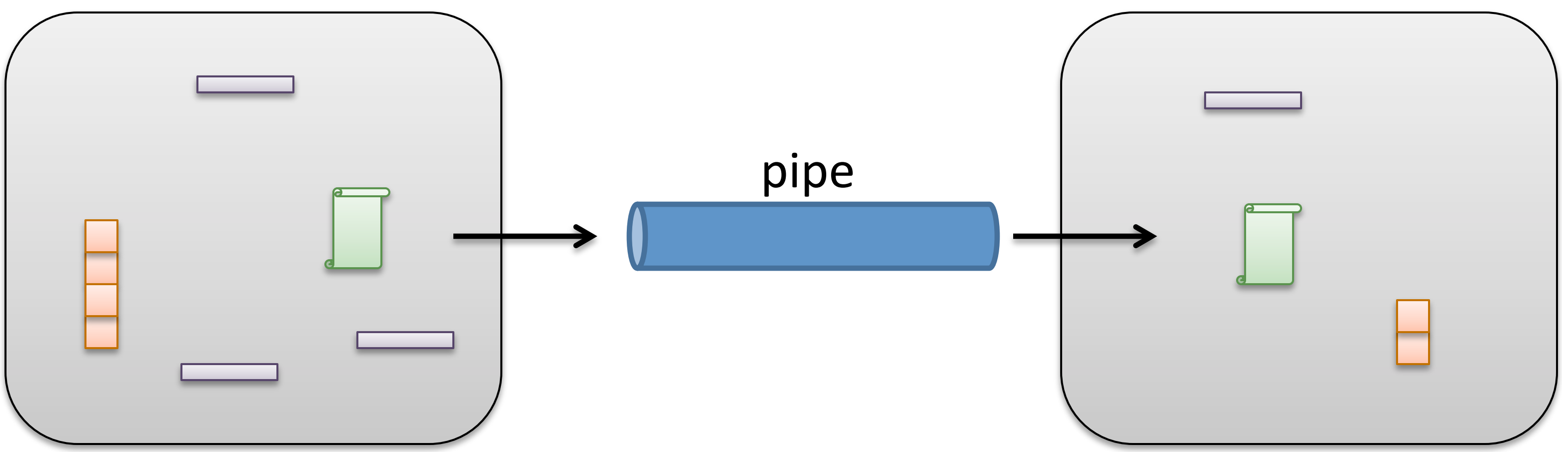
copies of data are made when either parent or child writes to the data

standard use:

```
match fork () with
| 0 ->  ... child process code ...
| pid -> ... parent process code ...
```

Interprocess Communication via Pipes

- A pipe is a first-in, first-out queue
- Data (a sequence of bytes) may be written on one end of the pipe and read out the other
  - writes block after the underlying buffer is filled but not yet read
  - reads block until data appears to be read
  - bad idea to read and write the same pipe in the same process!



- Creating a pipe:
  - pipe : unit -> file_descr * file_descr

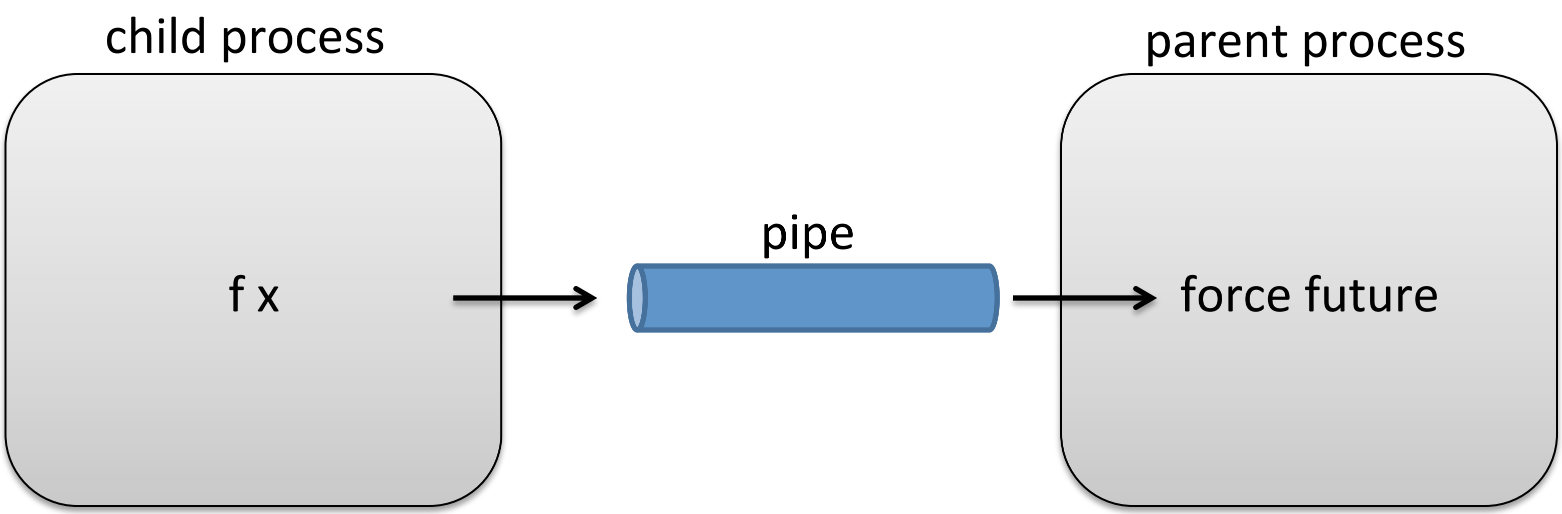
Futures via Processes

future f x runs
f x in a child process

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

result of f x serialized
and sent through a pipe
back to the parent



Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

pipe endpoint read by parent

process id of the child

Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

```
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (
     close fin;
     let oc = out_channel_of_descr fout in
     Marshal.to_channel oc (f x) [Marshal.Closures];
     Pervasives.exit 0 )
  | cid -> (
     close fout;
     {fd=fin; pid=cid}  )
```

create pipe to communicate

Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

```
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (
     close fin;
     let oc = out_channel_of_descr fout in
     Marshal.to_channel oc (f x) [Marshal.Closures];
     Pervasives.exit 0 )
  | cid -> (
     close fout;
     {fd=fin; pid=cid}  )
```

fork child

Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

```
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (
     close fin;
     let oc = out_channel_of_descr fout in
     Marshal.to_channel oc (f x) [Marshal.Closures];
     Pervasives.exit 0 )
  | cid -> (
     close fout;
     {fd=fin; pid=cid}  )
```

child uses the output (fout) and closes the input (fin)

parent uses the input (fin) and closes the output (fout)

Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

```
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (
     close fin;
     let oc = out_channel_of_descr fout in
     Marshal.to_channel oc (f x) [Marshal.Closure
     Pervasives.exit 0 )
  | cid -> (
     close fout;
     {fd=fin; pid=cid}  )
```

parent completes routine immediately; keeping the future data structure around to force later

Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

```
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (
     close fin;
     let oc = out_channel_of_descr fout in
     Marshal.to_channel oc (f x) [Marshal.Closures];
     Pervasives.exit 0 )
  | cid -> (
     close fout;
     {fd=fin; pid=cid}  )
```

child executes the future function

Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

```
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (
     close fin;
     let oc = out_channel_of_descr fout in
     Marshal.to_channel oc (f x) [Marshal.Closures];
     Pervasives.exit 0 )
  | cid -> (
     close fout;
     {fd=fin; pid=cid}  )
```

then marshalls the results, sending them over the pipe

... and then terminates, its job complete

Marshalling / unmarshalling

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Marshal.html

In Java this
is called
“serialize”

Module Marshal

module Marshal: sig .. end

Marshaling of data structures.

This module provides functions to encode arbitrary data structures as sequences of bytes, which can then be written on a file or sent over a pipe or network connection. The bytes can then be read back later, possibly in another process, and decoded back into a data structure. The format for the byte sequences is compatible across all machines for a given version of OCaml.

Warning: marshaling is currently not type-safe. The type of marshaled data is not transmitted along the value of the data, making it impossible to check that the data read back possesses the type expected by the context. In particular, the result type of the Marshal.from_* functions is given as 'a, but this is misleading: the returned OCaml value does not possess type 'a for all 'a; it has one, unique type which cannot be determined at compile-type. The programmer should explicitly give the expected type of the returned value, using the following syntax:

- (Marshal.from_channel chan : type).

Anything can happen at run-time if the object in the file does not belong to the given type.

Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

```
let force (f: 'a future) : 'a =
  let ic = in_channel_of_descr f.fd in
  let res = ((Marshal.from_channel ic) : 'a) in
  close f.fd;
  match waitpid [] f.pid with
  | (_,WEXITED 0) -> res
  | _ -> failwith "process failed to terminate in force"
```

reads the data from the future's pipe

closes the file descriptor

Futures via Processes

future interface

```
type 'a future
val future : ('a -> 'b) -> 'a -> 'b future
val force : 'a future -> 'a
```

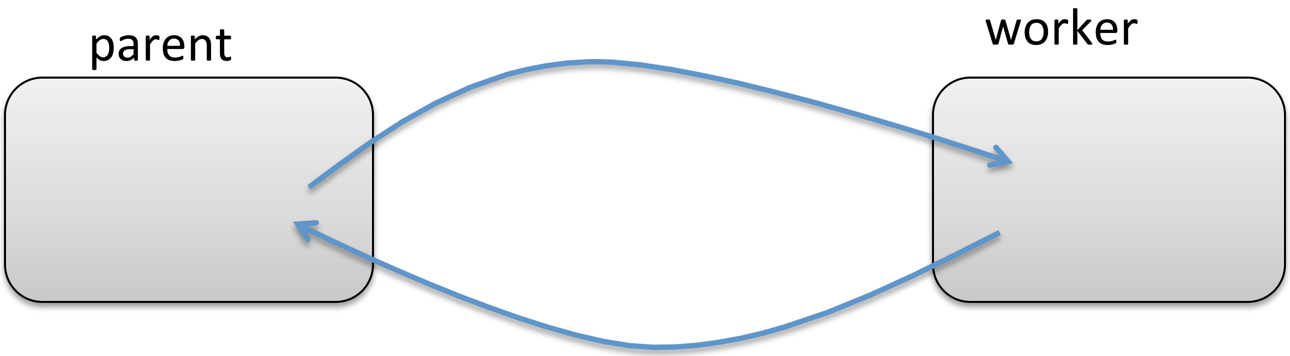
```
type 'a future = {
  fd  : file_descr;
  pid: int
}
```

```
let force (f: 'a future) : 'a =
  let ic = in_channel_of_descr f.fd in
  let res = ((Marshal.from_channel ic) : 'a) in
  close f.fd;
  match waitpid [] f.pid with
  | (_,WEXITED 0) -> res
  | _ -> failwith "process failed to terminate in force"
```

wait until child terminates; prevents "fork bomb" (other techniques could be used here)

Costs of "fork"

- Futures enable a rather simple communication pattern:



- But the cost of starting up a process and communicating data back and forth is high

Unix "fork" system call copies the entire address space into the child process. That includes all the closures and heap data structures in your entire program!

- Operating system does it ***lazily***, using virtual-memory paging.
- That means this pattern: if (fork()) {parent…} else {exec();} does not pay a price, does no copying

But the pattern on the previous slides has no "exec();" call.

Another problem with "fork"

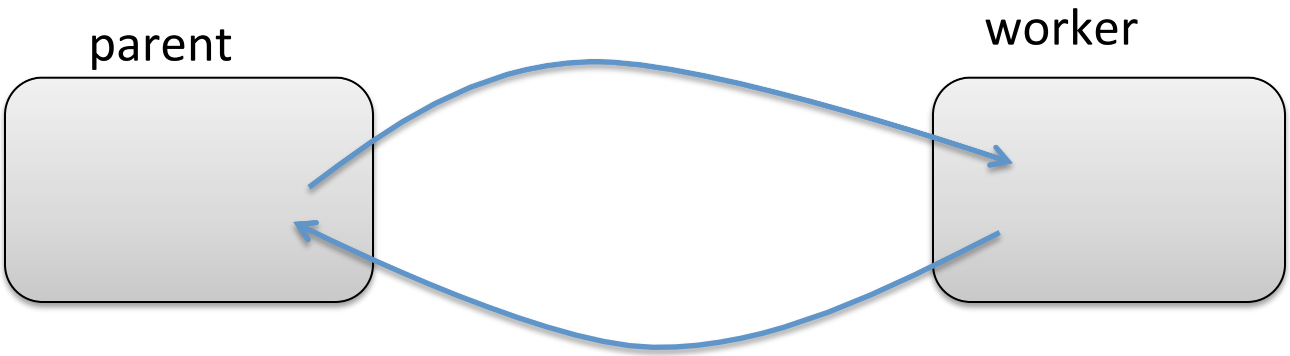
```
let future (f: 'a -> 'b) (x: 'a) : 'b future =
  let (fin, fout) = pipe () in
  match fork () with
  | 0 -> (close fin;
          let oc = out_channel_of_descr fout in
          Marshal.to_channel oc (f x) [Marshal.Closures];
          Pervasives.exit 0 )
  | cid -> (close fout; {fd=fin; pid=cid}  )
```

Parent process and child process must share memory!

- This is possible on two different cores of the same multicore chip
- Sometimes possible with two chips on the same circuit board.
- **Not scalable to massive parallelism in the data center!**

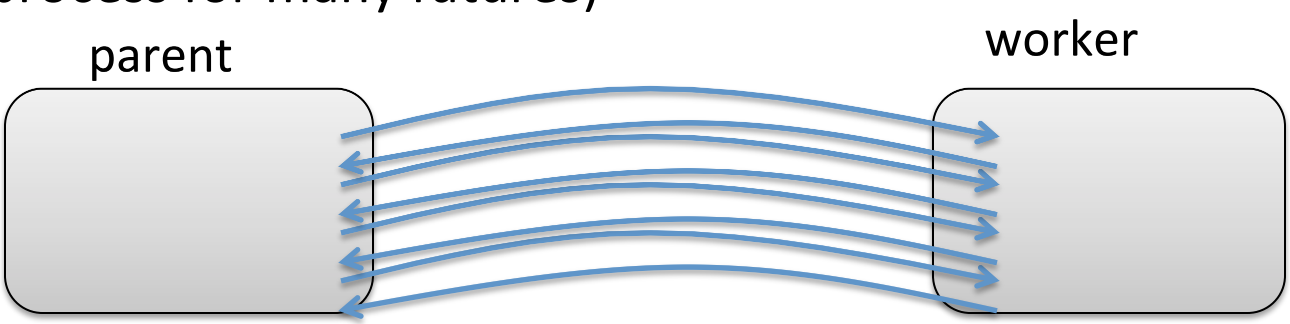
Message Passing

- Futures enable a rather simple communication pattern:



But the cost of starting up a process and communicating data back and forth is high

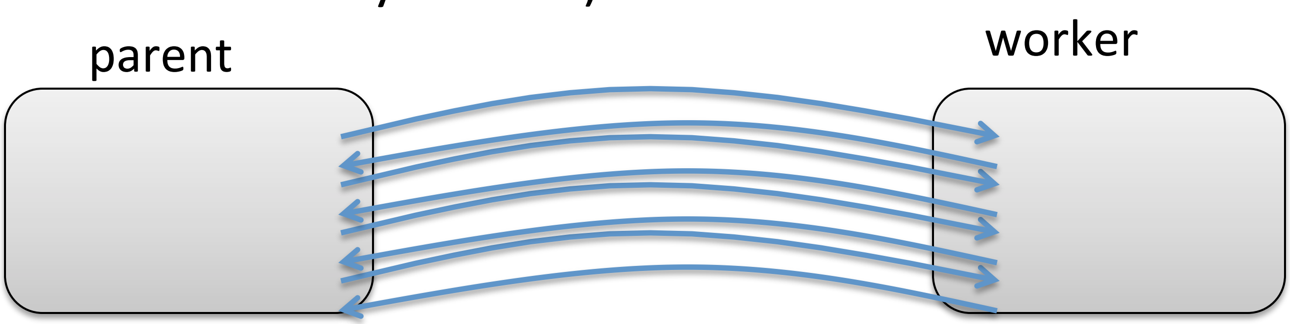
- Instead: spawn 1 worker and have it do many tasks
  - (the implementation of futures could be optimized to reuse 1 process for many futures)



Message Passing

Also: when creating the worker (with “fork”), don’t send data at the same time! No need to share memory; the “fork” can be remote on another machine (in the data center).

- Instead: spawn 1 worker and have it do many tasks
  - (the implementation of futures could be optimized to reuse 1 process for many futures)



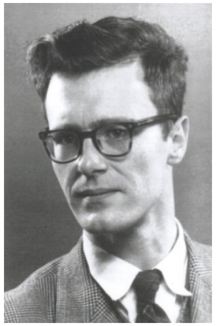
History: Shared Memory vs. Message-Passing

In 1968 and 1973, Dijkstra and Hoare described the principles of shared-memory computing with semaphores (locks, mutual exclusion).

In 1978, a new paradigm,

“Communicating Sequential Processes”, was introduced.

CSP uses synchronous channels with *no shared memory.* Nicer than that Dijkstra-Hoare shared-memory stuff.

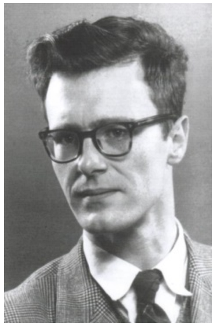


Edsger W. Dijkstra
1930 - 2001

C. Antony R. Hoare
1934 -

CSP was invented by

based on ideas from



Communicating Sequential Processes (CSP)

The CSP paradigm has evolved quite a bit since Tony Hoare's original invention.

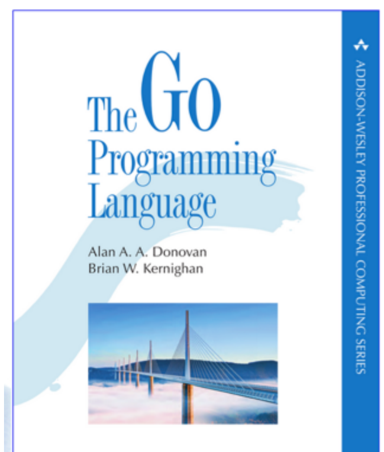
1978: CSP
Tony Hoare

1985: **Squeak**
Luca Cardelli and Rob Pike

1991: **Concurrent ML**
John Reppy

1994: **Newsqueak**
Rob Pike

2007: **Go**
Robert Griesemer, Rob Pike, and Ken Thompson



2015 Go book by
Donovan and Kernighan

Gratuitous remarks

Go is a pretty good language:

Safe (like ML, Haskell, Java, Python; unlike C, C++)

Garbage-collected (like ML, Haskell, Java, Python; unlike C, C++)

Enforces abstractions (like ML, Haskell, Java; unlike Python, C, C++)

Good concurrency mechanisms (better than ML, Java, Python, C, C++)

Has higher-order functions (like ML, Haskell, sorta Java; unlike C, C++)

Avoids language bloat (like ML, Haskell, C; unlike C++)

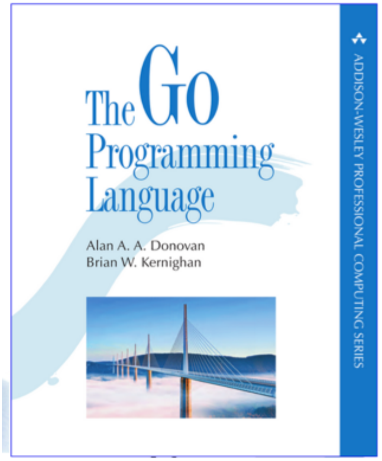
Open source (like ML, Haskell, C, Python; unlike Java)

But:

No polymorphism (unlike ML, Haskell, Java)

Not functional (too many features that depend on side effects)

Therefore: Not quite Nirvana



CSP -> MPI (1990s – now)

MPI (Message Passing Interface)

is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

MPI has become a *de facto* standard for communication among processes that model a parallel program running on a distributed memory system. Actual distributed memory supercomputers such as computer clusters often run such programs. MPI programs can also run on shared memory computers.

Most MPI implementations consist of a specific set of routines (i.e., an API) directly callable from C, C++,Fortran and any language able to interface with such libraries, including C#, Java or Python.

[Adapted from Wikipedia]

Back to CSP

In 1978, a new paradigm, "Communicating Sequential Processes", was introduced.

CSP uses synchronous channels with *no shared memory.* Nicer than that Dijkstra-Hoare shared-memory stuff.

CSP was invented by

Tony Hoare

PL Theorists love "Little Languages"

The lambda calculus:

- just variables, functions, function application
- the essence of functional programming

CSP:

- just process creation, channel creation, data send, data receive and choice
- the essence of concurrent programming

Programming languages are complicated.
There is great benefit to studying them in a minimal context.

Operations on channels in CSP

spawn f x	*create a new (non-shared-memory) thread*
c←new()	*make a new channel*
c!x	*send datum "x" on channel "c"*
c?x	*receive datum "x" from channel "c"*

select [c?x → f(x) | d?y → g(y) | e?z → h(z)]

block until at least one channel is ready; then receive on a ready channel

Channels are both
a **communication** mechanism and
a **synchronization** mechanism

SYNCHRONOUS channel:

c!x *blocks* until some thread does a matching c?y

ASYNCHRONOUS (buffered) channel:

c!x can proceed even if no channel is trying to read

Typical pattern of use

```
(* repeatedly send i on c forever *)
let rec f c i =
   c ! i; f c i

(* repeatedly print what you get on c or d forever *)
let rec consume c d =
   select [  c?x → print x
           | d?y → print y];
   consume c d

let zeros = new() in
let ones = new() in
spawn (f zeros) 0;           (* send 0s forever *)
spawn (f ones) 1;            (* send 1s forever *)
consume zeros ones           (* print 0s and 1s as you receive them *)
```

```
(* sample output *)
11101000100010011000000011111101010100111
```

Assignment 7 channels

No need for “select”; any given thread is waiting on just one channel.

Channel creation is combined with thread creation in a simple “design pattern.”

Assignment 7: bidirectional channels

an ('s, 'r) channel for ME looks like this:

Message Passing API

```
type ('s, 'r) channel

val spawn  : (('r, 's) channel -> 'a -> unit) -> 'a -> ('s, 'r) channel
val send    : ('s, 'r) channel -> 's -> unit
val receive : ('s, 'r) channel -> 'r

val wait_die : ('s, 'r) channel -> unit
```

What can you tell from *just* the type of **spawn**?

```
let  f (c: (int,bool) channel) (y: string) = ...

in   spawn f x
```

```
type ('s, 'r) channel

val spawn   : (('r, 's) channel -> 'a -> unit) -> 'a -> ('s, 'r) channel
val send    : ('s, 'r) channel -> 's -> unit
val receive : ('s, 'r) channel -> 'r

val wait_die : ('s, 'r) channel -> unit
```

Summary

- A few disciplines for parallel and concurrent programming:
  - futures
  - locks
  - message-passing
  - parallel collections
- Higher-level libraries (futures, collections) that hide the synchronization primitives are *easier to use* and *more reliable* than lower-level synchronization primitives on their own (locks, message passing)
- On the other hand, higher-level libraries are often less flexible
  - data represented as a particular collection
  - computation needs to fall into the map-reduce (or series-parallel graph) frameworks