Assignment 7: *Implement* and *use* parallel map-reduce

COS 326
Andrew W. Appel
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
End goal: develop a system for efficiently computing US population queries by geographic region
Assignment 7

Libraries build layered concurrency abstractions

Concurrency primitives

Hardware

Applications

Library

message passing

Library

inverted index
dating service
geographic queries

Library

parallel sequences
futures

Library

unix processes
## Map-Reduce API for Assignment 7

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

Process 1

Process 2

Communication channel (pipe)

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

What the shell does (simplified)

```c
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")
}
```

You learned this in COS 217
create a child process using `fork : unit -> int`

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

standard use:

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

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

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

future f x runs f x in a child process

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
}
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

```ocaml
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} )
```

future interface

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

```
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

```ocaml
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

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

```ocaml
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} )
```

type 'a future = {
  fd : file_descr;
  pid: int
}
In the context of futures via processes, we define a future interface:

### Future Interface Definition

- **Type Definition:**
  ```
  type 'a future = {
    fd : file_descr;
    pid : int
  }
  ```

- **Function Definitions:**
  ```
  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})
  ```

This code snippet demonstrates how to create a future, where the results are marshalled and sent over the pipe. The process then terminates, indicating its job is complete.
Module **Marshal**

```ocaml
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

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

```ocaml
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"
```

type 'a future = {
  fd : file_descr;
  pid: int
}
Futures via Processes

future interface

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

```ocaml
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)
• 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!**
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)
Instead: spawn 1 worker and have it do many tasks

- (the implementation of futures could be optimized to reuse 1 process for many futures)

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).
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.

CSP was invented by

Edsger W. Dijkstra 1930 - 2001
C. Antony R. Hoare 1934 -
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
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)  
*these two issues are related; how?*  

Therefore:  **Not quite Nirvana**
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]
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
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

SYNONOMUS channel:

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

ASYNONOMOUS (buffered) channel:

c!x  can proceed even if no channel is trying to read
let rec f c i = (c!i; f c i)
let rec consume c d =
  (select [c?x → print x | d?y → print y]; consume c d)

let zeros←new(); ones←new();
spawn (f zeros) 0; spawn (f ones) 1;
consume zeros ones

Result:
1110100010001001110000000111111101010100111
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`?

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

in  spawn f x
```

```ocaml
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
```