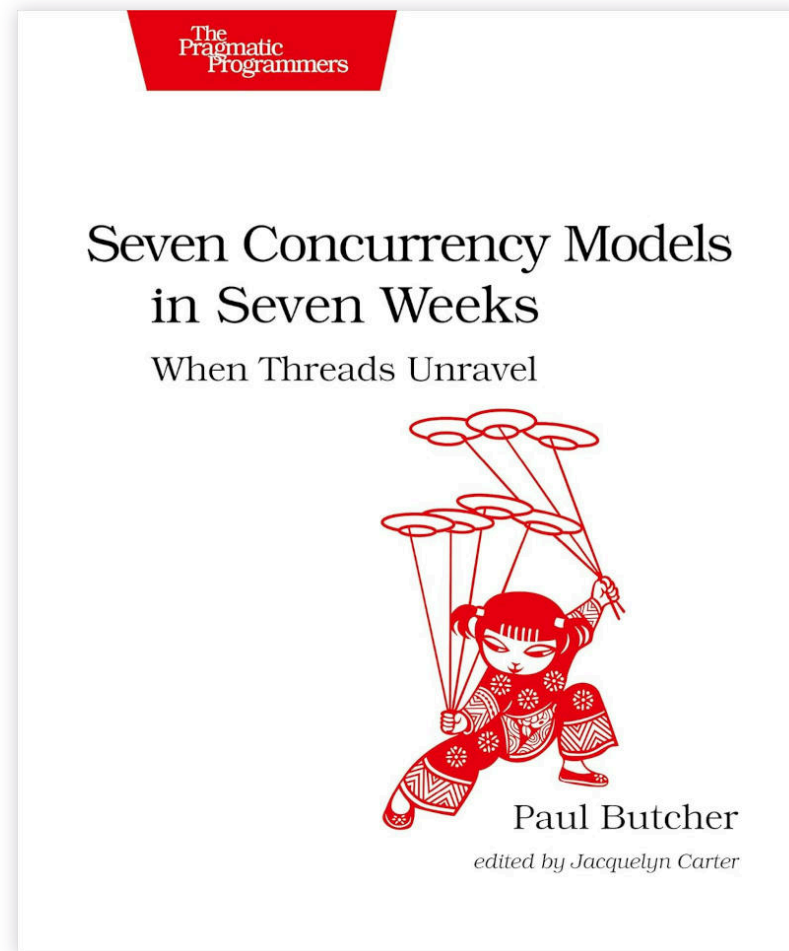


Modelos de concurrencia alternativos

Modelos de concurrencia alternativos

- Hilos y candados (Clojure)
- Programación funcional (Clojure)
 - Promise
 - Future
 - pcalls, pvalues, pmap
 - Delay
 - Atoms
 - Refs & STM
 - Agents
 - Reducers (fork/join)
 - core.async (go blocks & channels)
- Corutinas y generadores (Python)
- Paralelismo en el GPU
- Big Data



Clojure: Hilos y candados

```
(def candado (Object.))

(def hilo1 (Thread. (fn []
  (println "1: esperando candado")
  (locking candado
    (println "1: candado adquirido")
    (Thread/sleep 3000)
    (println "1: candado liberado")))))

(def hilo2 (Thread. (fn []
  (println "2: esperando candado")
  (locking candado
    (println "2: candado adquirido")
    (Thread/sleep 1000)
    (println "2: candado liberado")))))

(run! #(.start %) [hilo1 hilo2])
(run! #(.join %) [hilo1 hilo2])
```



Promise

- `(promise)` crea una nueva promesa.
- `(deliver p v)` asigna el valor `v` a la promesa `p`.
- `(deref p)` o `@p` bloquea hasta que la promesa tenga un valor y lo devuelve.

```
(defn tarea-muy-costosa [n]
  (Thread/sleep 3000)
  (+ n 10))

(def p (promise))

(.start (Thread. (fn []
                  (let [r (tarea-muy-costosa 1)]
                    (deliver p r)
                    (println "deliver")))))

(println "deref 1:")
(println @p)

(println "deref 2:")
(println @p)
```



Future

- `(future expr)` evalúa la expresión en un hilo separado.
- `(deref f)` o `@f` bloquea hasta que el resultado esté disponible y lo devuelve.

```
(defn tarea-muy-costosa [n]
  (Thread/sleep 3000)
  (+ n 10))

(def f1 (future (tarea-muy-costosa 1)))
(def f2 (future (tarea-muy-costosa 2)))
(def f3 (future (tarea-muy-costosa 3)))
(println "Hilos lanzados")

(println "Deref 1:")
(println (map deref [f1 f2 f3]))

(println "Deref 2:")
(println (map deref [f1 f2 f3]))

; necesario para evitar una espera de 1 minuto antes de
; finalizar el proceso:
(shutdown-agents)
```



pcalls, pvalues y pmap

- `(pcalls f1 f2 f3)`: Ejecuta las funciones en hilos separados.
- `(pvalues expr1 expr2 expr3)`: Evalúa las expresiones en hilos separados.
- `(pmap f coll)`: Similar a `map`, pero ejecuta las aplicaciones de `f` en hilos separados.

```
(defn tarea-muy-costosa [n]
  (Thread/sleep 3000)
  (+ n 10))

(println (time (doall (map tarea-muy-costosa (range 4)))))
;; "Elapsed time: 12001.447199 msecs"
;; (10 11 12 13)

(println (time (doall (pmap tarea-muy-costosa (range 4)))))
;; => "Elapsed time: 3012.111684 msecs"
;; => (10 11 12 13)

(shutdown-agents)
```



Delay

- `(delay expr)` crea un objeto de tipo `Delay`, que asegura que la expresión se evalúa una única vez la primera vez que es desreferenciada.
- `(deref d)` o `@d` obtiene el resultado de la evaluación de la expresión, bloqueando si aun no se ha terminado de evaluar.

```
(defn tarea-muy-costosa [n]
  (println "evaluando!")
  (Thread/sleep 3000)
  (+ n 10))

(def d (delay (tarea-muy-costosa 1)))
(println "Delay creado")

(println (time @d))
(println (time @d))

(shutdown-agents)
```



Refs & STM

- `(ref valor-inicial)` crea un ref con el valor inicial.
- `(deref r)` o `@r` devuelve el valor actual del ref.
- `(dosync expr)` ejecuta la expresión en una transacción atómica.
- `(alter r f)` dentro de una transacción, actualiza el valor del ref.

```
(def prox-numero-libre (ref 0))
(def total-recaudado (ref 0))

(defn reservar-numero [precio]
  (dosync (let [n @prox-numero-libre
               t (alter total-recaudado + precio)]
            (alter prox-numero-libre inc)
            [n t])))

(dotimes [_ 8]
  (.start (Thread. (fn []
                    (Thread/sleep (rand-int 300))
                    (let [[n t] (reservar-numero (rand-int 5000))]
                      (locking *out* (println (format "número reservado: %d, total recaudado: %d" n t))))
                    (recur))))))
```




Atoms

- `(atom valor-inicial)` crea un átomo con el valor inicial.
- `(deref a)` o `@a` devuelve el valor actual del átomo.
- `(swap! a f)` actualiza en forma atómica el valor del átomo, y devuelve el valor anterior.
- `(swap-vals! a f)` similar a `swap!`, pero devuelve una secuencia con el valor anterior y el nuevo valor.

```
(def estado-rifa (atom {:prox-numero-libre 0
                        :total-recaudado 0}))

(defn reservar-numero [precio]
  (let [actualizar #(-> %
                        (update :prox-numero-libre inc)
                        (update :total-recaudado + precio))
        [viejo nuevo] (swap-vals! estado-rifa actualizar)]
    [(viejo :prox-numero-libre) (nuevo :total-recaudado)]))

(dotimes [_ 8]
  (.start (Thread. (fn []
                    (Thread/sleep (rand-int 300))
                    (let [[n t] (reservar-numero (rand-int 5000))]
                      (locking *out* (println (format "número reservado: %d, total recaudado: %d" n t))))
                    (recur))))))
```



Agents

- `(agent valor-inicial)` crea un agente con el valor inicial.
- `(deref a)` o `@a` devuelve el valor actual del agente.
- `(send a f)` envía una función `f` para actualizar el valor del agente en otro hilo.
- `(add-watch a k f)` ejecutará `f` cada vez que el valor del agente cambie.

```
(require '[clojure.pprint :as p])

(def rifa (agent {:proximo-numero-libre 0
                  :reservas []}))

(defn reservar-numero [persona precio]
  (send rifa (fn [estado]
    (let [n (:proximo-numero-libre estado)
          nueva-reserva {:persona persona
                          :numero n
                          :precio precio}]
      (-> estado
        (update :proximo-numero-libre inc)
        (update :reservas conj nueva-reserva))))))

(dorun (pmap #(reservar-numero % (rand-int 5000))
  ["Ana" "Beto" "Carla" "Diego" "Eva" "Fabián" "Gisela" "Hugo"]))

(await rifa)
(p/pprint @rifa)
(shutdown-agents)
```



Reducers

Las funciones de la biblioteca `clojure.core.reducers` permiten procesar grandes cantidades de datos mediante el patrón **fork/join**.

```
(require '[clojure.core.reducers :as r])

(def numeros (vec (range 100000)))


(println (r/fold 512 ;; tamaño de la partición
               +    ;; función para combinar el resultado de dos particiones
               +    ;; función para reducir una partición
               (r/filter odd? numeros)))
```

Communicating Sequential Processes (CSP)

CSP es un modelo de concurrencia basado en procesos livianos que se comunican mediante el paso de mensajes a través de canales.

La biblioteca `core.async` es una implementación de CSP en Clojure, inspirada en las *goroutines* y *channels* de Go.

- `(chan)` crea un canal.
- `(>! c v)` encola `v` en el canal `c` (bloquea si el *buffer* del canal está lleno).
- `(<! c)` desencola del canal `c` (bloquea si no hay nada para leer).
- `(close! c)` cierra el canal `c` (`<!` devolverá `nil`).
- `(go ...)` ejecuta el cuerpo en forma asíncrona en un go block. Devuelve un canal que contendrá el resultado.



```
(ns goblocks.core
  (:gen-class)
  (:require
   [clojure.core.async :as async :refer [<!! <! >! chan go close!]]))

(defn -main []
  (let [entrada (chan)
        _ (go (doseq [_ (range 10000)]
                (>! entrada (rand-int 100)))
            (close! entrada))
        workers (for [_ (range 100)]
                  (go (loop [suma 0]
                          (let [n (<! entrada)]
                            (if (nil? n)
                                suma
                                (recur (+ suma n)))))))
        salidas (async/merge workers)
        resultado (go (<!! (async/reduce + 0 salidas)))]
    (println (<!! resultado))))
```



Corutinas (Python)

Una **corutina** es una función que puede ser pausada y reanudada, permitiendo la concurrencia cooperativa.

Originalmente las corutinas en Python se usaban para implementar **generadores**, que son una forma conveniente de crear iteradores.

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

```
>>> f = fibonacci()
>>> next(f)
0
>>> next(f)
1
>>> next(f)
1
>>> next(f)
2
```

Corutinas (cont.)

Combinando las corutinas con un **scheduler** se logra la concurrencia:

```
def col():  
    print("col haciendo cosas...")  
    yield  
    print("col haciendo cosas...")  
    yield  
  
def co2():  
    print("co2 haciendo cosas...")  
    yield  
    print("co2 haciendo cosas...")  
    yield
```

```
def scheduler(corutinas):  
    cola = corutinas[:]  
    while cola:  
        try:  
            co = cola.pop(0)  
            co.send(None)  
            cola.append(co)  
        except StopIteration:  
            pass  
  
scheduler([col(), co2()])
```



Corutinas (cont.)

El scheduler se transforma en un **event loop**:

```
def get_page():
    print("Starting to download page")
    yield ("sleep", 1)
    print("Done downloading page")
    yield "<html>Hello</html>"

def read_db():
    print("Starting to retrieve data from db")
    yield ("sleep", 0.5)
    print("Connected to db")
    yield ("sleep", 1)
    print("Done retrieving data from db")
    yield "db-data"
```

```
import time, heapq

def scheduler(coros):
    ready = coros[:]
    sleeping = []
    while True:
        if not ready and not sleeping:
            break
        if not ready:
            deadline, coro = heapq.heappop(sleeping)
            if deadline > time.time():
                time.sleep(deadline - time.time())
            ready.append(coro)
        try:
            coro = ready.pop(0)
            result = coro.send(None)
            if len(result) == 2 and result[0] == "sleep":
                deadline = time.time() + result[1]
                heapq.heappush(sleeping, (deadline, coro))
            else:
                print(f"Got: {result}")
                ready.append(coro)
        except StopIteration:
            pass

scheduler([get_page(), read_db()])
```



Corutinas: `async/await`

Para trabajar con corutinas de forma más práctica, recientemente se agregó al lenguaje la sintaxis `async/await` y el módulo `asyncio`:

```
import asyncio

async def get_page():
    print("Starting to download page")
    await asyncio.sleep(1)
    print("Done downloading page")
    return "<html>Hello</html>"

async def write_db(data):
    print("Starting to write data to db")
    await asyncio.sleep(0.5)
    print("Connected to db")
    await asyncio.sleep(1)
    print("Done writing data to db")
```

```
async def read_db():
    print("Starting to read data from db")
    await asyncio.sleep(1)
    print("Connected to db")
    await asyncio.sleep(0.5)
    print("Done reading data from db")

async def get_and_write():
    page = await get_page()
    await write_db(page)

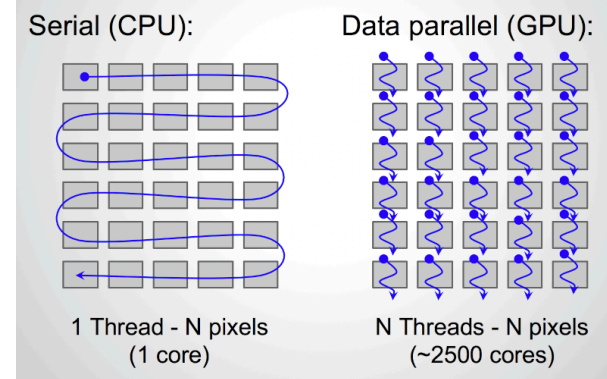
async def main():
    await asyncio.gather(
        read_db(),
        get_and_write(),
    )

asyncio.run(main())
```


Paralelismo en el GPU

Ejemplo OpenCL:

```
// Multiplies A*x, leaving the result in y.
// A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].
__kernel void matvec(__global const float *A, __global const float *x,
                    uint ncols, __global float *y)
{
    size_t i = get_global_id(0);           // Global id, used as the row index
    __global float const *a = &A[i*ncols]; // Pointer to the i'th row
    float sum = 0.f;                       // Accumulator for dot product
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```



Matrix size	1 CPU core	64 CPU cores	1 GPU	GPU speedup
(512, 512)	5.472 ms	517.722 μ s	115.805 μ s	47x / 5x
(1024, 1024)	43.364 ms	2.929 ms	173.316 μ s	250x / 17x
(2048, 2048)	344.364 ms	30.081 ms	866.348 μ s	400x / 35x
(4096, 4096)	3.221 s	159.563 ms	5.910 ms	550x / 27x

Big Data

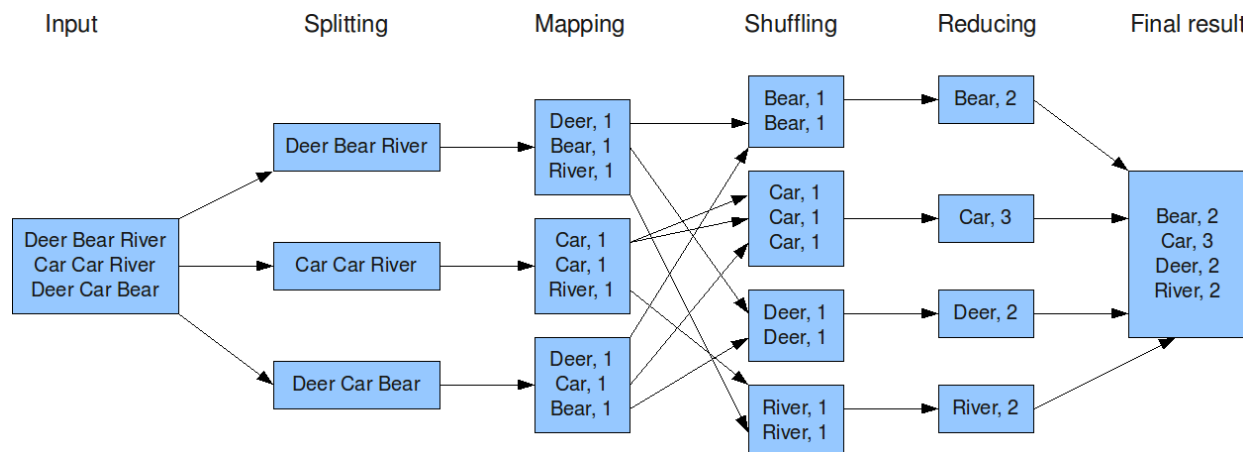
Se denomina **big data** al manejo de grandes volúmenes de datos, que no pueden ser procesados con los mecanismos tradicionales.

Aplicaciones que necesitan big data: búsquedas en Internet, redes sociales, gobiernos, salud, fintech, etc.

Existen diversos modelos y arquitecturas para el procesamiento de big data. Un ejemplo es **MapReduce**, que permite distribuir el procesamiento de grandes volúmenes de datos en múltiples nodos. El programador provee las implementaciones de dos funciones: `map` y `reduce`, y el sistema (ej. **Hadoop**) se encarga de distribuir la carga de trabajo y manejar fallos.

```
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
    // partialCounts: a list of aggregated partial counts
    sum = 0
    for each pc in partialCounts:
        sum += pc
    emit(word, sum)
```



www.ingenieria.uba.ar

f    /ingenieriauba

 /FIUBAoficial