**Emerging Programming Paradigms**

[ lezioni 30 marzo - 01 aprile ]

**Quick-Sort**

Il codice finale è disponibile su: [Codice/qsort.erl](https://drive.google.com/drive/u/1/folders/1I2XDnZ5p0jaGmEQOGlpsNv7XnMHUFhCq)

Un prima implementazione semplice può essere la seguente:

qsort([]) -> [];

qsort([H|TL]) ->

qsort([X || X <- TL, X =< H]) ++

[H] ++

qsort([X || X <- TL, X > H]).

main() ->

L = [rand:uniform(10000) || \_ <- lists:seq(0, 10000)],

qsort(L).

Si può aggiungere una funzione benchmark per calcolare il tempo impiegato dalla funzione qsort e valutarne l’efficienza.

benchmark(F, L) ->

timer:tc(?MODULE, F, [L]).

E la chiamiamo con benchmark(qsort, L).

Per rendere più affidabile la valutazione del tempo impiegato si esegue la funzione qsort più volte calcolando la media dei tempi di esecuzione.

benchmark(F, L) ->

T = [ timer:tc(?MODULE, F, [L]) || \_ <- lists:seq(0, 100) ],

lists:sum([ X || {X,\_} <- T ]) / (1000 \* length(T)).

La parte interessante è ora **parallelizzare** il codice. Una prima semplice implementazione può essere quella di creare un attore ogni qualvolta si vuole parallelizzare un’operazione. Un normale pc ha solo 4 core, avere in esecuzione un numero di attori molto superiore a 4 creerebbe una quantità di content switch molto importante allungando di molto i tempi di esecuzione.

psort([]) -> [];

psort([H|TL]) ->

Pid = self(),

**spawn\_link**(fun () -> Pid ! psort([X || X <- TL, X =< H]) end),

L2 = psort([X || X <- TL, X > H]),

receive R -> R end ++

[H] ++

L2.

Il problema di questa implementazione è che ci mette molto più tempo, a causa dei content switch e inoltre non ordina neanche i valori correttamente in quanto i messaggi passati tra processi non sono ordinati, ma arrivano quando riescono, impedendo il corretto ordinamento.

Per risolvere quest’ultimo problema si utilizzano i **cookies**, i quali vengono usati per tenere traccia dei messaggi. Il processo figlio invia la padre due cose: la ref che il padre ha generato e il risultato, permettendo al padre di ordinare correttamente i messaggi.

psort([]) -> [];

psort([H|TL]) ->

Pid = self(),

**Ref = make\_ref()**

spawn\_link(fun () -> Pid ! {**Ref**, psort([X || X <- TL, X =< H])} end),

L2 = psort([X || X <- TL, X > H]),

receive {**Ref**, R} -> R end ++

[H] ++

L2.

Questa implementazione ordina correttamente i dati, ma rimane molto più lenta di quella senza parallelismo. Per risolvere questo problema si può:

- avere un pool di attori già creati di dimensione prefissata che si intervallano per eseguire le operazioni in parallelo.

- si creano e distruggono attori in base al bisogno, limitati comunque da un bound superiore (non se ne creano all’infinito).

L’opzione di creare attori al volo può essere implementata come segue:

psort2(\_, []) -> [];

psort2(**0**, L) -> qsort(L); % se viene raggiunto il numero massimo di attori creabili i dati vengono ordinati sequenzialmente

psort2(**N**, [H|TL]) ->

Pid = self(),

Ref = make\_ref()

spawn\_link(fun () -> Pid ! {Ref, psort2(**N-1**, [X || X <- TL, X =< H])} end),

L2 = psort2(**N-1**, [X || X <- TL, X > H]),

receive {Ref, R} -> R end ++

[H] ++

L2.

Il miglioramento rispetto alla versione sequenziale c’è stato, ma è minimo. Ogni attore ha le sue pagine di memoria separate, quindi non sono necessari lock alla memoria, questo velocizza l’esecuzione globale del programma. Un modo per diminuire la complessità computazionale è cercare di muovere all'interno del secondo attore un numero di dati inferiore, e fare questo passaggio nel processo padre invece che nel processo figlio.

psort2([]) -> [];

psort2([H|TL]) ->

Pid = self(),

Ref = make\_ref(),

L1 = [X || X <- TL, X =< H] **% L1 ora lo salva il padre**

spawn\_link(fun () -> Pid ! {Ref, psort2(L1)} end),

L2 = psort2([X || X <- TL, X > H]),

receive {Ref, R} -> R end ++

[H] ++

L2.

L’approccio in cui si ha un pool di attori fissato inizialmente da cui si attinge può essere implementato come mostrato nel file.

ppsort3 ha un dispatcher, viene registrato il nome del dispatcher, il worker comunica al dispatcher di essere pronto e riceverà dal dispatcher il lavoro da fare (una funzione) il dispatcher viene creato con una lista di jobs da fare.

Il problema del implementazione del dispatcher è che se non ci sono worker poi non funziona la chiamata ricorsiva. Abbiamo fatto un messaggio “not available” per sistemare. [trascrizione da sistemare]

Questa soluzione è estremamente lenta rispetto alla soluzione precedente (spawn all’occorrenza dei processi in base al bisogno) perché quest’ultima fa degli spawn che permettono di parallelizzare molto lavoro. In questa soluzione invece a volte si impegna un intero worker per eseguire un lavoro esiguo (ordinare una lista piccola). Inoltre è presente anche un dispatcher intermedio che aggiunge una copia ulteriore dei dati.

**Porte**

Il binding di librerie in genere non è concettualmente difficile ma rompe completamente le proprietà di un linguaggio. Alla base dei linguaggi di programmazione funzionale c’è l’assunzione dell’immutabilità dei dati. Tramite il binding si rompe l’ipotesi di immutabilità, e inoltre usando binding a librerie esterne si rischia di essere forzati allo stile di programmazione della libreria. Un esempio è la libreria per la UX GNOME è studiato per essere usato da linguaggi imperativi (callback quando un bottone viene cliccato).

L’idea è che si vuole fare binding di una libreria esterna non funzionale non è un problema, è sufficiente inglobare il “mondo esterno” ad un attore e interagire con questo attore tramite send e receive senza rompere lo stile di Erlang. Questa interazione avviene tramite **porte**, una sorta di PID a cui un attore invia e riceve messaggi. Una porta assomiglia a un PID ma non è il PID di un attore ma il PID di un binding a qualcosa di esterno.

Un esempio:

|  |
| --- |
| read(P) ->  receive  {P, {data, Data}} ->  io:format("Received: ~p~n",[Data]),  read(P)  after 1000 -> ok  end.  test() ->  P = **open\_port**({spawn, "bc -i"}, [binary,{line,255}]),  read(P),  receive after 2000 -> ok end,  P ! {self(), {command, <<"3+5\n">>}},  io:format("Sent~n"),  read(P),  receive after 2000 -> ok end,  P ! {self(), close}. |

Il comando che permette di creare una porta è open\_port(). Con la funzione read(P) è possibile ricevere i messaggi ricevuti dalla porta P. È possibile inviare messaggi alla porta sempre con il punto esclamativo: P ! {self(), {command, <<...>>} il formato del messaggio inviato è importante, è necessario identificarsi con *self*, l’atomo *command* indica l’invio di un comando il cui contenuto deve essere in formato binario. Con il messaggio speciale close si chiude la porta: P ! {self(), close}. Non è possibile utilizzare exit con una porta, bensì solo con un PID.

La libreria *spawn* è un po’ la system di Erlang.

**Esecuzione distribuita**

Un programma **concorrente** viene eseguito su una singola macchina, magari utilizzando più thread in esecuzione che si alternano su più core. Se vengono eseguite più istruzioni insieme si parla di **parallelo**, se il processo è unico e si alterna nell’uso del processore è solo concorrente. Un programma è **distribuito** se si hanno più processori su una rete connessi in modo remoto tra loro.

Lo scambio di messaggi tramite un rete porta molti problemi: la possibilità di perdere messaggi, problemi di sicurezza, memorie non condivise, ecc.

Erlang è per costruzione appositamente creato per gestire tutti questi problemi, tranne la questione sicurezza in quanto i messaggi sono scambiati totalmente in chiaro. La cifratura dei canali di comunicazione deve avvenire su un altro layer, Erlang non implementa cifratura in quanto in origine era stato creato per la comunicazione sulla rete privata Ericsson.

Per assegnare un nome ad un nodo Erlang e comunicarvi in rete tramite esso è sufficiente:

rlwrap erl -name nome\_macchina\_1

Per permettere a più nodi di comunicare tra loro è sufficiente utilizzare una delle tante funzioni di Erlang che prenda in input il PID di un altro nodo, ad esempio net\_adm:ping(‘nome\_macchina\_2@cs.unibo.it’). A questo punto nome\_macchina\_1 e nome\_macchina\_2 sono associate allo stesso cluster perchè hanno comunicato tra loro. Con nodes() viene mostrata la lista dei nodi all’interno del cluster in cui si sta eseguendo.

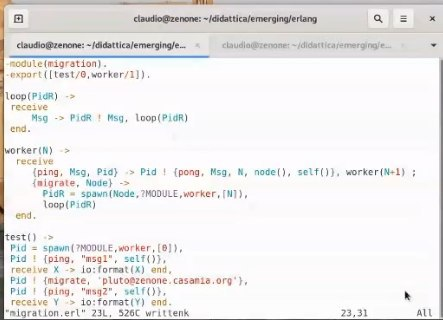
Con register(shell, self()) è possibile mandare messaggi all'attore identificato dall’atomo shell e verranno ricevuti dal nodo in esecuzione. Tramite {‘nome\_macchina\_2@cs.unibo.it’, shell} ! ciao il nodo nome\_macchina\_1 invia un messaggio all’altro nodo.

Con global:register\_name(atm, self()) permette di registrare globalmente il PID del nodo come atomo atm. Tutti i nodi ora gli potranno inviare messaggi tramite global:send(atm, hello), senza conoscerne il reale indirizzo.

Per **migrazione** di codice si intende prendere un attore che è in esecuzione su un nodo e spostarlo in esecuzione su un secondo nodo remoto per un certo periodo di tempo.

Erlang NON nasce con una primitiva per fare questa operazione. L’attore perderà il PID nella migrazione. Quello che andremo a implementare sarà una migrazione che duplicherà l’attore nel nodo dove vogliamo trasferirlo e poi trasformerà il primo attore in un semplice forwarder di messaggi per il secondo.

Un esempio di *migrazione* è disponibile nel file:



Quand’è che viene utile questa tecnica?

Bilanciamento del carico, per alleggerire un nodo si spostano attori in nodi scarichi.

Un programma è dati + codice. Se questi sono su due macchine diverse o si spostano i dati o il codice. Spostare il codice è meglio dal punto di vista della sicurezza e della dimensione.

**Dettagli sull’implementazione di Erlang**

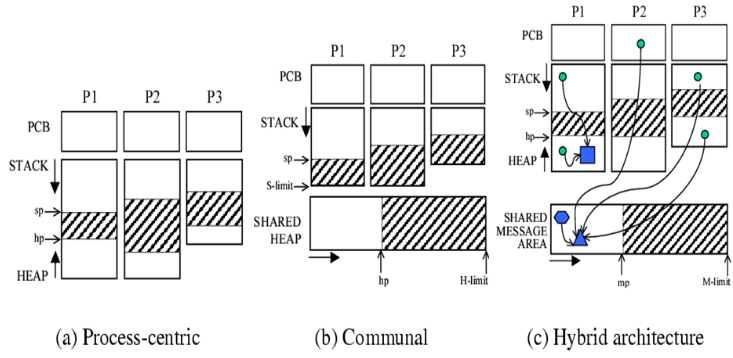
Le operazioni di send/receive tra due beam devono trasformare in parte dati in input e in parte in output, ovvero rielaborare i PID in base alla posizione del nodo (in base al fatto che sia locale o remoto).

Gli **atomi** sulla BEAM sono rappresentati come una parola sullo stack, quindi la rappresentazione interna è un numero. Durante la fase di interpretazione per associare un numero ad un atomo è possibile utilizzare il suo *hash*, questa non è la soluzione adottata da Erlang bensì da OCaml. Un problema nell’utilizzare l’hash è quello delle collisioni, ovvero due atomi che producono lo stesso hash. La BEAM può sicuramente riconoscere collisioni, tuttavia non esiste una soluzione se non utilizzare un atomo differente.

Un modo semplice per associare una numerazione ad un atomo è tramite un **valore progressivo** (primo atomo creato numero 1, ecc). Il problema però sorge quando questi numeri associati agli atomi vanno trasferiti tra due nodi. Erlang in questi casi fa sì che oltre allo scambio tra gli atomi venga inoltrata anche la tabella atomo-numero del nodo di partenza, questa operazione è costosa ma ogni volta vengono inviati solo gli atomi non già inviati in precedenza.

I PID hanno rappresentazioni diversi tra le macchine perchè devono tenere conto di come raggiungere quel nodo.

**Runtime System Architectures**

****

Erlang adotta un’**architettura ibrida**, ovvero ogni processo ha il proprio heap ma tutti condividono un’area per lo scambio dei messaggi.

Uno svantaggio della comune è che ovviamente servono dei lock e questi rallentano.

L’architettura ibrida decide quali dati mettere nella area condivisa, quelli a cui magari si accede in sola lettura.

Ogni processo legato a un core ha un lista di puntatori agli attori assegnati a quel processo.

**Erlang Term Storage**

Lo scopo è quello di implementare ***mnesia***, ovvero un database imperativo, condiviso e distribuito fra tutti i nodi di Erlang. Indipendentemente da quale nodo ci si trovi è possibile scrivere sul database e tutte le modifiche saranno visibili da tutti i nodi del cluster.

L’obiettivo è infatti rappresentare una memoria condivisa distribuita, una task molto complessa.

*mnesia* è una libreria ad alto livello che permette di creare un database distribuito (ma non SQL), l’utilizzatore può creare tabelle con colonne ed vi si salvano dei record formati da una chiave seguita da dei valori.

**Multitasking**

Il **multitasking con context switch** può essere di due tipi, **collaborativo** o **pre-emptive**.

Collaborative: il codice che scrivete è predisposto a passare il controllo agli altri thread. Il modo più semplice per implementarlo è tenere una lista di operazioni da fare globale e quando finiamo di fare un'operazione guardare se questa lista non è vuota e nel caso eseguire l’operazione successiva.

Il **multitasking collaborativo** (anche chiamato thread green)   
vantaggio:

estremamente rapido, c’è un solo processo in esecuzione e quando queste deve cambiare processo in esecuzione è una semplice chiamata di funzione del linguaggio. Ognuno di questi thread in esecuzione può accedere alla memoria degli altri thread in quanto tutti all’interno dello stesso processo.

svantaggi:

- tutto viene delegato al programmatore.

- potrebbe essere *unfair*in quanto sono i singoli processi a decidere quando passare l’esecuzione.

- system call bloccanti hanno la conseguenza che tutti i thread vengono sospesi.

L’opposto è un **sistema hardware** che manda un interrupt nel momento in cui termina un timer e fa conseguire il context switch.

Vantaggi (opposti al thread green):

- automatico

- fair

Svantaggi:

- lento

Cosa utilizza Erlang?

In Erlang, così come altri linguaggi pensati per la concorrenza, utilizza un meccanismo di thread green (**multitasking collaborativo**), in cui vi è un solo processo per core ed è l’attore che passa il controllo agli altri processi. La BEAM ogni tot di istruzioni eseguite da parte di un attore lo stoppa e fa eseguire del codice ad un altro attore. Tutto questo è possibile perché è presente una virtual machine: la BEAM appunto.

Mettendo questo controllo nella BEAM abbiamo i vantaggi dei due modelli.

Di contro questa situazione è possibile solo perché abbiamo una virtual machine.

**Blockchain**

Blockchain è una buzzword alquanto abusata.

Una **blockchain** è una possibile implementazione distribuita (**peer-to-peer**) di un **ledger**.

Un ledger è la trasposizione elettronica di un archivio notarile: è una struttura dati astratta, simile a uno **stack**, le cui uniche operazioni sono **empty**, **push** e **iter** (in sola lettura).

Le celle di un ledger contengono sequenze arbitrarie di bit, chiamate **transazioni**.

Esempi di transazioni: (descrizioni di) movimenti di beni, comprese criptovalute, da un soggetto A a un soggetto B; descrizioni di eventi quali passaggi in una catena produttiva; stati e transazioni di uno smart contract.

Uno **smart contract** o **distributed application** (DApp), è un attore il cui stato quiescente, codice incluso, è memorizzato sulla blockchain. Anche l'invio di un messaggio (chiamata di un metodo in certi linguaggi) è memorizzata sulla blockchain insieme al nuovo stato che viene calcolato al momento della push.

Gli smart contracts vengono usati per regolare le transizioni fra umani in quanto possono registrare trasferimenti di denaro/beni e in quanto automatizzano certe operazioni a seguito di certi messaggi. Gli umani possono fidarsi di cosa accadrà a seguito, per esempio, di trasferimento di denaro in quanto hanno la possibilità di leggere il codice perchè è di libero accesso.

Si potrebbe implementare un ledger come un singolo programma che diventa trusted ma un single point of failure ulteriormente più facile soggetto di attacchi informatici (hackers) ed economici (mazzette o ricatti finalizzati alla modifica dei dati).

Una blockchain è invece una implementazione peer-to-peer di un ledger.

Il problema della **consistenza** delle copie del ledger è **indecidibile**. La soluzione consiste nell'indebolire l'ipotesi di consistenza (le copie possono differire) e quella di **immutabilità** del ledger (devo poter cambiare i ledger per rimetterli in sync qualora mi accorga di discrepanze).

**Consistenza/immutabilità parziale**: sigarantisce che le parti più profonde del ledger (quelle più antiche) abbiano una probabilità molto bassa di essere fuori sync e di essere quindi mutate. Non bisogna quindi fidarsi delle transazioni memorizzate troppo in superficie perchè ancora soggette a cambiamenti.

Un **fork** è una situazione in cui un nodo si accorge che la sua copia del ledger differisce da una copia comunicata (tramite un protocollo di **gossiping**) da un altro nodo

Il fork viene risolto scegliendo **la catena più lunga**.

Gli algoritmi di gossiping sono push, quando ricevo una modifica la distribuisco agli altri.

Un secondo protocollo di gossiping viene usato per distribuire sulla rete le transazioni da includere via push.

Bisogna evitare che un nodo disonesto modifichi il ledger alterando una transazione, allungando la catena (magari con transazioni fasulle) e distribuendo agli altri nodi la nuova catena molto lunga.

Per farlo aggiungere una nuova cella al ledger deve essere un'operazione **costosa** in modo che non sia conveniente per un nodo disonesto modificare il passato remoto.

Soluzione: le transizioni sono accorpate in **blocchi** che sono le vere celle del ledger; ogni blocco include, oltre a un certo numero di transazioni, anche un puntatore al nodo precedente sotto forma di **hash** di quel nodo, in modo tale che alterare un blocco richieda alterare tutti i successivi (**Merkel's trees**); un numero magico, da trovare tramite ricerca brute force, che sia la soluzione a un **problema complesso** che richieda tanta **potenza di calcolo** (e quindi **denaro** sotto forma di **energia elettrica**) per essere risolto, ma la cui soluzione sia facile da verificare.

Il fatto che la testa della catena possa essere modificabile vuol dire che un utente deve attendere che siano creati nuovi blocchi per fidarsi che la transazione sia definitiva. Ci sono blockchain sviluppate per fare pagamenti più veloci, ma hanno altri svantaggi. Essendo un problema indecidibile bisogna fare tradeoff.

Esempio: in bitcoin il problema consiste nel trovare un numero tale per cui la hash del blocco contenente quel numero termini per k zero dove k è la difficoltà del problema. k viene scelto in modo tale che in media venga risolto nel mondo un problema ogni 10 min circa. All'aumentare della potenza di calcolo mondiale k viene fatto crescere.

Poiché creare blocchi è **costoso**, serve fare in modo che i nodi onesti siano **incentivati** economicamente a svolgere il lavoro di mantenimento dell'infrastruttura.

Come incentivo, chi crea un blocco è autorizzato a mettere una transizione speciale che assegna al creatore **una nuova unità di criptovaluta**. Di fatto il creatore sta pagando l'unità di criptovaluta con denaro fisico (bolletta elettrica).

In questo modo è garantito che un disonesto non riesca a modificare la catena a meno che non riesca ad avere più della potenza di calcolo dei nodi onesti combinati assieme. Ovviamente più nodi disonesti potrebbero **colludere** a tale fine.

Per evitare spreco di **immani quantità di energia elettrica**, le blockchain più recenti cercano di utilizzare altri meccanismi. Per esempio, invece della **proof-of-work** (devo dimostrare di avere speso soldi per risolvere il problema crittografico) usano la **proof-of-stake** (devo immobilizzare per un certo tempo prolungato, anche giorni o mesi, una certa quantità di capitale). In questo caso il sistema funziona se i disonesti non hanno più denaro da immobilizzare della somma degli onesti.

Anche se il codice è libero, non è detto che sia comprensibile. Questo ha sollevato un po’ di problemi con i giuristi.

Qui un esempio semplice di **smart contract** scritto in Erlang: [Codice/smart\_contract.erl](https://drive.google.com/drive/u/1/folders/1I2XDnZ5p0jaGmEQOGlpsNv7XnMHUFhCq)

Qui un esempio più elaborato (in stile scommessa) di **smart contract** scritto in Erlang: [Codice/smart\_contract\_pari\_dispari.erl](https://drive.google.com/drive/u/1/folders/1I2XDnZ5p0jaGmEQOGlpsNv7XnMHUFhCq)