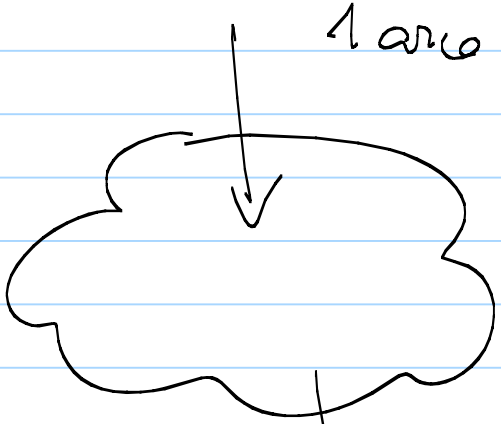
Skeleton → MDF

Note Title 16/03/2009



1 arco entrante

$(<t_1, t_2 \dots t_n>)$

struct

grafo MDF

Unità di composizione

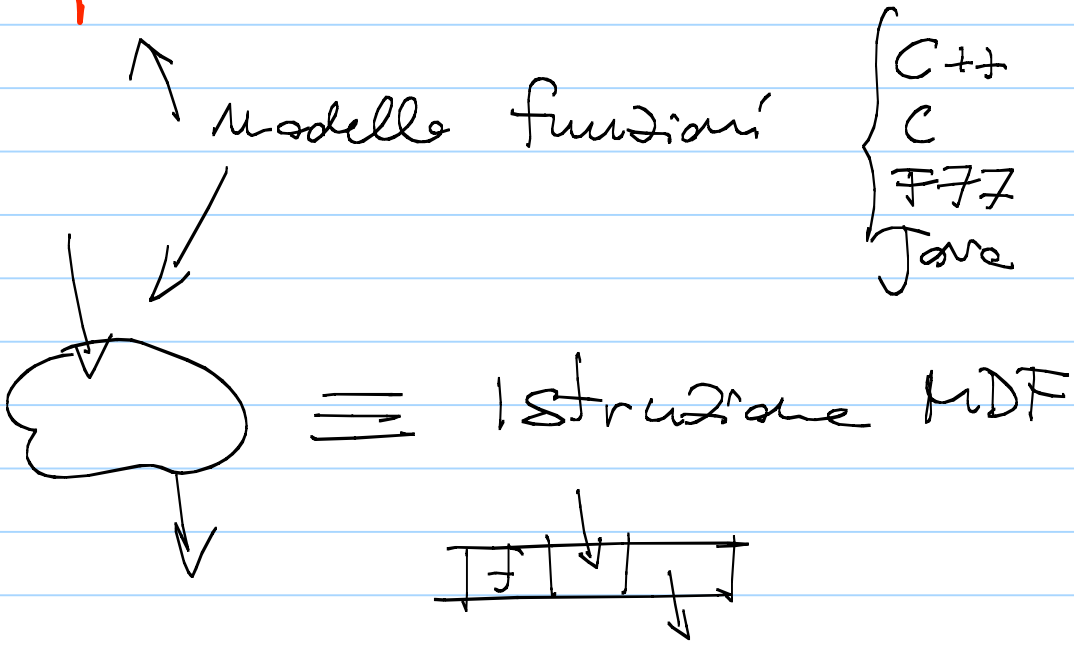
1 arco uscente

$\Rightarrow$ Sk := pipe(..) | farm(..) | ..

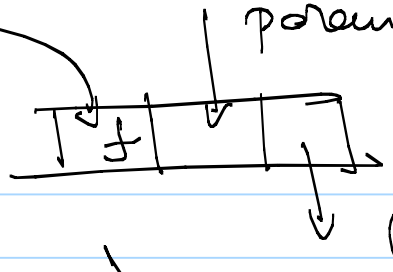
$\mathcal{C}[Sk] \rightarrow$ grafo MDF

Skeleton stream parallel (muskel)

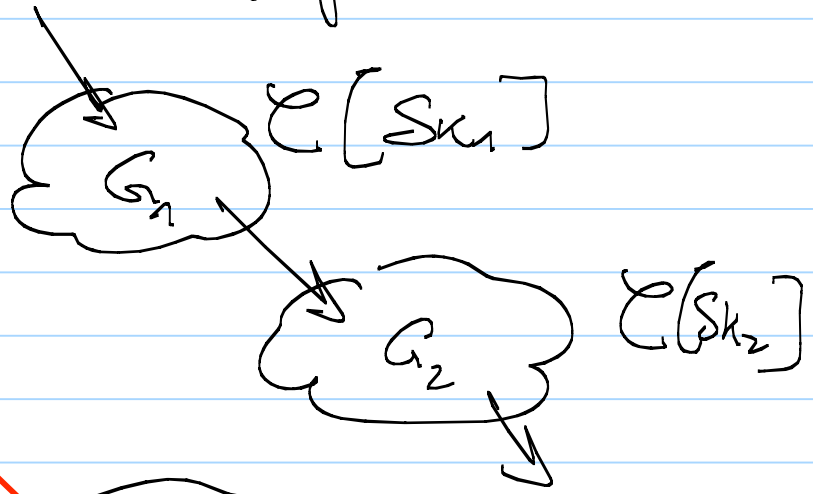
SK ::= pipe (SK, SK) | farm (SK) |
seq (....)



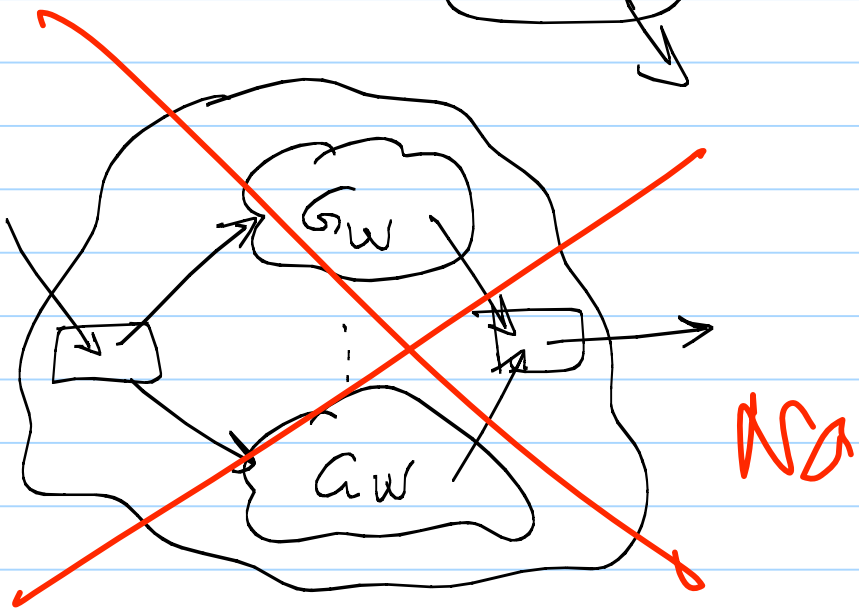
$\mathcal{C}[\![\, Seq(\cdot,\cdot) \,]\!] \equiv$



$\mathcal{C}[\![\, pipe(Sk_1, Sk_2) \,]\!] \equiv$



$\mathcal{C}[\![\, farm(Sk) \,]\!] \equiv$

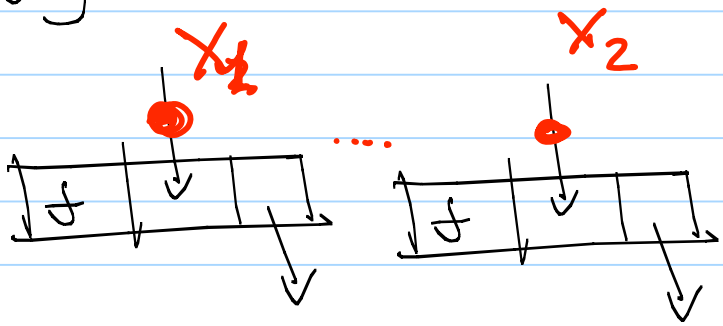


Stream parallelism

istanze multiple del grafo MDF risultato della compilazione

$$\mathcal{C}[\text{farm}(S_K)] = \mathcal{C}[S_K]$$

$\text{farm}(F) \Longrightarrow$



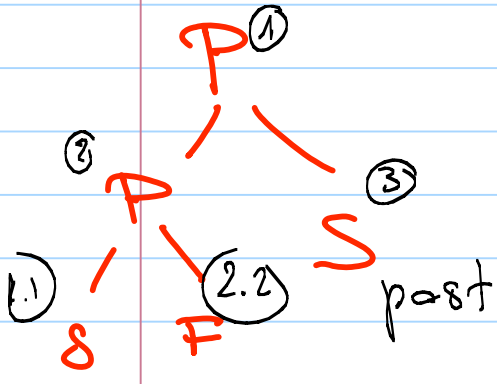
Sk_W è una skeleton

$$\mathcal{C}[farm(Sk)] = G_W$$

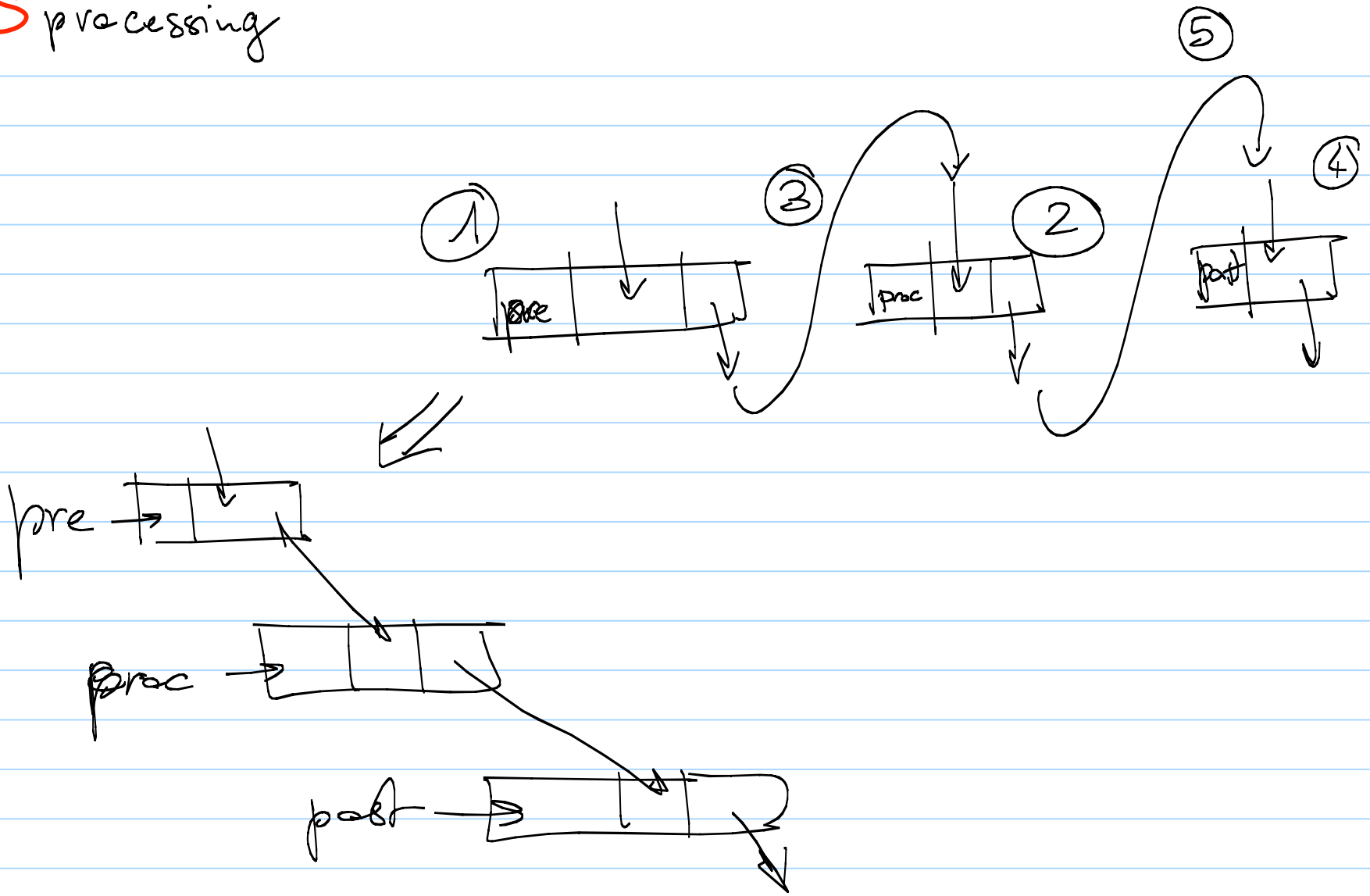
dove $G_W = \mathcal{C}[Sk]$

↑ vale × stream parallel

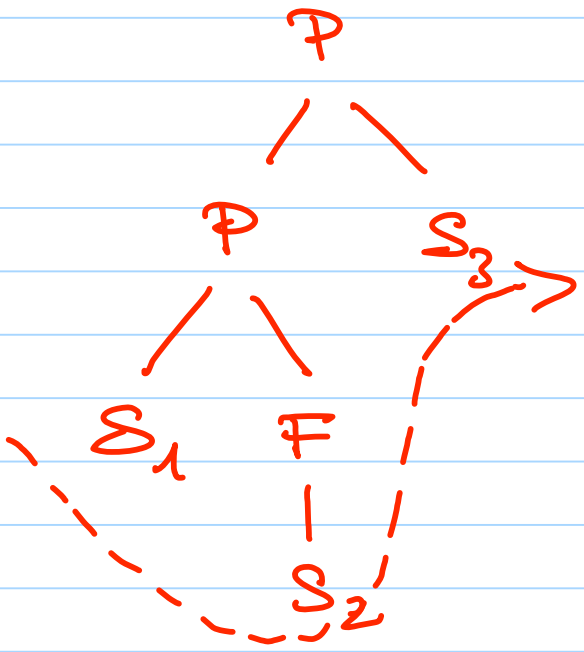
pipe (pipe (seq (pre) , farm (processing)) ,
seq (post))



pre | S processing



Catena di relazioni data flow



relazione HDF fra:
sequenziale del
programma a scheletro

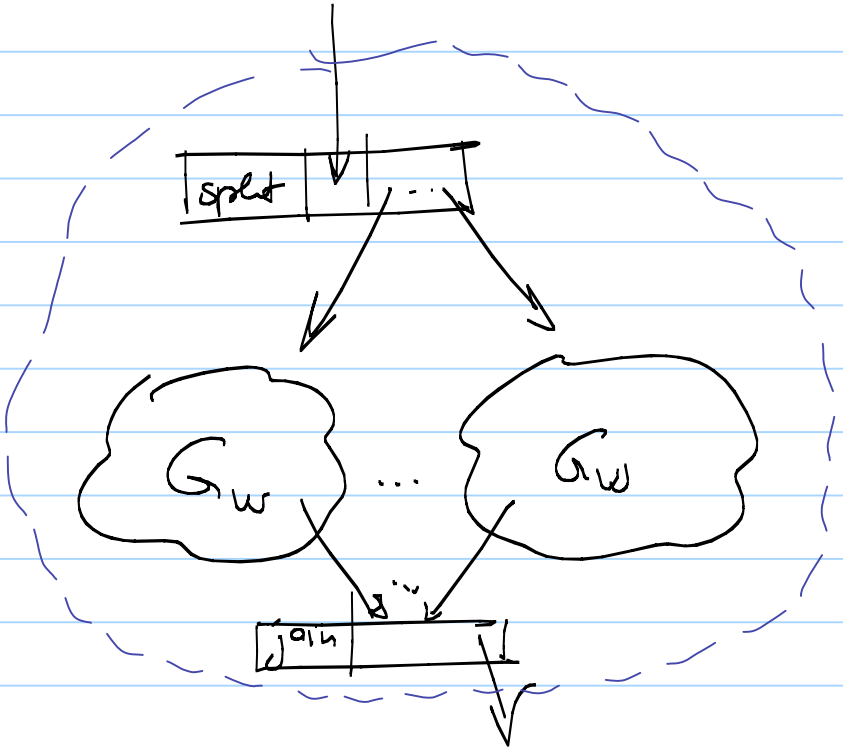
Data parallelism

$$Sk ::= \ldots \mid map(Sk) \mid reduce(Sk)$$

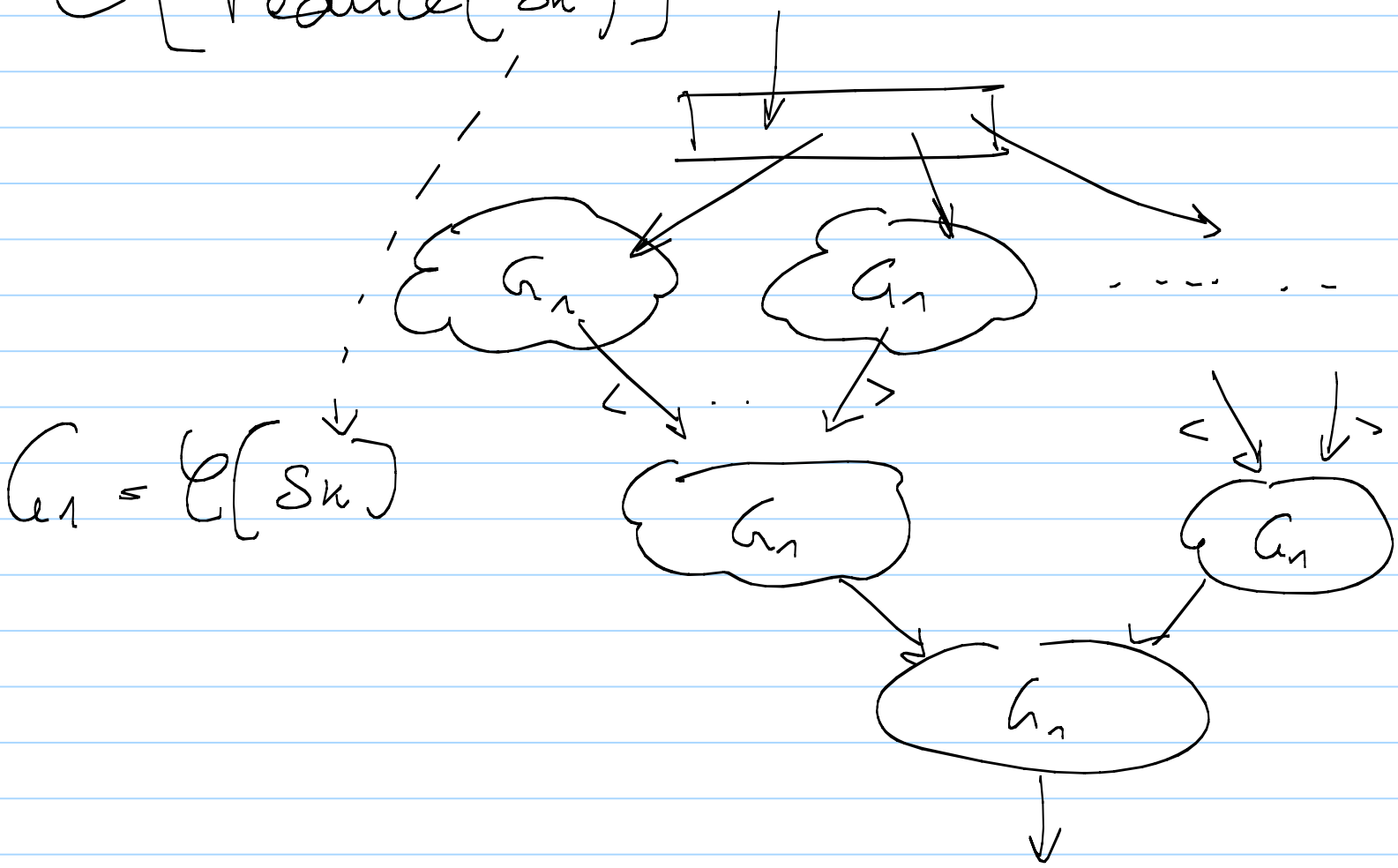
...

$$\mathcal{C}[map(Sk)] =$$

$$G_W = \mathcal{C}[Sk]$$

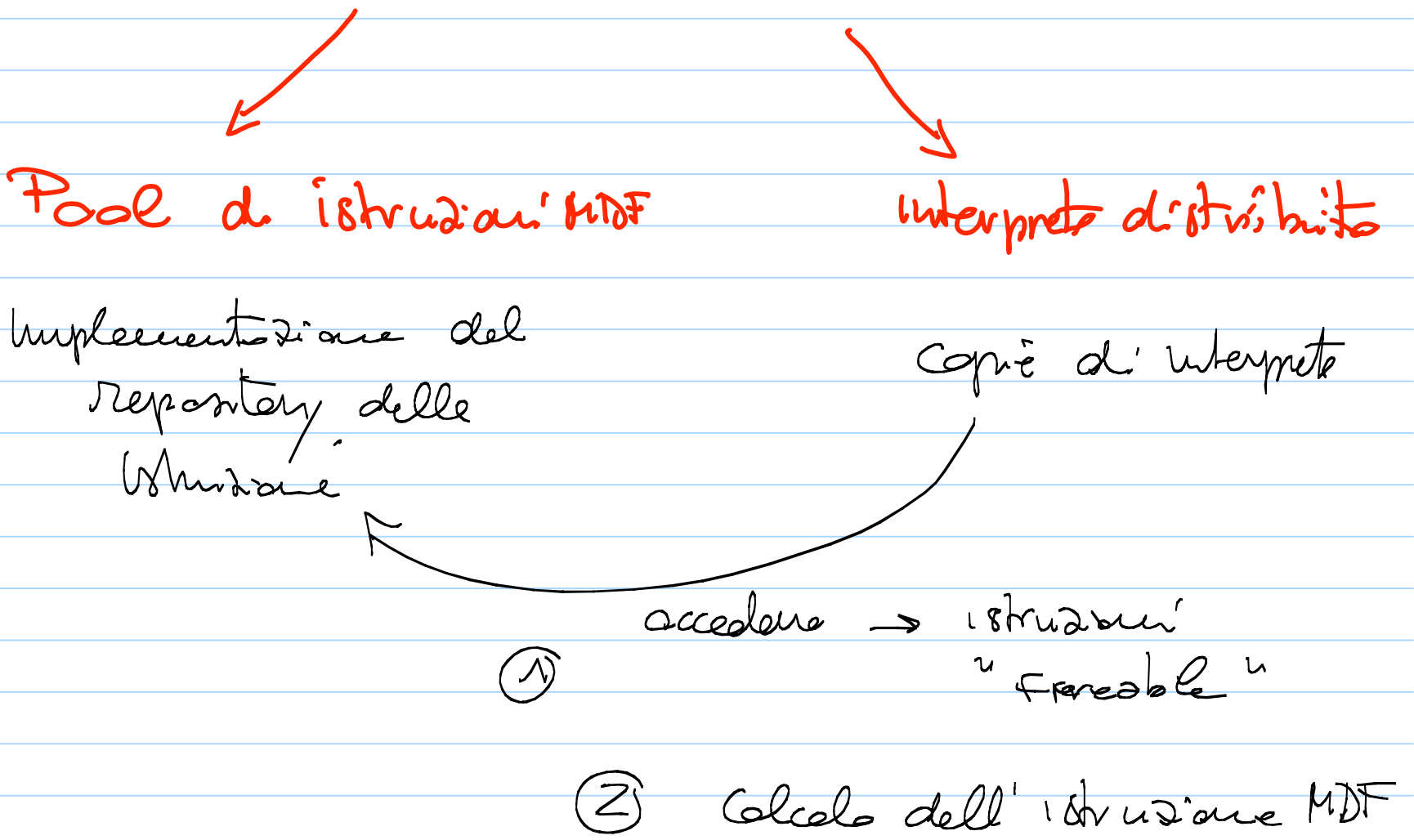


$$\mathcal{C}[\text{reduce}(S_K)] =$$



$$G_1 = \mathcal{C}[S_K]$$

Esecuzione distribuita di grafi MDF



(3) update del repository con i token generati

Repository

"Formato" delle istruzioni

- tag di grafo,
- tag di istruzione,
- id del codice,

Serve x distinguere le copie dei grafi che calcolano task diversi

Serve x distinguere le istruzioni

determina la funzione calcolata

< bit presenza, valore >*, token in ingresso

< tag di istruzione, #pos >* destinazioni

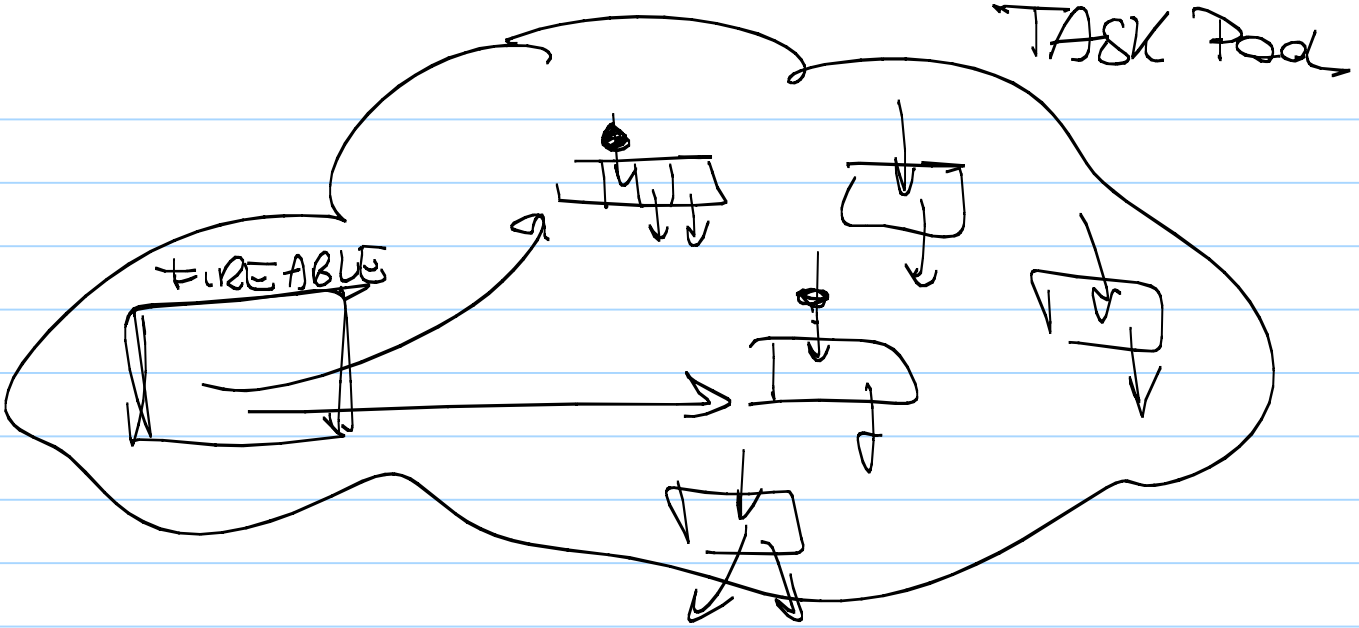
No "flag fireable" esplicito

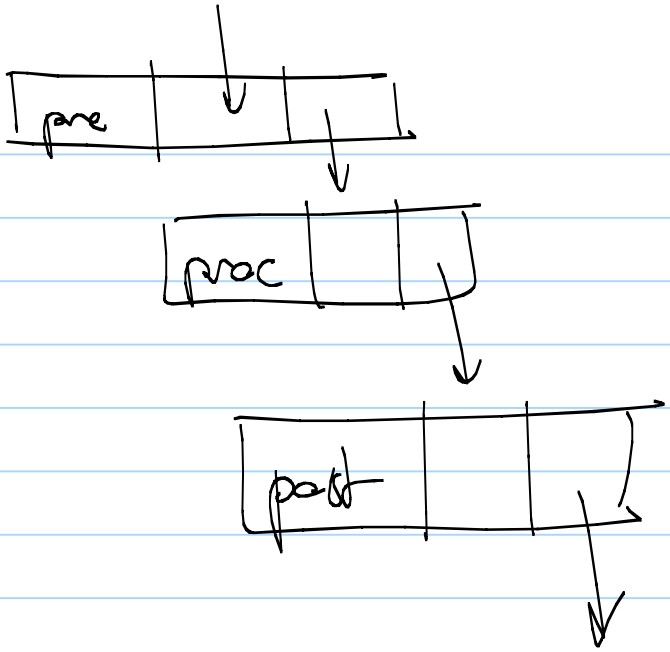
grande genera un token per ogni x

1) controlla se la rende fireable

2) se sì nella istruzione (x) nella struttura dati "istruzioni fireable"

Repository





$$MDF_i = \{ < gid_x , id_{pre} , \text{"pre"} , \langle\langle false , null \rangle\rangle , \langle\langle id_{proc} , \emptyset \rangle\rangle \; >$$

le MDF

Esempio del grafo risultato della compilazione

< gid x , id proc , "proc" , << false, null >>
<< id post , ∅ >> >

< gid x , id post , "post" , << false, null >>
<< null , ∅ >> >

}

le parti sottolineate in rosso cambiano quando istanziate una copia nel task pool

dopo l'arrivo di x_1 x_2 x_3

Repository = { $\langle x_1, id_{pre}, "pre", \langle\langle true, x_1 \rangle\rangle \ldots \rangle$

$\langle x_1, id_{proc}, \ldots$

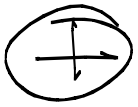
$\langle x_1, id_{post}, \ldots$

$\langle x_2, id_{pre}, "pre", \langle\langle true, x_2 \rangle\rangle \ldots$

$\langle x_2, id_{proc}, \ldots$

$\langle x_2, id_{post} \ldots$

$\langle x_3, id_{pre}, "pre", \langle\langle true, x_3 \rangle\rangle \ldots$



fireable (< <x_1, id_{pre}, @pointer>
<x_2, id_{pre}, @>
<x_3, id_{pre}, @> >)

dopo x_1 x_2 x_3

Caso 1 : arriva x_4

⟶ istanzio MDF → Repository

<x_4, id_{pre} ....
;

Cas 2 : Interprète d'une feuille instruction

↳ passe $\langle x_1, id_{proc}, @ \rangle$

sortie $\langle proc(x_1) \rangle$

↓

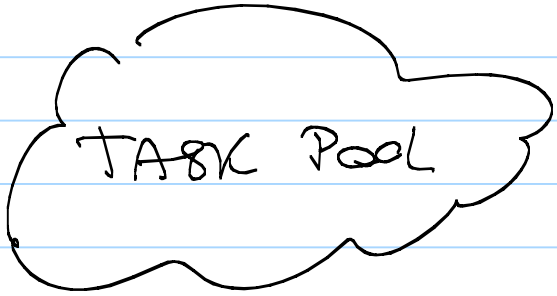
def $\langle\langle id_{proc}, 0 \rangle\rangle$

↓

in $\langle x_1, id_{proc}, \text{"proc"}, \langle\langle true, \rangle\rangle$

ne fait rien

Interpreti MDF distribuiti




```
while (true) {
    ask_fireable();
    compute();
    store_token();
}
```

ask_fireable()

↳ INT → TASK Pool è pure sincrona

TASK Pool → INT

↳ spedizione

"codice"

"parametri"

Ha bisogno di aperto

↓↓

pre-caricamento di tutti i codici che potrà eseguire

Come dati

all'inizio della compilazione di un programma

- viene letta la "f"
- la memoria inizia a cercare di codificare
- viene il "codice" (numero, identificatore) della funzione invece che la funzione

---

trattamento dei parametri (token dati in ingresso)

In caso di "mouse" in sola lettura

⇒ caching + token "riferimento" invece dei dati

push

Il TASK Pool

1) manda x_1
2) dice "riuso" quindi → cache
3) successivamente manda ref(x_1)

pop

Il TASK Pool

1) manda ref(x_1) sempre
2) INT REMOTO / DISTRIBUITO su cache miss include x_1 al TASK Pool

Ultimo passo sui "dati" trasferiti ad INTERP

→ AFFINITY SCHEDULING

Scheduler computazioni che usano i dati (token)

$\{t_1 \ldots t_n\}$

nell' INTERPRETE REMOTO

che già ha in cache

il maggior numero di $\{t_1 \ldots t_n\}$

responsabilità del TASK POOL ?

Compute()

accade poco nel loop dell'INT REMOTO

applicare codice su dati

→ meccanismi x l'accesso in cache

↙ ↓

adatt ottimizzazioni

→ meccanismi x serializzare (canonicare "indietro") i lavori rimanenti

< dato, id grafo, id istruzione, #token >

store_token();

in parte su INT REP

in parte su TASK POOL

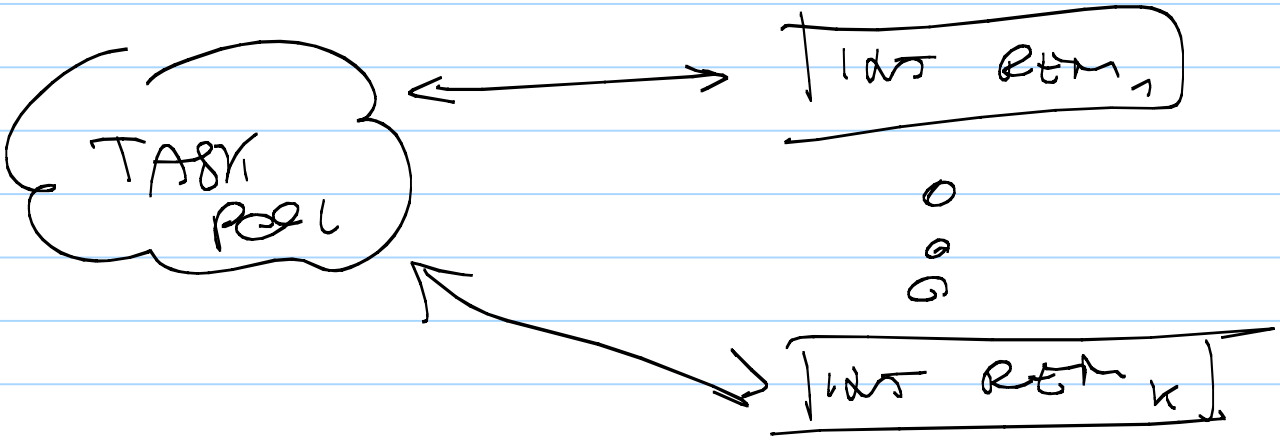
→ memorizza i token di "parte grafo"

+

assegna struttura dati "FIREABLE"

Tutte queste "specifiche" di singolo INT DISTRIBUITO / REMOTO

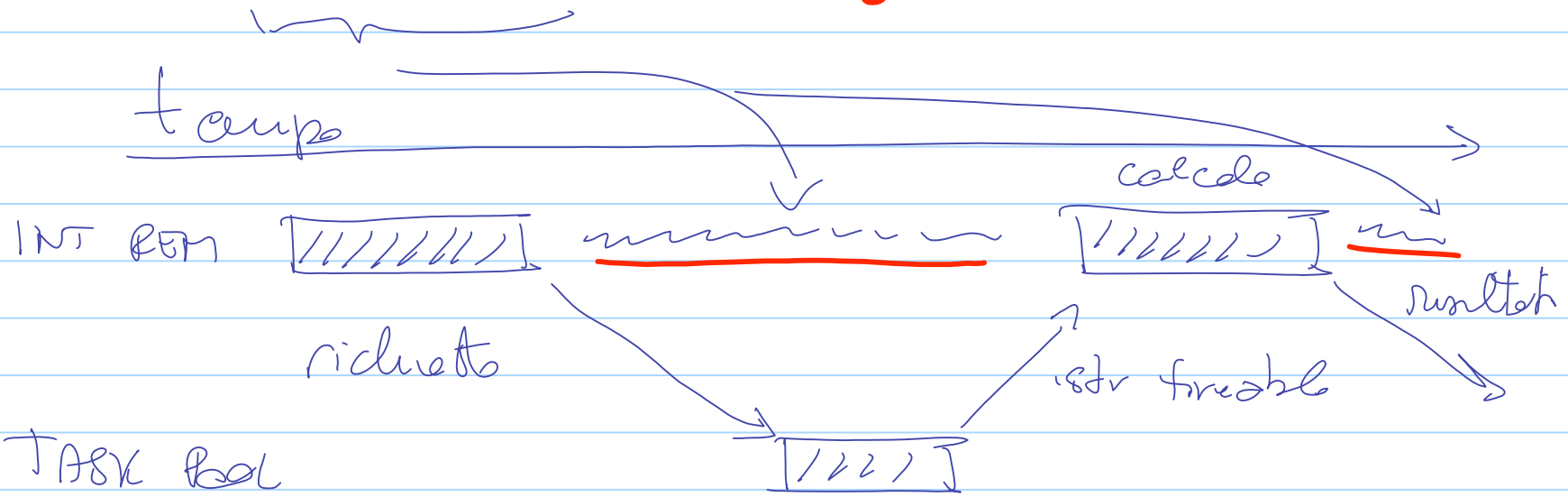
Come gestione "l'insieme" delle Copie di INT REMOTO



Considerazioni:

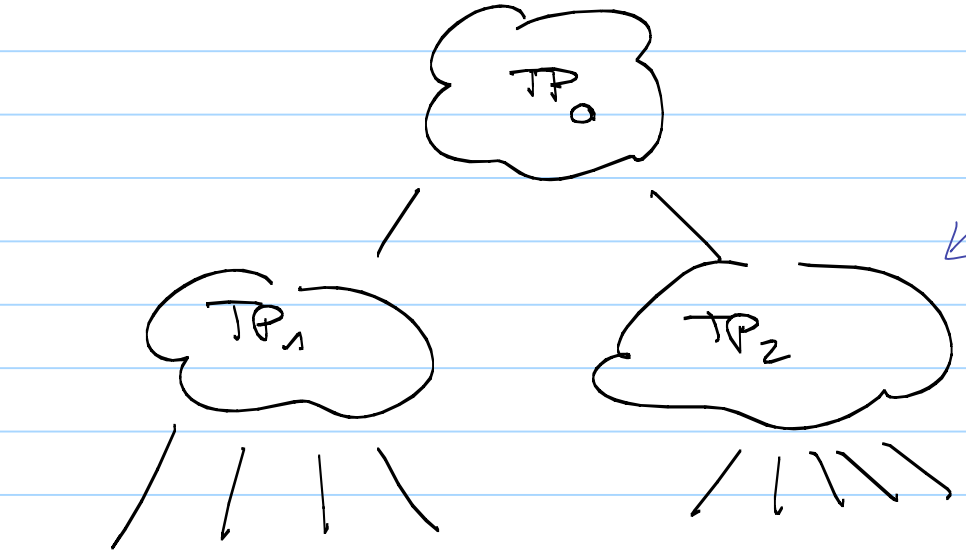
- TASK POOL ≡ coda di botteghe

- ATTESE INUTILI negli INT REMOTI



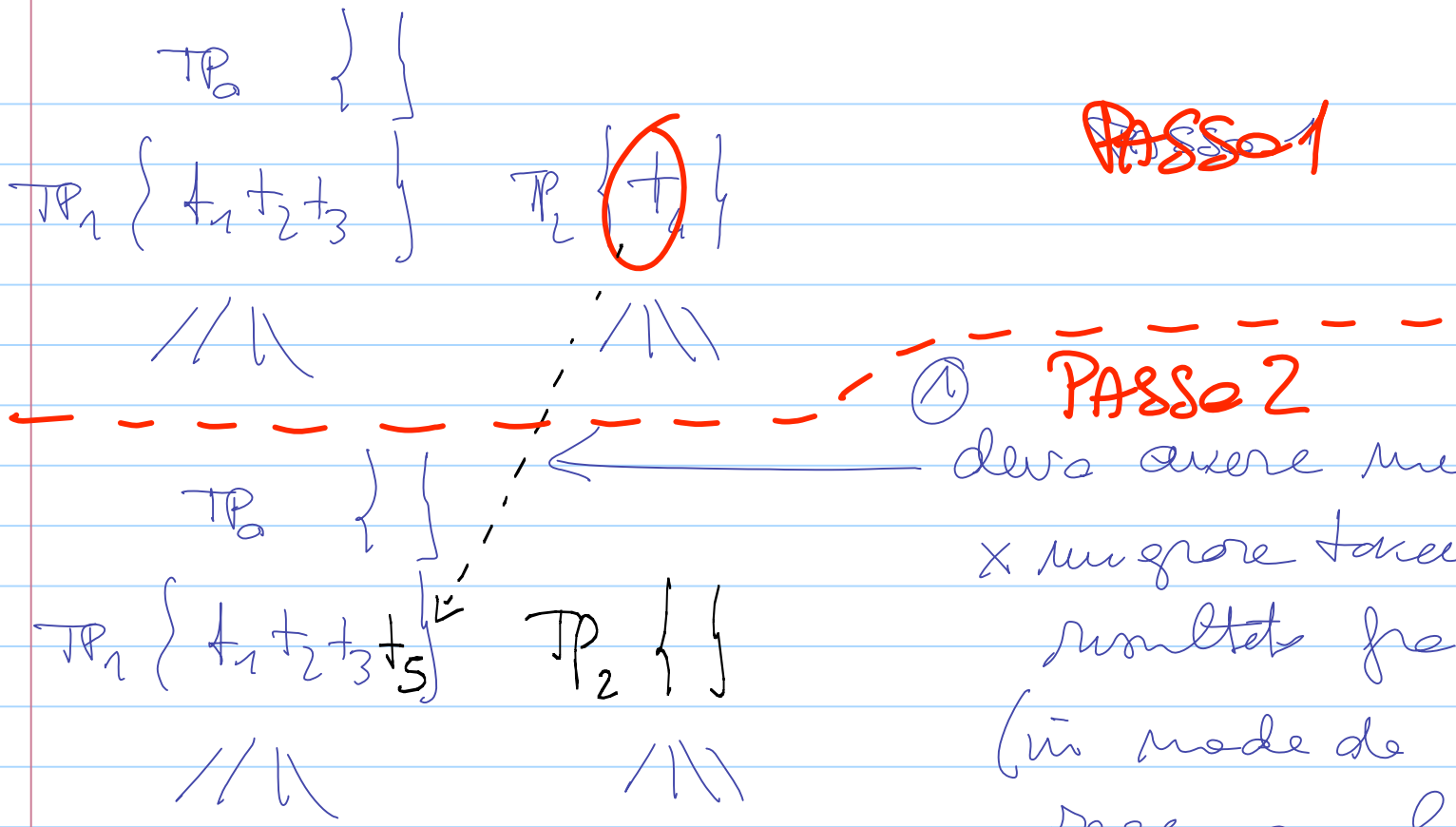
Task pool decentralizzate

a) albero



lavorano "in locale"

situazioni di "stallo" sono risolte mediante i livelli superiori



PASSO 1

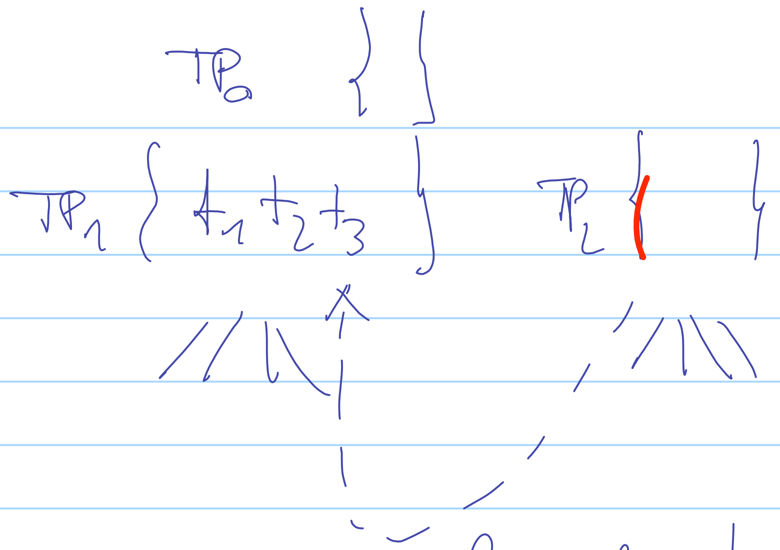
PASSO 2

deve avere meccanismi x maggiore tolleranza risultato fra TPi (in modo da scegliere la vista HDF destinazione)

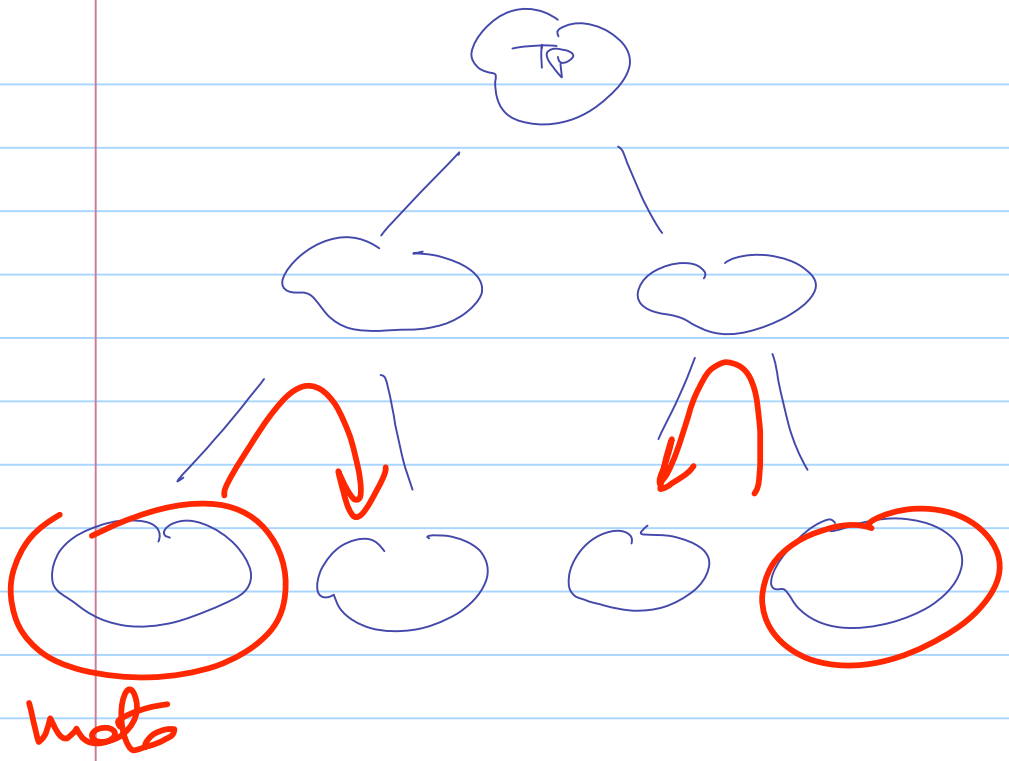
(2) deve avere meccanismi x fare load balancing

Passo 3

② nel dire che:

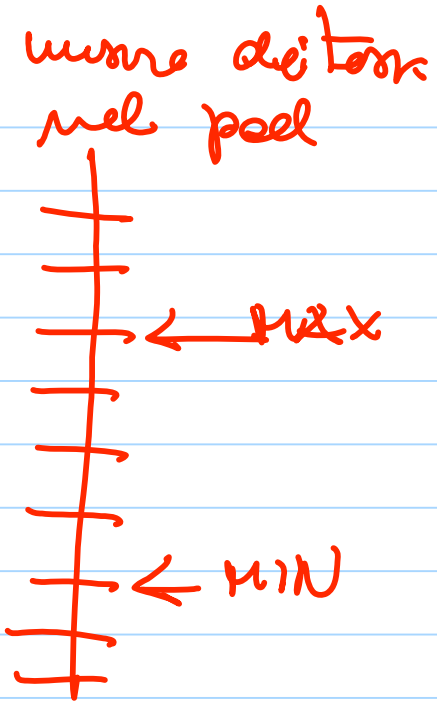


chiede test (MDF_i fireable) da calcolare



questo è meccanismo "on demand"

Meccanismi di proattività:



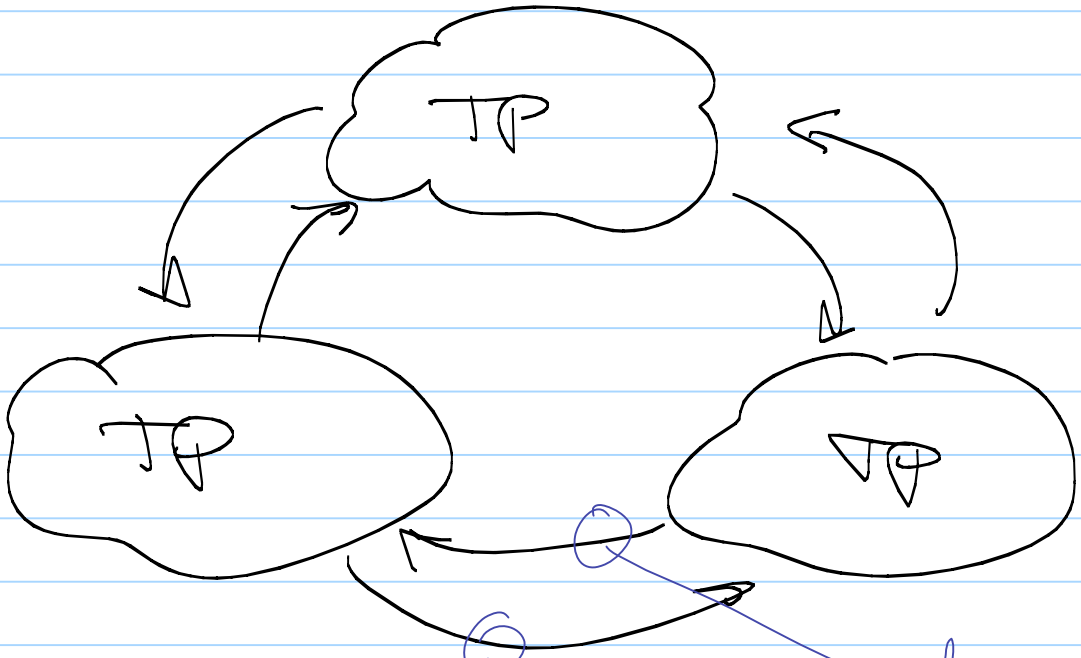
low water mark

high water mark

1) se scende sotto il MIN chiedo task

2) se salgo sopra il MAX do task da fare ad altri

Altre funzioni x il Task Pool



ANELLO

disponibilità di task

richiesta di task

Tempi di attesa degli INT. RETRY

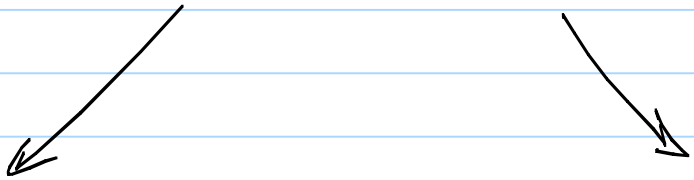
→ tecniche di "parallelismo in eccesso"

nell' INT RETRY

+ attività

↑ attende
↑ calcola

} comportamento asintotico



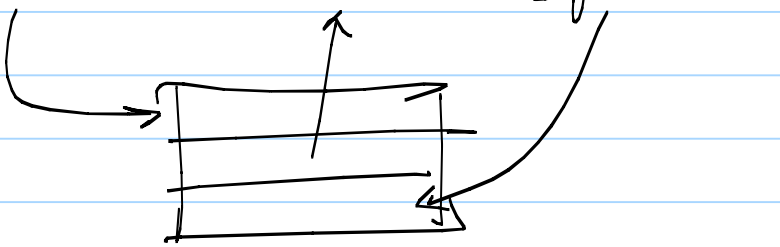
doppio o triplo buffering

alternativa: + copia dell'interpreto resto sulla stessa macchina

INT REM

thread thread thread

Riceve Calcola Spedisce



↓

affidamento sulla schedulazione