

Parallel and Distributed Systems: Paradigms and Models

AA 2018/19

**Workers: Luca Corbucci
Alessandro Berti**

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 1

Lezione 1	3
Lezione 2	9
Lezione 3	14
Lezione 4	26
Lezione 5	34
Lezione 6	44
Lezione 7	51
Lezione 8	64
Lezione 9	67
Lezione 10	76
Lezione 11	89
Lezione 12	95
Lezione 13	101
Lezione 14	111
Lezione 15	113
Lezione 16	116
Lezione 17	128
Lezione 18	132
Lezione 19	139
Lezione 20	145
Lezione 21	153
Lezione 22	164
Lezione 23	177
Lezione 24	181
Lezione 25	188
Lezione 26	193
Lezione 27	204

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 1

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 2

Lezione 28	215
Lezione 1	227
Comunicazione dei nodi	228
Stream Concept	229
Canali di comunicazione di FastFlow	229
Pattern di FastFlow (alto livello)	230
Lezione 2	236
Lezione 3	249
Lezione 4	264
Lezione 5	274
Lezione 6	289
Lezione 7	301

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 2

Lezione 1

Evoluzione delle CPU e passaggio al Multi Core

Negli ultimi anni abbiamo assistito ad una rivoluzione dell'hardware. Lo sviluppo di CPU multi core, l'avvento del cloud e la possibilità di creare macchine ad altre prestazioni sempre più potenti hanno comportato la necessità di sviluppare dei modelli di sviluppo efficienti per le architetture parallele e distribuite.

Fino all'inizio degli anni 2000 infatti si tendeva ad utilizzare macchine single core con un unico processore che di anno in anno veniva reso sempre più potente.

Secondo la legge di Moore, pubblicata nel 1965, il numero di componenti che potevano essere posizionati su un chip sarebbe raddoppiato ogni due anni circa. Aumentando il numero dei componenti sarebbe poi aumentata anche la potenza del chip.

Questo raddoppio dei componenti su un unico chip è stato portato avanti fino al 2005 circa quando si è arrivati ad un punto in cui non era più conveniente andare a produrre dei chip single core perché le performance non andavano a raddoppiarsi nel momento in cui si raddoppiava la complessità del chip ma aumentavano solamente della radice quadrata dell'aumento della complessità.

Si erano raggiunti due limiti:

- **Limite Teorico:** Mettendo sempre più componenti uno vicino all'altro comportava problemi nella dissipazione del calore
- **Limite Costruttivo:** diventa complicato andare a stampare i circuiti

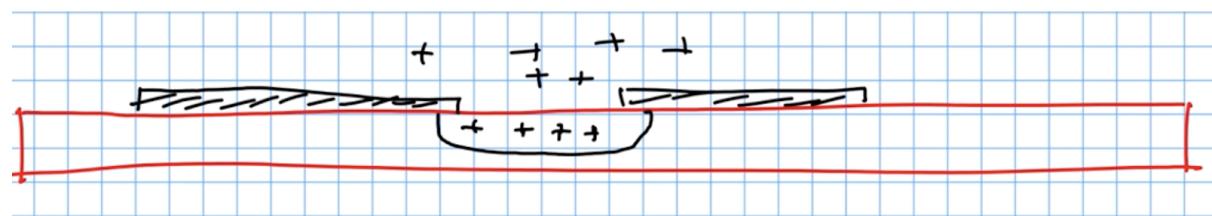
Dal 2005 in poi i produttori hanno smesso di lavorare su chip singoli e sono passati allo sviluppo di architetture multi core.

Con le architetture multi core si vanno a produrre chip più piccoli, che consumano meno e che possono dissipare meglio il calore.

Col tempo poi la sfida è diventata quella di inserire parti sempre più piccole sul chip.

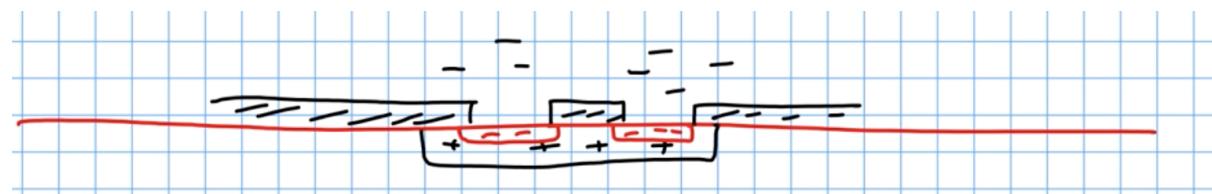
Come si produce un chip (Spiegazione molto ma molto semplificata)

Si utilizza un cristallo di silicio ovvero si prende la sabbia del mare e si eliminano le impurità andando a creare il cristallo. Questo viene poi tagliato e si ottiene un “blocco” di silicio di dimensione rettangolare. Sul silicio poi viene messo un inchiostro particolare e poi il silicio viene posizionato in un dispositivo che permette agli ioni positivi di penetrare all’interno del silicio nei punti in cui non abbiamo l’inchiostro. Poi viene tolto l’inchiostro e ottengo una parte di silicio con cariche positive di ioni nelle parti in cui non c’era l’inchiostro.

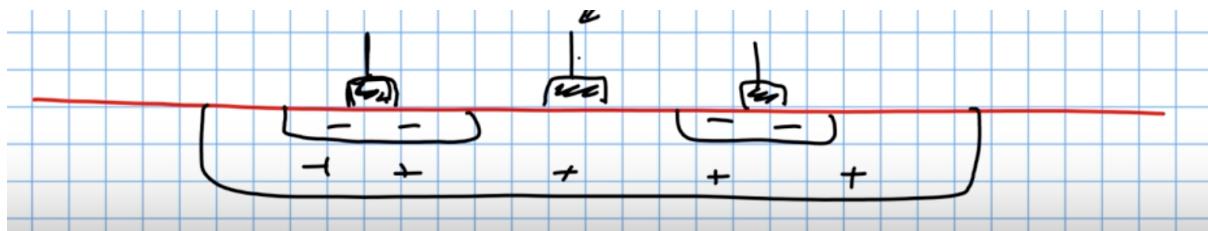


Poi si mette altro inchiostro e vengono inseriti ioni negativi, in questo modo ottengo dei blocchi con ioni negativi.

Tolgo di nuovo l’inchiostro e ottengo una zona positiva con piccole parti negative all’interno.



Quello che si fa ora è inserire sopra al silicio non solamente inchiostro ma delle parti in metallo (On/Off switch) che si comporteranno in modo differente a seconda delle cariche (positive o negative) che mettiamo sopra al metallo. A secondo della carica, il metallo permette il passaggio di “informazioni” da un blocchetto di metallo all’altro.



Ad esempio uno di questi on/off switch potrebbe essere utilizzato per creare una parte di una alu.

Quanti componenti possono essere inseriti sul chip dipende dalla distanza tra i blocchi di inchiostro.

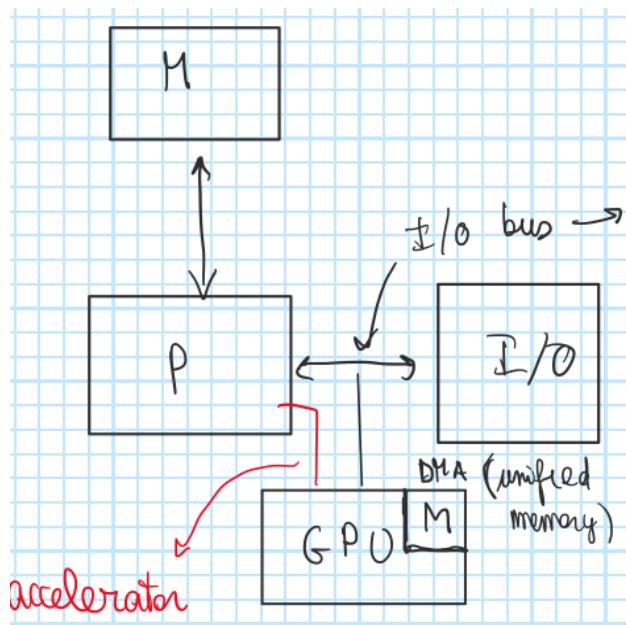
Quindi ora siamo intorno a 22nm, ci stiamo spostando verso 10nm, arrivati attorno a 6/7 nanometri non potremo più abbassare questo valore.

GPU

Dopo il passaggio al multi core c'è stato un secondo grande passo in avanti che consiste nell'utilizzo delle GPU (Graphical Processing Unit) come co-processori, in questo caso il codice che dovrebbe essere eseguito nella CPU viene spostato nella GPU e viene eseguito da questi processori.

Per lavorare con le GPU sono stati sviluppati software come OpenCL e Cuda.

All'interno dell'GPU abbiamo vari processori chiamati "Streaming Multi Processor" che a loro volta sono formati da core più piccoli e meno potenti che possono eseguire delle operazioni molto semplici in parallelo. Il risultato è che le computazioni sequenziali vengono eseguite nella CPU, i calcoli più complicati invece vengono svolti in parallelo nella GPU. Il processore dice a questi core più semplici quello che deve fare, senza indicazioni questi core non fanno nulla autonomamente.



Dato un modello come quello in figura, per programmare la GPU, almeno nella prima generazione, andavano eseguiti i seguenti step:

- Si deve preparare un kernel (CUDA o OpenCL)
- Si compila il kernel
- Vengono effettuati dei trasferimenti di dati, dal processore alla GPU spostiamo codice, comandi e dati, dalla GPU al processore invece spostiamo i risultati

Le ultime generazioni usano dispositivi DMA che implementano una memoria unificata (non è veramente unificata, ad esempio se faccio riferimento ad un certo indirizzo vado comunque in memoria principale e mi porto quello che serve nella GPU) e quindi diventa più semplice lo sviluppo con le GPU.

Dobbiamo essere attenti perchè il tempo necessario per spostare i dati alla GPU e riportarli alla CPU non deve essere maggiore del tempo che mi serve per eseguire il codice, altrimenti mi conviene farlo direttamente nella CPU senza stare a spostare i dati.

Con le GPU abbiamo anche il concetto di Accelerator che è differente rispetto a quello che succede nella CPU ed è efficiente se siamo in

grado di portare velocemente i dati nella GPU, questo perchè i vari core della GPU andranno ad eseguire il codice in parallelo.

FPGA

Oltre alle GPU un altro sistema che è stato sviluppato è chiamato FPGA ovvero Field Programmable Gate Array.

Un FPGA è una matrice che contiene milioni di celle che lavorano a circa 800 MHz.

Ognuna di queste celle può essere programmata per eseguire tre comportamenti differenti:

- Una piccola funzione binaria che prende in input da 3 a 5 bit e manda in output 1 solo bit
- 1 Bit register
- Router ovvero permette il passaggio del segnale che arriva da una funzione binaria o da un registro da nord, sud, est, ovest a nord, sud, est, ovest.

Si programmano come se stessimo scrivendo su una chiavetta USB, è possibile programmare anche qualcosa di complesso come ad esempio una ALU.

Le istruzioni che devono essere eseguite sono sulla stessa FPGA quindi non abbiamo la necessità di andare in memoria a prendere i dati che ci servono.

Gli FPGA di oggi sono formati da colonne di 2 tipi:

- DSP: Digital signal processor, questi permettono di eseguire moltiplicazioni e salvare il risultato in un accumulatore
- Memory Banks: alcuni MB di memoria

Se devo calcolare una funzione F con il DSP, facciamo il calcolo, salviamo il risultato nella Memory bank e poi possiamo calcolare una seconda funzione con il DSP.

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 8

Possiamo poi usare parte della FGPA per collegarci al bus PCI in modo che l'FPGA possa essere utilizzato come co-processore.

All'inizio era complicato programmare le FPGA perchè si usava un linguaggio RTL, poi partendo da questo file RTL si produceva un secondo file che è un Bit Stream e tutto il processo comprendeva l'utilizzo di problemi NP di cui quindi non conosciamo una soluzione efficiente. Il processo di compilazione quindi poteva richiedere anche ore.

Poi dato il bit stream, possiamo caricarlo nell'FPGA in pochi secondi.

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 8

Lezione 2

TOP 500

Quando si parla di soluzioni parallele è importante trovarne una che sia in grado di funzionare in modo efficiente ma che allo stesso tempo non vada a consumare troppa energia.

È possibile creare una classifica “pirramidale” delle macchine parallele più potenti ed efficienti, in cima alla piramide troviamo le macchine che sono elencate nel sito top500.org che elenca i 500 supercomputer più potenti al mondo andando a calcolare per ognuno uno score in base al tempo necessario per eseguire dei software pensati appositamente per il test. Oltre ai 500 super computer più potenti, troviamo anche il sito green500.org che invece elenca quelli che, pur essendo molto potenti, consumano meno energia.

Scendendo nella piramide troviamo i server, poi i dispositivi che usiamo tutti i giorni come laptop e cellulari, alla base troviamo i dispositivi IOT.

Esempi di lavori paralleli

Un primo esempio preso dalla vita reale che possiamo considerare per vedere meglio il funzionamento dei sistemi paralleli è la traduzione di un libro.

Se prendo il libro e lo traduco da solo mi servirà un tempo t , posso però dividere in due il libro e farlo tradurre a due persone, in questo caso il tempo necessario si dimezza. Possiamo dividere in tante parti il libro e, se abbiamo n persone, possiamo arrivare ad un tempo di traduzione pari a t/n . Il problema è che per dividere il libro e rimettere insieme le pagine dopo averle tradotte abbiamo un costo extra che deve essere preso in considerazione.

Se il costo extra che mi serve per dividere e rimettere insieme le pagine è maggiore rispetto al tempo necessario per la traduzione allora non mi

converrà più eseguire questo lavoro in parallelo con n persone che ci lavorano e quindi potrebbe essere meglio diminuire il valore di n.

Un altro esempio è il conteggio delle monete che le persone che stanno in una stanza hanno in tasca:

Ognuno conta le monete che ha in tasca

Si sommano tutti i totali e si ha la quantità di monete complessiva

Possiamo notare che la somma può essere fatta per gruppi di persone in modo da ridurre ancora di più il tempo necessario e rendere parallelo il lavoro, questo perchè non conta l'ordine con cui sommo i vari totali.

Rispetto all'esempio della traduzione le persone non hanno necessità di un input per iniziare il lavoro e quindi non perdiamo del tempo per dividerci il lavoro.

Google MapReduce

MapReduce è un modello di programmazione presentato da Google in un Paper¹ che ci permette di processare e generare grandi data sets. L'utilizzatore deve specificare una funzione map $f: A \rightarrow B$ che prende un input e crea una coppia $\langle key, value \rangle$.

Poi deve essere definita una funzione reduce che prende un set di coppie $\langle key, value \rangle$ e va a restituire la somma dei valori per le stesse chiavi.

Un esempio:

Se abbiamo creato con la funzione map due coppie, $\langle k,1 \rangle, \langle k,2 \rangle$, la funzione reduce calcola la somma delle coppie che hanno la stessa chiave, quindi, in questo caso restituisce $\langle k,3 \rangle$.

Ci sono molti problemi che possono essere risolti con MapReduce, ad esempio viene utilizzato anche per contare le parole presenti all'interno di un testo.

¹

<https://storage.googleapis.com/pub-tools-public-publication-data/pdf/16cb30b4b92fd4989b8619a61752a2387c6dd474.pdf>

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 10

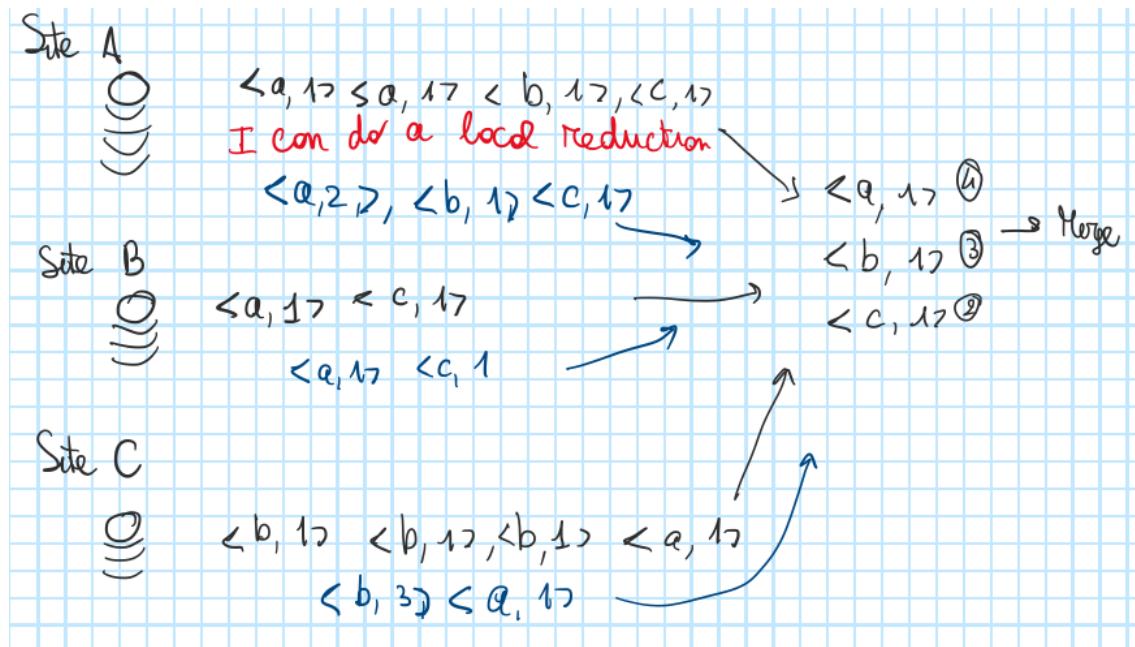
C'è una implementazione di MapReduce che, data una grande quantità di dischi su cui possiamo installare Java e Hadoop, permette di eseguire delle computazioni in parallelo. Noi decidiamo la funzione map e poi il framework decide indipendentemente come eseguire in parallelo il codice.

La ricerca delle tuple che hanno la stessa chiave viene effettuata dalla libreria, la fase di esecuzione della funzione reduce invece può essere complicata.

Se abbiamo vari cluster in giro per il mondo da cui estraiamo queste coppie <chiave, valore> alla fine dovremo trovare il modo per prendere questi dati ed eseguire il merge, questo non è un problema banale.

Quindi, prima abbiamo la fase di esecuzione della funzione Map (questa è la fase più semplice), poi una "Shuffle Phase" in cui andiamo a raggruppare le tuple con lo stesso valore (questa è la fase più complicata) e alla fine la fase di reduce in cui facciamo la somma (anche questa non è difficile).

Se ad esempio tutti quelli che hanno eseguito la fase di map inviano le tuple ad un unico computer, questo ordina le tuple e abbiamo quindi una lista di tuple ordinata, però c'è un centralization point e questo può anche limitare l'efficienza. In questo computer poi dobbiamo andare anche a fare la fase di reduce, quindi va fatta la somma e questo lo possiamo fare in parallelo.



Per migliorare l'efficienza ognuno dei Site può eseguire un ordinamento e una somma delle tuple (in blu nell'immagine) prima di spedirle. In questo modo riduco la quantità di dati che devono essere inviati e diminuisco il tempo che serve al Centralization Point per eseguire i calcoli.

Possiamo migliorare questa soluzione andando ad eliminare il Centralization Point.

Supponiamo di avere 3 Site per la ricerca che mi producono tuple ordinate e già sommate, poi abbiamo altri due site che contribuiscono a sommare le coppie con le stesse chiavi.

Quindi possiamo utilizzare una hash function che ci mappa le chiavi nei due site che vogliamo usare per i calcoli.

In questo modo sposto il lavoro che prima facevo su un solo server sui Site che calcolano l'hash e poi sui due che calcolano.

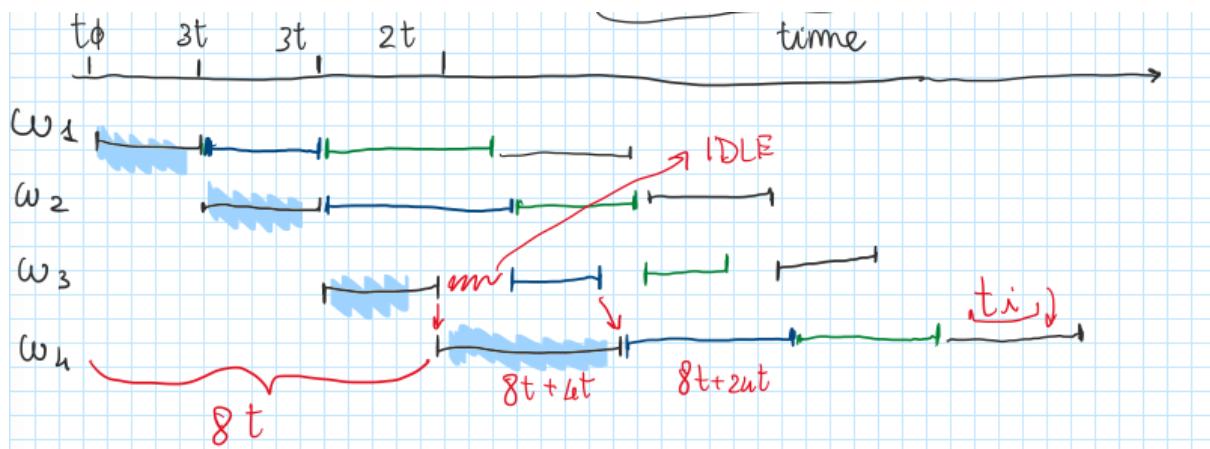
Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 13

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 13

Lezione 3

Car Building Pipeline

Consideriamo il problema di produrre una macchina andando ad utilizzare vari worker che si dividono il lavoro che deve essere effettuato.



La situazione è questa, abbiamo una pipeline in cui ogni worker lavora per un certo tempo e poi manda al worker successivo quello che ha prodotto, questo worker lavorerà a sua volta sulla macchina e passerà il lavoro svolto ad un worker successivo.

Quindi se il W_1 lavora sulla macchina attuale, il W_2 lavorerà sulla Prev e il W_3 sulla Prev(Prev).

Il problema che si pone in questo caso è che può esserci un periodo di attesa di un certo worker (in questo caso il 3) che attende che il precedente (il 2) finisca di lavorare per ottenere la macchina su cui lui deve lavorare. Questo comporta quindi uno spreco di tempo e a sua volta il W_3 , prima di inviare il pezzo, dovrà far aspettare il W_4 .

Questo è il problema del buffering che non si esaurirà mai e anzi, tenderà anche a crescere.

La somma dei vari periodi di tempo che ogni Worker necessita per svolgere il proprio task è chiamata latenza (evidenziata in celeste).

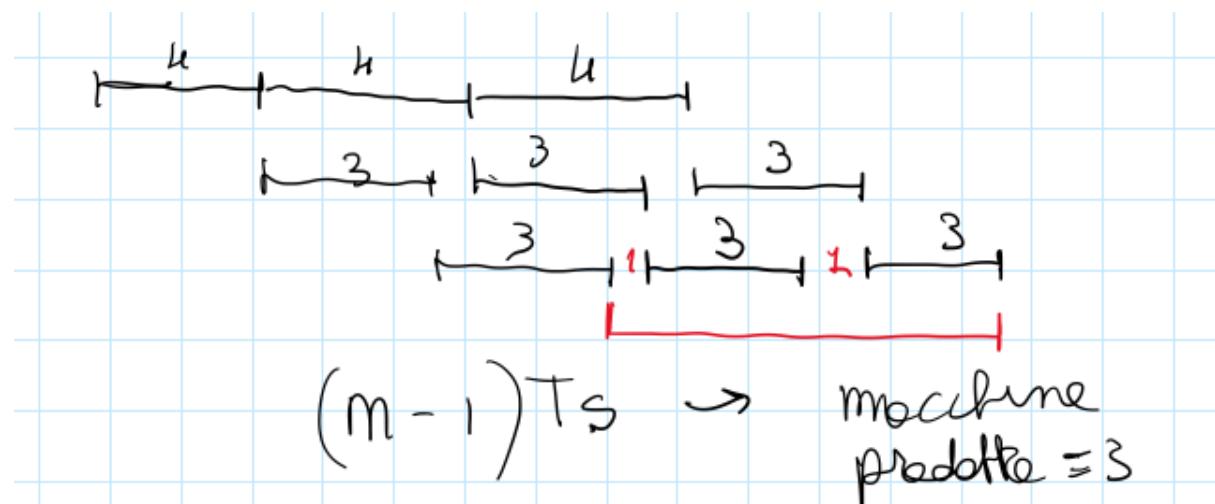
Per risolvere il problema ammettiamo che ogni worker lavori per un tempo T_i che equivale al tempo del worker più lento. In questo modo per produrre una singola macchina ci metto Latenza = $\sum T_i$. Definiamo il **service Time** come $T_s = \text{Max}\{T_i\}$ e questo mi indica il tempo di attesa che ho prima di iniziare a processare il task successivo.

Se voglio produrre m macchine con n worker abbiamo il completion Time che è definito come $\sum T_i + (m - 1)max\{T_i\}$. All'interno dell'ultima equazione il termine $\text{Max}\{T_i\}$ rappresenta il Service Time, questo viene moltiplicato per $m-1$ perchè produco m macchine ma per la prima non ho un service time perchè siamo nella fase di fill della pipeline.

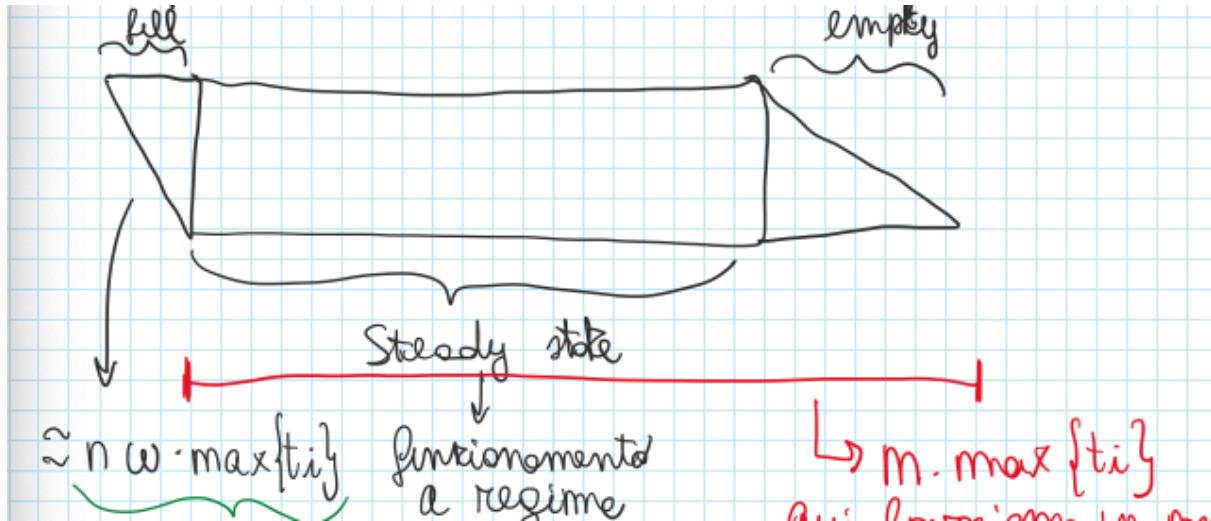
Se produco solamente una macchina latenza e completion Time coincidono e sono uguali a $\sum T_i$.

Se ad esempio produco 3 macchine abbiamo che la latenza è la somma di $4+3+3 = 10$, poi il $T_s=4$ e il completion Time è uguale a $\sum T_i + (m-1)T_s$ ovvero $= 10 + 2*4 = 18$.

La parte di tempo $(m-1)T_s$ equivale al tempo che passa tra la fine della produzione della prima macchina e la fine della produzione dell'ultima. Il completion time può essere approssimato a $m*T_s$.



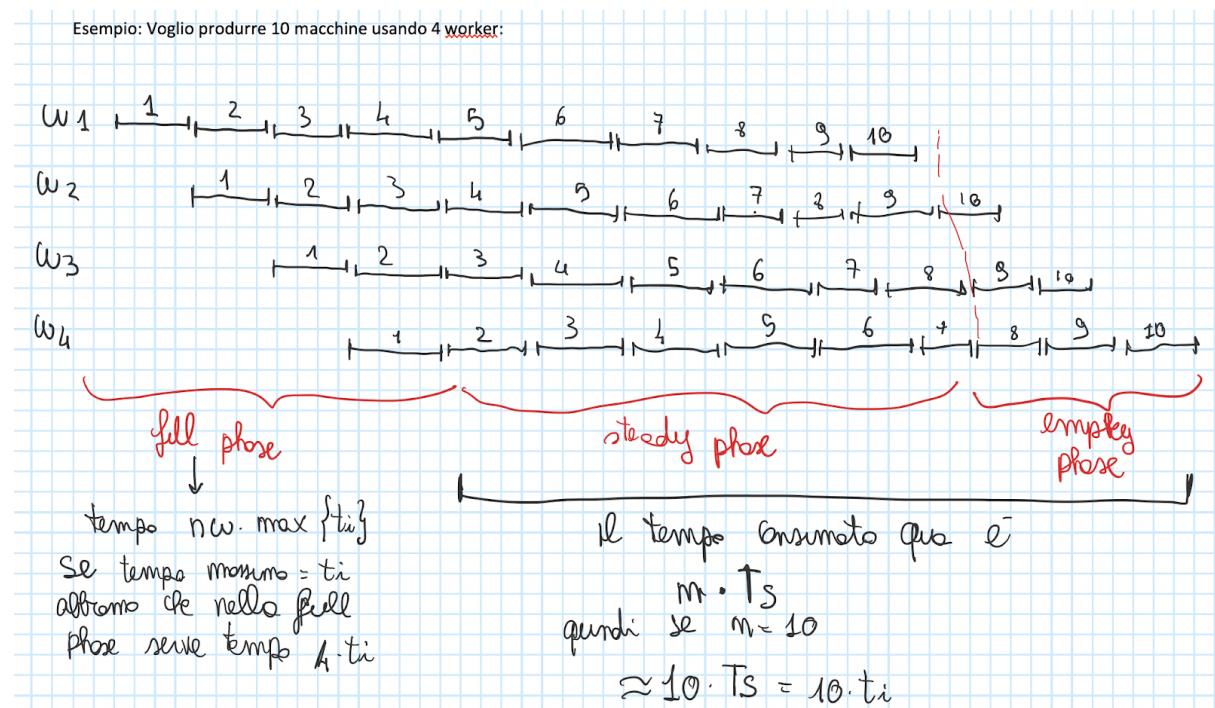
Consideriamo come esempio il caso in cui si vogliano produrre 4000 macchine, all'inizio della nostra pipeline abbiamo una fase di riempimento in cui i vari worker ottengono tutti un lavoro da fare, poi si passa in uno stato di funzionamento a regime e alla fine avremo invece una fase in cui i vari worker smettono di lavorare.



Nella fase di fill abbiamo un costo che è pari a circa $nw * \max\{t_i\}$ perché i vari worker all'inizio non hanno subito lavoro e quindi perdiamo del tempo per assegnarlo dato che per fargli arrivare un task è necessario che questo passi prima dai worker precedenti.

Nella fase di funzionamento a regime invece il tempo necessario per completare le m macchine (il completion Time) con nw worker diventa circa $m * T_s = T_c$.

Esempio di cui non sono sicuro:



Traduzione di un libro

Consideriamo di nuovo l'esempio della traduzione di un libro, in questo caso se traduco il libro da solo abbiamo un completion Time che dipende dal numero di pagine m da tradurre e dal tempo necessario per tradurre ogni singola pagina ovvero $Tc(1) = m * TranslatePage$.

Se utilizziamo nw worker la situazione cambia, in questo caso abbiamo infatti un tempo che sarà pari a $Tc(nw) = \frac{m}{nw} * Translate1Page$.

Vogliamo ora considerare il caso in cui non dobbiamo tradurre solamente un libro ma ne abbiamo da tradurre k.

In questo caso il lavoro da svolgere comprende varie fasi:

- Divisione di un singolo libro in varie parti
- Distribuzione delle parti
- Traduzione da parte dei vari worker
- Assemblaggio delle parti del libro tradotte
- A questo punto torniamo al primo punto e partiamo con un altro libro

Quindi questo vuol dire che se ho k libri allora avremo un completion

$$\text{Time pari a } Tc = k * \frac{m}{nw} * \text{Translate1Page}$$

Il problema in questo caso si presenta nel momento in cui uno dei worker ci mette più degli altri. In questo caso infatti abbiamo che se un certo worker sta ancora traducendo mentre gli altri hanno finito dovrà necessariamente attendere che lui completi prima di riassemblare il libro e poi distribuire il successivo.

Un possibile approccio alternativo al problema consiste nel non dividere i singoli libri ma nell'assegnare i libri da tradurre alle singole persone in modo da eliminare anche la fase di riassemblaggio. Questo funziona se il numero di libri da tradurre ovvero k è maggiore del numero di worker nw .

Ora se abbiamo uno studente lento possiamo tranquillamente lasciarlo lavorare mentre gli altri che avranno già finito otterranno un altro libro da tradurre, alcuni alla fine avranno tradotto di più e altri di meno, però non perdo tempo ad attendere chi ci mette più tempo.

Considerando che dati k libri creiamo dei blocchi di $\frac{k}{nw}$ libri da dare ai vari worker, possiamo calcolare il completion time usando la formula:

$$Tc = \frac{k}{nw} * (\text{Translate1Book})$$

La differenza rispetto alla prima formula sta solamente nel fatto che qua non mi blocco ad aspettare quelli che ci mettono più tempo.

Riassumendo....

Quelli visti fino ad ora sono dei pattern per organizzare il lavoro, per ora abbiamo visto due tipologie:

- **Data Parallelism:** è il caso della traduzione del libro in cui il lavoro comprende una fase di split e una fase di merge:

- Ho un task singolo
- Divido in vari task
- Eseguo il sub task
- Rimetto insieme il risultato

Quello che vogliamo fare con il pattern del data parallelism è ridurre la latenza ovvero cercare di impiegare meno tempo possibile per svolgere tutto il lavoro di traduzione.

In questo caso le persone non devono essere coordinate tra loro e possono lavorare in modo indipendente.

- **Stream Parallelism:** nel caso dell'esempio della car building chain abbiamo invece varie funzioni che lavorano in modo indipendente e che modificano i dati che abbiamo in input. Qua abbiamo una vera e propria catena e siamo obbligati a coordinarci.

L'obiettivo in questo caso è la riduzione del service time, quindi vogliamo che il tempo che ci metto usando il parallelismo non sia maggiore del tempo che ci metto a fare il lavoro da solo, questo vuol dire che il worker non deve attendere inutilmente del tempo prima di iniziare a lavorare.

Vogliamo che le macchine successive alla prima impieghino un tempo minore per essere prodotte rispetto al tempo che impiegherebbe una singola persona.

Pratica

Thread: un thread è un flusso indipendente all'interno di un programma e il programma è un processo che viene eseguito in una CPU.

I thread sono eseguiti condividendo la memoria a cui possono accedere.

In C++ per utilizzare i thread va inclusa la libreria thread:

```
#include <thread>
```

Il vecchio modo che si utilizzava per la creazione dei thread consiste nell'uso di pthread e in particolare della chiamata a pthread_create.

Ora in C++ si utilizza la chiamata a:

new thread(function name, argv1,...).

Con questo nuovo metodo andiamo a creare il nuovo thread, viene passata come parametro la funzione che dovrà essere eseguita dal thread con i vari parametri da passare alla funzione.

Un primo esempio di come funzionano i thread in C++:

```
#include <iostream>
#include <thread>

void body1(int n){

    for(int i=0;i<n;i++){
        std::cout << "This is thread n. ";
        std::cout << n << std::endl;
    }

    return;
}

int main(int argc, char * argv[]){
    if (argc == 1)
    {
        std::cout << "Serve un parametro" << std::endl;
        return 0;
    }

    int nThread = atoi(argv[1]);

    // Creo il vettore con i thread ID
    std::thread * t[nThread];

    std::cout << "Running " << nThread << " Threads" << std::endl;

    // Assegno ad ogni posizione del vettore un nuovo thread
    // ogni thread ha come parametro La funzione body che voglio
    // che venga eseguita e poi viene passato come parametro i
    for(int i=0;i<nThread;i++){
        std::cout << "Starting Thread " << i << std::endl;
        t[i] = new std::thread(body1,i);
    }
}
```

```
}

// Serve la join per fare in modo che si aspetti
// la fine di ogni thread.

for (int i = 0; i < nThread; i++)
{
    //Serve la -> perchè è un puntatore
    t[i]->join();

}

return 0;
}
```

Possiamo fare un primo esempio di thread in C++ considerando la traduzione del libro, assumiamo di swappare i caratteri da minusconi a maiuscolo e viceversa invece di fare la traduzione, introduciamo inoltre una attesa attiva per fare in modo di simulare meglio il funzionamento dei thread.

È possibile risolvere il problema in modo sequenziale andando a leggere il contenuto di un file in una stringa e poi andando a modificare di carattere in carattere.

In parallelo dobbiamo necessariamente suddividere la stringa in vari range, senza però copiare il contenuto della stringa, questo perchè i thread vengono eseguiti con una memoria in comune.

Se ho la stringa con il testo e la divido in due assegnando il lavoro ai due thread, impiegherò circa la metà del tempo rispetto al caso sequenziale, contando anche che devo dividere le stringhe.

Labda function in C++: per definire una labda function in C++:

- Si usa come tipo della funzione auto e non ne viene specificato uno come nelle normali funzioni
- Mettiamo il nome della funzione seguito da un uguale

- Poi mettiamo tra parentesi tonde il parametro da passare alla funzione e tra parentesi quadrate il parametro che viene preso dall'ambiente in cui ci troviamo.

Per calcolare il tempo utilizziamo la libreria chrono, poi andiamo a calcolare il tempo all'inizio e alla fine dell'esecuzione.

Un'alternativa consiste nella creazione di un oggetto che possiamo utilizzare proprio per calcolare il tempo passato, quando l'oggetto viene creato inizializziamo il tempo e quando viene distrutto calcoliamo il tempo attuale e lo sottraiamo al tempo di partenza calcolando il tempo trascorso.

Questo è possibile perchè se mettiamo il codice all'interno di parentesi graffe, quando chiudiamo la graffa l'oggetto verrà distrutto e quindi potremo calcolare il tempo passato senza difficoltà.

Profiling Tool: se vogliamo capire quali parti del codice prendono una maggiore quantità di tempo per essere eseguite si può usare un tool di profiling come ad esempio gProf.

Per usarlo compiliamo il codice con -g per il debugging e con -pg per il profiling. Quando eseguo l'eseguibile verrà prodotto un file con il risultato e un file di output di gProf che mi permetterà di capire quali parti del codice necessitano di un tempo maggiore per essere eseguite.

Uso dei processori: si usa il comando htop che oltre a mostrare i vari processi in esecuzione mostra anche quali sono i core che stanno funzionando di più o di meno.

Codice del traduttore:

Libreria esterna con attesa attiva e “traduttore”

```
void active_delay(int msecs){  
    auto start = std::chrono::high_resolution_clock::now();  
    auto end = false;
```

```
while(!end){
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    auto msec =
std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
    if(msec>msecs)
        end = true;
}
return;
}

auto translate_char(char c){
    active_delay(1);
    if(islower(c))
        return toupper(c);
    else
        return tolower(c);
}
```

Main con scelta tra sequenziale e parallelo:

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <chrono>
#include <ctime>
#include <vector>
#include <thread>

#include "Translator.hpp"

using namespace std;

typedef struct{
    int start;
    int end;
} RANGE;

int main(int argc, char * argv[]){

    if (argc == 1){
        std::cout << "Serve un parametro" << std::endl;
        return 0;
    }

    string filename = argv[1];

    // Per prima cosa viene aperto il file, poi creiamo text in cui
    // salviamo il testo e poi abbiamo Line che verrà usata per
```

```
// accumulare il testo che leggo prima di appenderlo a text.
ifstream fd(filename);
string text = "";
string line;

while(getline(fd,line)){
    text.append(line);
    text.append("\n");
}
fd.close();

// primo caso, vogliamo eseguire la traduzione sequenziale
if(argc == 2){
    auto start = std::chrono::high_resolution_clock::now();

    // la funzione transform prende come parametro un range, nel nostro caso begin e
end
    // poi un secondo punto in cui verrà salvato l'output, nel nostro caso questo è
il
    // secondo begin. Poi prende una funzione che prende in input i vari elementi del
range
    transform(text.begin(), text.end(), text.begin(), translate_char);
    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    auto usec =
std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
    std::cout << "Spent " << usec << " usec to translate " << filename << "
Sequentially" << endl;
}
// In questo caso vogliamo considerare una esecuzione in parallelo della traduzione
else{
    // Quando avviamo dobbiamo indicare quanto vogliamo andare in parallelo
    int nw = atoi(argv[2]);

    /*
        Definizione di una Lambda in C++.
        Non deve essere indicato il tipo della funzione quindi mettiamo auto
        poi mettiamo il nome della funzione, dopo l'uguale abbiamo tra le parentesi
        tonde il parametro che deve essere indicato quando chiamo la funzione
        invece tra le parentesi quadrate c'è un parametro che viene preso
dall'ambiente
        in questo caso si tratta della variabile text che viene passata come
riferimento.
        Nel body della Lambda abbiamo il for che lavora su text ed effettua la
modifica al testo
            chiamando la funzione definita in Translator.hpp.

        Nel nostro caso la funzione compute_chunk riceve un range di testo su cui
lavorare e
            modifica quello specifico testo. L'idea è di chiamarla in parallelo da vari
thread in modo
            da effettuare le modifiche in parallelo.
    */
}
```

```
auto compute_chunk = [&text](RANGE range) {
    for(int i = range.start; i < range.end; i++)
    {
        text[i] = translate_char(text[i]);
    }
    return;
};

auto start = std::chrono::high_resolution_clock::now();
vector<RANGE> ranges(nw);
int m = text.size();
int delta {m/nw};
vector<thread> tids;

// L'array dei range è fatto in modo di dividere in blocchi l'array
for(int i = 0; i<nw; i++){
    ranges[i].start = i*delta;
    // Questo qua sotto è un if, prima del ? abbiamo la condizione
    // dopo le due possibili soluzioni
    ranges[i].end = (i != (nw-1) ? (i+1)*delta : m);
}

// Creiamo i vari thread inserendo all'interno del vettore i thread ID dei
// vari thread che creiamo. Quando creiamo il thread gli indichiamo la funzione
// che deve essere eseguita e poi passiamo il parametro che in questo caso è
// il range i.
for(int i=0; i<nw; i++){
    tids.push_back(thread(compute_chunk,ranges[i]));
}

for (thread & t : tids){
    t.join();
}

auto elapsed = std::chrono::high_resolution_clock::now() - start;
auto usec =
std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
std::cout << "Spent " << usec << " usec to translate " << filename << " In"
parallel with " << nw << " Threads" << endl;
}

return 0;
}
```

Lezione 4

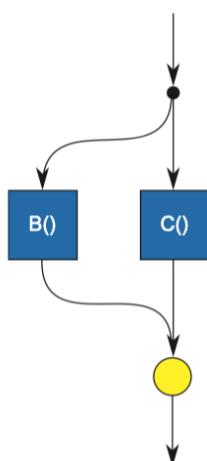
Fork-Join model

Nel modello Fork-Join il flusso del processo viene suddiviso in tanti flussi su cui successivamente viene eseguita la join in modo da fare una ricombinazione.

Ogni flusso che creiamo è indipendente e non è necessario che svolgano operazioni simili tra loro.

Dopo la join solamente un flusso continua a lavorare.

Possiamo rappresentare il modello Fork-Join con un grafo:



Le varie librerie che mi permettono di gestire al meglio la parallelizzazione dei task implementano in modi differenti il modello Fork-Join:

OpenMP è una libreria implementata in g++ e in icc, per usarla con g++ serve il flag -openmp.

OpenMP implementa la parallelizzazione dei task utilizzando un solo flag #pragma all'interno del codice.

Se ad esempio devo parallelizzare il lavoro di un for che esegue delle modifiche sugli elementi di un array, posso mettere il tag #pragma prima del for e in questo modo OpenMP andrà a

suddividere l'array in un numero di parti uguali al numero di core disponibili e poi assegnerà una parte del lavoro ad ogni core.

È essenziale che le varie operazioni eseguite all'interno del for siano indipendenti tra loro.

Un esempio considerando il traduttore:

```
#pragma omp parallel for num_threads(nw)
for(unsigned int i=0; i<text.size() ; i++ ) {
    text[i] = traduttore(text[i]);
}
```

- Cilk è un'altra libreria simile a OpenMP, in questo caso i task all'interno di un for vengono parallelizzati utilizzando una sintassi differente e andando ad utilizzare un “cilk_for” al posto del classico for.
- Un'altra alternativa è una libreria chiamata GrPPI² che ha l'obiettivo di portare nel C++ alcuni pattern delle computazioni parallele. Sfrutta OMP, TBB, i thread di C++ e FastFlow.
In questo caso ad esempio il for in parallelo viene effettuato con una map:
grppi::map(seq;....)

Performance, Portability, Programmability

Per scegliere quale di queste librerie e pattern utilizzare non è importante considerare solamente il tempo necessario all'esecuzione ma va considerata la tripla P ovvero Performance, Portabilità e Programmabilità o Produttività.

- **Performance:** il tempo può essere suddiviso in due parti, abbiamo il tempo di compilazione del codice e poi il throughput Ts.
Un concetto legato al tempo è l'energia perchè dobbiamo considerare che possiamo calcolare l'energia consumata dall'esecuzione del programma con la formula time*power.

² <https://github.com/arcosuc3m/grppi>

L'obiettivo è di produrre un codice che necessiti di poco tempo per essere eseguito ma che allo stesso tempo non consumi troppa energia.

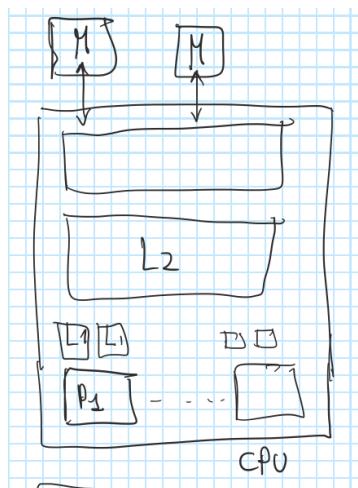
Andando a rendere parallelo un certo codice, vorremmo che l'esecuzione passasse da un tempo t ad un tempo t/n dove n è il numero di worker. In alcuni casi però ci sono dei problemi che non conviene parallelizzare e che quindi rimangono sequenziali.

- **Portability:** Le macchine sequenziali hanno tutte una struttura molto simile tra loro, sono praticamente delle "Von Neumann Machine" ovvero hanno una memoria, un processore e la possibilità di svolgere operazioni di I/O.

Tra una e l'altra può aumentare ad esempio il numero di processori ma in ogni caso spostiamo dati da e verso la memoria. Le macchine parallele invece possono avere una struttura che è molto diversa l'una dall'altra. Ad esempio possiamo avere dei cluster di macchine connesse tra loro, oppure macchine Multicore con Memoria condivisa, abbiamo le GPU e le FPGA.

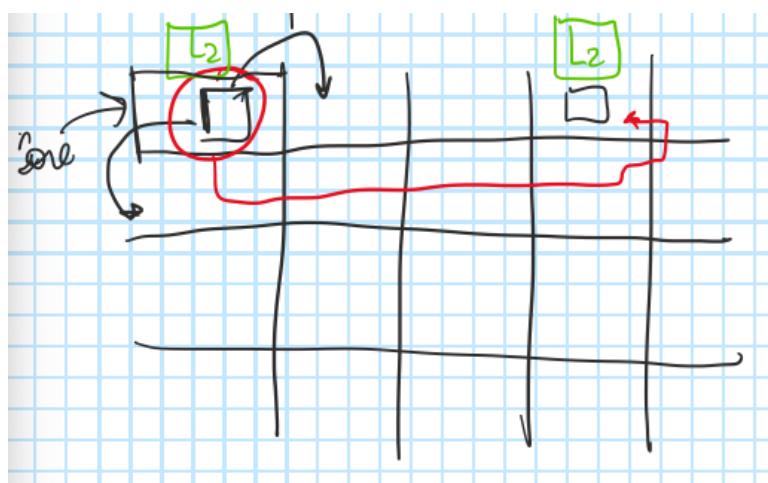
Se consideriamo solamente la categoria delle "Shared Memory Multicore" abbiamo ad esempio una macchina con una memoria principale condivisa poi ci sono delle memorie cache sempre più piccole e un processore con più di un core.

Per essere più precisi i vari core non condividono tutti i livelli della cache ma solamente dal secondo livello in poi, quindi il primo livello di cache è esclusivo per ognuno dei core. All'interno della CPU troviamo i vari core con la cache condivisa e non, all'esterno abbiamo poi la memoria principale.



Questa appena indicata è la struttura di processori come i5, i7 e anche gli Xeon, se però consideriamo un altro tipo di processore, come ad esempio lo Xeon Phi, abbiamo comunque una architettura con Shared Memory MultiCore ma è differente rispetto a quella descritta ora.

Lo Xeon Phi XNL può essere visto come una mesh a due dimensioni in cui in ognuno dei blocchi abbiamo un dispositivo collegato al bus (ognuna delle righe è un bus, quindi se mi voglio spostare dal primo blocco al quarto uso il bus e faccio solamente una operazione, è la linea rossa).



Questo dispositivo ha al suo interno un paio di core con la sua cache più una parte della cache L2 che è condivisa con il resto del

chip. Se sono nel primo blocco e cerco qualcosa che sta nella cache L2, questo qualcosa può essere nella parte di cache L2 che ho nel mio blocco oppure può essere nella cache L2 degli altri blocchi.

Questa versione è la Xeon Phi XNL, c'è anche un altro Xeon Phi che è chiamato KNC.

In questo KNC abbiamo una serie di anelli (bus) che collegano i vari core (in entrambe le direzioni). In questo modo se due core devono comunicare usiamo il bus per spostare le informazioni, tutta la comunicazione è gestita dall'hardware.

Le differenze tra i due modelli sono tante.

Se scrivo un programma parallelo cosa posso fare per garantire la portability?

Consideriamo lo XEON Phi KNL, abbiamo una mesh di core e 8 memorie condivise. Cosa succede se un thread alloca dello spazio molto grande e poi esegue il fork di 4 thread per utilizzare questa struttura dati che ha creato.

Consideriamo che il thread viene eseguito su uno dei core della mesh e nella memoria collegata inserisce la struttura dati che alloca. Supponiamo che i thread che forko vengono eseguiti da altri core della mesh, ognuno di questi 4 thread provocherà dei cache fault che porteranno poi i dati che sta cercando nella sua cache da quella dell'altro thread o dalla memoria M2. Il tempo per lo spostamento di questi dati è proporzionale alla radice quadrata del numero dei core.

Se invece prendo un processore Xeon per server o laptop abbiamo un certo bus che è collegato alle varie memorie e quindi lo spostamento dei dati verrebbe eseguito in un tempo che è costante.

Quindi vanno considerati molti aspetti se si pensa alla portability e possiamo garantirla in vari modi:

- Possiamo considerare le varie macchine su cui verrà eseguito il programma usando if e else. Questa non è una grande soluzione.
In pratica qua ricompiliamo la soluzione che troviamo in base alla macchina su cui deve girare il codice.
 - Posso scrivere in alternativa un codice che non garantisce portability e poi utilizzare dei tool che sono in grado di produrre del codice ottimizzato per le specifiche macchine che vogliamo utilizzare. Attualmente questi tool però non sono troppo ben funzionanti.
-
- **Programmability o Productivity:** questo mi indica quanto è complicato scrivere in parallelo un certo programma.
In particolare ci sono dei pattern che possono essere utilizzati e delle librerie che, implementando questi pattern, vanno a semplificare il lavoro producendo lo stesso risultato finale.
Questi pattern sono implementati in modo efficiente ed è meglio utilizzarli quando si deve risolvere un certo problema perché si evita di implementare da 0 le stesse cose che sono sicuramente implementate meglio e con meno errori.

Mechanism e Abstraction

Per cercare di ottenere le 3 P indicate fino ad ora bisogna ragionare in termini di Mechanism e Abstraction.

- Ci servono dei **meccanismi** per arrivare ad un buon risultato delle 3 P:
 - Serve un meccanismo per organizzare la computazione in parallelo di thread e processi. Abbiamo bisogno di un metodo che ci consenta di svolgere in parallelo i vari task. Quando parliamo di creare thread e processi è necessario anche essere in grado di valutare il costo del fork di thread e processi.

- Dobbiamo essere in grado di comunicare, se abbiamo vari thread che lavorano in parallelo, questi devono essere in grado di comunicare tra loro. La comunicazione può avvenire in tanti modi differenti e ognuno di questi modi ha una performance differente.

Nell'esempio della traduzione del libro per esempio dobbiamo avere una memoria condivisa perchè non è possibile che la stringa sia copiata in tutti i thread.

Oltre alla shared Memory appena citata abbiamo anche la comunicazione con Memory Passing, in questo caso abbiamo una comunicazione Point to Point con un thread che produce e uno che consuma ma possiamo avere anche una comunicazione collettiva con un set di server che lavorano e un set di receiver che ricevono i dati.

- Anche la sincronizzazione è importante perchè ci sono tanti modi per gestire la sincronizzazione e anche in questo caso abbiamo performance differenti a seconda del metodo scelto. Abbiamo la possibilità di utilizzare le lock, oppure le condition variable o le "barriers" che è un metodo di sincronizzazione collettiva.

- **Strategies:** possiamo ad esempio utilizzare un certo meccanismo per gestire una computazione, un esempio di strategy può essere ad esempio assegnare il lavoro tramite un metodo di comunicazione usando poi le barrier come metodo di sincronizzazione. Questa può essere una delle possibili strategy.
- Dal mechanism utilizzando le strategy devo fornire al programmatore delle **abstractions**.

Un esempio di astrazione ad esempio è il parallel for, svolgo un numero di task uno dopo l'altro e non hanno dipendenze tra loro, questo comporta che io possa dividere i task tra loro senza dipendenze.

Una prima strategy può essere dividere l'array in parti uguali e mandarle ai vari worker, una seconda strategy invece può essere un assegnamento in stile round robin.

Un'altra strategy ancora consiste nel creare una serie di chunk che hanno una dimensione che diventa, a mano a mano che vado avanti a dividere, sempre più piccola. In questo modo ogni thread prende un chunk, poi finisce e prende il successivo, ogni chunk prende tempo differente.

All'inizio del lavoro i thread lavorano in tempo differente ma quando arrivo alla fine il lavoro che devo fare è minore quindi le differenze tra il tempo necessario ai vari thread è minore e quindi è possibile che finiscano allo stesso tempo.

Spostandosi dal Mechanism alla strategy dobbiamo considerare anche il costo dei Mechanism che utilizziamo, ad esempio dobbiamo considerare il tempo per eseguire una join o un fork di un thread.

Se consideriamo lo XEON KNL servono circa 100 microsecondi, se invece abbiamo una macchina i5 o i7 abbiamo che questo tempo è di circa 15 microsecondi (sono range indicativi).

Se vogliamo implementare l'astrazione partendo dai Mechanism per garantire performance, portability e Programmability, dobbiamo sempre considerare questo overhead che è dato da:

- Creazione dei thread o Join dei thread
- Utilizzo della memoria, ad esempio accedo a differenti parti della memoria e a differenti cache. Inoltre la cache tra i vari core viene mantenuta “coerente” e quindi quando ne modifichiamo una devo andare ad aggiornare anche le altre degli altri core.
- Accessi I/O
- Comunicazione
- Utilizzo di funzionalità offerte dai linguaggi di programmazione, se creiamo e distruggiamo oggetti può avere delle conseguenze dal punto di vista dell'overhead perché vengono fatte modifiche alla memoria.
- Lock e Mutua Esclusione

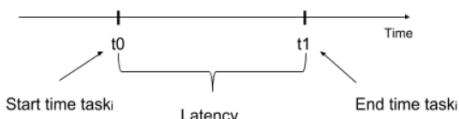
Lezione 5

Misure di Performance

Il motivo principale per cui si parallelizza un software è legato alla performance.

Il miglioramento delle performance che si può ottenere con la parallelizzazione è legato, ad esempio, alla riduzione del tempo totale necessario per eseguire un singolo task oppure alla riduzione della frequenza con cui vengono eseguiti i task.

In particolare si parla di **Latenza** per descrivere il tempo che è necessario per svolgere un singolo task.



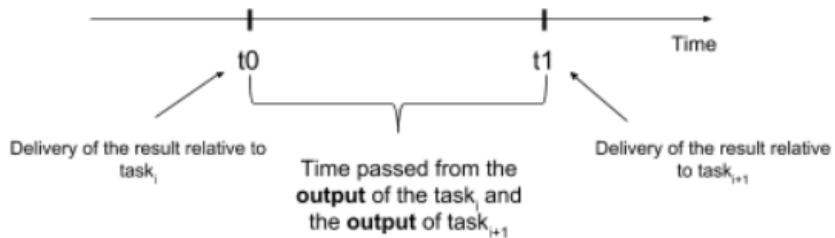
Il lavoro inizia al tempo t_0 e finisce al tempo t_1 , la latenza è rappresentata da $t_1 - t_0$.

Se invece abbiamo n task da eseguire allora consideriamo il tempo iniziale t_0 in cui iniziamo e il tempo finale t_n in cui terminiamo, indichiamo con **completion time** il tempo $t_n - t_0$ che serve per completare tutto il lavoro.

L'obiettivo è diminuire il Completion Time con una esecuzione in parallelo.

Oltre alla latenza c'è un'altra misura delle performance che è il **Throughput** ed indica il numero di task che vengono completati in ogni secondo.

Throughput = Task/Secondi.



Il Throughput diminuisce molto quando si passa ad una esecuzione parallela.

Il tempo che passa da t_0 a t_1 è definito come service time.

Un metodo per parallelizzare questo tipo di task che prendono ognuno in input l'output del precedente task è utilizzare una pipeline che è formata da vari stage. L'utilizzo della pipeline però aggiunge dell'overhead perché i vari stage devono comunicare e devono essere sincronizzati quindi si perde un po' di tempo.

Per migliorare le performance ci sono delle situazioni in cui è preferibile abbassare la latenza e altre situazioni in cui invece è preferibile alzare il Throughput.

Ci sono anche delle situazioni in cui la latenza potrebbe essere migliore nel caso sequenziale che nel parallelo, il Throughput invece sarà sempre migliore nel caso parallelo.

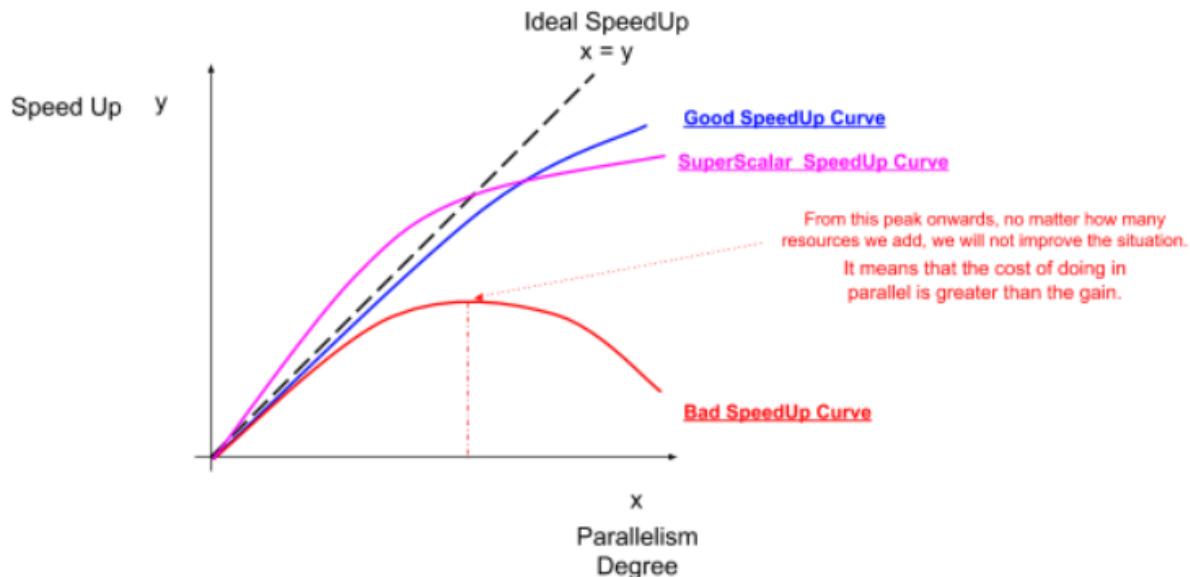
Speedup

Ci sono varie metriche che ci permettono di comprendere se la parallelizzazione ha portato gli effetti desiderati e se effettivamente l'aumento di worker ha comportato una diminuzione del completion time.

Una prima metrica è la speedup, in questo caso dividiamo il tempo necessario per l'esecuzione sequenziale per il tempo necessario per l'esecuzione in parallelo.

$$Sp = T_{seq}/T(N)$$

Nel grafico qua sotto vediamo varie possibili curve della speedup:

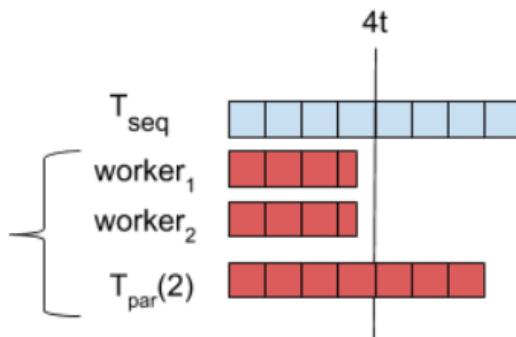


- Speedup lineare: in questo caso abbiamo la situazione migliore perchè ci troviamo ad avere un aumento delle prestazioni che è lineare rispetto al numero di worker. Se abbiamo un grado di parallelismo pari a P allora vuol dire che l'algoritmo verrà eseguito P volte più velocemente rispetto al caso sequenziale. Questa speedup viene considerata ottima.
 - La Bad Speedup Curve: ce l'abbiamo quando aumentando troppo il grado di parallelismo otteniamo delle prestazioni nettamente peggiori del caso sequenziale.
 - Superscalar Speedup: In questo caso abbiamo una situazione un po' particolare perchè aumentando il grado di parallelismo fino ad un certo valore abbiamo che le performance migliorano di più rispetto alla ideal speedup. Poi dopo quel valore le performance iniziano a scendere.
- La superlinear speedup potremmo ottenerla, ad esempio, se il programma parallelo che scriviamo è differente da quello sequenziale e questo comporta che l'accesso alla cache sia svolto in modo differente e più veloce oppure potrei avere necessità di una maggiore quantità di cache, cosa che posso avere se ho vari

worker ma non se ne ho uno soltanto (Ad esempio se ho un solo worker a cui servono 64K di cache ma la L1 è solo di 32K allora devo accedere alla L2 e perdo tempo, se però ho vari worker, questi lavorano sulla cache L1 e quindi non hanno la perdita di tempo dell'accesso alla L2).

Dal punto di vista teorico invece la Superscalar Speedup è impossibile:

- Ammettiamo di avere un tempo ottimo di $8T$ per l'esecuzione sequenziale
- Dividiamo il lavoro tra due worker e otteniamo che ogni worker lavora meno di $4T$
- Se prendiamo quello che fanno i due worker e lo eseguiamo in modo sequenziale in un for otteniamo un tempo che è minore di $8T$ ma questo non è possibile perché $8T$ è l'ottimo.



Scalability

La scalability è un'altra proprietà legata alla performance, in questo caso viene considerato solamente il programma parallelo e si considera il tempo per l'esecuzione con un unico worker e l'esecuzione con N worker.

Quindi la formula per il calcolo della scalability è $Sc(N) = T(1)/T(N)$. Questa formula sembra molto simile a quella della speedup, in realtà c'è una differenza, in questo caso infatti non consideriamo l'esecuzione

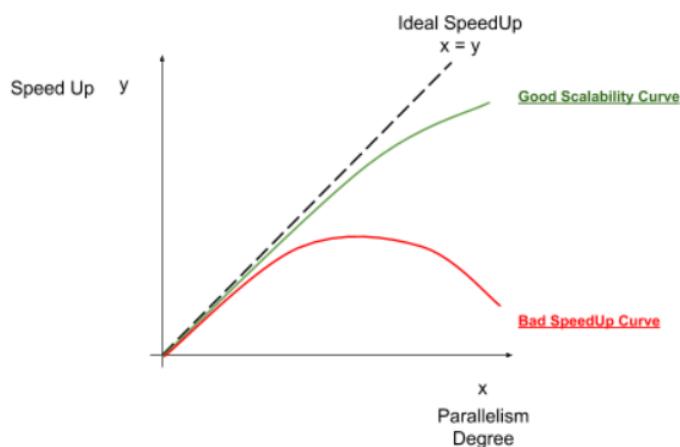
sequenziale ma l'esecuzione parallela con un solo Worker, la differenza sta nel fatto che ci sono situazioni in cui possiamo avere un tempo di esecuzione del programma sequenziale minore rispetto al tempo di esecuzione del programma parallelo con un unico worker ovvero $T(1) > T_{seq}$.

Questa differenza nel tempo di esecuzione è data dal fatto che le librerie thread safe introducono dell'overhead che ad esempio può essere legato al fatto che vengono creati dei thread.

Rimane anche da considerare che la scalability non è proprio una vera e propria misura di performance perchè noi stiamo considerando un tempo parallelo con 1 solo worker rispetto al tempo parallelo con N worker.

Quindi potremmo avere una parallelizzazione che con N worker funziona meglio rispetto al caso in cui abbiamo solamente un worker ma potrebbe essere scritta male rispetto alla versione sequenziale producendo quindi un codice che è peggiore della versione sequenziale.

È del tutto possibile avere una scalability ottima che però corrisponde ad una pessima speedup.



Energia

La parallelizzazione può ridurre anche il consumo di energia, se utilizziamo DVFS che è una tecnica per lo scaling della frequenza dei core ci troviamo nella situazione in cui aumentando la frequenza potremmo eseguire un numero maggiore di istruzioni per ogni secondo aumentando quindi l'efficienza.

Il consumo di energia è dato dalla somma del consumo di energia dinamico e statico.

Il consumo di energia statico è quello che paghiamo sempre, è indipendente dalla frequenza ma dipende dal voltaggio.

Il consumo dinamico invece è proporzionale a

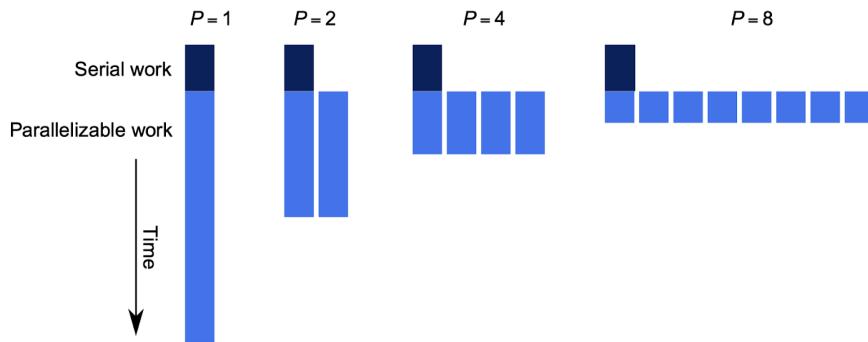
$$\mathcal{P}_{dynamic} \propto V^2 f.$$

Quindi proporzionale al voltaggio e alla frequenza, possiamo approssimare questo valore del consumo dinamico con f^3 ovvero con la frequenza al cubo.

La principale innovazione tra le varie generazioni di laptop riguarda il TDP ovvero il Thermal Design Power e questo riguarda la quantità di energia che è necessaria al processore per stare in azione e per rimanere IDLE ovvero bloccato con l'opportunità di essere risvegliato mantenendo in memoria quello su cui stavo lavorando.

Amdahl Law

La Amdahl Law è un risultato negativo perché ci pone dei limiti sul valore massimo che la speedup può assumere.



In particolare se prendiamo una computazione sequenziale, dato il tempo possiamo considerare che ci sono alcune parti della computazioni che possiamo parallelizzare e altre parti che invece non possiamo parallelizzare.

Se chiamiamo f la porzione di computazione che non è parallelizzabile abbiamo che il miglior tempo sequenziale sarà:

$$T_{seq} = f * T_{seq} + (1 - f) * T_{seq}$$

Ora possiamo provare a calcolare la speedup nel caso in cui utilizziamo n worker:

$$Sp(N) = \frac{T_{seq}}{f * T_{seq} + \frac{(1-f)*T_{seq}}{n}}$$

A questo punto calcoliamo il limite ammettendo di poter usare un numero infinito di worker avremmo che comunque la speedup verrebbe limitata da $1/f$.

$$\lim_{(n- > \infty)} Sp(N) = \frac{T_{seq}}{f * T_{seq} + \frac{(1-f)*T_{seq}}{n}} = \frac{1}{f}$$

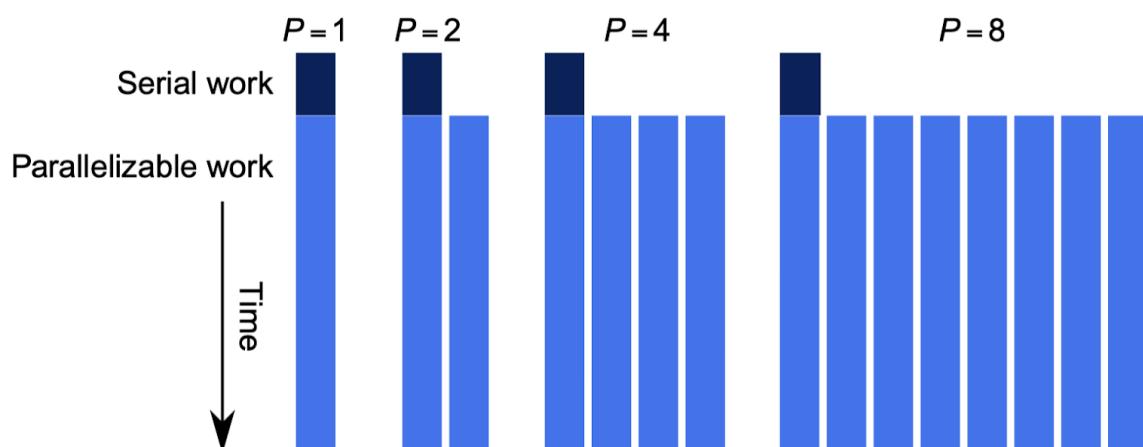
Quindi la speedup è limitata dalla frazione di lavoro che non può essere parallelizzato.

Ad esempio se questa frazione è 10/100 abbiamo uno speedup massimo di 10x, se è il 1/100 abbiamo uno speedup massimo che è 100x.

Gustafson Law

Secondo la Gustafson Law, la dimensione del problema aumenta a mano a mano che i computer diventano più potenti e in particolare si può dire che il lavoro richiesto dalla parte parallela del problema cresce molto più velocemente rispetto alla parte sequenziale.

Se questo è vero quindi aumentando la quantità dei dati abbiamo che pagheremo sempre di meno per la parte sequenziale f , a patto che la dimensione della f rimanga costante o non aumenti troppo.



Work-Span Model

Si tratta di un modello più utile rispetto alla Amdahl Law per stimare il tempo di esecuzione dei programmi perchè considera anche la parallelizzazione imperfetta.

Questo modello non solo ci fornisce un upper bound per l'esecuzione (esecuzione in sequenziale dei vari task) **ma ci fornisce anche un lower bound** (esecuzione con un numero infinito di worker).

Il work span model presenta i vari task che devono essere eseguiti andando a creare un grafo aciclico diretto, un nodo del grafo rappresenta un task e questo può lavorare solamente se i nodi che sono collegati a questo hanno finito di produrre il task necessario.

È fondamentale che i nodi all'interno di questo grafo abbiano un costo unitario, il work-span model non funziona se abbiamo dei tempi di esecuzione differente per ognuno dei nodi che troviamo nel grafo.

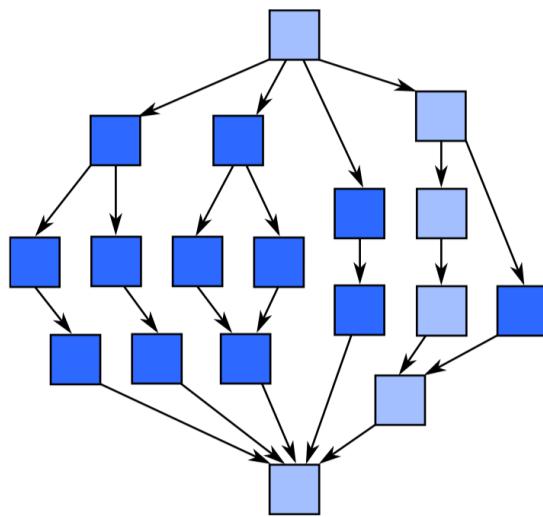
Questo modello è greedy perchè ogni nodo lavora appena ne ha la possibilità senza perdere tempo.

Consideriamo i seguenti tempi:

- Work Time: Tempo totale che servirebbe per eseguire l'algoritmo in modo sequenziale, quindi sommiamo il tempo di esecuzione dei vari task corrispondenti ai nodi.
- Span Time: È il costo che si ottiene su una macchina ideale con un numero infinito di processori. Il costo dello span time equivale alla somma dei task che troviamo nel critical path del grafo.
- Critical Path: è il percorso più lungo (in termini di tempo) formato da task che abbiamo all'interno del grafo.

Esempio:

Consideriamo il seguente grafo:



In questo caso il critical Path è 6 perchè abbiamo che per andare dal punto iniziale al finale abbiamo da attraversare 6 nodi.

Il costo totale è 18 perchè sono 18 nodi ovvero 18 task da eseguire.
Lo speedup viene limitato da $Sp \leq Work/Span$, quindi in questo caso non avremmo uno speedup maggiore di 3.

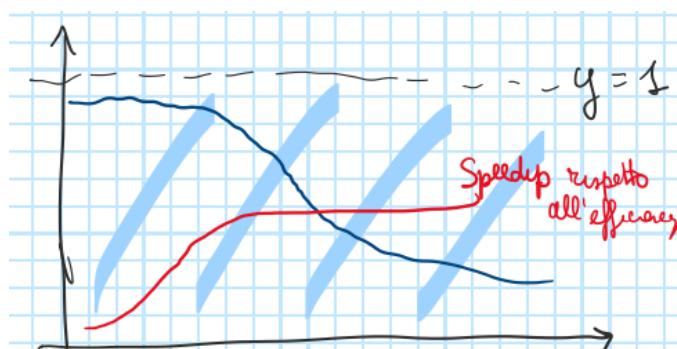
Lezione 6

L'efficiency è un'altra delle metriche relative alla performance e al parallelismo, in particolare la possiamo definire come il tempo parallelo ideale con n worker diviso per il tempo che abbiamo attualmente utilizzando n worker in parallelo ovvero $E(n) = \text{tid}(n)/T(n)$.

L'efficienza misura quanto un eventuale miglioramento dell'hardware può aver apportato anche un miglioramento in termini di performance, il valore ideale è 1 che corrisponde alla speedup lineare.

In particolare possiamo mettere in relazione la speedup con l'efficienza perchè il tempo ideale che abbiamo nell'esecuzione parallela è dato da $\text{Tid}(n) = T_{\text{seq}}/n$ ovvero dal tempo di esecuzione sequenziale diviso per n che è il numero dei worker.

Quindi l'efficienza diventa $E(n) = T_{\text{seq}}/(n*T(n))$, dato che $T_{\text{seq}}/T(n)$ è la speedup allora vuol dire che $E(n) = Sp(n)/n$.



Guardando il grafico possiamo dire che l'efficiency sta sicuramente sotto alla $y=1$.

All'inizio abbiamo una efficiency alta perchè non spendiamo tanto per organizzare la parallelizzazione poi il valore diminuisce perchè abbiamo un overhead alto introdotto dalla maggiore parallelizzazione (avviare i thread, sincronizzarli, farli comunicare).

Allo stesso tempo la speedup all'inizio è bassa perchè abbiamo una N (numero di processi paralleli) bassa, poi all'aumentare della N allora

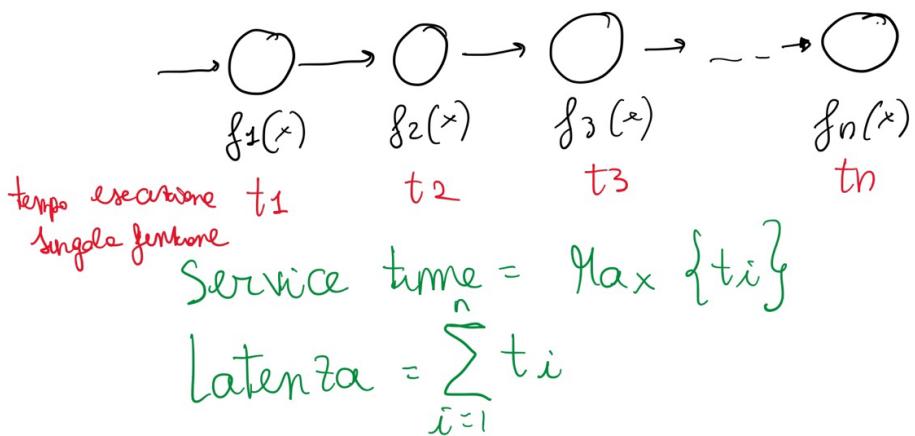
aumenta la speedup ma diminuisce l'efficiency perchè abbiamo più overhead per creare tutti i thread.

Ottimizzare per ottenere l'efficienza è una cosa particolare, possiamo utilizzare questa misura per capire quanto i cambiamenti dell'hardware migliorano poi il tempo di esecuzione dell'algoritmo. Se aggiungo sempre più core e non ottengo miglioramenti vuol dire che non sto gestendo bene le risorse che ho aggiunto.

Ci sono dei casi in cui usare 4 thread o 8 non porta miglioramenti e aggiungere nuovi core non porta alcun miglioramento al tempo di esecuzione. (Esempio delle macchine per andare a Firenze, se usiamo una macchina con 4 persone ci metto lo stesso tempo di 2 macchine con 2 persone.)

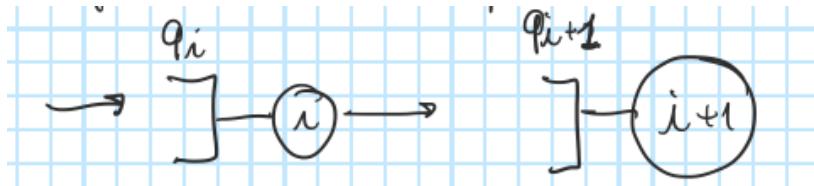
Prendiamo in considerazione una pipeline, abbiamo varie stations e ognuna esegue una funzione $f(x)$, per eseguire ognuna di queste funzioni è necessario un tempo t_i . Il Service time in questo caso è dato dal massimo t_i necessario per calcolare la funzione f_i e mi indica il tempo che un certo worker starà fermo ad attendere che il precedente termini il lavoro che ha iniziato e che lui deve usare come input.

La latenza invece è la somma di tutti i t_i ovvero dei tempi per eseguire le varie funzioni:

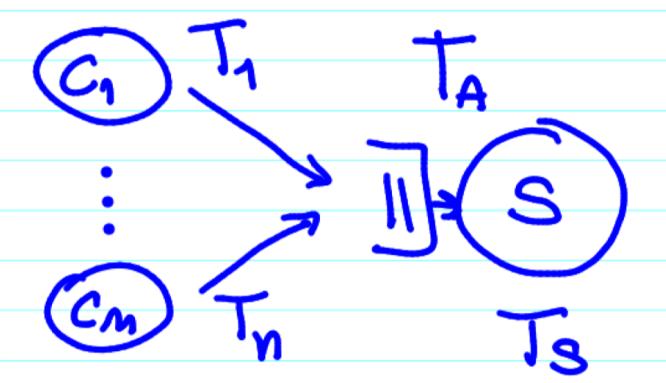


Nella realtà l'implementazione è differente, ognuna di queste stazioni ha una front queue da cui prende i dati su cui lavora, poi una volta che ha

finito di svolgere la funzione manda l'output nella front queue della stazione successiva.



Si considera l'utilisation Factor che mi indica quanto sto utilizzando la coda e quanti elementi sono presenti all'interno.

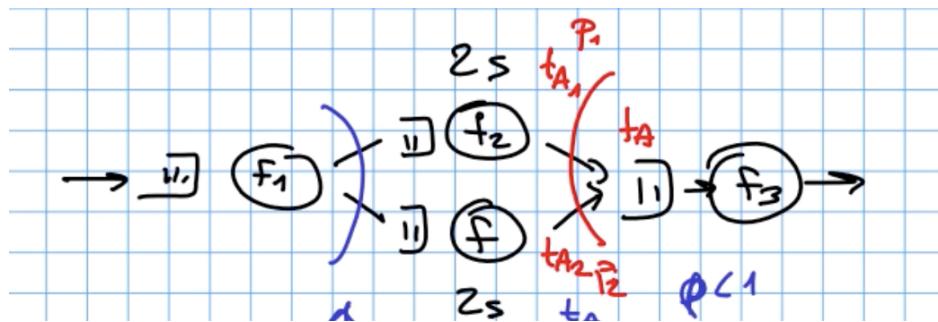


L'utilization Factor è dato da $\rho = \frac{Ts}{Ta}$ ovvero dal service time che mi indica quanto tempo ci mette S a lavorare e dal Ta che invece mi dice ogni quanto arrivano dati nella coda.

L'obiettivo è non riempire la coda, quindi se ho un service time che è maggiore rispetto al Tai è un problema perché arrivano più dati di quanti io ne riesca a processare. Quindi l'utilisation Factor è buono quando il valore è minore di 1 ovvero quando il service time è minore del tempo che serve per portare dati nella coda. Esempio: ogni 2 secondi arriva un nuovo dato ma io ci metto 1 secondo a processarlo, quindi utilization Factor = $\frac{1}{2} < 1$.

Quando abbiamo un utilization Factor > 1 si verifica una bottleneck che va trovata e deve poi essere rimossa.

Per rimuovere la bottleneck nel caso della pipeline possiamo duplicare lo stage in cui si verifica la bottleneck.



In questo modo otteniamo due code e in parallelo abbiamo due worker che svolgono la funzione f_2 .

In questo modo abbiamo la funzione f_1 che manda in output dei dati, questi vanno con probabilità $1/2$ nella prima coda e con probabilità $1/2$ nella seconda coda e quindi i due worker anche se hanno un service time maggiore del T_a unito, riescono a lavorare in parallelo senza creare la bottleneck.

Ad esempio, se i due worker ci mettono due secondi a lavorare e ogni secondo arriva un nuovo dato, il primo dato arriva nella prima coda e lavora per due secondi il primo worker, dopo un secondo arriva un altro dato che va nella seconda coda e lavora il secondo, poi quando arriva il terzo dato lo posso rimettere nella prima coda perché il primo worker ha già finito di lavorare.

Per quanto riguarda l'output dei due worker, possiamo considerare i due T_a separati e sommarli per capire ogni quanto tempo arriverà qualcosa nella coda del worker successivo.

Tutto questo ci permette di mantenere la coda dei due worker con un utilization factor minore di 1, si tratta in realtà di un discorso soprattutto teorico perchè risolvere il problema in questo modo ha un costo.

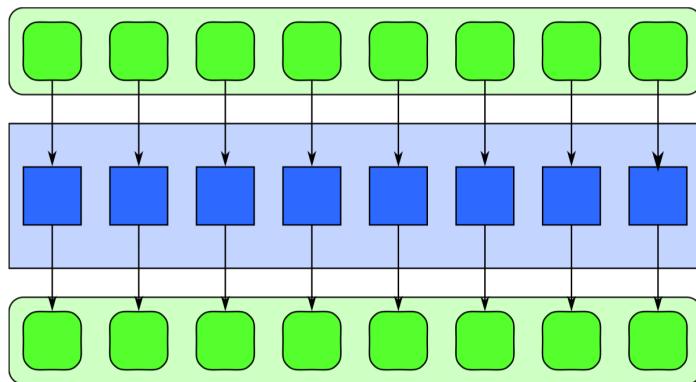
Parallel Patterns

Data Parallel Pattern

Si tratta di pattern per la programmazione parallela che puntano soprattutto a diminuire la latenza L .

I pattern più importanti di questo tipo sono:

- Map: abbiamo in input dei dati strutturati, ad esempio un vettore, definiamo una funzione $f:a \rightarrow b$ tale che svolga un compito "semplice" senza andare a modificare lo stato globale. Ogni esecuzione della funzione con input un valore che abbiamo nell'array deve essere indipendente. In output avremo alla fine un vettore della stessa dimensione che però avrà al suo interno non più gli stessi elementi di prima ma gli elementi a cui abbiamo applicato la funzione f .

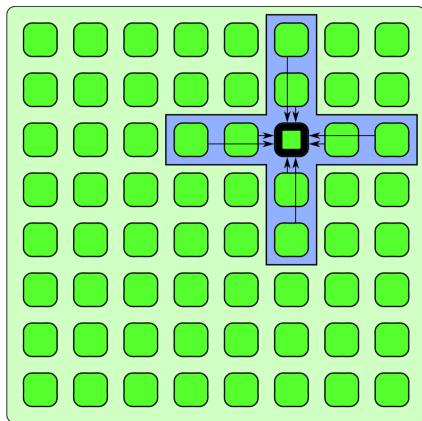


- Reduce: abbiamo in questo caso una funzione f che è in grado di combinare i vari elementi dell'array in un unico elemento. Quindi la funzione è del tipo $f: a^*a \rightarrow b$. Un esempio può essere una funzione che prende i vari elementi dell'array e mi restituisce in output la somma.



- Stencil: si tratta di una generalizzazione del pattern Map in cui con una funzione non accediamo e modifichiamo un elemento dell'array alla volta ma accediamo anche ad un gruppo di vicini di quell'elemento.
La funzione è $f: a^n \rightarrow b$.
Quindi ad esempio potremmo passare alla funzione l'elemento x_i su cui mi trovo ma anche il precedente e il successore.

Questo tipo di pattern viene utilizzato per lavorare sulle immagini.

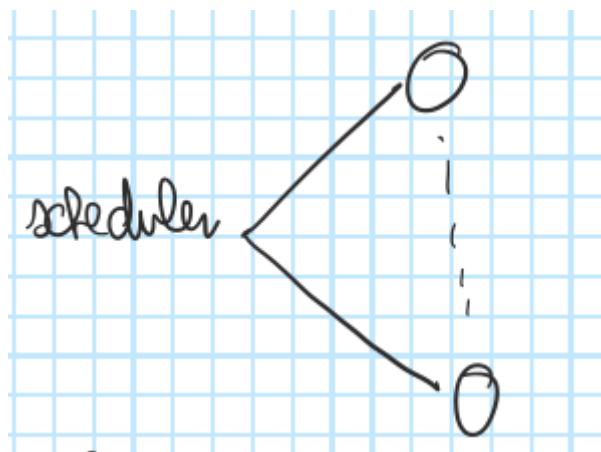


Stream Parallel Pattern

In questo caso l'obiettivo è aumentare il throughput ovvero vogliamo aumentare i dati prodotti in un certo periodo di tempo.

Abbiamo due pattern in particolare che possono essere utilizzati per arrivare a questo scopo:

- Pipeline: in questo caso andiamo a suddividere il lavoro che deve essere svolto in tanti task più piccoli, ogni worker svolge solamente un lavoro e poi passa quello che ha creato al worker successivo che a sua volta svolgerà una funzione su quei dati che ha ottenuto. Questo ci permette di aumentare il throughput perchè abbiamo tanti worker che lavorano in parallelo su lavori differenti.
- Farm: in questo caso abbiamo uno scheduler che assegna ai vari worker il lavoro che deve essere effettuato. Il worker svolgono tutti lo stesso lavoro e non sono specializzati come invece avviene nella pipeline.

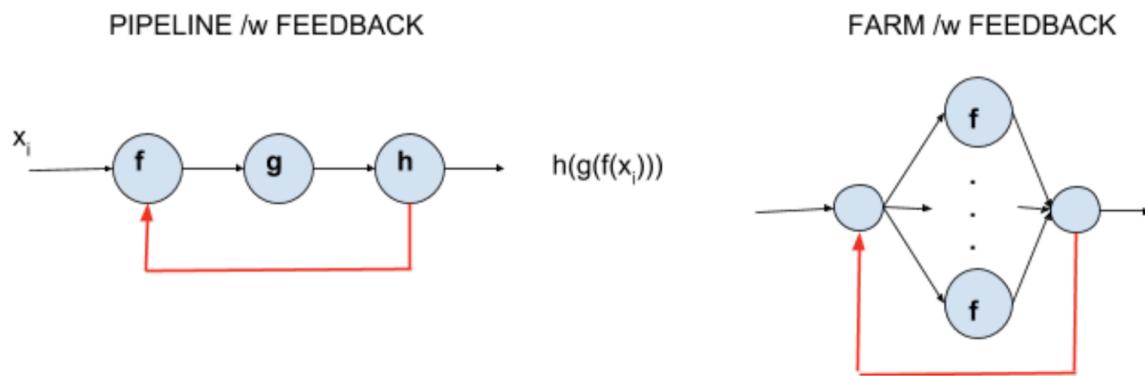


Alla fine il risultato dei vari worker va all'output stream.

Lezione 7

Tra i vari stream parallel pattern abbiamo la pipeline in cui ogni step esegue una funzione diversa e la farm in cui ogni worker svolge la stessa funzione con input differenti.

Quando utilizziamo questi pattern abbiamo la possibilità di utilizzare anche un “feedback” ovvero una sorta di modifier per il pattern che ci permette di utilizzare un canale solamente per le comunicazioni tra i vari worker.



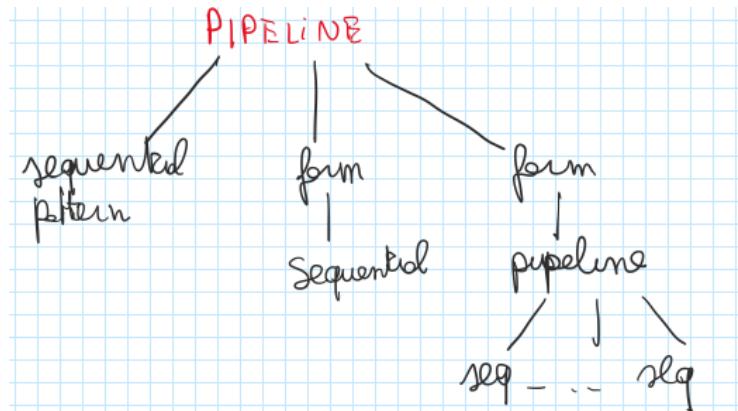
Ad esempio, potrei avere la necessità di comunicare un risultato calcolato dal worker h al worker f perché magari il worker f può poi utilizzarlo per migliorare il suo funzionamento e quindi migliorare i prossimi risultati che calcolerà.

Un esempio è il caso degli stock market, ad esempio se h prende una certa decisione magari potrebbe voler dire a f di fare attenzione a qualcosa in particolare. I feedback vengono inviati da destra verso sinistra sia nella pipeline che nella farm.

Pattern Composition

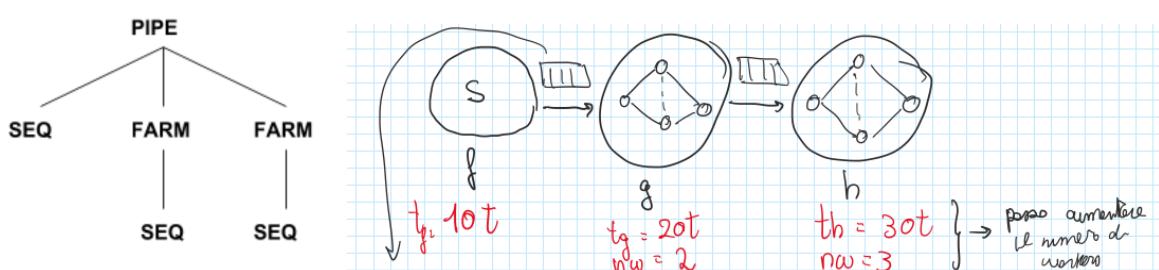
Avendo vari pattern possiamo pensare di utilizzarli andandoli a comporre (anche più di una volta). Andremmo a creare un albero di pattern, un esempio è quello nella foto qua sotto, abbiamo una pipeline e ogni stage

della pipeline potrebbe avere una implementazione sequenziale oppure una implementazione con una farm che a sua volta internamente ha degli stage sequenziali oppure potremmo avere gli stage della pipeline implementati come farm che a sua volta è implementata con stage organizzati come una pipeline che a sua volta è formata da stage sequenziali.



L'idea di utilizzare dei nested pattern è simile all'idea di utilizzare un approccio RISC (in quel caso si usano istruzioni base che rendono il processore più veloce e il grosso del lavoro lo fa il compilatore) spostando ovviamente questo concetto alla parallelizzazione.
Quindi andremo a rendere sempre più semplici i task che devono essere effettuati.

Se consideriamo questo composed pattern:



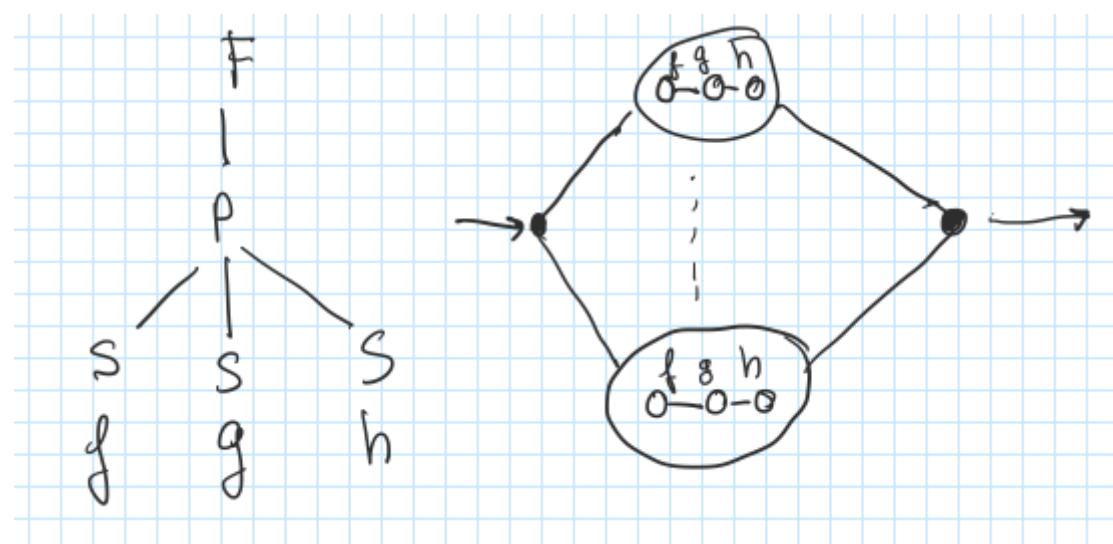
Abbiamo una pipeline con 3 stage, un primo stage che è sequenziale, poi uno stage con una farm che è formata da stage sequenziali e poi uno stage con una farm anch'essa con stage sequenziali.

Dovremmo evitare le bottleneck, se le code che abbiamo nella pipeline si riempiono troppo velocemente abbiamo un rallentamento.

Prendiamo il caso in cui g è più lento di f e h è più lento di f ma è anche differente dal tempo necessario a g. Il numero dei worker di g e di h dovrebbe essere quindi differente a seconda di quanto vogliamo aumentare la velocità.

Se ad esempio abbiamo il $tf=10$, poi $tg=20$ e $th=30$, possiamo mettere 2 worker che svolgono la g e tre worker che svolgono la funzione h e otteniamo per ognuno un service time di 10 e quindi otteniamo una pipeline che funziona correttamente.

Con la composition possiamo anche fare qualcosa di differente, ad esempio consideriamo la seguente composizione:



Abbiamo una farm ma all'interno abbiamo che ogni worker è una pipeline che svolge in sequenza f, g e h.

In questo caso il service time viene calcolato come massimo del tempo necessario per svolgere ogni parte degli stage, quindi dato che h ci mette 30t ed è il massimo, $T_s = 30t$ ed è il massimo.

Anche in questo caso possiamo aumentare il numero dei worker e portandolo a 3abbassiamo il T_s a 10t come fatto in precedenza.

C'è una differenza sostanziale rispetto alla prima situazione, nella prima infatti tutti sono sempre impegnati e hanno sempre una funzione da svolgere. Nel secondo caso invece abbiamo dei periodi in cui alcuni worker non fanno niente, ad un certo punto infatti la coda tra il terzo e il secondo si riempirà e quindi il secondo non farà niente, lo stesso discorso vale tra il secondo e il primo, solamente il terzo lavora sempre. Quindi questa seconda soluzione è meno efficiente della prima.

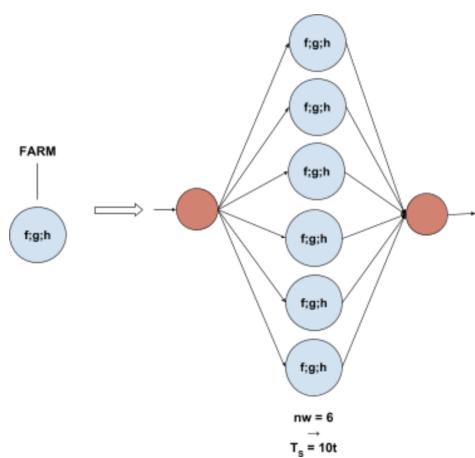
Se contiamo il numero delle processing element delle due soluzioni:

- La prima soluzione utilizza 6PE, quindi 6 thread o core
- La seconda ne usa 3×3 perchè ne ha 3 che formano la farm ma dentro ogni stage ce ne sono altri 3, quindi in tutto 9 thread.

Entrambe le soluzioni però hanno lo stesso throughput. In efficiency abbiamo un aumento del 30% nella seconda soluzione, quindi qua l'efficienza fa la differenza perchè la seconda soluzione non è efficiente come la prima.

Abbiamo una terza possibilità per comporre i vari pattern.

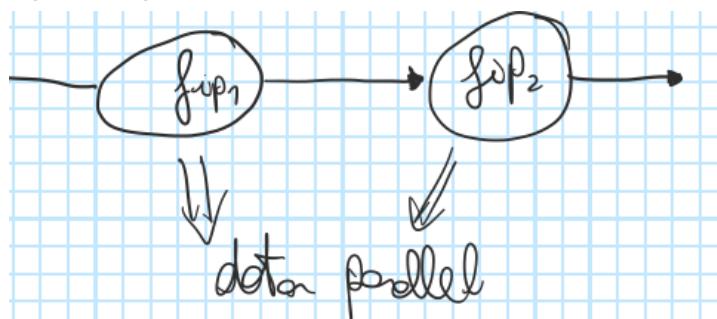
Potremmo utilizzare un thread che si occupa di calcolare f,g e h e che quindi ha un service time pari a $60t$. Se però inseriamo 6 di questi thread in una pipeline abbassiamo a $10t$ il service time e quindi riusciamo ad avere la stessa prestazione della prima soluzione con 6 PE + 2 PE per gestire la distribuzione dei task e per prendere gli output.



Compositionality

Possiamo anche unire i due tipi di pattern visti ovvero il data parallelism che cerca di diminuire la latenza e lo stream parallelism che invece aumenta il throughput.

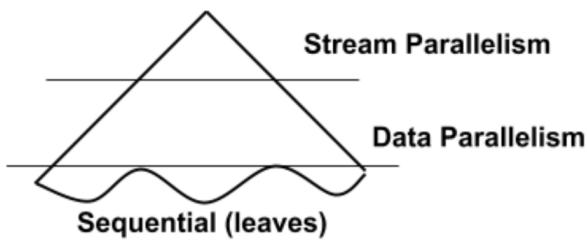
Ad esempio se dobbiamo ridurre il rumore di una bitmap e poi vogliamo fare altre trasformazioni, potremmo volere una pipeline e all'interno di ogni stage vorremmo lavorare in modo data parallel.



Rimuovere il rumore richiede che un controllo dei pixel e un confronto con i pixel vicini in modo da capire se ci sono grandi differenze di colori. Questa operazione può essere svolta in parallelo, per ogni pixel devo considerare i pixel vicini, posso dividere l'immagine in parti uguali che però si sovrappongono. Se poi nel secondo stage devo trasformare in black e white posso dividere in parti uguali l'immagine e poi posso lavorare in parallelo.

Questo lavoro lo possiamo fare con una pipeline, nel primo stage ho uno stencil mentre nella seconda una map.

Componendo questi due pattern otteniamo un two tier composition model in cui abbiamo, per ogni composition che possiamo creare, una prima parte dell'albero che è di tipo stream parallel, una parte più bassa che è data parallel e la parte delle foglie che è sequenziale.

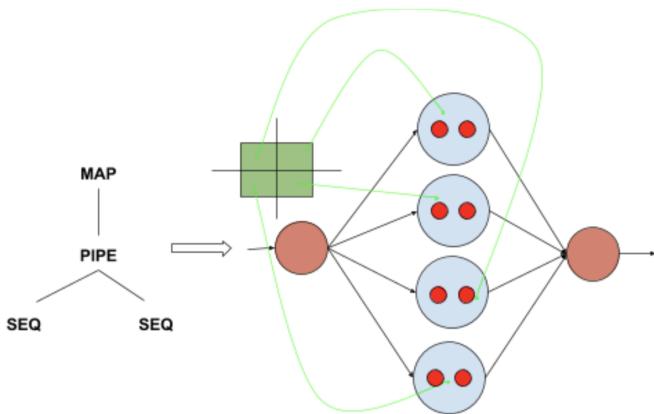


Ad esempio se consideriamo l'esempio del noise reduction e del black e white, possiamo organizzare una pipeline che ha al suo interno due operazioni sequenziali ovvero PIPE(Seq N, Seq BW), la prima parte necessita di tempo T_n e la seconda tempo T_{bw} .

Quindi il $T_s = \text{Max}(T_n, T_{bw})$.

Se invece creiamo una pipeline che al suo interno utilizza un pattern data parallel abbiamo PIPE(stencil(seq), map(seq)) ovvero abbiamo un service time che è sicuramente minore di $\text{Max}(T_n, T_{bw})$ perché abbiamo anche ridotto la latenza con la parallelizzazione.

Una soluzione che non rispetta il two tier composition è la seguente perché non abbiamo la pipeline all'esterno ma la inseriamo all'interno e al suo interno abbiamo delle operazioni svolte in modo sequenziale.



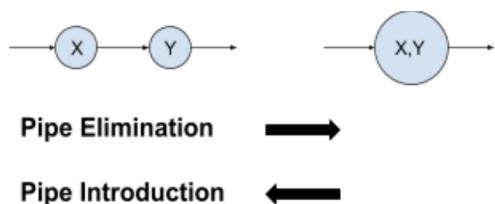
Questa soluzione deve considerare il costo per lo split dell'immagine e la ridirezione e non mi porta un reale guadagno anzi comporta dei problemi nella realizzazione architettonale.

Regola per eliminare la pipeline

La pipeline viene eliminata andando da sinistra a destra e si introduce una composizione dei due task che venivano svolti all'interno della pipeline che ora vengono svolti in modo sequenziale.

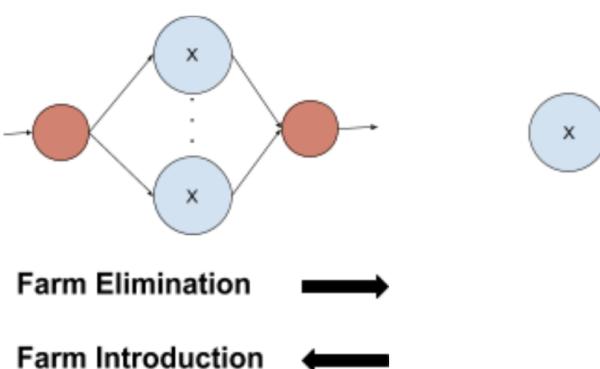
$\text{pipe}(x,y) = \text{Comp}(x,y)$ —> Eliminazione da sinistra a destra

Andando invece da destra verso sinistra si ottiene una pipeline perché la composition viene divisa in vari task eseguiti nella pipeline.



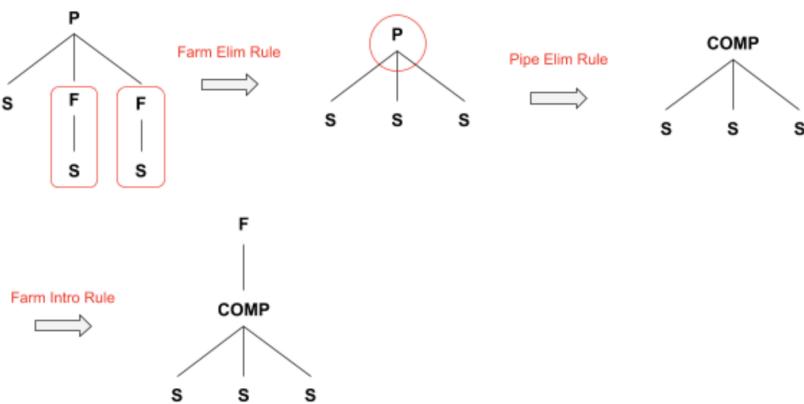
Lo stesso discorso vale con la farm:

$\text{Farm}(X) = \text{Seq}(X)$ —> Eliminazione da destra a sinistra, introduzione da sinistra a destra.



Il Ts che si ottiene con l'introduzione della composition è maggiore di quello che abbiamo usando la pipeline o la farm.

Date queste regole possiamo considerare un esempio:



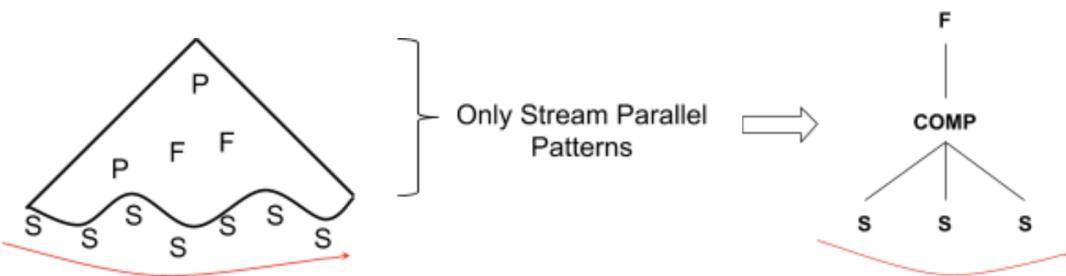
Il primo albero è uguale all'ultimo dal punto di vista della computazioni ma in termini di prestazioni abbiamo dei miglioramenti.

Questo risultato lo possiamo provare in generale e otteniamo una **forma normale** per la stream parallel computation.

Nella forma normale non devo gestire le code dei nodi sequenziali perché abbiamo fatto una composizione di tutti i nodi sequenziali e quindi è come se fosse un'unica funzione.

Ci troviamo con un albero in cui nella parte alta abbiamo solamente Stream Parallel Pattern e nella parte delle foglie abbiamo delle computazioni sequenziali. Le foglie vengono prese da sinistra a destra e messe in una composition mantenendo l'ordine perchè una foglia potrebbe produrre l'output che viene usato come input dalla foglia successiva.

Sopra a questa nuova organizzazione con la composizione, si mette una farm, abbiamo quindi una farm con un numero di worker e ogni worker effettua in modo sequenziale quello che veniva svolto dalle varie foglie.



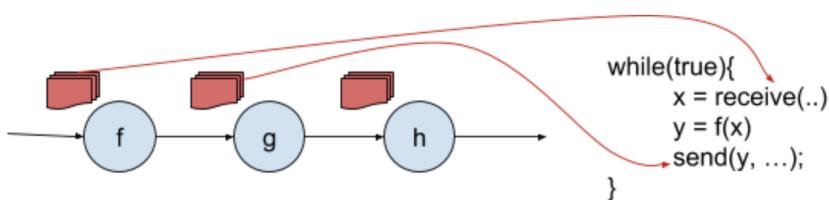
Implementation of the Stream Parallel Pattern

L'obiettivo di questo pattern è aumentare il throughput ovvero vogliamo aumentare i dati prodotti in un certo periodo di tempo.

Abbiamo una implementazione di questo pattern tramite l'utilizzo di una pipeline e una implementazione tramite una Farm.

Pipeline

Consideriamo uno Shared Memory MultiCore, abbiamo 3 thread che eseguono tre funzioni differenti e abbiamo bisogno di un metodo di comunicazione efficace da poter utilizzare per lo scambio di informazioni tra i thread.



Possiamo utilizzare ad esempio una coda, ognuno di questi thread prende dalla coda il dato che poi passa alla funzione e manda in output nella coda successiva il dato che ha prodotto.

Pop e push nella coda devono essere però sincronizzati, ad esempio possiamo utilizzare una meccanismo di lock/unlock per gestire la sincronizzazione, questa è la soluzione più semplice e abbiamo il seguente codice per receiver e sender:

Receiver:	Sender:
Lock(mutex)	Lock(Mutex)
Pop dalla coda	Push nella coda
Unlock(mutex)	Unlock(Mutex)

Questo metodo però non considera il caso in cui abbiamo la coda piena o vuota, possiamo utilizzare ad esempio le condition variables.

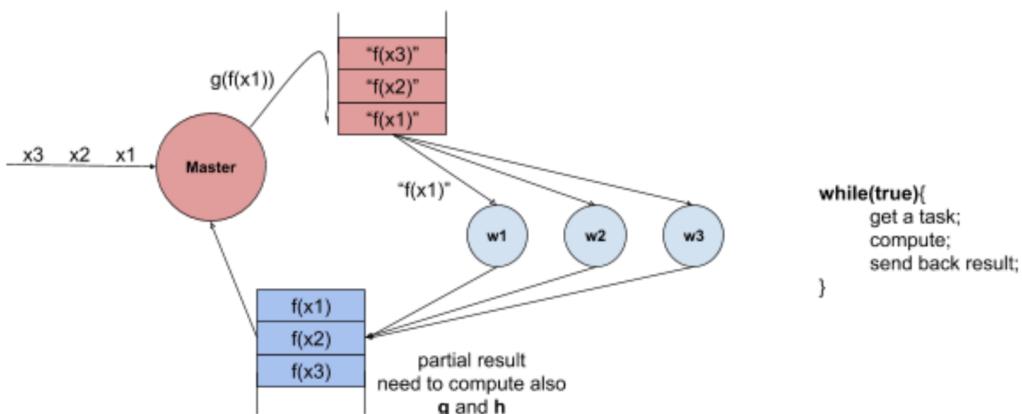
Associamo una mutex alla condition variable in modo che nel momento in cui accedo alla coda (quando c'è qualcosa da leggere o quando c'è spazio per scrivere), la condition variable capisce che deve fare la lock. Alla fine verrà eseguito l'unlock della mutex e verrà inviata una notify ai thread che erano in attesa sulla stessa condition variable in modo che possano accedere alla coda.

Ammettiamo di avere una coda che funziona nel modo appena spiegato, quindi una coda che sia in grado di capire se la coda è piena e in tal caso fermarsi e non aggiungere altri dati e anche se la coda è vuota. Questa coda avrebbe l'operazione send() e l'operazione receive(). Questo non è l'unico modo che abbiamo per implementare una soluzione del genere.

Consideriamo la computazione della pipeline in modo differente:

- Consideriamo 4 dati, x_1, x_2, x_3, x_4 e tre funzioni messe in una pipeline che sono f, g e h .
- Ognuno dei dati viene elaborato dalla funzione f , quindi viene calcolato $f(x_1)$ poi $f(x_2)$, $f(x_3)$ e $f(x_4)$. Poi calcoliamo $g(f(x_1))$, $g(f(x_2))$ e poi lo stesso discorso con $h(g(f(x_1)))$.
- Ci sono delle dipendenze tra i vari calcoli perché la funzione $g(f(x_1))$ non la posso calcolare fino a quando non ho calcolato $f(x_1)$ e lo stesso discorso vale con le altre funzioni. Se però x_1 e x_2 sono disponibili nello stesso momento, posso comunque calcolare $f(x_1)$ e $f(x_2)$ in parallelo.

Quindi la pipeline viene implementata in questo modo:



Ogni volta che al master arriva uno dei dati x_i , mette in una coda la computazione che deve essere effettuata su quel dato, la coda rossa mi indica quali sono le funzioni che devono essere ancora eseguite e che devo far calcolare ai vari worker. Sotto abbiamo un pool di worker che eseguono un loop in cui prendono il task da eseguire dalla coda, lo calcolano e poi mandano in output il risultato nella seconda coda (quella blu). Nella coda blu abbiamo un risultato parziale, poi il master leggerà da questa coda e quando ad esempio trova $f(x_1)$ capirà che deve poi essere effettuata l'operazione $g(f(x_1))$ e mette questa operazione all'interno della coda rossa.

L'idea in questo caso è che nessuno dei worker è dedicato al calcolo di una certa funzione, calcolano f, g e h a seconda della necessità del momento e del dato che prendono in input.

In questo modo avrò sempre del lavoro da fare, con la pipeline originale se metto solamente tre thread quello che ottengo è una latenza che è un terzo rispetto a quella originale (da x a $x/3$).

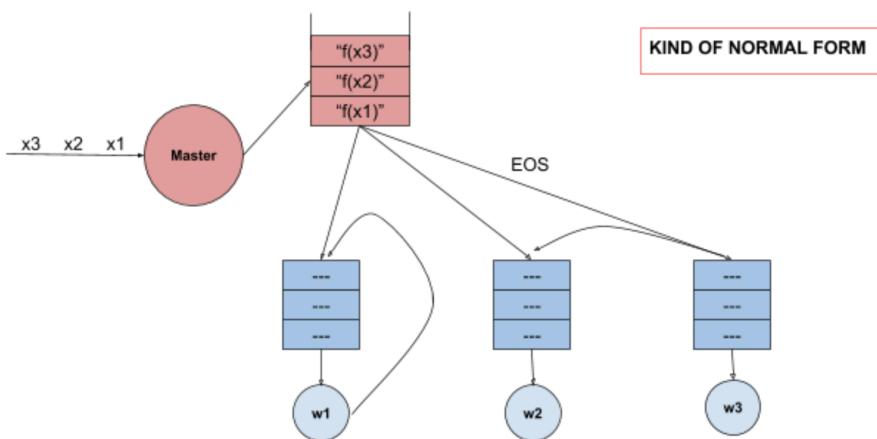
Qui invece la latenza rimane la stessa ma ho più thread che lavorano in parallelo e il service time diminuisce, quindi perdo meno tempo per il service time e viene aumentato il Throughput.

Questa soluzione ha anche dei difetti:

- L'ordine degli output potrebbe essere differente da quello che ci aspettiamo
- Se calcoliamo $f(x_2)$ e $f(x_3)$ prima di $g(f(x_1))$ potrei avere un aumento della latenza
- Abbiamo un thread in più per il master
- Abbiamo un problema con la cache visto che siamo in una shared memory architecture. Ad esempio se calcoliamo $g(f(x_1))$ nel worker 3 e $f(x_1)$ sul worker 1 con i due thread che non sono nello stesso core, avremo uno spostamento di dati tra le cache dei due core. Questo non si sente molto in questo caso, se invece abbiamo una architettura distribuita si sente maggiormente questo problema perché lo spostamento dei dati non avviene tramite un bus ma

avviene tramite un cavo. In questo caso quindi è importante la locality.

Se ho tante workstation che devono comunicare tra loro abbiamo il problema della queue generale ovvero della coda rossa in cui memorizzo l'operazione da effettuare perchè questa è nel master, quindi lontana dai worker. Possiamo risolvere questo problema in questo modo:



Abbiamo il master con la sua coda, ognuno dei worker (localizzati su macchine diverse) ha la propria coda, quando un worker deve lavorare guarda la sua coda, se è vuota va a prendere il task nella coda generale, altrimenti prende il task dalla sua coda. Quando ha finito di calcolare mette il risultato nella sua stessa coda.

Quando uno di questi worker ha la sua coda vuota e andando alla coda generale la trova vuota, chiede il task da eseguire ad una delle code degli altri worker.

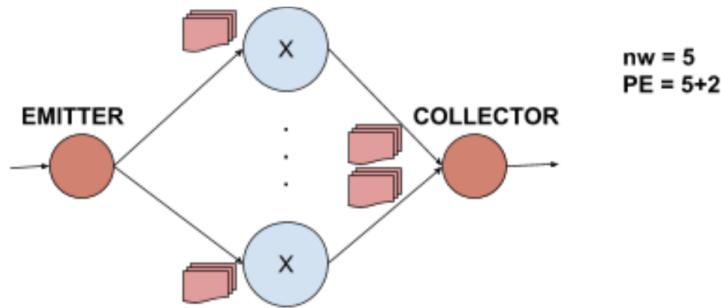
Quando poi il worker finisce di eseguire l'ultima funzione della pipeline ovvero $h(g(f(x_1)))$, manda questo risultato al master.

Questa forma di pipeline assomiglia ad una normal form perchè prima calcoliamo f poi g e poi h nello stesso nodo e mi permette di garantire una locality perchè un worker mi calcola tutto quello che riguarda un certo dato iniziale e in questo modo non deve spostare dati perchè ha tutto nella cache.

Farm

Consideriamo l'implementazione con la farm nel caso di FastFlow. Abbiamo un thread scheduler, chiamato emitter e un thread gatherer, chiamato collector.

Se diciamo che la parallelism degree della farm è 5 vuol dire che in realtà il numero di PE è 7 perché abbiamo anche questi altri due thread.



C'è anche un altro metodo che permette di implementare la farm. Abbiamo una coda con i task in input e una serie di worker che vanno nella coda, prendono un task, calcolano la funzione e mettono in una coda differente il risultato. Il codice è lo stesso per tutti i worker e se usiamo una struttura dati come questa potremmo avere un problema come prima perché abbiamo una sola coda da cui prendere i dati e un solo scheduler che distribuisce i task quindi è preferibile inserire la coda prima dei worker e dopo. Abbiamo una coda prima del worker in cui accede l'Emitter e il worker, poi una dopo a cui accede il worker e il collector.

Lezione 8

MakeFile

```
cxx = g++  
cxxflags = -std = c++17  
ldflag = -pthread
```

Nel Makefile inseriamo le direttive per il compilatore in modo da poter compilare il codice semplicemente scrivendo “make nomeDelFile”. Indichiamo il tipo del compilatore, lo standard del C++ da utilizzare e la libreria pthread da utilizzare.

Mutex

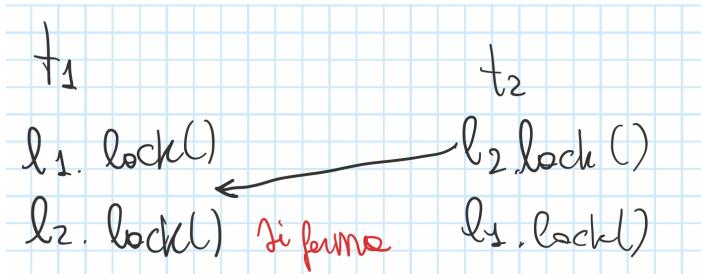
Un modo stupido per utilizzarne le mutex, C++ non lo supporta più:

```
#include <mutex>  
mutex l;  
  
l.lock()  
//Codice da eseguire, se qua ad esempio ho uno sleep  
passano tutti i //secondi che indico, senza la lock  
invece passano solamente una volta //e non per tutti i  
thread.  
l.unlock()
```

È essenziale che dopo aver preso la chiave questa venga anche rilasciata altrimenti rimango con la chiave e gli altri thread non possono lavorare. Quindi, se ad esempio ho un return tra la lock e l'unlock devo assicurarmi di sbloccare la chiave prima di fare il return.

Esistono varie tipi di lock, una documentazione è nel sito cppreference.com:

- UniqueLock: lock che deve essere utilizzata con le condition variables perchè garantisce delle proprietà che mi servono se uso le condition variables.
- GuardLock: passiamo alla lock una mutex, quando esce dallo scope di quella funzione, la mutex viene rilasciata in automatico. È preferibile usare questo metodo al posto della classica lock.
- Scoped_Lock: simile alla guardLock, ci permette di avere una o più mutex per tutta la durata dello scope. Questa può essere utile se siamo in un caso come questo:



Qui potremmo trovarci con il threat t_1 che prende la lock su l_1 e poi subito t_2 prende la lock su l_2 , quindi sono entrambi fermi.

Variabili atomiche

Quando vogliamo eseguire delle operazioni su delle variabili facendo in modo che siano eseguite in parallelo senza però andare ad eseguire esplicitamente la lock e l'unlock possiamo utilizzare un tipo atomico per la variabile.

Ci sono una serie di operazioni che possono essere svolte in modo atomico, un esempio sono l'incremento e il decremento di una variabile senza dover inserire la lock prima e l'unlock dopo.

Store e load della variabile quindi vengono fatti in modo atomico.

Avere una variabile atomica ha un costo quindi non ha senso farle tutte atomiche perché altrimenti andremmo ad avere dei problemi anche dal punto di vista delle performance.

Esempio di definizione di variabile di tipo atomico:

```
std::atomic<int> x{0};
```

Gestire la sincronizzazione di una coda

Una situazione più complessa consiste nella sincronizzazione di una coda, quando tolgo qualcosa dalla coda voglio essere sicuro che la coda non sia vuota, se devo attendere non voglio avere una attesa attiva e quindi devo utilizzare una variabile di condizione.

La variabile di condizione ha due operazioni principali:

- Wait: mi fermo se non rispetto la condizione
- Notify: notifico l'altro thread che è in attesa sulla condition variable, ad esempio, se la coda è vuota mi fermo, quando un altro thread aggiunge qualcosa nella coda allora mando una notify al thread fermo che controlla di nuovo la condizione e se a quel punto trova qualcosa nella coda allora lo estrae.

```
std::unique_lock<std::mutex> lk(m);
cv.wait(lk, []{return ready}); // Quando attendo la lock
// viene rilasciata subito in automatico
```

Metodi Async

Il metodo Async serve per poter eseguire una parte del codice in modo asincrono rispetto all'esecuzione del flusso principale.

Eseguiamo un fork e eseguiamo il codice in un thread indipendente, viene restituito un puntatore che è definito come “future”.

Dal thread principale dobbiamo eseguire la get() su questa future in modo da ottenere il risultato calcolato dal thread, se il risultato non c'è ancora il thread si blocca.

```
launch::async;
```

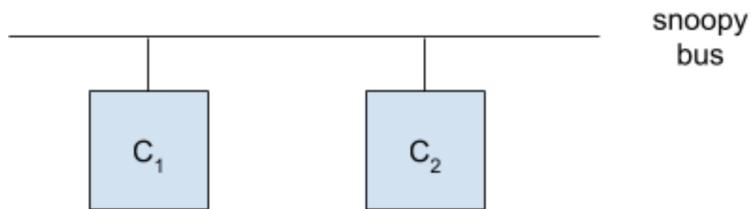
Lezione 9

Vediamo due possibili fonti di inefficienza nelle computazioni in parallelo:

- **Locality:** l'idea della locality è quella di andare a memorizzare vicini tra loro i dati che dobbiamo utilizzare durante l'esecuzione del programma. L'espressione principale della locality è la cache che ci consente di avere una memoria che ha un costo di accesso molto basso in cui inseriamo però pochi dati. Se i dati che vengono inseriti all'interno della cache tengono conto della locality allora riusciamo ad ottenere una performance migliore durante l'esecuzione del programma. Tra l'accesso alla cache e l'accesso alla memoria abbiamo una differenza in termini di tempo che si nota, se ci mettiamo 1 secondo per accedere alla cache allora ce ne vorranno 100 per accedere alla memoria perché c'è un ordine di magnitudine che li separa.

Abbiamo a disposizione vari livelli di cache, da quella di primo livello che ha una velocità di accesso simile a quella di un registro fino ad arrivare alla memoria principale. Aumentando la distanza dal processore aumenta anche lo spazio disponibile ma anche il tempo necessario per l'accesso ai dati.

Quando si esegue del codice in parallelo bisogna considerare la cache, infatti su una architettura multicore la cache è parzialmente condivisa. Quando si parla di cache e di computazione in parallelo è importante anche la **coherency**, infatti se cerchiamo di accedere allo stesso indirizzo in due cache differenti, dovremo ottenere lo stesso valore. Ci sono vari modi per implementare questa soluzione, all'inizio si utilizzavano dei bus chiamati snoopy bus, il problema però è che questo sistema è limitato dal numero di dispositivi che possiamo collegare al bus.



Un’alternativa allo Snoopy Bus è il “Directory Based Coherency”, questo metodo per mantenere la coherency utilizza una speciale directory per tenere traccia dello status dei vari blocchi della cache.

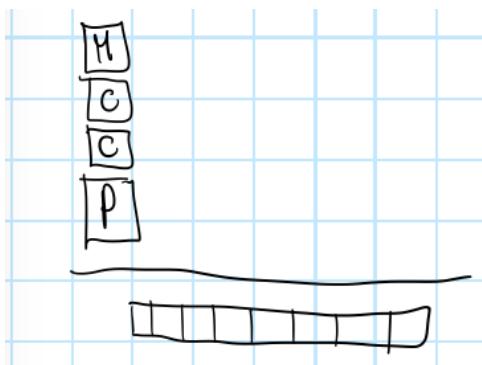
Lo status mi indica quali nodi stanno condividendo quel blocco in quel momento e in che stato si trova quel blocco della cache.

Il problema si verifica quando dobbiamo scrivere nella cache perché tutte le locazioni che dipendono da quel valore devono essere aggiornate e per farlo dobbiamo bloccare le read su quelle locazioni da parte di altri core.

Nelle architetture con un alto numero di core per chip abbiamo vari metodi per la gestione della cache:

- * Non abbiamo proprio una cache
- * Se c’è la cache non c’è la consistenza

Un esempio è il processore della playstation, viene usato un processore con un core molto grande e poi questo core condivide un bus con 8 core più piccoli, ognuno di questi core ha una memoria che però non ha impatto sulle altre.



Un altro esempio è parallela che ha inserito molti core su un unico

chip utilizzando delle mesh orizzontali e verticali, ogni core ha una piccola parte di memoria che è organizzata tramite PGAS, questo modello ci permette di salvare molto spazio.

Concetti legati alla cache:

Grain: Possiamo calcolare il grain facendo la divisione tra il tempo che spreco per il calcolo e il tempo che spreco per comunicare.

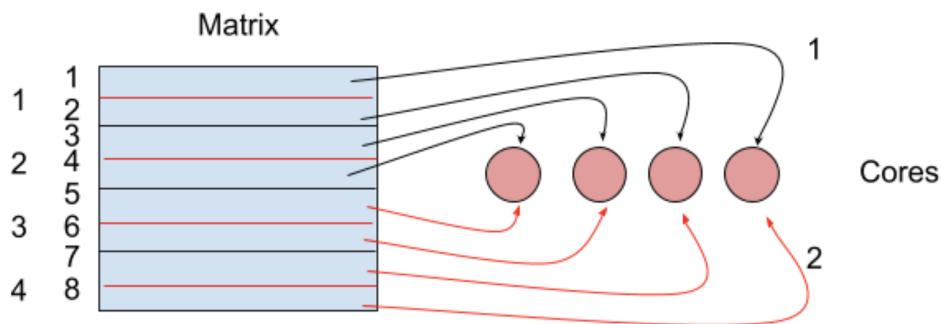
$$\text{Grain} = \frac{T_{compute}}{T_{communicate}}$$

Quindi:

Il tempo per la comunicazione lo consideriamo come overhead quindi deve essere basso rispetto al tempo necessario per la computazione.

In questo contesto diventa importante per l'utilizzo della cache e per ridurre il tempo di computazione il modo in cui divido i dati su cui devo lavorare.

Prendiamo l'esempio della moltiplicazione delle matrici, se devo moltiplicare due matrici, devo suddividere i dati all'interno del processore, quindi o divido A in vari blocchi orizzontali oppure posso dividere B in blocchi orizzontali. Mi conviene fare una divisione per blocchi orizzontali perchè la matrici vengono memorizzate per righe. Se però abbiamo una matrice grande e la dividiamo in 4 parti perchè ho 4 core, potrei trovarmi nel caso in cui tutto il blocco di dati non entra nella cache e quindi una tecnica migliore consiste nello splittare i blocchi in modo che vengano esauriti per ogni blocco gli elementi della prima riga e poi delle successive.



Conoscere la dimensione della cache e delle strutture dati che vogliamo utilizzare ci permette di fare una scelta migliore e di capire bene in che modo è meglio implementare il nostro programma parallelo.

False Sharing: Negli algoritmi paralleli si tende a suddividere i dati in blocchi in modo che ogni thread lavori su un blocco di dati.

Ad esempio se vogliamo implementare un DPI abbiamo che ogni pacchetto che arriva viene assegnato ad un thread, possiamo utilizzare una hash table in modo da capire per ogni pacchetto a quale thread devo assegnarlo.

In questo modo avrei quindi una struttura dati condivisa tra tutti i vari worker e ogni worker però avrà accesso solamente ad una parte della struttura.

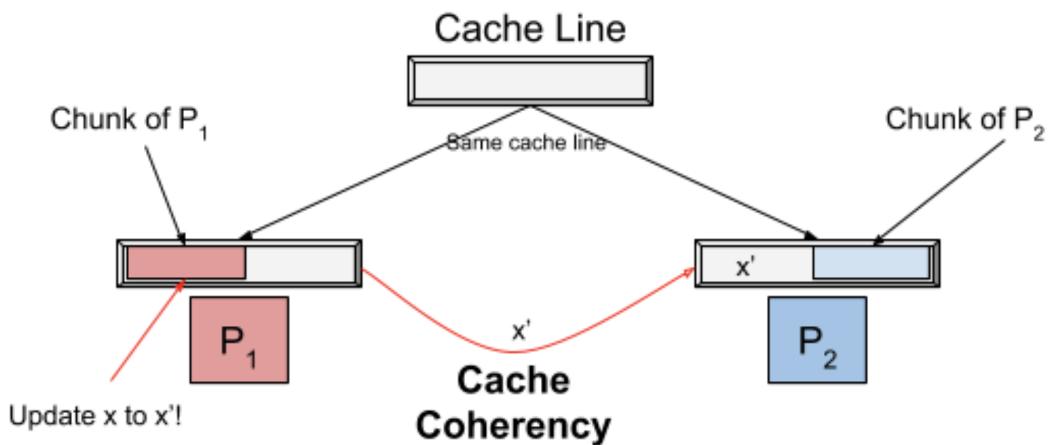
Questo metodo di lavoro in cui ognuno lavora su una parte della struttura si chiama Single Owner Computes Rule.

Può essere un problema avere una struttura di questo genere, condivisa tra tutti gli utenti perché potrei trovarmi con della cache che potrebbe essere condivisa tra due worker. In questo caso, anche se i due worker lavorano su parti differenti della cache, a causa della coherency sarebbero obbligati ad aggiornare la cache ogni volta che l'altro apporta delle modifiche.

Ammettiamo di avere una linea di cache dove troviamo oltre dei bit di controllo e poi i dati veri e propri.



Abbiamo due copie della cache nei due worker, quando uno dei due accede alla sua cache e fa delle modifiche sulla sua parte, anche l'altro dovrà fermarsi ed aspettare forzatamente che venga aggiornata anche la sua cache.



(L'idea è che P_1 aggiorna X in L_1 poi X è aggiornato in L_2 e poi P_2 è costretto ad aggiornare X nella sua L_1 anche se non gli serve).

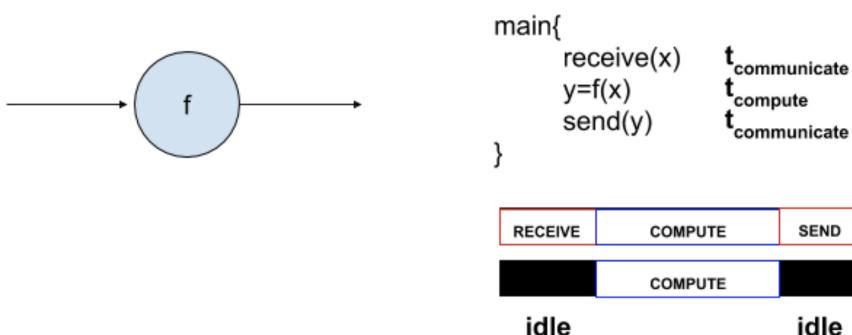
I due problemi appena descritti sono importanti per la memory pressure e per la memory intensity. In particolare la memory intensity mi indica quante load e store faccio rispetto alla quantità di computazione (Load+Store/Computations) e se questo valore è alto vuol dire che devo considerare ancora maggiormente il grain e il false sharing.

- **Excess Parallelism:** in un software parallelo le varie attività devono comunicare e sincronizzarsi durante l'esecuzione, se per esempio abbiamo 4 contesti nella macchina allora potremmo inserire più di 4 attività parallele perchè in questo modo riusciamo ad oversplittare la computazione.
Se ad esempio un thread in un contesto è fermo perchè attende

dati da altri o magari perchè sta scrivendo i dati, questo può essere deschedulato e al suo posto potrà essere inserito un altro thread di quelli creati “in più” in modo che intanto possa lavorare lui.

Ci sono alcune soluzioni che sono efficienti, si tratta del double e del triple buffering in cui cerchiamo di organizzare la computazione in modo che ci siano più cose da fare.

Ad esempio, prendiamo un thread che riceve, effettua dei calcoli e scrive in output il risultato. Il problema di questo thread è che sarà in uno stato “idle” sia mentre aspetta sia quando invia, quindi avremo un tempo di comunicazione (speso sul device di I/O) dovuto a lettura e scrittura e poi avremo il tempo d'esecuzione della funzione.



Mentre il thread è fermo ad aspettare o ad inviare quindi converrebbe fare altro, quindi ad esempio potrei fare la parte di computazione f relativa a qualcosa che ho già ricevuto.

Ammesso che comunicazione e computazione hanno un tempo di esecuzione simile allora posso organizzare la memoria in tre buffer e posso usare tre thread.

Utilizziamo 3 thread differenti in modo che non deve essere gestita la sincronizzazione, i tre thread lavorano “sfalsati”, nel senso che uno riceve, uno calcola e uno invia in ogni fase, in questo modo aumenta anche la quantità di dati in output

Un thread che è in stato di idle viene deschedulato e viene fatto subentrare uno che può calcolare.

Lo steady state di questo triple buffer è il seguente:

Buffers		Phase 1	Phase 2	Phase 3
B1	t1	rcv(B1)	f(B1)	snd(B1)
B2	t2	f(B2)	snd(B2)	rcv(B2)
B3	t3	snd(B3)	rcv(B3)	f(B3)

Quindi la strategia consiste nel creare più thread di quanti possono effettivamente lavorare in parallelo in modo da mascherare la latenza dovuta alla comunicazione.

Questa tecnica viene utilizzata dalle GPU in modo da non pagare il costo del trasferimento dei dati dalla CPU.

La stessa cosa la possiamo ottenere anche con il double buffer, in questo caso però abbiamo che le fasi di scrittura e di lettura sono uniche e non divise come nell'esempio precedente.

Questa strategia comporta dei miglioramenti anche per le performance, consideriamo la speedup rossa che è quella ideale in cui arrivati al limite fisico dovuto al numero di contesti abbiamo poi una linea orizzontale. In realtà quello che fa realmente la macchina è differente perché grazie al numero di thread maggiore del numero dei contesti abbiamo che lo speedup è quello blu perché in ognuna delle fasi c'è almeno un thread che calcola. (Nell'immagine c'è un errore, è 4 contesti, 8 thread per fare l'excess parallelism).<

In definitiva possiamo dire che l'excess parallelism diminuisce il tempo necessario all'esecuzione del programma in parallelo ma aumenta l'overhead dovuto alla creazione dei thread perché ora ne creiamo un numero maggiore rispetto al caso base del programma in parallelo.

Dobbiamo ovviamente stare attenti a non inserire un numero di

thread troppo alto perchè altrimenti l'overhead cresce troppo rispetto al miglioramento atteso in termini di tempo risparmiato.

Usando una Shared Memory Multicore dobbiamo gestire i problemi di data races, si sfruttano quindi dei meccanismi per la concorrenza.

Abbiamo lock, mutex e condition variables ad esempio.

L'utilizzo di questi meccanismi aumenta i possibili problemi, consideriamo ad esempio la locality, abbiamo una variabile X salvata da qualche parte e due processori che la portano nella loro cache e poi uno dei due la incrementa. Per la coherency anche l'altro deve aggiornare il valore di X, il problema è che poi devo anche avere un meccanismo per la sincronizzazione in modo da evitare data races, quindi avrò una mutex salvata in una cache che però deve essere replicata anche nell'altra cache, quindi in pratica l'utilizzo delle lock introduce un numero maggiore di condivisione di dati. Quindi si deve evitare di utilizzare le lock quando non servono e quando abbiamo delle alternative.

Introduzione a OpenMP

Consideriamo il problema del for in parallelo, con OpenMP è possibile eseguire in parallelo un for a patto che le varie iterazioni siano del tutto indipendenti tra loro e quindi possano essere parallelizzate.

Una soluzione di questo genere la otteniamo così:

```
#pragma omp parallel for
for(int i=.....){
    Body
}
```

Questo codice è una contrazione di due differenti pragma di OpenMP:

```
#pragma omp parallel nthread(n)
#pragma omp for
```

Con numThread indico quanti thread devono lavorare in parallelo, se questo parametro non viene specificato, la parallelism degree viene presa in automatico dal sistema operativo che nel file /proc/cpuinfo indica questo dato.

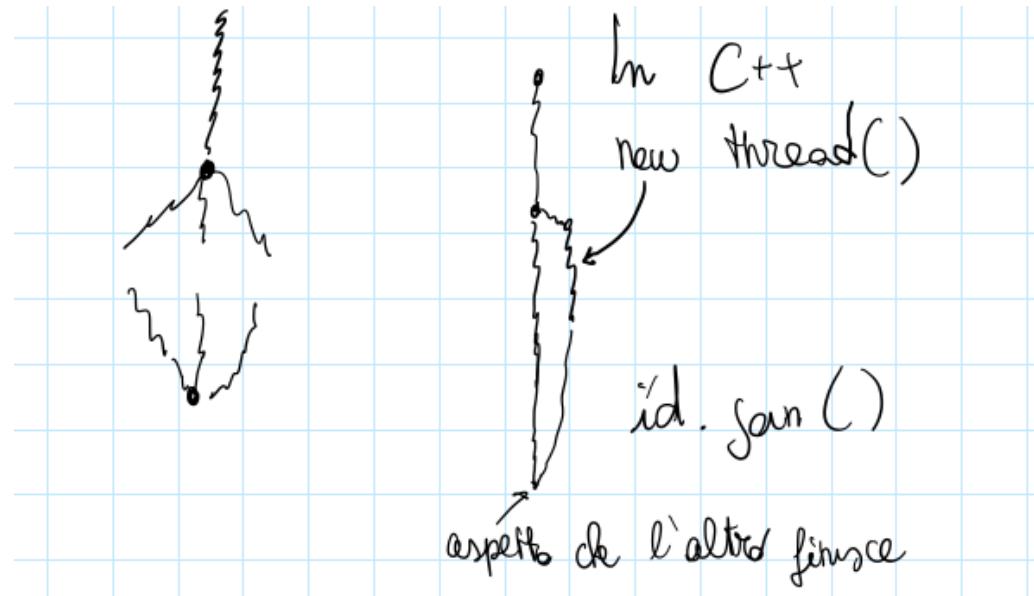
Lezione 10

OpenMp

OpenMp è un'astrazione che consente di progettare sistemi paralleli. OpenMp ha avuto una buona fortuna perchè è basato sull'uso di annotazioni nel codice, è basato sull'utilizzo di #pragma che non hanno effetti sul codice vero e proprio ma hanno invece effetto nel momento in cui viene compilato il programma.

Per abilitare il supporto ad OpenMp è necessario passare il flag -fopenmp al compilatore, senza questo le varie annotazioni non hanno alcun effetto.

Il modello di OpenMp riflette in qualche modo il modello Fork-Join che abbiamo visto anche in C++.



Arriviamo ad un punto del programma in cui dividiamo la computazione e creiamo vari thread, alla fine dovremo poi fare la join.

OpenMp alza il livello dell'astrazione e permette la creazione di un thread pool in cui però i vari thread possono svolgere funzioni differenti tra loro, cosa che invece non posso fare nel classico C++.

Consideriamo anche il modello della memoria, assumiamo di avere un memory shared, nel classico C++ quando creiamo il thread gli passiamo

il metodo che deve essere svolto e qualche parametro che poi diventa parametro della funzione. Se però vogliamo modificare qualcosa fuori dal thread dobbiamo utilizzare delle variabili globali che devono essere state definite prima del fork.

In OpenMp invece i thread ereditano l'ambiente in cui vengono creati ma poi ci sono dei metodi che possono essere utilizzati per annotare le variabili. In questo modo riusciamo a dire al thread il tipo di variabile che viene passata.

Questo metodo per segnare le variabili corrisponde all'uso:

- Private(x) : è qualcosa che modifica una pragma precedente e dice a quella pragma che la variabile x è stata creata locale e privata al thread che la utilizzerà.
- Shared(x)
- FirstPrivate(x): copia locale ma è stata inizializzata prima di creare il thread.
- LastPrivate(x): copia nel thread e poi l'ultimo valore che assume è quello che viene dato nel thread.

Si inizia con:

```
#pragma omp parallel num_thread(n){  
}
```

Il comando viene inserito tra le parentesi ed è meglio mettere le parentesi anche se non servirebbe. In questo caso definiamo il numero di thread, in alternativa viene preso dal sistema operativo.

Posso anche specificarlo in altro modo:

```
omp_get_num_thread()  
// Abbiamo anche la possibilità di settare una variabile  
// che mi indica quanti thread voglio usare  
export omp_num_threads = 4
```

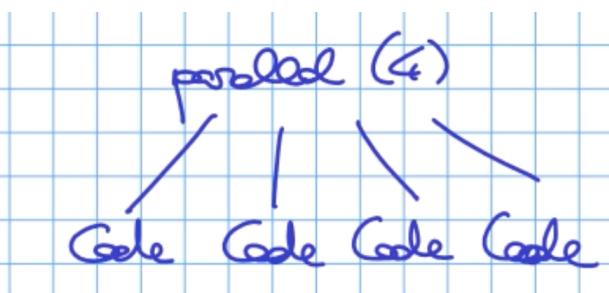
Se dentro al pragma precedente vogliamo eseguire qualcosa con un singolo thread possiamo usare:

```
#pragma omp single{  
}
```

C'è anche un pragma che mi consente di aggiornare le variabili che vengono utilizzate all'interno in modo atomico come se stessimo utilizzando una variabile atomica. All'interno di questo pragma dobbiamo anche inserire un comando di update

```
#pragma omp atomic{  
    update command  
}
```

Quando abbiamo una situazione di questo tipo:



Ciascuno dei thread ha un ID differente, se dentro ad uno di questi thread c'è un blocco single allora gli altri thread si bloccano e aspettano che questo finisca.

Dopo il comando parallel c'è uno scheduler che mi gestisce la creazione dei vari thread e che mi permette di eseguire i thread. Non sappiamo all'inizio su quale core verrà eseguito il thread.

Il parallel command in questo caso replica il codice nei vari thread.

Un primo esempio:

```
#include <iostream>
#include <omp.h>

int main(int argc, char * argv[]){
    if(argc != 1){
        auto nw = atoi(argv[1]);
        #pragma omp parallel num_threads(nw)
        {
            std::cout << "This is thread " <<
omp_get_thread_num() << " of " << omp_get_num_threads() <<
std::endl;
        }
    }
    else{
        #pragma omp parallel
        {
            std::cout << "This is thread " <<
omp_get_thread_num() << " of " << omp_get_num_threads() <<
std::endl;
        }
    }
    return 0;
}
```

C'è anche la possibilità di definire delle section, in questo caso definiamo varie sezioni che vengono eseguite in parallelo ma con codice differente all'interno e non come nel caso del pragma parallel in cui tutti i thread hanno lo stesso codice.

```
#pragma omp parallel
{
#pragma omp sections
{
```

```
#pragma omp section
{
    sec1
}

#pragma omp section
{
    sec 2
}
}
```

Quindi utilizzo le due sezioni per differenziare il codice che i due thread poi eseguono. All'interno delle due section posso inserire anche il pragma single.

```
#include <iostream>
#include <omp.h>

int main(int argc, char *argv[])
{

if (argc != 1)
{
    auto nw = atoi(argv[1]);
    #pragma omp parallel num_threads(nw)
    {
        #pragma omp sections
        {
            #pragma omp section
```

```
{  
    std::cout << "This is first section, thread " <<  
    omp_get_thread_num() << " of " << omp_get_num_threads() <<  
    std::endl;  
}  
  
#pragma omp section  
{  
    std::cout << "This is second section, thread " <<  
    omp_get_thread_num() << " of " << omp_get_num_threads() <<  
    std::endl;  
}  
}  
}  
}  
  
}  
}
```

Quando chiudiamo l'ultima parentesi del pragma parallel mettiamo in conto che dovremo attendere i thread che stanno ancora lavorando. Possiamo trovarci in una situazione differente, se ad esempio abbiamo creato due thread che però sono dipendenti tra loro nel senso che uno lavora con le variabili che produce l'altro, allora potremmo avere che il primo thread si blocca ad un certo punto in attesa dell'altro. Per sincronizzare questi due thread possiamo utilizzare una

```
#pragma omp barrier
```

all'interno delle due sections in modo che quando il thread arriva alla barrier si blocca e passa nello stato idle fino a che anche il secondo thread non arriva alla sua barrier e lo risveglia.

Altra cosa che possiamo fare con OpenMp è il Parallel For che ci dice che un certo for deve essere eseguito in parallelo.

Quando usiamo il parallel for stiamo dicendo al compilatore che nel body del for non ci sono dipendenze.

```
# pragma for
{
    for(.....){
        ....
    }
}
```

La Pragma For (Map pattern) è un pragma dove la variabile i che usiamo come iteratore è considerata privata dei vari thread.

Spesso è interessante avere anche un pattern per la reduce e non solamente per la map.

Se vogliamo fare una reduce che mi somma tutti gli elementi della map però non abbiamo più un for indipendente perchè il valore che abbiamo per la somma ad ogni iterazione dipende dalle precedenti iterazioni.

Ad esempio se noi abbiamo un array con 4 elementi e abbiamo 2 thread, uno somma i primi due e uno gli ultimi 2, però nessuno poi fa la somma completa perchè non l'ho specificato.

Quello che possiamo fare è utilizzare una pragma che mi parallelizza il for andando però ad eseguire una riduzione con l'operatore + sulla variabile in cui salviamo i risultati intermedi. In questo modo alla fine del loop otteniamo la somma di tutte le somme intermedie.

```
#include <iostream>
#include <omp.h>
```

```
#include <vector>

int main(int argc, char *argv[])
{

    if (argc != 1){
        auto nw = atoi(argv[1]);

        const int n = 1024;
        std::vector<int> v(n);
        auto actualSum = 0;
        for(int i=0; i<n; i++){
            v[i] = i+1;
            actualSum += v[i];
        }

        int sum = 0;
        #pragma omp parallel for num_threads(nw)
        reduction(+:sum)
            for (int i = 0; i < n; i++){
                sum += v[i];
            }

        std::cout << "Sum is " << sum << " should be " <<
actualSum << std::endl;
    }

    return 0;
}
```

Se vogliamo implementare la riduzione possiamo fare in due modi:

- Creiamo una variabile atomica “sum” e poi creiamo due thread che fanno un for in parallelo, un primo thread su metà della lista e l’altro thread sull’altra metà. Poi ogni volta che prendono un nuovo

elemento i due thread fanno la somma alla variabile globale sum.

Quindi non accade mai che i due thread svolgono una somma nello stesso momento.

- Si usano due thread separati che fanno le stesse cose della prima soluzione però solamente alla fine sommano le loro somme parziali alla somma totale. L'overhead è proporzionale al numero dei thread che vengono creati ma i miglioramenti in termini di tempo sono proporzionali al numero di thread che vengono utilizzati.

Per capire in che modo possiamo migliorare le prestazioni di un programma possiamo utilizzare dei tool come ad esempio VTune o GProf che svolgono una analisi del codice e cercano di identificare i “critical loop” ovvero quei loop che rallentano l'esecuzione.

Viene svolta una dependency analysis e poi possiamo mettere una #pragma prima delle parti di codice che possiamo parallelizzare.

C'è un'altra questione da considerare quando utilizziamo il parallel for, all'interno della pragma del parallel for abbiamo il for vero e proprio con il codice da eseguire in parallelo, alla fine di questo parallel for è come se ci fosse una barrier, quindi, se un thread finisce il parallel for si ferma nello stato idle e attende gli altri thread che devono arrivare alla barrier. Se ad esempio dopo il primo parallel for ne abbiamo un altro che potrebbe già essere eseguito questo può essere un problema perché abbiamo un thread idle che attende, il problema si bypassa andando ad inserire la parola nowait alla fine del pragma del parallel for.

C'è un altro pragma utile:

```
#pragma omp task
{
    code
}
```

Il codice all'interno di questo pragma viene eseguito come se fosse

indipendente, tra i vari thread che sono disponibili uno viene preso ed esegue questo codice, se ad esempio ho un for con dentro due task allora è possibile eseguire questi due task in modo indipendente.

Oltre al semplice task possiamo utilizzare il pragma taskwait in cui viene creata una sorta di barrier, in questo modo all'interno di una regione di codice parallelo è necessario attendere che ogni task creato sia anche terminato.

È anche possibile specificare le dipendenze tra i vari task:

```
#pragma omp task depend(in:x) depend(out:y) depend  
(inout:z){  
    code  
}
```

Questo task può essere eseguito in parallelo usando i thread però garantisce che vengano rispettate le dipendenze, ad esempio se c'è un altro task che ha come dipendenza in:y allora questo dovrà essere eseguito dopo il primo.

Il vantaggio in questo caso è che da un certo punto di vista abbiamo lo stesso vantaggio delle async call in C++.

Con le async call in C++ abbiamo la possibilità di svolgere una funzione in modo asincrono e questo restituisce una future che poi restituirà un valore in futuro.

Dire che un task ha una dipendenza è come creare un task async e poi fare la get attendendo che ci sia qualcosa in input.

Ancora sul parallel for:

Ci sono vari metodi per schedulare le iterazioni del parallel for:

Se scriviamo una cosa del tipo:

```
#pragma omp for schedule(argument)
```

Possiamo scegliere il parametro dell'argument tra static, dynamic e guided. Lo static scheduling mi dice che possiamo dividere l'iteration space in varie parti uguali, ad esempio possiamo specificare (static, 128 items) e quindi se abbiamo un blocco completo di 1024 elementi allora ne creiamo 8 indipendenti di 128 elementi e questo non dipende dal numero di thread.

Il dynamic scheduling: le iterazioni sono distribuite ai vari thread in blocchi.

Il guided scheduling: è simile al dinamico ma abbiamo un metodo differente per calcolare i blocchi.

C'è anche auto che può essere passato come argomenti, in questo caso la divisione viene fatta dal compilatore o quando viene eseguito il programma, lasciamo quindi libertà.

Come implementare una pipeline con OpenMP?

Tra il producer e il consumer abbiamo un buffer di una sola posizione e un bit di presenza che viene fissato a 0 inizialmente. Quando il producer mette qualcosa nel buffer modifica a 1 il presence bit. Quando il consumer vede il presence bit a 1 allora lo setta a 0 e prende il valore su cui deve lavorare.

Anche se non è un grande esempio di pipeline comunque fa vedere come funzionano le pipeline in OpenMp.

Esempio di codice di OpenMP per implementare una pipeline:

```
#include <iostream>
#include <thread>
#include <vector>
#include <chrono>
#include <omp.h>
#include <atomic>

using namespace std::literals::chrono_literals;

std::atomic<int> presenceBit;

int buffer;

int main(int argc, char * argv[]){
    presenceBit = 0;
    buffer = 0;
    auto delay = 1500ms;
    auto nw = atoi(argv[1]);
    const int n = 10;
    #pragma omp parallel num_threads(nw)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                std::cout << "This is the producer section, it is thread " <<
omp_get_thread_num() << " of " << omp_get_num_threads() << std::endl;
                for(int i=0;i<n;i++){
                    std::cout << "Waiting buffer free to send ..." << std::endl;
                    while(presenceBit == 1){
                        std::this_thread::sleep_for(1ms);
                    } buffer = i;
                    presenceBit = 1;
                    std::cout << "Sent!" << std::endl;
                    std::this_thread::sleep_for(delay);
                }
            }

            #pragma omp section
            {
                std::cout << "This is the second section, it is thread " <<
omp_get_thread_num() << " of " << omp_get_num_threads() << std::endl;
                for (int i = 0; i < n; i++)
                {
```

```
        while (presenceBit == 0)
        {
            std::this_thread::sleep_for(1ms);
        }

        std::cout << "Received " << buffer << std::endl;

        presenceBit = 0;
        std::this_thread::sleep_for(delay);
    }
}

return 0;
}
```

Lezione 11

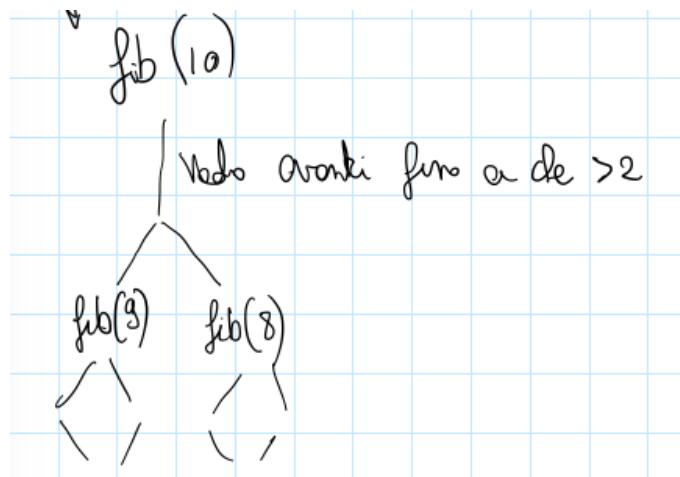
Un possibile utilizzo della pragma con depend:

```
#pragma omp task depend(in:x)  
  
#pragma omp task depend(out:x)
```

In questo caso il primo task non parte fino a che il secondo non termina.

Un'altra pragma è quella del taskwait, in cui indichiamo che devo attendere tutti i task dichiarati dentro al pragma prima di andare avanti a fare altro.

Questa pragma viene utilizzata per attendere tra una chiamata e l'altra della stessa funzione, ad esempio possiamo utilizzarlo per il calcolo della funzione di Fibonacci.



Nel caso dell'esempio abbiamo una chiamata iniziale a fib(10) all'interno del pragma, vengono lanciati due thread, uno che calcola fib(9) e uno che calcola fib(8). Questi due thread poi generano altre chiamate e alla fine ci fermeremo quando arriviamo al valore 2. Quando terminano tutti i calcoli (ad ogni chiamata) abbiamo un taskwait che prende i due risultati

intermedi e fa la somma, serve il taskwait perchè deve attendere che i precedenti siano completati prima di fare la somma.

```
#include <iostream>
#include <omp.h>

#define DEBUG

int fibonacci(int i) {
    int res;

    if (i<2)
        res = 1;
    else {
        int i1 = 0;
        // missing the shared -> does not work : i1 = firstprivate (default)
        #pragma omp task shared(i1)
        {
            i1 = fibonacci(i-1);
        }

        int i2 = 0;
        #pragma omp task shared(i2)
        {
            i2 = fibonacci(i-2);
        }

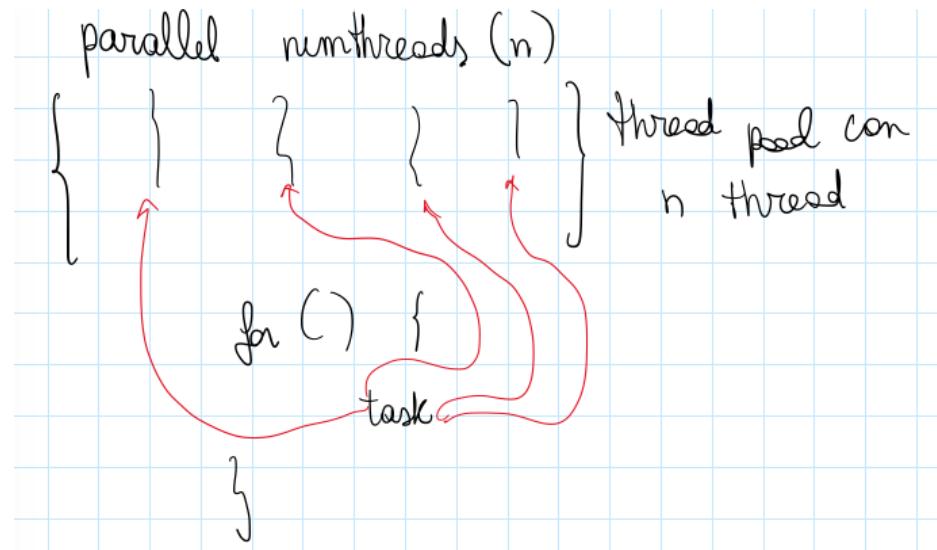
        // Ora attendiamo che finiscano tutti e due poi facciamo la somma.
        #pragma omp taskwait
        res = i1 + i2;
    }
    return res;
}

int main(int argc, char * argv[]) {

    int n = atoi(argv[1]);
    int nw = atoi(argv[2]);
    int res = 0;
    #pragma omp parallel num_threads(nw)
    {
        res = fibonacci(n);
    }
    std::cout << "Eventually: fibonacci(" << n << ")=" << res << std::endl;

    return(0);
}
```

Nel main ho il pragma del parallel con il numThread, quando ho questa pragma genero un thread pool con il numero di thread che viene indicato, ogni volta che trovo un task, questo viene assegnato ad uno dei thread che sono disponibili. Il match tra task e thread viene effettuato da OpenMP.

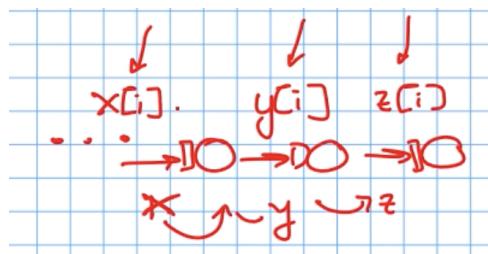


I thread nel thread pool in generale sono thread normali e non sono sticky, quindi viene fatta una distribuzione “round robin” dei task ai vari thread in modo che un core non lavori più degli altri.

Ci sono altri metodi per gestire i thread senza però utilizzare il dependency nella definizione del pragma.

Un esempio è **PipeFg**, in questo caso, date le funzioni f e g vogliamo fare in modo che queste vengano trattate come dei task e vogliamo usare le dipendenze in modo che i due task vengano eseguiti in modo concorrente.

Questo vuol dire avere due thread, uno deve calcolare $f(x)$ e uno $g(f(x))$, questo secondo sarà idle fino a quando il primo non avrà completato il suo compito. C’è concurrency ma non c’è parallelism in realtà.



Usare la closure “Depend” per implementare una pipeline non è il modo più intelligente da utilizzare perchè per una pipeline che usa un elemento X, poi Y e poi Z ci servono dei vettori e quindi abbiamo una limitazione dal punto di vista della memoria da utilizzare.

La nostra idea è che OpenMP è un po' troppo primitivo perchè si devono gestire ancora oggi dei concetti troppo a basso livello.

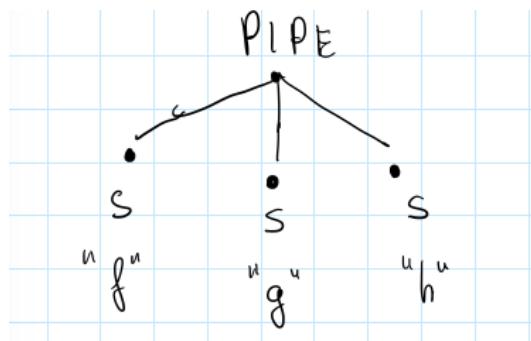
Refactoring Rule

Ammettiamo di avere un problema da risolvere, possiamo scrivere un programma parallelo che risolve il problema sfruttando un pattern, ad esempio la pipeline o la farm.

Utilizzando un albero di questo genere che mi indica il modo in cui funziona la pipeline e la farm possiamo utilizzare la refactoring rules.

La refactoring rule mi permette di applicare delle regole (eliminazione/introduzione di pipeline e farm) per fare in modo di ottenere un altro albero che calcola le stesse cose ma è più efficiente. Cosa posso fare per esplorare lo spazio delle soluzioni possibili per un certo problema?

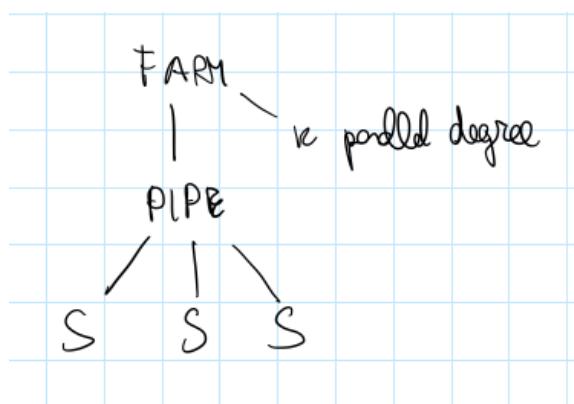
Partiamo con una computazione formata da tre stage, il primo stage calcola f, poi il secondo g e l'ultimo calcola h.



Se so che uno degli stage richiede un tempo maggiore rispetto agli altri, vediamo subito che c'è una bottleneck quindi per risolvere questo problema ed eliminare la bottleneck posso mettere una farm che all'interno ha un numero di worker che mi permette di abbassare il tempo di esecuzione di quello stage in modo da uniformarmi al tempo minore degli stage.

Vorrei poter fare questa scelta del numero di worker in modo automatico, non devo essere io che scelgo il numero di worker ma deve essere scelto durante l'esecuzione, lascio libertà alla libreria che prenderà la decisione migliore, questo si chiama autotune (viene fatto un tuning dei parametri in modo che si ottenga un risultato migliore).

Possiamo fare un'altra trasformazione per lo stesso albero:



Posso avere un albero in questo modo?

Dato che questi aspetti sono molto astratti, quello che posso fare è fare una astrazione del mio programma guardando solamente il pattern tree cercando di capire se all'interno del pattern tree c'è qualcosa di meglio

che posso fare senza però andare a scrivere tutte le possibili situazioni e poi vedere quale è migliore.

Programmare queste soluzioni può essere un problema.

È stato sviluppato RPLSH che è un tool che permette di esplorare lo spazio dei vari possibili pattern trasformation.

All'interno di questa libreria abbiamo la possibilità di creare delle pattern expression e di combinarle. Per ogni parte di questa espressione abbiamo un tempo di esecuzione differente. Abbiamo la possibilità di “giocare” con i vari tipi di pattern e ottenere una pattern expression migliore.

Come faccio a trasformare i pattern durante l'esecuzione?

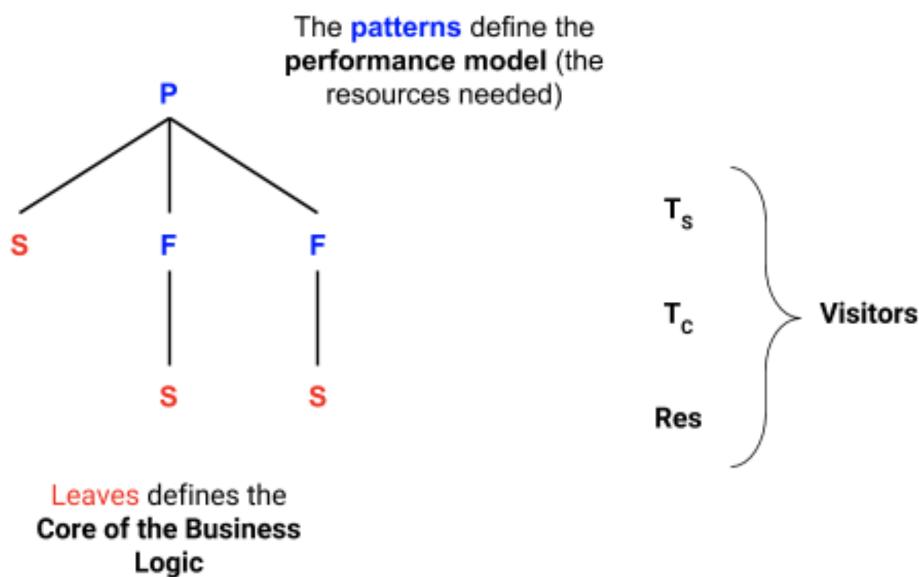
Abbiamo la possibilità di chiamare i differenti pattern e di chiamare differenti entry dei framework che mi permettono di usare i pattern.

Se ho ad esempio una pipeline con degli stage sequenziali, il pipeline manager dovrebbe capire che uno stage è troppo lento guardando il Tc e la lunghezza della coda e poi deve dire al thread che sta eseguendo quello specifico stage di non eseguire più la stessa cosa ma di eseguire una farm di questo stage. Devono essere resi disponibili dei metodi per fare delle chiamate alle varie librerie per permettere queste modifiche.

Lezione 12

Dato l'albero dei pattern visto anche nella precedente lezione, abbiamo i nodi in blu che sono i pattern e dati i pattern conosciamo la semantica, le performance e parte dell'implementazione che mi dice la quantità di risorse necessarie per usarle.

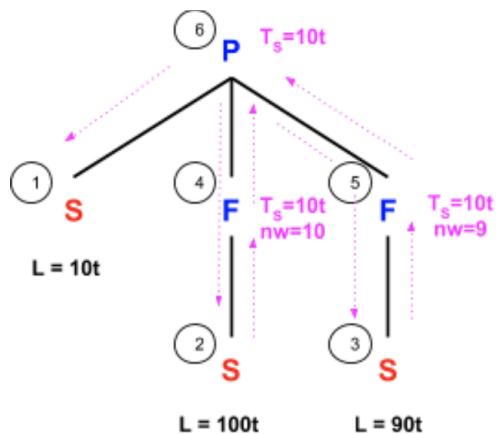
L'insieme delle foglie invece sono sequenze di codice sequenziale e sono il "Business logic code".



In particolare tutti i modelli che abbiamo per Service time, completion time e resources sono implementati come "Visitors".

Avere una implementazione come "visitors" vuol dire che quello che faccio per capire il service time di un albero è una visita fino alla prima foglia, poi salgo e riscendo alla foglia successiva, così vado avanti fino alla fine.

Ad esempio, se devo calcolare il service time: parto dal root, arrivo al nodo 1 che è un nodo sequenziale e latency = Ts ed è anche una foglia, dopo aver visitato il nodo 1 torno verso l'alto e poi scendo di nuovo, vado in 4 e poi in 2 dove trovo un'altro Ts, poi faccio ancora una volta la stessa mossa e arrivo alla foglia 3 che ha il suo Ts.



Climbing back from node 2 to node 4, since it is **complete subtree**, we can define the performance model of the pattern in 4. (Farm) Thus, defining the T_s of the Farm.

$$T_s(Farm) = \max(T_{emitter}, T_{collector}, \frac{T_{worker}}{nw})$$

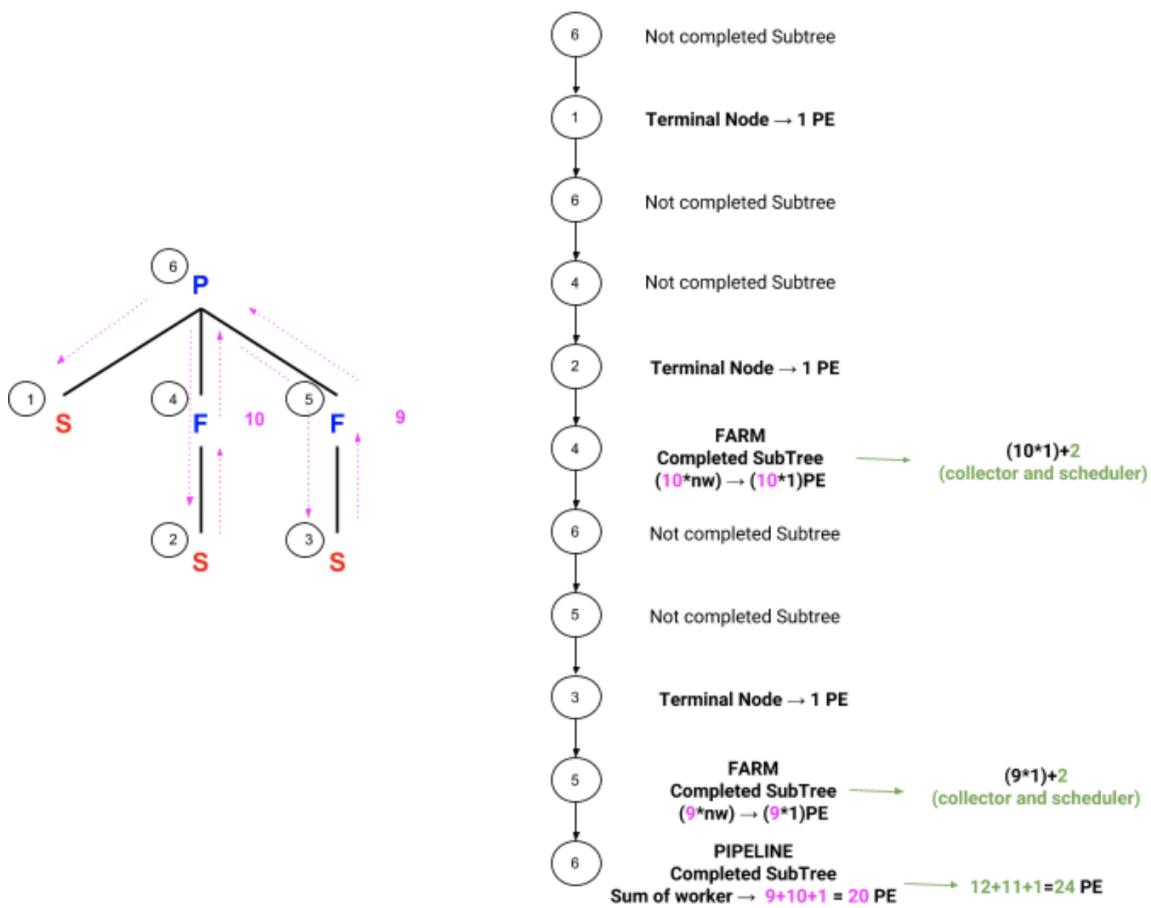
Quando torniamo dalla foglia 2 al nodo 4 qua scriviamo il T_s e dato che vogliamo avere un tempo uguale alla prima foglia prendiamo in considerazione il numero di worker pari a 10 e quindi dividiamo il T_s della foglia per 10. Quindi nel nodo 4 scriviamo un T_s pari a 10. La stessa cosa la facciamo con la foglia 3 e il nodo 5, in questo caso abbiamo una latenza di 90 e assumendo di avere 9 worker associamo al nodo 5 un T_s pari a 10.

A questo punto passiamo al nodo 6 che è una pipeline e possiamo dire che il service time è il massimo dei vari stage, in questo caso abbiamo che il service time totale è 10 perché i tre nodi hanno lo stesso service time.

Per la Farm il service time è il valore massimo tra T_s dell'mitter, del worker e il time dei worker diviso il numero di worker.

Con una visita simile a quella che abbiamo già fatto possiamo anche calcolare le risorse necessarie a questo pattern tree, quindi partendo dal root facciamo la seguente visita 6->1->6->4->2->4->6->5->3->5->6.

Per ogni foglia in cui ci troviamo calcoliamo le processing unit necessarie poi nel nodo superiore facciamo il calcolo completo, per esempio il nodo 4 ha 12 PE perchè la farm ha 10 worker più un emitter e un gatherer.



Ottimizzazione con sensori e attuatori

Supponiamo di avere una structure parallel application a cui sono collegati dei sensori e degli attuatori che sono poi collegati ad un decision process.

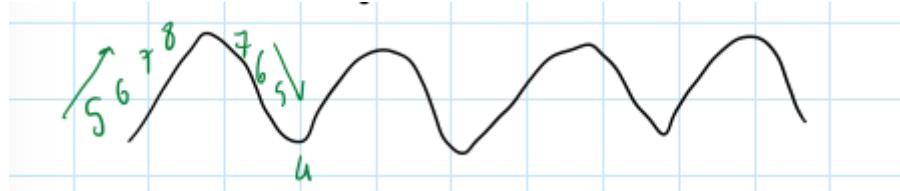
I sensori devono essere in grado di capire se ho rallentamenti nell'esecuzione del programma e se le code sono troppo piene.

Gli attuatori poi devono fare delle modifiche per risolvere il problema che è stato evidenziato.

Organizzando la soluzione in questo modo si presenta però un altro problema, quanto deve essere veloce la mia reazione quando il sensore trova un problema?

Se ho la possibilità di modificare immediatamente la situazione può capitare che il sensore dice che una coda è troppo piena e allora io modifco subito quella parte del programma, però può capitare che

aumento troppo il parallel degree, possiamo finire facilmente in una situazione in cui aumento e diminuiscono di continuo le risorse destinate alla computazione senza però arrivare ad uno steady state.



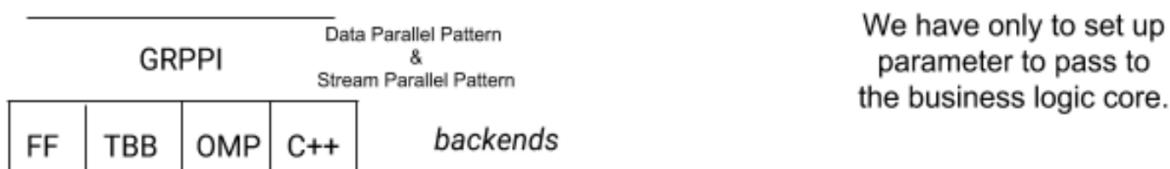
Quindi possiamo fissare un tempo, ad esempio posso decidere che faccio una modifica, poi per 30 secondi aspetto e poi eventualmente modifichiamo nuovamente la stessa parte del programma.

Se invece siamo proattivi e cerchiamo di anticipare le modifiche probabilmente nell'immediato futuro arriveremo ad uno stato che non ci piace e non è buono per le performance. Ad esempio, decido che la mia app deve analizzare 1 milione di pacchetti al secondo, se vedo che partiamo da un numero basso e poi saliamo fino ad arrivare ad un numero vicino ad un milione, potrei fare una modifica e alzare il numero massimo a 1.5 milioni senza che venga superato il limite. Questo è qualcosa che faccio in anticipo e potrebbe capitare che in realtà non supero mai 1 milione quindi già prima avevo dedicato troppe risorse e ora ne ho dedicate ancora di più senza che sia necessario.

GrPPI Framework

GrPPI (Generic parallel pattern interface) è stato sviluppato inizialmente come interfaccia per parallel pattern che però potesse essere compatibile con i metodi e le interfacce di C++. Ci consente un accesso al data e stream parallel pattern senza esporre l'implementazione.

APPLICATION



Possiamo vedere Grppi come un layer perché sotto ad esso abbiamo vari backend come ad esempio FastFlow, TBB, OMP e i thread nativi di C++, per lavorare con Grppi non dobbiamo conoscere il backend che c'è dietro ma solamente la libreria Grppi.

Non possiamo usare questi framework tutti insieme, dobbiamo decidere quale di questi utilizzare. A seconda della situazione e del problema da risolvere i vari backend avranno delle performance differenti.

Ci sono alcuni concetti da considerare:

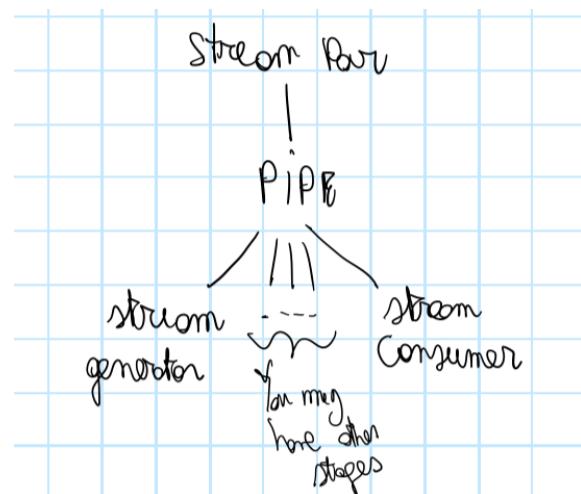
- **Stream**: abbiamo sia i data parallel pattern che lo stream parallel pattern. Lo stream parallel è implementato con un concetto astratto, non abbiamo un tipo stream perchè gli stream sono implementati con una sequenza di oggetti <T>.

Option<bool> è un tipo per cui la variabile può avere un valore assegnato true o false oppure può non avere un valore, quindi {}. Se dichiariamo “x : optional<int>” possiamo utilizzare un intero oppure {} per indicare che non c'è un valore associato.

Se abbiamo qualcosa allora inseriamo il valore altrimenti le parentesi.

Come conseguenza abbiamo che nello stream parallelism dobbiamo definire una pipeline che ha almeno due stage, uno è lo stream generator e l'ultimo è lo stream consumer.

Nel mezzo poi possiamo anche avere altri stages.



Non vale la stessa cosa per i data parallel pattern, possiamo ad esempio definire una pipeline specificando dei parametri, il primo è un executor che è quello che indica quale backend utilizziamo, poi abbiamo almeno due stage che sono delle funzioni che devono essere eseguite.

Se come primo stage uso qualcosa che produce interi e poi metto una seconda funzione che consuma floats c'è un problema.

In generale definiamo la pipeline gli stage, lo stream generator e lo stream collapser.

`pipeline(executor, function1, function2,...)`

Con la farm il discorso è molto simile.

La farm ha due parametri, uno è il numero di worker e uno è la funzione che deve essere eseguita.

`Farm(nw, fw).`

Quando si usa una map invece dobbiamo specificarla in questo modo:

`map(executore, iterator begin, iterator end, iterator begin, f)`

Il primo e il secondo iteratore mi indicano inizio e fine del vettore da cui prendiamo i dati mentre l'ultimo iteratore mi indica dove devo scrivere i dati.

Lezione 13

I data Parallel Pattern sono i seguenti:

- Map
- Reduce
- Stencil
- Scan

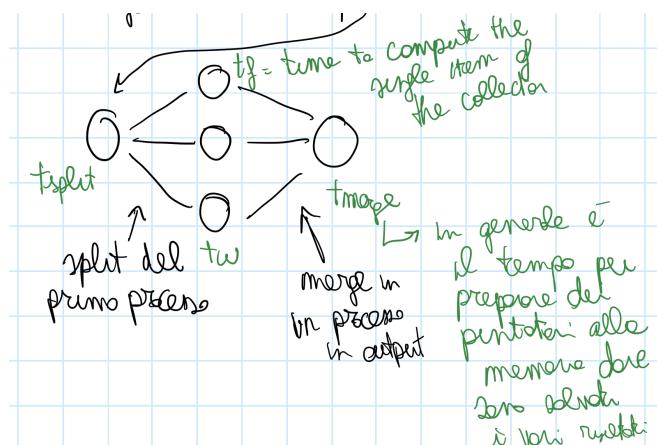
E sono quelli che vengono utilizzati per diminuire la latenza.

Consideriamo, ad esempio, la **Map**, abbiamo in input un `Collector<Type>`, poi usiamo una funzione e abbiamo in output un `Collector<TypeOut>`.

Per implementare questo tipo di parallelismo dobbiamo dividere i dati in parti uguali e assegnare queste parti ad un thread pool che esegue la map sulla sua parte di dati.

Alla fine poi i dati prodotti da ciascuno di questi thread verranno uniti in un unico risultato finale.

Possiamo vedere questa serie di operazioni che devono essere svolte all'interno di un "grafo", abbiamo inizialmente un tempo iniziale che viene speso per eseguire lo split dei dati in un tempo T_{split} , ogni thread si prende dei dati e calcola la funzione sui dati che ha preso in un tempo T_f . Poi alla fine abbiamo il tempo del Merge dei dati che è un T_{merge} e dipende dal tempo che serve per preparare i puntatori alla memoria dove verranno salvati i risultati.



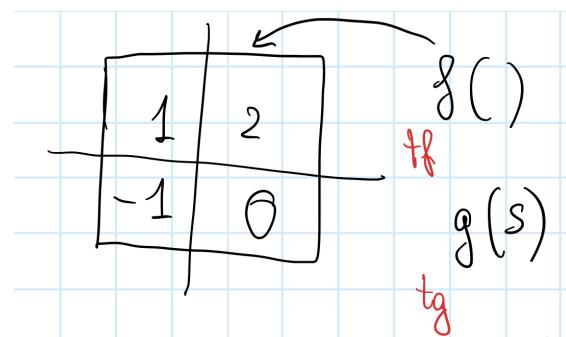
Per quanto riguarda la fase di split dei dati su cui si deve lavorare, data ad esempio una matrice $n \times n$, devo prendere una dimensione e la devo dividere per il numero di worker. La cosa più logica è dividere per righe e dare ad ogni thread i dati su cui deve lavorare.

Il problema è che se divido semplicemente N/nw avremo una partizione che ha più dati delle altre, quindi la tecnica più comune è prendere l'ultima parte e dividere i dati in eccesso tra le varie partizioni. Quindi ad esempio invece di avere una divisione 3,3,5 preferisco avere una divisione 4,4,3, in questo modo ogni worker tranne l'ultimo ha $(N/nw) + 1$ dati su cui lavorare.

Questa suddivisione dei dati segue la owner computes rules ovvero abbiamo vari worker che lavorano su parti differenti dei dati e non c'è possibilità che il worker i possa toccare la parte di dati che è stata assegnata al worker $i+1$.

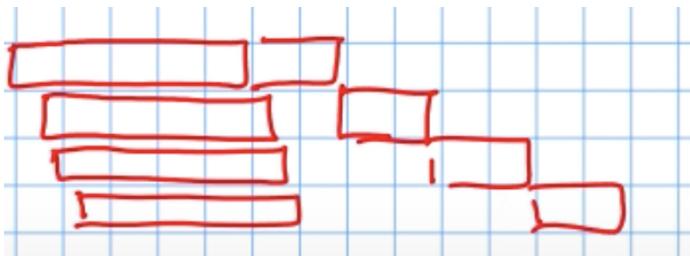
Prendiamo ad esempio una suddivisione dei dati in cui abbiamo una situazione del genere, vari dati e due funzioni, la funzione f la utilizziamo per trasformare i dati presenti all'interno della matrice, poi abbiamo una seconda funzione g che invece viene utilizzata per calcolare il risultato finale che poi viene scritto in uno stato finale globale.

Ad esempio la funzione g potrebbe essere utilizzata per sommare i valori della matrice per riga.



Il tempo che viene speso per calcolare la funzione g è importante sulla scalabilità della computazione perché possiamo pensare che per ogni elemento su cui dobbiamo lavorare spendiamo un tempo T_f che è

qualcosa che si sovrappone tra le varie computazioni che devono essere svolte, poi però devo calcolare un Tg che deve essere eseguito in mutua esclusione e quindi mi può cancellare tutti gli sforzi che ho fatto per avere una esecuzione in parallelo.



In una situazione come questa, non considerando l'overhead che ho all'inizio quando partono i vari thread per calcolare Tf, possiamo calcolare il tempo totale. Abbiamo m elementi su cui lavorare e per ognuno un tempo Tf che deve essere diviso per il numero di workers. Quindi abbiamo $(m * Tf) / nw$, questo va sommato con il numero di elementi moltiplicato per il tempo Tg e questo non lo posso dividere per il numero di workers.

Se ad esempio abbiamo un tempo per il calcolo di Tf che è $10Tg$ allora ci troviamo ad avere una prima parte del calcolo che possiamo abbassare nel momento in cui alziamo il valore di nw, però avremo sempre la seconda parte che invece non può essere abbassata. Quindi ci troviamo ad avere una speedup massima che dipende da Tf/Tg .

Consideriamo poi il pattern dello **Stencil**, in questo caso per aggiornare un certo elemento dobbiamo considerare non solo quell'elemento ma anche gli elementi vicini.

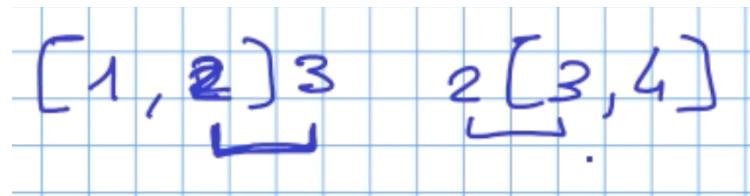
Se ad esempio devo fare la Map e partendo da un vettore voglio aumentare di 1 il valore degli interi che sono presenti all'interno, posso farlo in place senza dover allocare altro spazio.

Fare la stessa cosa con la stencil non è possibile, prendiamo ad esempio un array [1,2,3,4] e vogliamo calcolare la media degli elementi che sono vicini tra loro, vorrei ad esempio un array risultato di questo tipo [1.5, 3, 4.5, 3.5]. Il problema qua è che se io faccio un aggiornamento immediato dell'array in place potrei trovarmi in una

situazione in cui i primi due elementi sono corretti e aggiornati e poi quando vado a prendere il terzo elemento per fare la media mi trovo con il secondo che è già stato aggiornato e quindi comporta un valore errato della media, ad esempio $[1.5, 3, 3, 4] \rightarrow [1.5, 3, 5, 4]$.

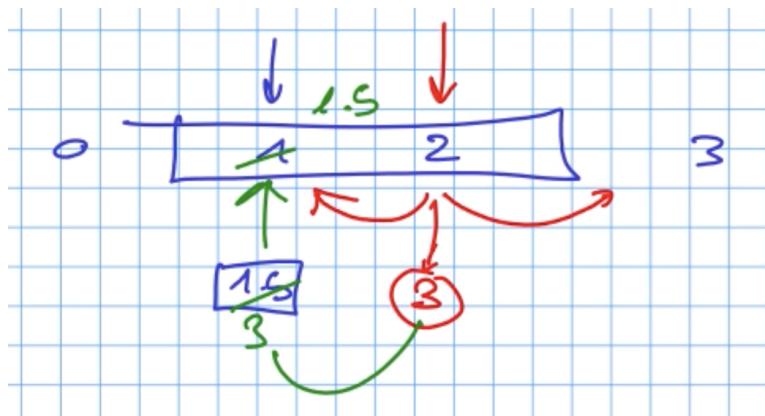
Quindi lo stencil non lo posso fare totalmente in place. Però posso copiare parte dei dati per fare in modo di ottenere il risultato corretto, in questo modo posso fare la computazione in parallelo e posso anche fare la sostituzione in place.

Ad esempio se facciamo una divisione dei dati in due parti abbiamo due array e devo copiare il primo elemento della seconda parte alla fine del primo vettore e l'ultimo elemento della prima parte all'inizio del secondo vettore per ottenere il risultato corretto.



Problemi di questo tipo non ce li abbiamo quando usiamo uno stream parallelism (pipeline, farm), questo perchè ogni task della farm è indipendente dagli altri e anche per la pipeline vale lo stesso discorso. Ad esempio la media nello stesso modo in una pipeline la posso fare in vari modi:

- La soluzione più semplice comporta la creazione di un vettore differente
- La soluzione più elegante consiste nell'utilizzo di un buffer in cui inserisco il risultato temporaneo, questo buffer lo committo nel momento in cui mi sposto avanti di una posizione e quindi non ho più bisogno di quel valore.



Ovvero, voglio calcolare la media di 0, 1 e 2, calcoliamo 1.5 e lo salviamo nel buffer. Poi calcoliamo la media di 1, 2 e 3, calcoliamo 2, a questo punto per la prossima iterazione 1 non ci servirà più quindi 1.5 lo committiamo e lo salviamo al posto di 1 e nel buffer ci memorizziamo 3.

Sempre restando in tema di Stream Parallel Pattern abbiamo la possibilità di fare una composizione di pattern, ad esempio partendo da $\text{Comp}(\text{Map}(f), \text{Map}(g))$ possiamo arrivare ad avere $\text{Map}(\text{Comp}(f,g))$, questa si chiama **Map Fusion Rule**.

Quindi nella prima composizione abbiamo due esecuzioni della map che quindi richiedono una struttura intermedia per mantenere i risultati del calcolo della f, nel secondo caso invece abbiamo solamente una map. Questo è un modo molto utile, specialmente se andiamo da sinistra a destra, per aumentare la quantità di computazione che svolgiamo per ogni blocco di dati che abbiamo creato dividendo i dati iniziali, in questo modo possiamo tollerare meglio l'overhead che abbiamo quando facciamo la divisione dei dati e il merge delle partizioni.

Consideriamo l'operazione di **reduce** quindi partiamo da una collezione di dati e vogliamo fare una somma degli elementi che sono all'interno di questa collezione.

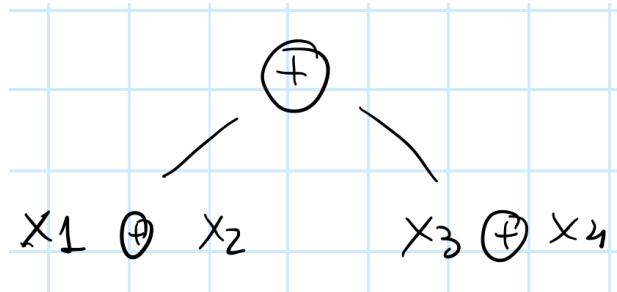
Per implementare questa reduce abbiamo due possibili metodi per posizionare le parentesi e quindi fare la somma finale ma in entrambi i casi va comunque fatta in modo seriale.

$$\begin{aligned} & \left((x_1 \oplus x_2) \oplus x_3 \right) \oplus x_4 \\ & x_1 \oplus (x_2 \oplus (x_3 \oplus x_4)) \end{aligned}$$

Cerchiamo di capire le performance che possiamo raggiungere, abbiamo m posizioni all'interno dell'array e nw thread per il calcolo in parallelo.

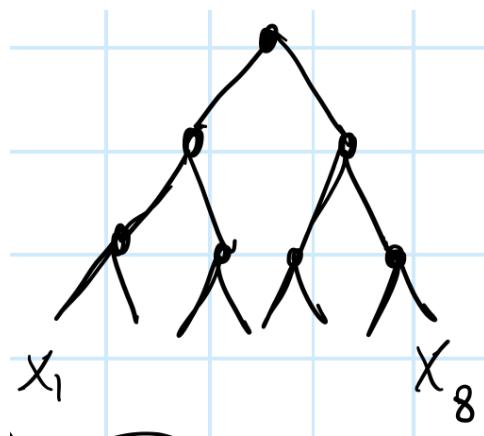
Abbiamo una operazione F che calcola la singola somma.

Quindi ad esempio se abbiamo 4 elementi possiamo dividerli in due blocchi e fare prima due operazioni di somma e poi sommare i risultati.



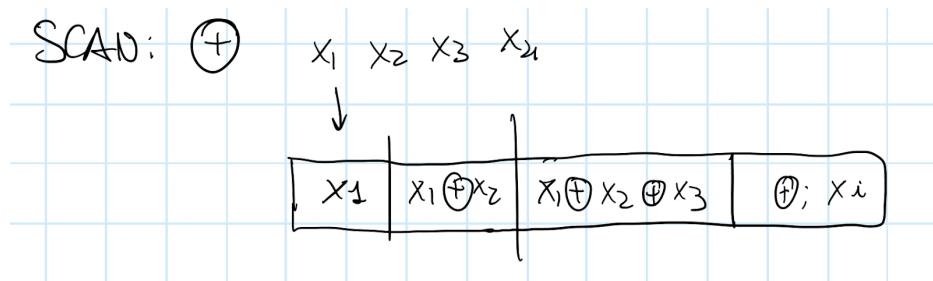
Se ne abbiamo 8 l'albero diventa più profondo e devo fare alcuni calcoli in più, manterrò una coda in cui inserisco i dati dei risultati parziali e poi ognuno dei thread prende uno di questi risultati parziali e fa la somma che deve essere fatta.

In generale quindi partendo con m task e nw workers, abbiamo che ogni thread dovrà calcolare $(m/nw - 1)$ operazioni di somma e in più dovranno essere calcolate $T_+ + (nw-1)$ operazioni per le somme superiori dei risultati intermedi.



Un pattern simile a questo della reduce è lo **Scan** che viene utilizzato quando facciamo una scansione del vettore.

In generale questa scansione del vettore può essere fatta in tanti modi, in generale è un algoritmo sequenziale perché scorro il vettore e sommo l'item attuale con il precedente.



In generale il tempo per l'esecuzione sequenziale dipende dal numero di elementi che abbiamo e dal tempo che impieghiamo per fare una singola operazione di somma.

Possiamo anche farlo in parallelo, dividiamo il vettore in varie parti e calcoliamo lo scan delle due parti, poi l'ultimo valore che ho sommato nella prima parte lo devo andare a sommare a tutti i valori della seconda parte.

Quindi la scan in parallelo può essere suddivisa in varie parti:

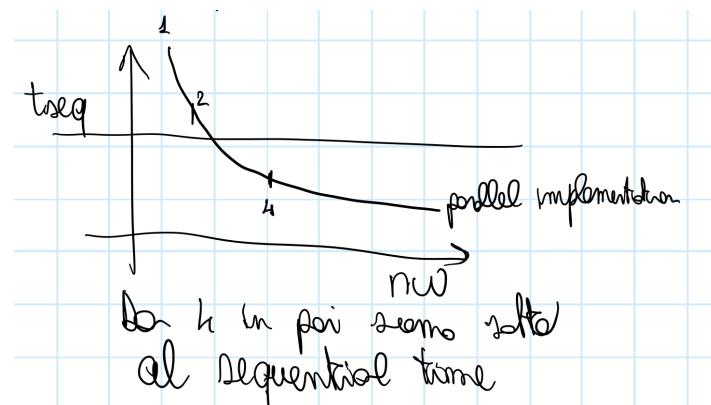
- Split delle partizioni e scansione delle partizioni locali
- Poi devo tornare all'inizio di ognuna di queste partizioni e devo sommare quello che ho calcolato nella partizione precedente.

Se ho m elementi e n_w worker abbiamo:

- m/n_w elementi in ogni partizione, facciamo la scansione di quella partizione e questo mi costa $(m/n_w - 1)T_+$
- Prendiamo l'ultimo elemento di ogni segmento e facciamo la scansione del vettore con gli elementi finali sommandoli, abbiamo n_w elementi in questo vettore e quindi avremo un costo $(n_w - 1)T_+$ per ottenere il vettore finale con gli elementi che dovrò sommare.
- Poi devo rifare la scansione di ogni segmento sommando l'elemento corrispondente presente nel vettore somma. Quindi di nuovo un tempo $(m/n_w)T_+$ per eseguire l'update.

In totale avremo un costo $((m/n_w - 1)T_+) + ((n_w - 1)T_+) + ((m/n_w)T_+)$ ovvero $2((m/n_w - 1)T_+) + ((n_w - 1)T_+)$.

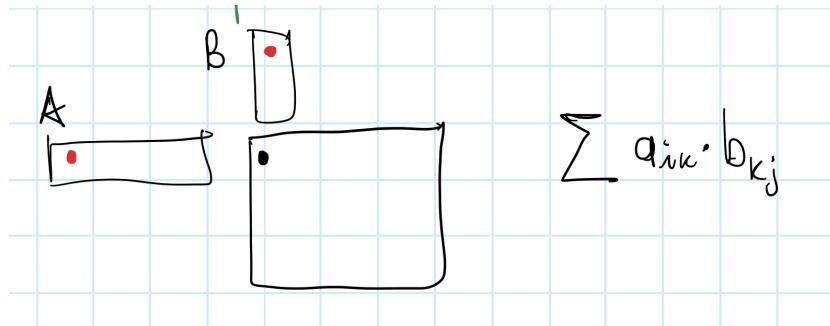
Abbiamo quindi più operazioni del caso sequenziali ma molte di queste possono essere svolte in parallelo.



Possiamo considerare il tempo sequenziale confrontandolo con il tempo parallelo, vediamo che se abbiamo un numero di worker maggiore di 4 avremo che siamo comunque sotto al tempo sequenziale.

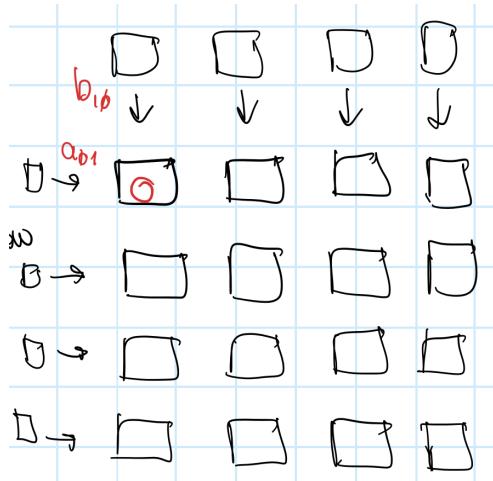
Consideriamo il caso della moltiplicazione delle matrici.

Abbiamo una matrice A e una B che devono essere moltiplicate, prendiamo una colonna della matrice B e una riga dalla matrice A e facciamo in modo che l'elemento della matrice finale sia la somma dell'elemento in posizione k della riga i di A moltiplicato per l'elemento in posizione k della colonna j della matrice B.



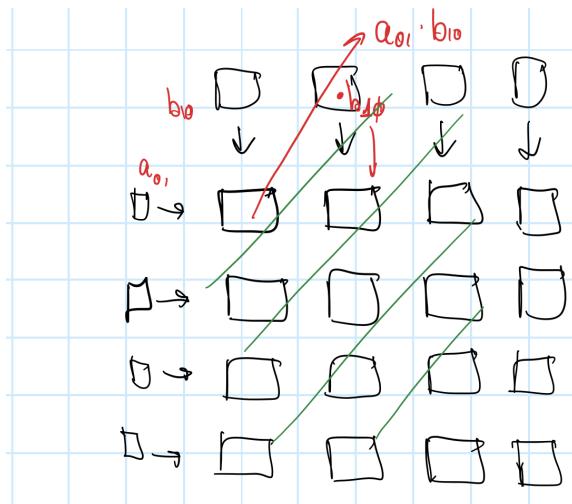
Questo è il modo classico per pensare l'algoritmo, possiamo però pensarla in modo differente, abbiamo degli elementi che comunicano in una mesh a due dimensione e sono capaci di calcolare la moltiplicazione e la somma.

Una singola iterazione k calcola la moltiplicazione e fa la somma, supponiamo che qualcuno mi manda per ogni ciclo di clock gli elementi della riga i e della colonna j.



Abbiamo quindi vari step:

- Al primo step abbiamo (in posizione 0,0 della matrice finale) A[0,0] e B[0,0], faccio una moltiplicazione e una somma e poi tengo il risultato.
- Al secondo step poi abbiamo che viene inviato il secondo elemento del vettore, quindi A[0,1] e B[1,0], viene fatto il calcolo e poi quello che era in posizione 0,0 viene spostato nella cella vicina dove viene fatto il calcolo considerando anche B[0,1].



Quindi abbiamo 1 elemento alla volta da ognuna delle due matrici.

Se la matrice è m^*n abbiamo:

- n step per far muovere gli elementi attraverso la prima riga e la prima colonna
- n step per attraversare tutta la matrice.

Quindi abbiamo $2*n$ step, utilizzando un n^2 array di executor, alla fine abbiamo la matrice moltiplicata in place all'interno dell'array.

Questo metodo per il calcolo delle matrici è stato riscoperto ora perché è utile per fare il training delle reti neurali.

Lezione 14

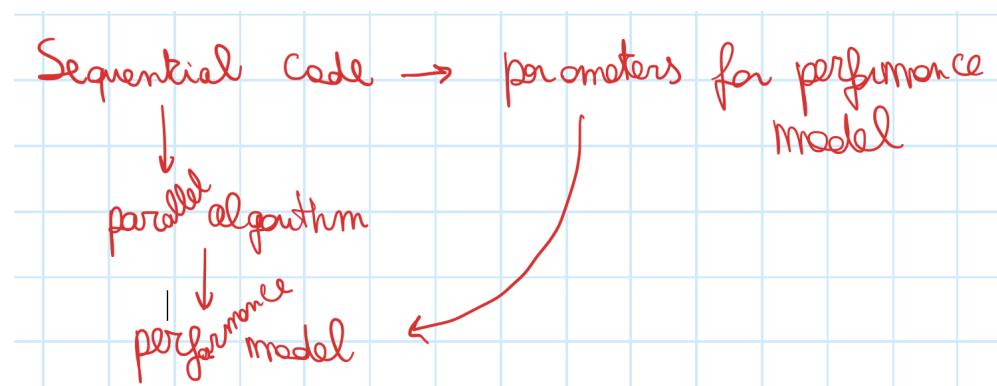
Nella prima ora ha analizzato il progetto dello scorso anno, Parallel Prefix, c'è una possibile soluzione efficiente per fare la parallelizzazione di questo problema (Disponibile il file jupyter).

Per quanto riguarda lo sleep di un thread:

- Se utilizziamo la funzione `std::this_thread::sleep(time)` mettiamo in pausa il thread che quindi non continuerà ad essere eseguito sul processore ma andrà a sospendersi lasciando spazio ad un altro thread.
- Se invece faccio attesa attiva ho un thread che continua a lavorare e controlla magari se una condizione è valida o no, in questo caso non lascio il processore e non posso fare altro in parallelo.

Quando scriviamo una implementazione parallela di un problema che inizialmente abbiamo risolto in modo sequenziale dobbiamo considerare che se ci metto di meno a fare l'esecuzione sequenziale che quella parallela anche se aumento i thread vuol dire che l'overhead è maggiore rispetto al costo effettivo delle varie operazioni.

Consideriamo un codice sequenziale, da questo dobbiamo comprendere quali sono i parametri che dobbiamo utilizzare per istanziare il modello, questi sono i parametri del performance model.



Con l'algoritmo sequenziale e dall'algoritmo parallelo possiamo cercare di capire il performance model.

Questo performance model, utilizzando i parametri che abbiamo scelto in precedenza, può darci un'idea delle performance del nostro algoritmo. Devo cercare di capire con il performance model quali sono le parti che mi prendono più tempo e come perdo tempo (overhead).

Consideriamo un altro problema:

Abbiamo una immagine e vogliamo posizionare sopra a questa immagine una seconda immagine in bianco e nero che fa da watermark. Il lavoro che deve essere svolto in questo caso può essere identificato con un pattern di tipo map in cui, abbiamo una funzione f che deve essere eseguita per ognuna delle immagini:

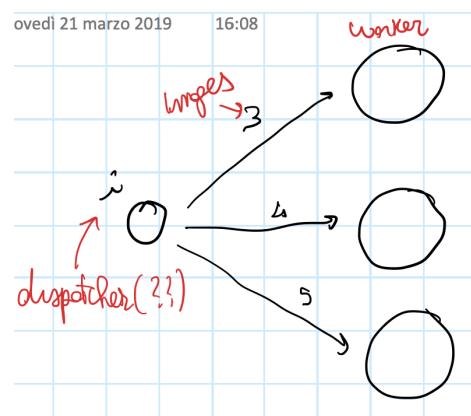
- Prima carico l'immagine
- Poi la processo
- Poi memorizzo la versione aggiornata dell'immagine

L'algoritmo dovrà fare i seguenti passaggi:

- Fase iniziale di Init: leggiamo le immagini dal disco
- Set up del funzionamento in parallelo
- Applicare la funzione Map

Lezione 15

Consideriamo sempre il problema dell'Image processing della scorsa lezione in cui abbiamo dei worker che svolgono il lavoro e delle immagini in input che devono essere prese dai worker per applicare un watermark.



Consideriamo i seguenti tempi per l'esecuzione dell'algoritmo:

- L'immagine viene caricata in circa 0.6 secondi
- L'immagine viene processata in 0.4 secondi
- L'immagine viene poi scritta sul disco in 1.4 secondi

In totale abbiamo un tempo T_w di 2.45 secondi.

Ammettiamo di avere una banda di 300 MB/s, possiamo concludere che:
 $T_w = T_w_{\text{parallelo}} + T_w_{\text{seriale}}$, dove il tempo seriale è il tempo per l'accesso al disco.

Abbiamo uno stream di n_w worker che eseguono questo lavoro, quindi il tempo T_w deve essere diviso in n_w worker ma poi rimane comunque il tempo seriale che non può essere parallelizzato perché ogni worker fa questa operazione e non la posso parallelizzare perché lavorano su risorse condivise.

Quindi le performance dello stream sono: $(T_w_{\text{parallel}}/n_w) + (T_w_{\text{seriale}} * n_w)$.

Un'altra versione dell'algoritmo per risolvere questo problema.

Possiamo utilizzare OpenMP e ci sono comunque varie possibili soluzioni:

- Possiamo creare una sections e all'interno varie sezioni in cui processiamo le immagini in modo indipendenti l'una dall'altra
- Possiamo utilizzare un parallel for, questa è la soluzione più semplice e veloce
- Possiamo utilizzare il pragma parallel all'esterno e poi all'interno il pragma omp task in cui in parallelo i vari thread svolgono tutti lo stesso lavoro ma su immagini differenti

Un'altra alternativa è utilizzare i pattern.

In questo caso partiamo con la versione sequenziale, abbiamo un for e all'interno del for per ognuna delle immagini eseguiamo tutte le trasformazioni che vogliamo fare.

Possiamo utilizzare GrPPI per fare una trasformazione del codice sequenziale e inserire l'utilizzo di alcuni pattern.

Ad esempio con GrPPI possiamo introdurre una pipeline al cui interno inseriamo tre stage, il secondo sarà una farm in cui vengono svolte le varie operazioni sulle immagini.

Se consideriamo i tempi che sono stati indicati all'inizio possiamo vedere che abbiamo un tempo molto elevato per l'accesso al disco, quindi una bottleneck è causata proprio da questo accesso al disco.

Vogliamo trovare il modo di bypassare questa bottleneck e anche qua abbiamo vari metodi per organizzare la computazione:

- Possiamo creare una farm al cui interno inseriamo una pipeline che svolge le tre operazioni: farm(pipe(load, process, save)).
- Una farm al cui interno mettiamo una funzione che svolge le tre operazioni ovvero Farm(comp(Load, process, save))
- Una farm in cui all'interno abbiamo una pipeline in cui però la seconda fase della pipeline è una farm che fa il processing delle immagini ovvero Farm(pipe(load, farm(process), save))

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 115

Se vogliamo valutare quale di queste è la soluzione migliore per il nostro problema dobbiamo considerare il tempo di esecuzione.

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 115

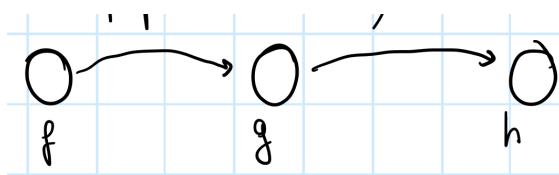
Lezione 16

Quando si sviluppa un algoritmo, quello che si fa è pensare ad una serie di pattern paralleli che poi vengono trasformati in un programma.

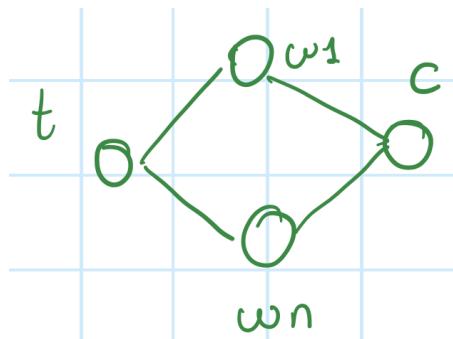
L'implementazione di un algoritmo parallelo può essere:

- Template Based
- Macro Data Flow based

Nel caso del **template based**, dopo aver riconosciuto i pattern che utilizziamo, dobbiamo usare il template scegliendo i parametri mancanti. Un esempio, abbiamo una pipeline(....), questo è già un template perchè ho una certa astrazione e non devo indicare subito alcuni parametri come ad esempio il tipo di comunicazione tra i vari stage della pipeline. Posso avere vari mechanism per implementare i canali di comunicazione tra le varie parti della pipeline (queue, TCP/IP....), anche se conosco l'implementazione del template necessito comunque dei parametri per far funzionare tutto. Quando scegli i parametri allora hai una implementazione. Posso poi calcolare il tempo complessivo per l'esecuzione $T_s = \max\{t_{s_i}\}$

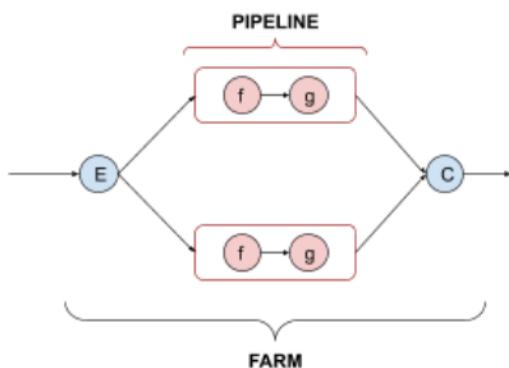


Poi posso anche capire se ho una composizione che migliora le performance della precedente.

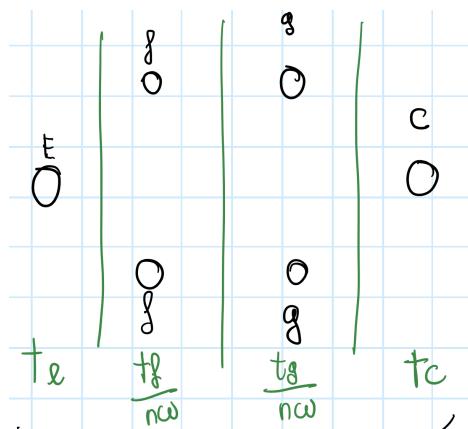


Ad esempio potrei utilizzare una farm(f, nw) e anche in questo caso ho un template. Anche in questo caso posso calcolare il tempo necessario per l'esecuzione che è $T_s = \max\{T_e, T_c, T_w/nw\}$.

È anche possibile implementare una farm di pipeline e in questo caso devo fare un mix dei due template per ottenere farm(pipeline(f, g), 2).



Internamente alla farm abbiamo una pipeline che è quindi è un altro template con i vari parametri scelti per l'inizializzazione. Conoscendo il service time della pipeline possiamo poi calcolare il service time della farm completa ovvero $T_s = \max\{T_e, T_c, \{\max(T_f, T_g)/nw\}\}$. Il T_s della farm va diviso per il numero di worker che abbiamo per ottenere la bandwidth finale.

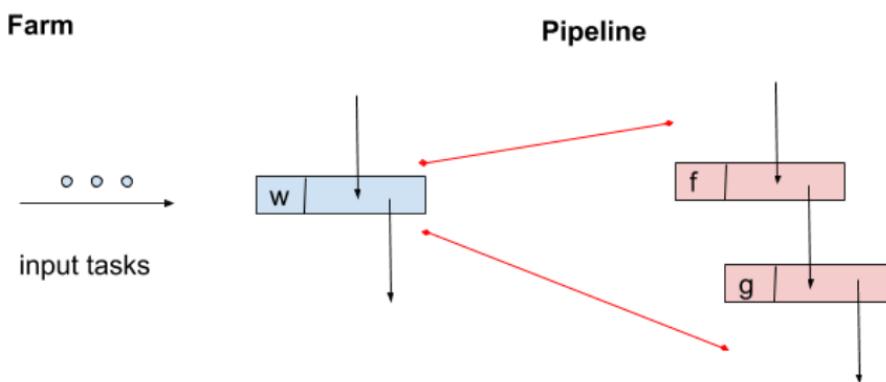


Il massimo tra i tempi in verde è il service Time della farm.

Il secondo caso è quello del **Macro Data Flow Based**, in questo caso abbiamo in mente un activity graph ovvero un grafo di dipendenze. Abbiamo un grafo di istruzioni, ogni istruzione ha uno o più input token (dati da processare), una funzione da eseguire e uno o più output. Partiamo ad esempio con una composizione del tipo $\text{farm}(\text{pipe}(f,g),2)$ dove f e g sono le funzioni che vengono eseguite nei due stage della pipeline e 2 è il numero di workers.

Per ognuno dei task deve essere istanziato un nodo che calcola la funzione, la farm non ha dipendenze perchè quello che viene dato in input alla farm sono dei dati che sono indipendenti tra loro, poi invece g è dipendente dal risultato che deve arrivare da f .

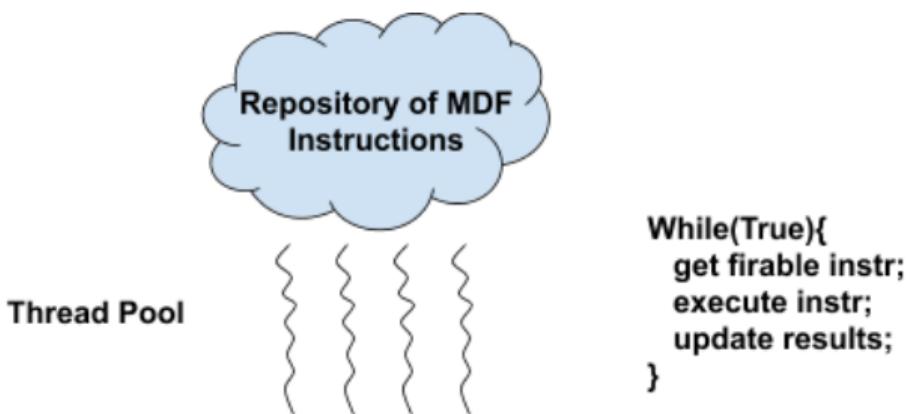
All'interno di questo grafo delle dipendenze stiamo utilizzando dei template.



Nel caso in cui dovessi avere 3 dati in input, avrei tre copie del grafo e in ognuna di queste copie avrei i nodi con la f e la g.

Gli archi del grafo rappresentano le dipendenze tra i nodi e mi permettono di spostare i dati tra un nodo e il successivo, tutte le dipendenze quando sono soddisfatte possono essere assegnate ad un nodo executor.

Come può essere implementato?



Abbiamo a disposizione un thread pool, una repo di istruzioni MDF e ognuno dei thread ha un codice che mi dice che deve essere presa l'istruzione pronta per essere eseguita (e che rispetta le dipendenze, poi deve essere eseguita e poi devono essere aggiornati i risultati finali).

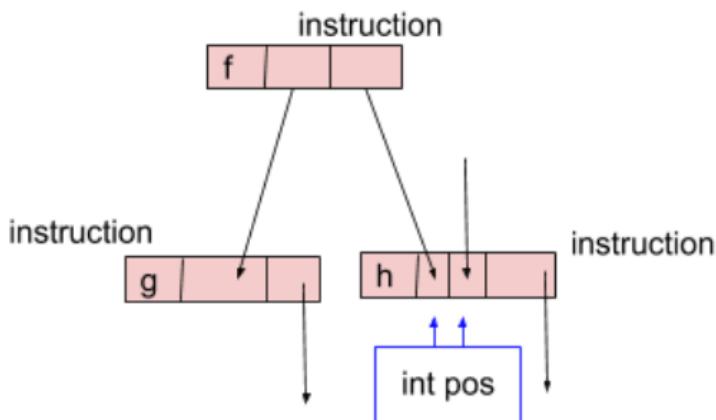
Consideriamo l'implementazione di una singola istruzione, vanno considerati alcuni parametri:

- Una funzione **<Tin, Tout> *f** che è la funzione che vogliamo calcolare
- Un **contatore count** che mi indica quanti input ho
- Un **vettore <Tin> x** che sono gli input tokens ovvero dei puntatori ai dati. Potrebbe avere valore NULL se non ho dati oppure potrebbe contenere i dati che vengono prodotti dalla funzione f e che devono essere dati in pasto alla funzione g.
- Un **contatore presence** che mi indica quanti dati sono presenti all'interno del vettore definito in precedenza, questo mi serve per

capire quando i dati presenti all'interno del vettore sono abbastanza per essere dati alla funzione successiva con cui ho la dipendenza.

- Un **Vector<ID Istruzione, int>** che mi dice dove devo mettere i risultati della computazione. Se la mia funzione produce a e b allora questo vettore mi indica dove devo memorizzare questi due valori. Ho anche bisogno di un intero che mi indichi la posizione dell'istruzione in cui devo memorizzare i dati.

Consideriamo una istruzione di questo tipo:



In questo caso devo sapere quale delle due istruzioni è la prima e quale è la seconda perchè le istruzioni potrebbero non essere commutative quindi devo capire in quale posizione dovrà andare a finire l'output token di f.

Quindi, ogni thread del thread pool fa il fetching delle istruzioni dalla repository MDF, quando prende una istruzione che ha le dipendenze rispettate e abbiamo che count == presence allora posso lavorare su quel task.

Una volta che ho l'istruzione non ho bisogno di alcun tipo di sincronizzazione con il resto della computazione perchè se riesco a prendere questa istruzione dalla repo vuol dire che ho già ottenuto il diritto a lavorare e nessuno userà i puntatori dei due vettori, altrimenti vuol dire che il grafo è stato prodotto male.

Quindi, quale sarebbe la differenza tra

- Qualcosa che è in grado di calcolare ogni dipendenza del grafo
- Il template che abbiamo visto prima con la farm

Nessuna.

Da un lato utilizziamo il template della farm per implementare il Macro Data Flow interpreter.

Dall'altro alcune delle cose che avresti dovuto considerare se avessi eseguito solamente una farm (con una condivisione di dati nella farm) ci sono comunque ma se il grafo è stato creato nel modo giusto non abbiamo problemi di contention e di critical access perchè tutte le dipendenze sono gestite dal grafo.

Sia se vogliamo produrre un Template o un Macro Data Flow Interpreter osserviamo che i meccanismi alla fine sono gli stessi e possiamo riconoscere le cose che accadono in parallelo, quelle che si scambiano i dati e quelle che devono essere sincronizzate.

Possiamo quindi utilizzare delle strutture che invece di farci gestire mutex, shared queue, join... ci permettano di utilizzare qualcosa più ad alto livello? Possiamo riutilizzare la stessa implementazione in tanti casi differenti?

Si, abbiamo dei set di meccanismi e ci sono anche vari gruppi che hanno provato a creare dei set di meccanismi ad alto livello rispetto al thread start-join model, non astratti come un pattern completo ma che possono essere usati per fare anche altre cose come ad esempio implementare un nuovo pattern o una nuova applicazione.

Parallel Building Block

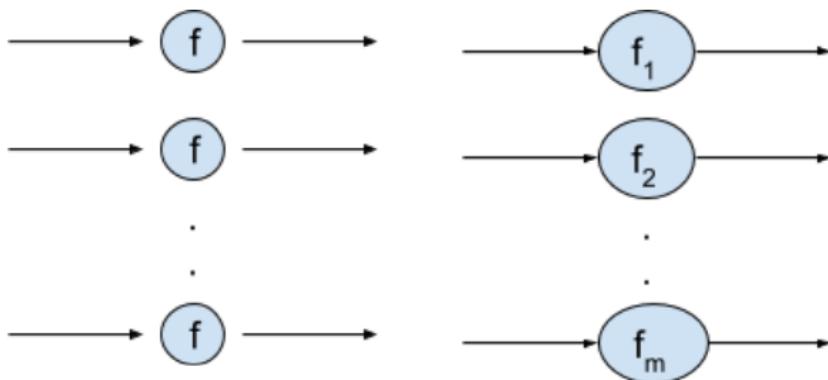
Distinguiamo tre differenti categorie di building blocks:

- **Computational building block:** include la possibilità di avere un numero di repliche di altri building block. Questo tipo di building block può essere utilizzato nella map per definire le varie partizioni o nella farm per calcolare gli item dello stesso input stream. Posso anche usarlo nel primo stage della reduce per calcolare la somma

delle partizioni.

Distinguiamo due tipologie di computational building block, entrambe prendono n valori in input e n li mandano in output.

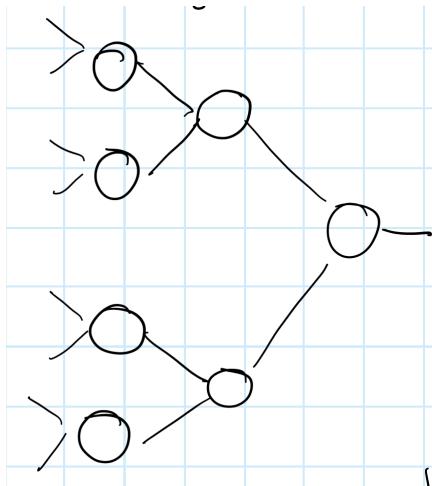
Un primo tipo è quello che ha tanti nodi che eseguono la stessa funzione, un secondo tipo invece ha tanti nodi che eseguono funzioni differenti tra loro.



Un'altra possibilità è quella di collegare tra loro i nodi attraverso le dipendenze, in questo modo otteniamo una pipeline che allo stesso tempo è anche un computational pattern, ognuno degli stage della pipeline qua può calcolare funzioni diverse tra loro oppure tutti la stessa funzione.

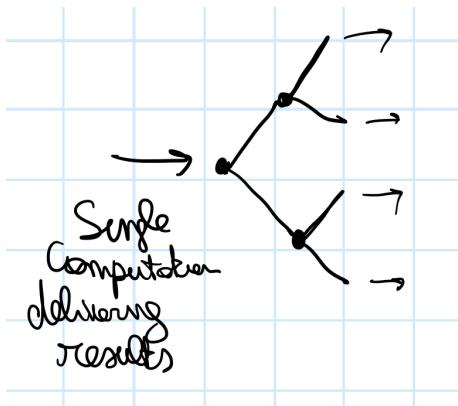
Abbiamo anche la possibilità di effettuare la reduce, possiamo infatti creare una sorta di albero in cui i vari nodi eseguono tutti la stessa identica funzione e da un nodo all'altro passo i risultati

intermedi fino ad arrivare al risultato finale della reduce.

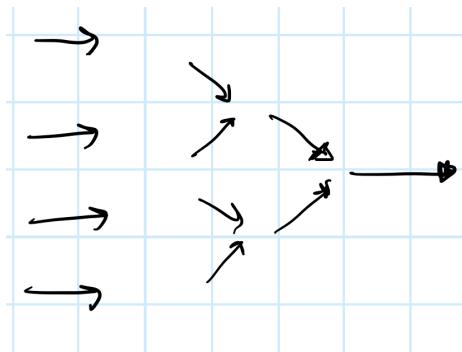


- **Communication Building Block:** assumiamo di avere un albero bilanciato, questo building block ci permette di connettere degli elementi che eseguono dei calcoli.

Possiamo avere ad esempio una singola computazione che mi manda in output i vari risultati, in questo caso si parla di comunicazione 1 a N:

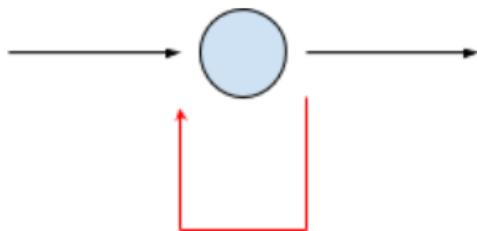


Oppure possiamo avere una situazione opposta che prende N input e mi calcola un solo output finale.



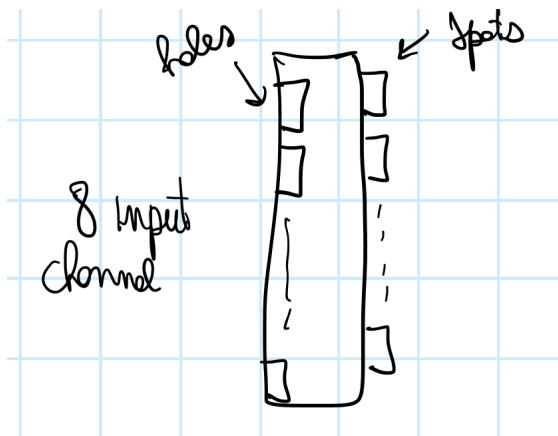
Nel caso del pattern communication la politica che applico per distribuire i dati determina vari tipi di computazione, posso definire delle policy per definire lo scheduler dell'output di ciascun nodo.

- **Modifiers:** è la possibilità di calcolare qualcosa dato un dato che arriva da un input channel e che deve andare a finire in un output channel.



Un building block di questo tipo è il feedback modifier che prende dei dati in input e fa il merge con i dati che arrivano da un feedback channel (in rosso nella foto). Il building block produce qualcosa che poi posso o mandare avanti o mandare indietro usandolo come feedback.

Dobbiamo immaginare questi building block come i mattoncini lego, quindi con dei fori e degli "spots".



È essenziale che questi mattoncini vengano collegati tra loro in modo da far combaciare il numero di input channel e di output channel. Quindi ad esempio un blocco 1-N lo posso collegare con uno che ha N-N.

Se consideriamo il **computational building block** avremo:

- N input e N output

Nel **communication building block**:

- 1 Input e N output
- N input e 1 output

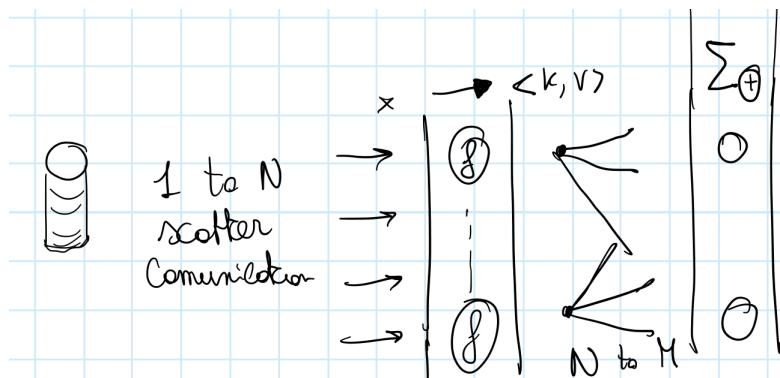
Se torniamo al caso del Template Based Implementation:

- La **pipeline** è un building block unico in cui ogni nodo manda fuori un output e prende un solo input
- La **farm** è un'unione di vari building block, abbiamo un communication building block 1 to N che mi serve per la prima parte di divisione dei dati, questo deve implementare una unicast policy nel senso che il dato deve finire in un unico nodo. Poi abbiamo uno stream di N workers che calcolano tutti la stessa funzione e in più abbiamo una comunicazione N to 1 finale implementata con un communication building block (deve implementare la gather policy)
- La Map la possiamo implementare utilizzando un 1 to N communication block, un gruppo di N workers e poi un N to 1 communication block. La differenza con la farm sta nel primo communication building block perchè qua i dati sono sparsi in output perchè abbiamo una collezione e la splittiamo. Il building block finale deve implementare una policy all gather in cui vengono presi i risultati dei vari worker e viene mandata in output una collection.

- Se vogliamo implementare la Map Reduce di google: se abbiamo i dati in un solo posto inizialmente devo suddividerli quindi avremo una comunicazione 1 a N, poi alla fine se vogliamo che i risultati

finiscano in un solo posto allora mi serve un gather che mi porti gli N output in una sola posizione.

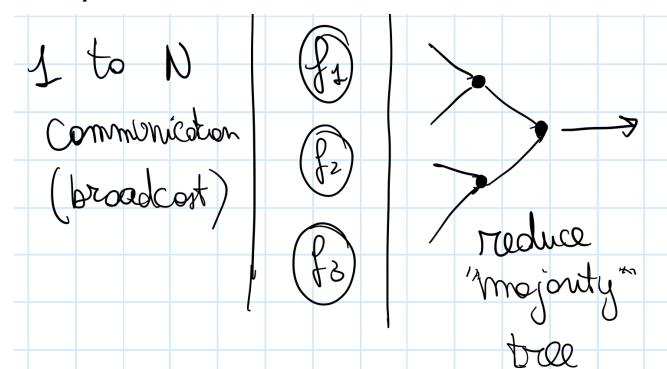
Internamente abbiamo una policy che è una hashing policy nel senso che da N nodi mando gli output in M nodi.



Consideriamo la funzione f che stiamo utilizzando all'interno dei vari pattern, vogliamo una f che sia "Fault Tolerant".

Possiamo utilizzare più funzioni che calcolano la stessa cosa implementandole però diversamente.

Nel caso del disegno abbiamo una comunicazione 1 to N Broadcast, poi le n funzioni diverse tra loro poi un majority tree che prende tutti i risultati, associa uno score ad ogni risultato e poi quelli con lo score più alto passano avanti.



Questi costrutti che abbiamo definito ci permettono di definire delle applicazioni parallele che hanno un certo grado di astrazione, se abbiamo un'applicazione parallela con la shared memory e vogliamo

passare ad una distributed network allora possiamo mantenere la struttura che abbiamo definito utilizzando i building block modificando solamente la libreria che utilizziamo.

Questo comporta anche che riusciamo a ottenere le 3P ovvero:

- **Performance** perchè le librerie dei building block sono implementate in modo efficiente
- **Portability** perchè abbiamo una libreria di building block per le shared memory e una per le architetture distribuite
- **Programmability** perchè lavoriamo ad un livello più alto e non devo definire mutex...

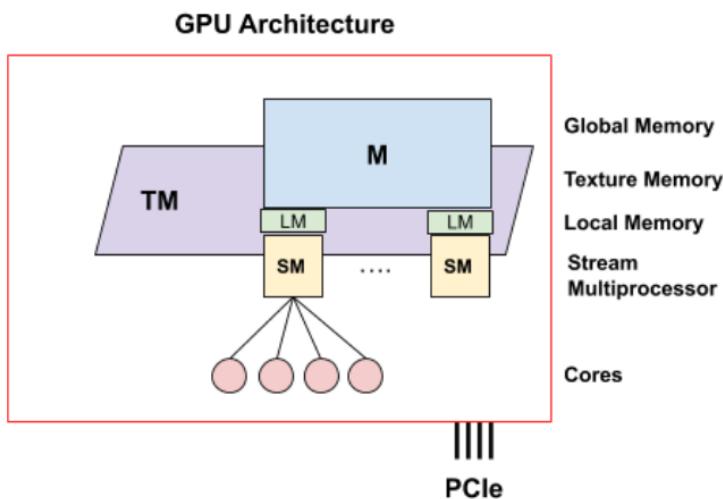
Lezione 17

Introduzione alle GPU

Le GPU che troviamo in moltissimi dispositivi hanno molti core, questo numero di core è differente rispetto al numero di core che troviamo nelle CPU. I core delle GPU sono in grado di eseguire solamente alcuni tipi di operazioni parallele, in particolare si tratta dei pattern data parallel quindi map, reduce e stencil. Non sono invece adatte per eseguire operazioni di pattern come lo stream parallelism.

Con le GPU non ci interessa troppo la velocità nell'esecuzione del singolo task ma puntiamo a parallelizzare il più possibile.

Nelle GPU abbiamo almeno una memoria che è completamente staccata dalla CPU, l'architettura è la seguente:



- Abbiamo vari SM che sono gli streaming multi processor, ogni SM ha vari core e varie cache (cache più piccole della cache della CPU perchè abbiamo più core e il tempo per prendere i dati quindi è meno importante)
- Ognuno di questi core sono in grado di eseguire solamente alcune istruzioni e le eseguono solamente per lo streaming multi processor. Le istruzioni sono le stesse nei vari core e vengono eseguite tutte insieme su dati differenti tra loro.

- Ogni SM ha una local memory che fa da cache per la global memory
- La GPU ha una memoria globale che è più lenta da accedere rispetto alle local memory dei vari SM
- C'è anche una Texture memory che è più lenta della Local Memory ma è più veloce della memoria globale.

I core accedono alla parte di memoria locale dell'SM, possono accedere molto velocemente a questa memoria e in generale gruppi di 16/32 core possono accedere contemporaneamente a più posizioni differenti della memoria locale.

Tutti questi core quindi fanno tutti la stessa cosa su dati differenti, fanno operazioni semplici come ad esempio load, store, combine e lo fanno molto velocemente.

L'unica cosa è che non possono fare cose diverse tra loro contemporaneamente, non posso avere un parallelismo di questo tipo tra i core e quindi se devo eseguire task diversi tra loro mi diventa una esecuzione sequenziale.

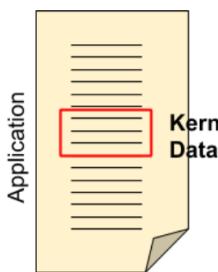
Questo tipo di architettura non è in grado di mantenersi in vita da sola e non è in grado di eseguire il boot di un sistema operativo.

Anche se GPU e CPU (ovvero una APU) si trovano sullo stesso chip, c'è comunque la necessità di connettere i due componenti con una interfaccia PCI express.

Questo bus che viene utilizzato per la comunicazione tra CPU e GPU è molto più lento di una qualsiasi memory interface.

Come posso sfruttare la GPU in un software parallelo?

Generalmente quello che si fa è preparare un programma che svolge una serie di operazioni, quando identifico una parte del programma che è eseguibile con un pattern data parallel devo capire se questo può essere eseguito su una GPU o no.



Cosa si deve fare per fare in modo di avere una esecuzione sulla GPU?

In generale abbiamo dei passi standard poi a seconda del tool che utilizziamo (ad esempio Cuda) abbiamo che lo spostamento dei dati dalla memoria è a carico del programmatore.

I passi necessari:

- Abbiamo bisogno di un compilatore pensato appositamente per le GPU
- Deve essere svolta una sincronizzazione tra quello che accade sulla CPU e quello che accade sulla GPU, per fare questo possiamo utilizzare il bus PCIe.
Questa operazione di sincronizzazione è più lenta di accedere i dati in memoria.
Dato che utilizziamo il bus PCIe dobbiamo valutare quanto sia conveniente utilizzare la GPU, nonostante l'alto bandwidth se passiamo pochi dati alla volta alla GPU paghiamo un overhead troppo alto rispetto al guadagno che abbiamo.
- Dobbiamo muovere i dati dalla CPU alla GPU e viceversa, qui abbiamo un lavoro che è a carico dello sviluppatore.

GPU	CPU
<ul style="list-style-type: none">- Data Parallel (Nvidia dice che le GPU sostituiranno le CPU ma non è vero perchè in realtà sono sempre CPU ARM attaccate alle GPU)	<ul style="list-style-type: none">- Utilizzo generale
<ul style="list-style-type: none">- Le performance/Watt sono	<ul style="list-style-type: none">- Bandwidth migliore rispetto

migliori nelle GPU rispetto alle CPU perchè nelle GPU abbiamo core più piccoli che sono replicati tante volte mentre nelle CPU i core sono molto più complessi. Questo può essere utile se vogliamo eseguire operazioni di tipo data parallel.	alle GPU
--	----------

Quando le GPU vengono sfruttate maggiormente?

- Per Video o Graphic processing
- Per High performance computing, tutti i codici che vengono usati per applicazioni scientifiche sono paralleli quindi si usano le GPU
- AI: per il training delle reti neurali si moltiplicano le matrici e quindi abbiamo la necessità di un calcolo parallelo.

Esistono anche alcuni dispositivi che sono creati utilizzando solamente le GPU, in particolare si tratta di Nvidia Jetson che è tipo un raspberry che utilizza solamente alcuni core ARM per avviare l'OS e poi solamente GPU.

Lezione 18

OpenCL è un framework che è stato pensato per lavorare su GPU, CPU e FPGA. Per far funzionare OpenCL abbiamo bisogno di alcune librerie di driver. OpenCL non vuole essere dipendente dalle GPU, deve essere completamente indipendente e quindi deve essere portabile. Il programma che scriviamo usando OpenCL può essere eseguito su vari dispositivi con performance differenti tra loro a causa dell'implementazione delle schede su cui viene fatto girare.

Per usare OpenCL abbiamo bisogno delle seguenti librerie:

- OpenCl-Headers
- OpenCL-CIHP-headers
- OCI-icd-openCL-dev
- Clinfo

E poi servono dei driver per eseguire il codice scritto in OpenCL:

- <https://github.com/intel/compute-runtime/releases>

Come già detto le GPU sono degli acceleratori, nel mercato delle GPU ci sono molte aziende che hanno interessi, Intel, AMD, ATI, Apple, Nvidia che ha sviluppato CUDA che è un sistema proprietario che permette di eseguire del codice solamente sulle schede video di Nvidia.

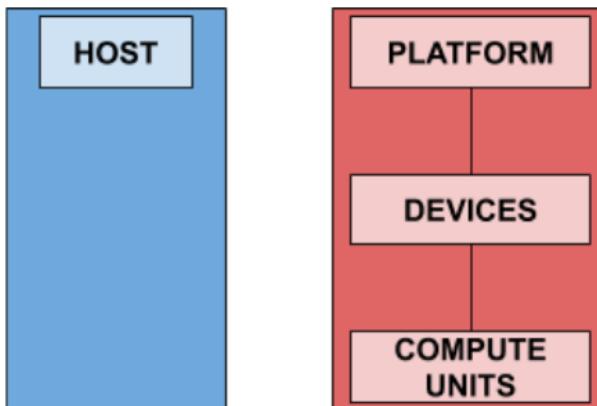
Nello sviluppo di questi sistemi abbiamo dei concetti fondamentali:

- Abbiamo la memoria dell'host.
- C'è la memoria della scheda video che è divisa in memoria globale, memoria locale e memoria privata. La global memory viene acceduta da tutte le attività, la local memory la accedo solamente da un gruppo di attività mentre ogni attività del gruppo ha una sua memoria principale che somiglia ai registri del

processore.



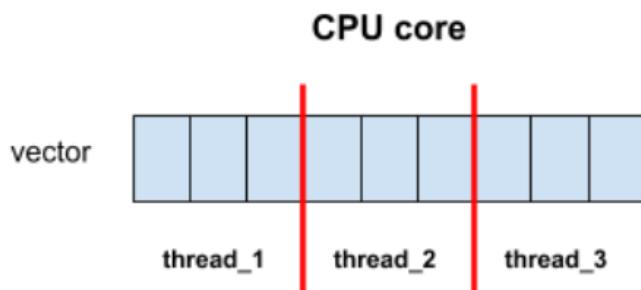
- Poi abbiamo che su ogni scheda video sono presenti le varie compute units



Inizialmente il trasferimento dalla memoria dell'host alla memoria della scheda video era a carico del programmatore, poi OpenCL nella versione 2.0 ha introdotto il concetto di shared virtual memory (SVM), questa è una astrazione della memoria. Questa astrazione della memoria permette di accedere dalla scheda video alle locazioni di memoria dell'host come se queste fossero memoria del dispositivo. La maggior parte delle chiamate che il programmatore era abituato a svolgere in passato non sono più necessarie a questo punto.

C'è un secondo concetto fondamentale che vale sia in OpenCL che in CUDA, questi dispositivi sono progettati per eseguire delle operazioni in parallelo.

In generale quando scriviamo un codice in parallelo e lo eseguiamo nella CPU abbiamo un vettore e vogliamo applicare una funzione agli elementi del vettore (una funzione che è indipendente dal punto del vettore in cui viene applicata).



In OpenCL e in CUDA l'idea di base è diversa, dobbiamo pensare in termini di **space of points**, non abbiamo una struttura che contiene solamente puntatori ma dobbiamo intenderlo come un insieme di punti (coordinate). Per ognuno di questi punti dobbiamo eseguire un tipo differente di computazione, quando poi uno di questi punti ha una dipendenza con un altro punto allora abbiamo un degrado delle prestazioni.

Soltamente si parla di **Grid of Groups** e ognuno di questi gruppi ha un task al suo interno.



Questo space of points può avere differenti dimensioni, 1 dimensione, 2 dimensioni, 3 dimensioni, questo ci permette di chiedere “chi sei” in base a questa dimensione.

Con la chiamata get_global_id(0) potremo ottenere l'indice della prima dimensione, quindi, se avessimo passato 1 sarebbe stato l'indice della seconda dimensione.

Consideriamo la somma di due vettori A e B che viene memorizzata nel vettore C, avremmo questo codice:

```
function_name(A,B,C){  
    auto i = get_global_id(0)  
    C[i] = A[i] + B[i]  
    return  
}
```

Consideriamo la gerarchia con Grid, Groups e Task che abbiamo menzionato in precedenza, c'è anche un'altra cosa che va tenuta in considerazione. Grid, Groups e Task hanno delle funzionalità differenti anche dal punto di vista della sincronizzazione.

I task non possono fare molto ma partecipano alla sincronizzazione che è legata ai groups, ad esempio possiamo utilizzare barriers o memory fence per la sincronizzazione. I task nei gruppi devono attendere che ognuno di questi task arrivi alla barrier o ogni memory operation dei task è completata ma **non posso** fare nessun tipo di sincronizzazione tra i task (point to point), questo funziona solamente al livello del groups.

Ci sono anche dei parametri che dobbiamo considerare per quanto riguarda la GPU, con il comando clinfo riusciamo a vedere molti di questi parametri per la nostra GPU.

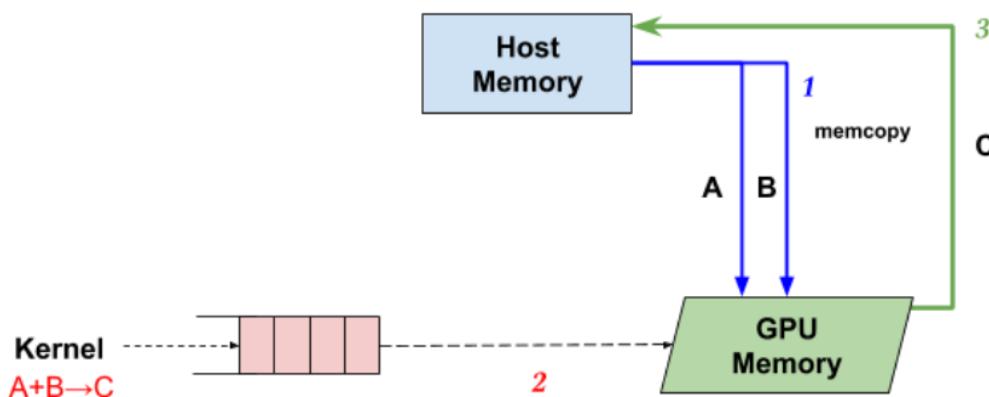
In particolare cambiando il group size e scegliendo la dimensione corretta allora si riesce anche a determinare l'efficienza perchè se riusciamo a trovare una divisione del lavoro in grado di adattarsi alla divisione delle risorse che abbiamo sulla GPU lavoriamo meglio, altrimenti con dei parametri errati avremmo delle risorse più usate di altre e alcune che invece non utilizziamo per niente.

Come facciamo a direzionare il programma alla GPU?

Abbiamo vari concetti in OpenCL:

- **Platform:** sono i dispositivi che vogliamo accedere, ad esempio le GPU
- **Context:** specifica quale parte del dispositivo vogliamo accedere, viene utilizzata una command queue che serve per direzionare le operazioni verso il dispositivo.

Attraverso la command queue inviamo alla GPU i kernel ovvero le unità minime di esecuzione.

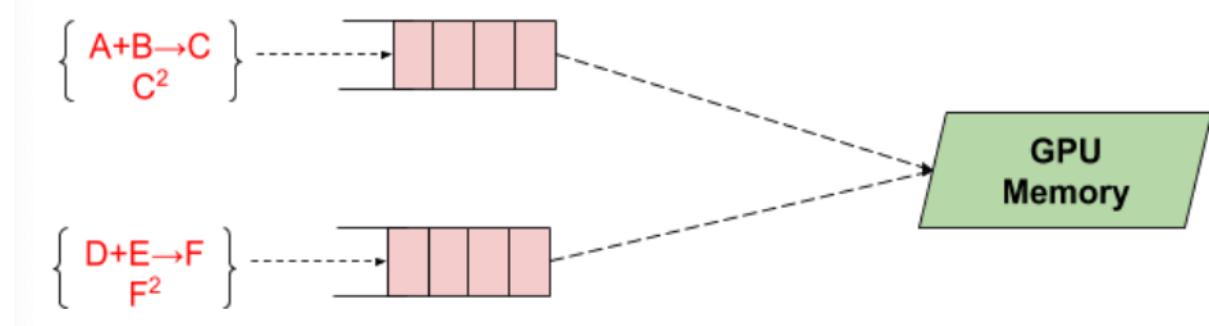


La command queue può essere:

- Ordinata: nel caso di una command queue ordinata abbiamo due kernel che vengono inviati all'interno della coda, ad esempio uno sarà relativo alla funzione F e uno alla funzione G. Il fatto che sia presente un ordinamento mi indica che la funzione F deve essere eseguita prima della G. Quindi si crea una dipendenza tra le due funzioni, ad esempio la funzione F potrebbe essere la somma di due vettori mentre la G potrebbe essere il quadrato degli elementi del vettore risultato.
- Non ordinata: in generale si preferisce (ed è più utile) avere una coda che non è ordinata, ci troviamo ad avere una GPU come server di kernel che quindi dovrà mantenere queste varie funzioni ed eseguirle senza però avere delle dipendenze tra le varie esecuzioni. Sarà la GPU a decidere come eseguire le varie funzioni.

Consideriamo due esempi, vogliamo eseguire due operazioni:

- $A+B = C$
- $D+E = F$



Avremo due kernel che eseguono la somma e poi il quadrato degli elementi del vettore finale, in questo caso possiamo avere due command queue ordinate, perchè comunque abbiamo una dipendenza somma-> quadrato, allo stesso tempo non abbiamo un ordine che mi indica se devo prima eseguire la prima somma o prima la seconda, quindi questo può essere deciso dalla GPU e alcune delle operazioni che appartengono alle due somme possono anche avere un overlap.

Quindi in definitiva:

- Con CUDA ci basta scrivere una funzione che viene dichiarata come kernel e il compilatore la compilerà e la manderà alla GPU che poi la esegue
- OpenCL invece deve avere il codice passato come una stringa, questo codice viene passato attraverso una call al compilatore. Questo permette di avere un compilatore semplice all'interno della libreria ma non è il massimo per il programmatore perchè se c'è un errore nel codice del kernel dovremo fare altre call per ottenere il log dell'errore.

Messaggi finale per concludere:

- Consideriamo il codice che utilizziamo per sommare i due vettori in un vettore risultato. Quanto è specifico questo codice per fare le somme rispetto a fare le moltiplicazioni?

Troviamo una parte specifica per fare le somme solamente all'interno del kernel. Quindi mi basta scrivere un altro kernel per fare una operazione differente dalla precedente.

- Dato lo zip pattern in cui abbiamo $\text{zip}(v1, v2, f) \rightarrow v3$ dove $v3[i] = f(v1[i], v2[i])$.

Consideriamo lo $\text{zip}(+)$ e una seconda applicazione del quadrato a tutti i numeri del vettore $a(x^2)$.

Quindi vogliamo prendere $v1$ e $v2$, sommarli ottenendo $v3$ e poi per ogni elemento di $v3$ calcolare il quadrato.

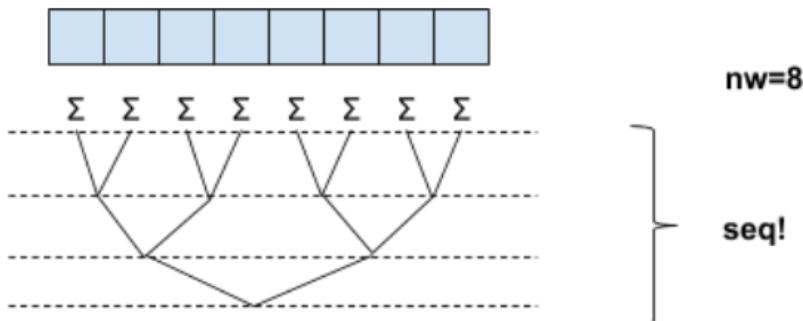
Posso utilizzare una procedura zip e una procedura “apply to all” che mi restituisce il risultato o posso fare qualcosa di più efficiente?

L' improvement maggiore qua ce l'abbiamo se riusciamo a migliorare la composizione del pattern perchè idealmente vogliamo muovere $v1$ e $v2$ verso la GPU, calcolare il kernel dello zip e dello square e ottenere il risultato.

Lezione 19

Consideriamo l'operazione di reduce che prende in input una serie di numeri e deve restituire la somma di questi numeri.

$$x_1 \dots x_m = x_1 \oplus x_2 \oplus \dots \oplus x_m$$



Affinchè sia possibile eseguire queste somme in parallelo, è necessario che le somme siano associative e commutative.

L'albero che si crea per sommare questi numeri ha una altezza che è logaritmica rispetto al numero di elementi che devono essere sommati. C'è una cosa importante da considerare, a seconda della dimensione dei dati, il tempo necessario per sviluppare questo albero binario che mi permette di calcolare la somma totale è molto maggiore rispetto al guadagno in termini di prestazioni che ottengo.

Quindi in molte situazioni conviene svolgere la parte che sta nell'albero in modo sequenziale, ammesso però che i dati da sommare non siano troppi. Quindi, se ho circa 100 thread e la somma ha la dimensione di una operazione di tipo floating point non c'è modo di fare tutta questa operazione meglio in parallelo che in modo sequenziale.

OpenMP: consideriamo OpenMP, in questo caso abbiamo una attenzione speciale per l'operazione di reduce, tanto che questa è implementata in parallelo.

InnerProduct

$$x_1 * y_1 + x_2 * y_2 + \dots + x_m * y_m$$

Sequential

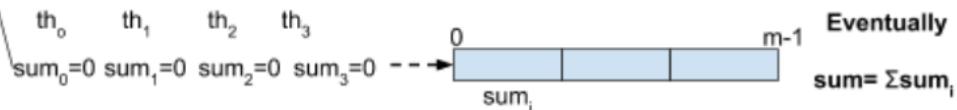
```
sum = 0;
for (int i = 0; i < m; i++)
    sum += x[i]*y[i]
```

In this way, the for is splitted among nw threads. But the problem is that **sum** depends on the previous operation...

Parallel

```
sum = 0;
#pragma omp parallel for num_threads(nw) reduction( +:sum)
for (int i = 0; i < m; i++)
    sum += x[i]*y[i]
```

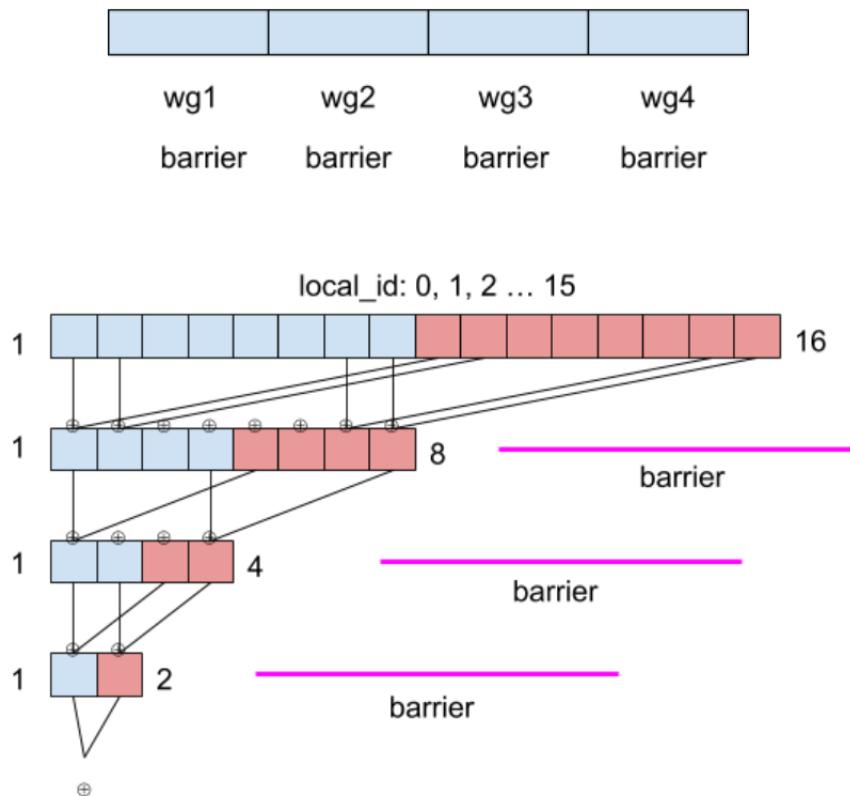
Addind **reduction**, the problem is resolved since, now, openMP knows that I want to reduce on the **sum** variable using the **+** operator



L'implementazione sequenziale somma alla variabile sum il valore di una moltiplicazione. Se voglio parallelizzare il loop con OpenMP posso utilizzare il parallel for andando però ad indicare nel pragma anche "reduction" con il nome della variabile in cui vogliamo accumulare la somma. Quando il sistema trova la dicitura reduce con la variabile in cui salvare i dati fa in modo che venga creato un numero di variabili adeguato al numero di thread che stiamo utilizzando, questo perchè altrimenti usando una sola variabile avrei una dipendenza tra le varie iterazioni del for.

Al termine di ognuna delle somme dei thread devo garantire comunque la sincronizzazione, quindi devo inserire delle barriere alla fine di ognuna delle somme in modo che quando passo al livello successivo dell'albero tutti i thread abbiano terminato la loro somma parziale.

Se vogliamo fare questa computazione sfruttando la GPU e quindi utilizzando OpenCL partiamo da un vettore che viene suddiviso in varie posizioni, ognuna di queste posizioni viene assegnata ad un workgroup e per ogni workgroup abbiamo una barrier che mi garantisce la sincronizzazione.



I valori che si creano ogni volta che facciamo una di queste somme parziali devono essere memorizzati all'interno di un vettore parziale. Alla fine della storia dovrò avere tante somme parziali quanti sono i workgroup del global space, per questo motivo devo indicare come parametro questa global dimension che varia a seconda della GPU che stiamo utilizzando e che possiamo ottenere con il comando clinfo.

Esempi:

- Se abbiamo $1K = 2^{10}$ di elementi e abbiamo $32 = 2^5$ elementi per ogni work group allora avremo 2^5 workgroup ognuno con 2^5 elementi

Se eseguo tutto questo processo di reduce all'interno del work group quello che ottengo è un vettore di 32 posizioni in cui vengono memorizzate le somme parziali che vengono calcolate da ognuno dei gruppi.

Consideriamo la somma finale che otteniamo alla fine di questo albero di somme, quello che ho è un valore locale che rappresenta la somma e che deve essere memorizzato nel vettore delle 32 posizioni nella posizione in cui voglio posizionare la somma finale della riduzione.

Se consideriamo che tutto il codice viene eseguito da tutti gli indici nel mio spazio allora dovrò fare attenzione per evitare che troppi core mappino qualcosa in questa posizione.

Questo ultimo valore locale deve essere memorizzato in un vettore di somme parziali nella posizione del work group dall'unico thread che possiede questo livello che è il thread (o core o point nello space) che ha l'indice 0 nel workgroup.

Consideriamo il codice per eseguire la reduce in una GPU:

```
local_id = get_local_id(0); // id all'interno del group
group_size = get_local_size(0); //solamente quelli che hanno Local_id =
0 possono scrivere
// nell'array globale
...
...
localSum[local_id] = inputVector[get_global_id(0)];
for(stride = group_size/2; stride > 0; stride/=2){ /
// Calcola la somma dell'elemento assegnato con l'elemento più Lontano
da me
barrier(CLK_LOCAL_MEM_FENCE);
    if(local_id < stride)
        localSums[local_id] += localSums[local_id + stride];
}
if(local_id == 0)
    partialSums[get_group_id(0)] = localSums[0];
```

Questo è un metodo che possiamo utilizzare per calcolare la reduce in un numero logaritmico di passi.

Non è un metodo troppo elegante però perchè alla fine del procedimento abbiamo un vettore di somme parziali, quindi dobbiamo completare l'operazione e abbiamo due possibilità:

- Ripetiamo il processo fino a che la dimensione della somma locale è minore della dimensione del workgroup
- Trasferiamo tutti i dati nella CPU e usiamo un for classico attraverso il vettore di somme parziali

Tutto dipende dalla dimensione del vettore, con 32 elementi è meglio usare la CPU, inoltre dobbiamo anche tenere in considerazione dove dovremo poi utilizzare i dati del vettore, se serviranno nella GPU allora conviene lavorare direttamente nella GPU, altrimenti conviene spostare i risultati parziali nella CPU.

OpenCL 2.0

Con la versione 2.0 di OpenCL sono state introdotte alcune collective operations, quindi delle operazioni che possiamo utilizzare per fare la reduce. Ad esempio abbiamo `work_group_reduce_xxxx()` dove xxxx può essere una somma o un massimo o un minimo, se poi dovesse servirci un operatore differente allora dobbiamo implementarlo noi.

Questa operazione di reduce è simile alle barriers, in un workgroup se vogliamo sommare gli elementi di un vettore dobbiamo utilizzare questo tipo di operazione indicando il valore che vogliamo sommare alla somma totale. Tutti i thread dovranno fare questa operazione.

Possiamo utilizzare una singola linea di codice per fare questo:

```
float sum = work_group_reduce_sum(x[get_local_id(0)])
```

Quello che stiamo dicendo è che vogliamo calcolare una `sumFinale` dove `sum` è il risultato dell'applicazione della collective operation.

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 144

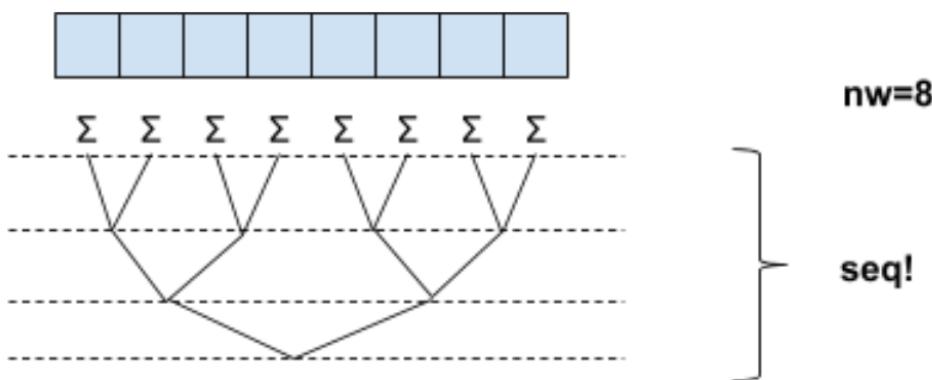
Ogni core nel work group raggiunge questa linea di codice e poi quando l'avrà eseguita avremo la somma di ciascun livello, ovvero di ciascun group. Abbiamo comunque il problema che vanno sommate le somme parziali.

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 144

Lezione 20

Reduce

$$x_1 \dots x_m = x_1 \oplus x_2 \oplus \dots \oplus x_m$$



Nella reduce eseguiamo un operatore (rappresentato con un cerchio con un + perchè vogliamo che abbia le stesse proprietà che ha l'operatore +).

Questo operatore deve essere commutativo e associativo perchè la reduce la vogliamo svolgere in parallelo e avere queste due proprietà ci consente di poter eseguire le somme parziali e poi la somma dei risultati parziali in un'unica variabile finale.

Quindi abbiamo un vettore di valori e ad esempio possiamo dividere il vettore in due parti e poi calcolare la somma degli interi della prima parte in modo sequenziale e della seconda parte e poi alla fine sommare i due risultati parziali.

Se il numero dei thread aumenta possiamo splittare ancora di più il vettore ottenendo una struttura ad albero in cui facciamo le varie somme in parallelo.

A seconda della dimensione dei dati, il tempo che spendo per creare tutta la struttura ad albero è maggiore del gain che ottengo andando a parallelizzare, quindi spesso è conveniente fare la parte che viene fatta con l'albero in modo sequenziale.

L'operatore di reduce è importante e spesso viene utilizzato. OpenMP ha una sua implementazione parallela dell'operazione di reduce.

InnerProduct

$$x_1 * y_1 + x_2 * y_2 + \dots + x_m * y_m$$

Sequential

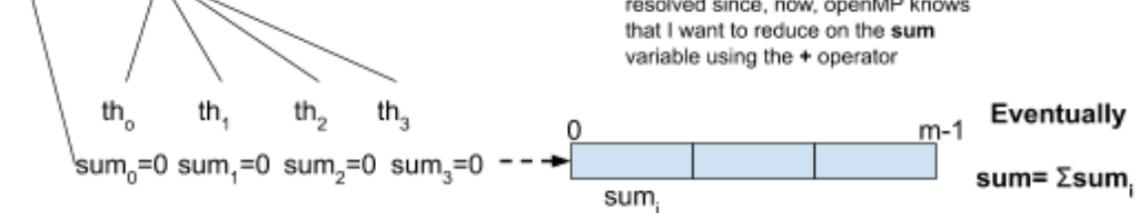
```
sum = 0;
for (int i = 0; i < m; i++)
    sum += x[i]*y[i]
```

In this way, the for is splitted among nw threads. But the problem is that **sum** depends on the previous operation...

Parallel

```
sum = 0;
#pragma omp parallel for num_threads(nw) reduction( +:sum)
for (int i = 0; i < m; i++)
    sum += x[i]*y[i]
```

Addind **reduction**, the problem is resolved since, now, openMP knows that I want to reduce on the **sum** variable using the **+** operator



Prendiamo il caso della moltiplicazione di due vettori, il risultato poi deve essere memorizzato in una variabile. In modo sequenziale possiamo fare un for che scorre i due vettori e memorizza in una variabile il risultato.

Se vogliamo parallelizzare questo for possiamo utilizzare open MP e la pragma che viene offerta per questo tipo di operazione:

```
#pragma omp parallel for num_threads(nw) reduction(+: sum)
```

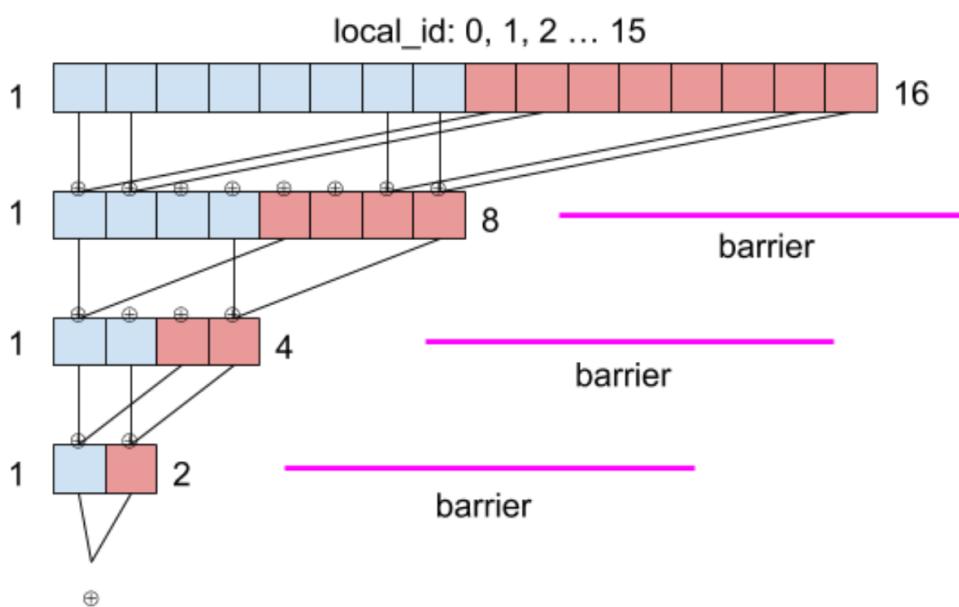
L'ultimo parametro lo mettiamo perchè vogliamo fare la reduce, in questo caso infatti il problema è che c'è una dipendenza tra le varie operazioni che devono essere svolte. Quando troviamo il parametro reduction allora il sistema deve organizzare la computazione in modo

che venga creato un numero di variabili adeguato al numero di thread. Alla fine i valori che troviamo in queste variabili che corrispondono ai vari thread vengono sommati in una variabile sum finale. In ogni caso devo garantire una sincronizzazione e per farlo devo fare in modo che ognuno dei thread si fermi in attesa degli altri risultati parziali ogni volta che finisce il calcolo della somma parziale.



Quindi in pratica mettiamo una barrier alla fine del calcolo di ogni somma parziale e questo mi permette di andare al livello successivo dell'albero con le somme.

Consideriamo questo problema della reduce con le GPU, dovremmo pensare ad una soluzione per questo problema fatta con OpenCL. Anche con OpenCL dovrei suddividere il vettore in varie parti e poi ad ogni iterazione dovrei fare le somme parziali



In questo caso dipende tutto dalla dimensione del workgroup perchè la sincronizzazione la devo fare all'interno dei workgroup e quindi dovrò fare un numero di somme parziali che corrisponde al numero di workgroup che abbiamo (cercando di scegliere una dimensione del workgroup che è quella indicata dal produttore della GPU).

Un esempio, abbiamo un vettore con 2^{10} elementi, in ogni workgroup vogliamo 2^5 elementi, quindi abbiamo 2^5 workgroup di 2^5 elementi ciascuno.

Quindi in pratica otterrò un vettore di 32 posizioni e ognuna conterrà la local sum corrispondente.

Quando arrivo con la computazione alla fine dell'albero (devo generare la somma finale), quello che ho è un valore “locale” che rappresenta la somma e deve essere memorizzato comunque all'interno del vettore di 32 posizioni. Devo fare attenzione perchè questo valore finale (che viene messo nel vettore in una delle posizioni) non può essere memorizzato da uno qualsiasi dei vari thread che abbiamo ma solamente dal primo thread tra quelli che hanno calcolato le somme parziali.

Consideriamo il codice per eseguire la reduce all'interno della GPU in un numero logaritmico di step:

```
Local_id = get_local_id();
group_size = get_local_size();
...
localSums[local_id] = inputVector[get_global_id()];
for(stride = groupsize/2, stride > 0, stride /=2){
    barrier(CLK_LOCAL_MEM_FENCE);
    if(local_id < stride)
        localsums[local_id] += localSums[local_id + stride];
}

if(local_id == 0)
    partialSums[get_group_id(0)] = localSums[0];
```

Questo metodo non è troppo ben funzionante perchè alla fine abbiamo un vettore di somme parziali e non subito la somma complessiva. Quindi dobbiamo comunque completare l'operazione di somma e ci troviamo con due possibilità:

- Ripetiamo il processo fino a quando la dimensione delle somme parziali è minore rispetto alla dimensione del workgroup e quindi possiamo calcolarla di nuovo ricorsivamente
- Trasferiamo il vettore delle somme parziali nella CPU e poi in modo sequenziale calcoliamo la somma complessiva degli elementi contenuti all'interno.

Tutto dipende però dalla dimensione del vettore, con 32 elementi come nel caso dell'esempio è meglio utilizzare la CPU. Dobbiamo poi considerare anche dove i dati calcolati verranno utilizzati, se servono nella GPU è inutile che li spostiamo nella CPU, se invece servono nella CPU allora va bene anche spostarli e finire lì il calcolo.

Risolviamo lo stesso problema con OpenCL 2.0

Con la versione 2.0 di OpenCL è stata introdotta una primitiva che svolge il ruolo del collector in ognuno dei workgroup:

work_group_reduce_xxx() dove al posto di xxx ci va sum/max/min o un'altra operazione (implementabile).

Questa primitiva che è stata introdotta mi permette di calcolare la reduce in ognuno dei workgroup, tutti i partecipanti alla collective operation devono partecipare per fare in modo che l'operazione vada a buon fine e funzioni bene, in un certo senso questa primitiva funziona un po' come una barrier.

In un workgroup se vogliamo sommare il vettore dobbiamo chiamare questa operazione con l'intero che vogliamo sommare alla somma totale, tutti i thread poi devono fare questa operazione.

Al posto di scrivere il codice che abbiamo scritto prima possiamo scrivere una sola riga che è:

```
float sum = work_group_reduce_sum(x[get_local:_id(0)]);
```

Qua stiamo dicendo che vogliamo calcolare una somma globale dove la somma è il risultato dell'applicazione della collective operation.

Tutti nel gruppo, prima o poi arriveranno a questa linea e dovranno svolgere l'operazione, quando usciamo poi dalla work_group_reduce sappiamo che la somma è la somma di tutti gli elementi.

Abbiamo comunque il problema che abbiamo la somma locale dei vari workgroup e non la somma globale di tutto il vettore.

Questa è una cosa a cui non avevano pensato all'inizio quelli che producono le GPU perchè solitamente le GPU sono pensate per calcolare in modo efficiente la Map, possono fare operazioni stencil, possono combinare cose che sono read only con cose che sono write only, le operazioni collective invece non sono proprio pensate per le GPU. Non è un problema solamente delle GPU, è una cosa più generale, l'idea prima era di usare Pc normali in una rete per fare una sorta di computer più "performante" e in questi tipi di computer c'era sempre il problema di implementare operazioni di reduce e operazioni barrier. Infatti se guardiamo ad alcuni dei computer dei top500, sono presenti degli hardware specializzati per implementare una comunicazione broadcast che è quello di cui necessitiamo per implementare una barrier.

Se guardiamo le specifiche di OpenCL ci sono varie operazioni workgroup_xxx che fanno varie cose ad esempio la workgroup reduce e la workgroup scan.

La scan è fatta in modo inclusive ed exclusive e come minimo supportano la somma, possono essere usate per prendere un vettore che ha al suo interno la somma degli elementi del vettore fino a quel punto del vettore.

C'è anche un'altra chiamata work_group_broadcast che viene utilizzata per implementare una operazione broadcast che permette di inviare un valore ad ognuno dei thread del workgroup.

Altre chiamate che possono essere utili sono work_group_all e work_group_any, queste le usiamo come una sorta di barrier e quando il workgroup arriva a questa chiamata allora deve controllare nel primo

caso che tutti i valori siano true (è un AND) oppure che uno sia true (OR).

Le idee viste fino ad ora con la reduce sono introduttive per le implementazioni della map, dello zip e della reduce stessa.

- Nel caso della Map abbiamo la funzione f e per ognuno degli elementi del vettore viene applicata questa funzione f, quindi avremo $x_i \rightarrow f(x_i)$
- Nel caso dello zip abbiamo la funzione g che prende due vettori e produce un vettore finale che dipende dagli input, ovvero:
 $z_i = g(x_i, y_i)$. Ad esempio per il prodotto scalare tra due vettori abbiamo uno zip(*) che quindi fa una moltiplicazione tra i due vettori seguito poi da una somma dei risultati.
- Nella reduce a seconda dell'operazione che facciamo (ad esempio una somma) sommiamo tutti gli elementi del vettore

Approccio “Library Based”, vogliamo utilizzare un codice che chiamo come una libreria, abbiamo tre possibili soluzioni differenti:

- Devo avere per prima cosa dell'initialization code, un codice che ad esempio mi fa prendere informazioni riguardo alla platform, al device, mi fa creare le command queue.
- Poi posso avere delle call tali che se scrivo per esempio:
 - `Map(std::vector<float> x, ...)`: questa call usa quello che è stato inizializzato in precedenza e poi dichiara il buffer, copia cose nel buffer (dall'host al device), esegue il kernel e copia indietro i risultati (dal device all'host).
- Un'altra call possibile: `map(clBuffer, ...)` userò direttamente il buffer senza avere la necessità di dichiarare lo spazio nella memoria del dispositivo, riempirla con i dati iniziali....

La map è un caso particolare perchè vogliamo lavorare su un vettore che è fonte dei dati e anche il posto in cui mettiamo i risultati.

La map in questo caso funziona bene quando è nello stage intermedio, se però è il primo operatore che invochiamo sul vettore non funziona più

bene perchè i dati in input non sono già nella GPU, cosa che invece accade quando la map è uno degli stage intermedi.

Nel caso generale consideriamo l'utilizzo dello zip, lo zip prende tre parametri: zip(A, B, C), dove A e B sono i vettori di input o un buffer mentre C è il vettore di output o un CLBuffer.

Questa non è una buona idea perchè qua stiamo chiedendo al programmatore di gestire l'allocazione delle risorse oltre al funzionamento del pattern mentre la sua volontà era semplicemente quella di usare un pattern data parallel.

Vorremmo poter fare una astrazione, in generale può essere fatto in una libreria ma ho bisogno di avere la possibilità di dire qualcosa alla libreria come se fosse un meta-programming.

Una possibile soluzione al problema è un framework di programmazione prodotto da alcuni ricercatori svedesi che si chiama skepu2.

I vettori in questo caso vengono chiamati smart container e sono una sorta di memory manager che cercano di gestire al meglio il modo in cui i dati sono memorizzati, dove i dati vengono spostati e dove vengono aggiornati.

In pratica viene utilizzato questo framework per fare in modo di astrarre il tipo di dato che viene usato nella GPU e nella CPU garantendo anche uno scambio di dati ottimizzato, vogliamo fare in modo di non specificare se il vettore che viene passato è un std::vector o un clBuffer.

Lezione 21

Con Pool evolution pattern³ si intendono quegli algoritmi che seguono una programmazione di tipo evolutivo, partiamo con una popolazione iniziale P poi questa si evolve con dei meccanismi che apportano modifiche alla popolazione. Solamente gli individui che si adattano meglio riescono a sopravvivere e poi riproducendosi porteranno alla nascita di persone che a loro volta hanno le stesse modifiche dei padri. Portando questa idea dell'evoluzione della specie nel campo della programmazione possiamo fare un parallelo tra gli individui e un set di oggetti che vengono manipolati con una regola evolutiva che va a modificare le feature dell'oggetto. Inoltre possiamo definire una funzione che è in grado di misurare la goodness dell'oggetto ovvero quanto l'oggetto è in grado di evolversi e quindi di sopravvivere al cambiamento. Preferiamo avere un cambiamento che sia lento perché il risultato ottenuto sarà migliore, se invece facciamo cambiamenti più rapidi otterremo una convergenza in tempi più rapidi ma in generale alla fine la funzione di fitness ci restituirà dei risultati che sono peggiori rispetto a quelli che otteniamo con cambiamenti lenti.

Parlando di pool evolution pattern abbiamo il seguente glossario da considerare:

- P = popolazione ovvero un set di individui
- S = funzione $P \rightarrow P$ che seleziona gli individui che sono più propensi al cambiamento
- E = Funzione evolutiva: Individuo \rightarrow Individuo, prende uno o più individui e applica il cambiamento
- F = funzione filter: $P \rightarrow P$, guarda alla popolazione e seleziona gli individui in base ad alcuni criteri
- T = funzione di terminazione: $P \rightarrow \text{Bool}$, controlla se l'obiettivo che ci eravamo posti è stato raggiunto o no

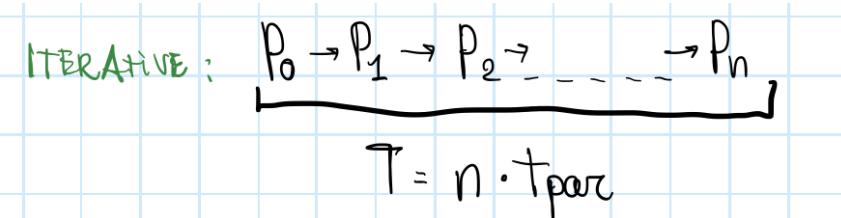
³ Pool Evolution: A Parallel Pattern for Evolutionary and Symbolic Computing
<https://link.springer.com/article/10.1007%2Fs10766-015-0358-5>

L'evolution pattern può essere espresso in questo modo:

```
while(not (t(P))){  
    N = e(S(P))  
    P = P U f(N,P)  
}
```

La funzione f filtra i nuovi individui rispetto alla popolazione attuale e poi il risultato di questa funzione viene unito alla popolazione già esistente.

Di base questo problema può essere risolto in modo iterativo perché ogni volta che ho una evoluzione della popolazione, devo considerare la popolazione precedente e da questa apportare le modifiche.



Non possiamo banalmente creare un thread che calcola P_0 mentre un altro thread calcola P_1 perché c'è una dipendenza tra le due iterazioni. Questo è un problema tipico quando abbiamo delle dipendenze, questo vuol dire che idealmente se anche facessi uno speedup della singola iterazione di un certo tempo, il fatto che faccio una sola iterazione o ne faccio 1000 non mi va a modificare lo speedup.

Quindi, cosa possiamo fare per parallelizzare la singola operazione?

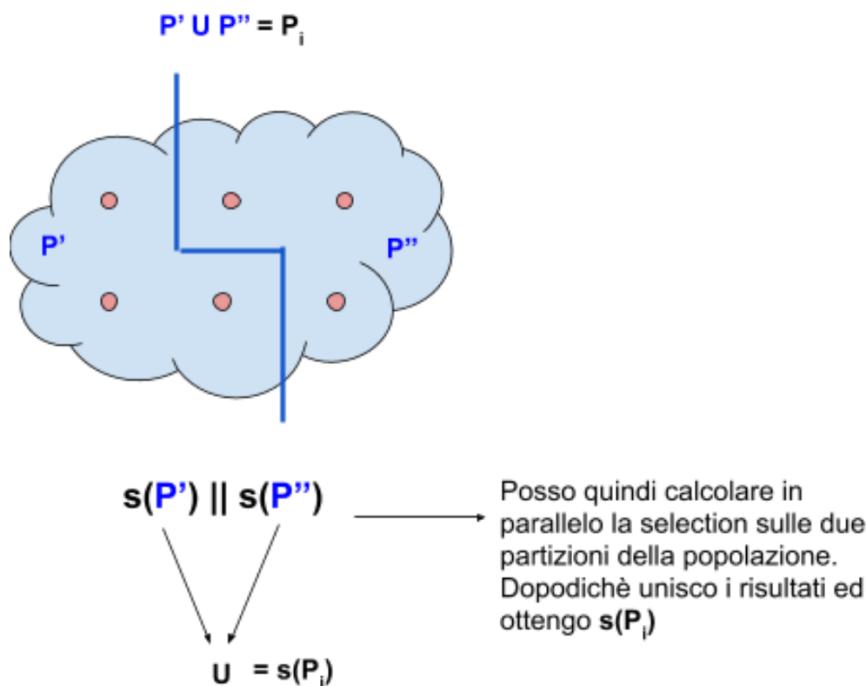
```
while(not (t(p))){  
    N = e(s(p))  
    P = P ∪ f(N, p)  
}
```

} parallelize

Prendiamo la prima delle due istruzioni, abbiamo $N = e(s(P))$ ovvero prima selezioniamo una parte della popolazione e poi applichiamo la trasformazione che mi fa evolvere la popolazione.

La selezione la possiamo migliorare e lo speedup dipende dal tipo di funzione che vogliamo utilizzare per splittare.

Quindi quando facciamo la popolazione possiamo pensare di suddividere tutta la popolazione in tante piccole popolazioni e per ognuna di queste facciamo l'operazione di selezione unendo alla fine i risultati. Si tratta quindi di una sorta di map.



La funzione evolutiva E è quella che applica i cambiamenti agli individui, non deve prendere in considerazione gli individui che stanno attorno a quello che stiamo considerando, quindi si tratta di una funzione che fa, anche in questo caso, una map.

Quindi, anche in questo caso possiamo apportare le modifiche in parallelo assegnando ai vari thread i task che devono modificare la popolazione.

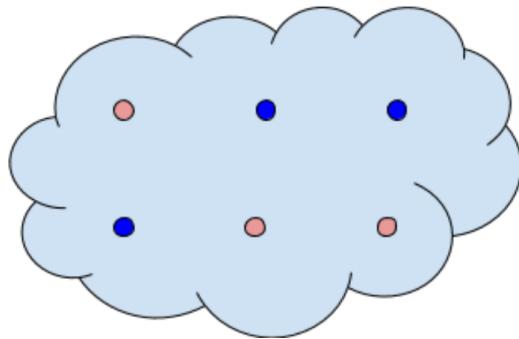
È anche possibile fare una sola map invece che due, questo è possibile se le funzioni da calcolare s ed e sono semplici, in questo caso possiamo creare una singola funzione che prima fa la s e poi fa la e, in

questo modo applichiamo la map una sola volta ottenendo comunque lo stesso risultato.

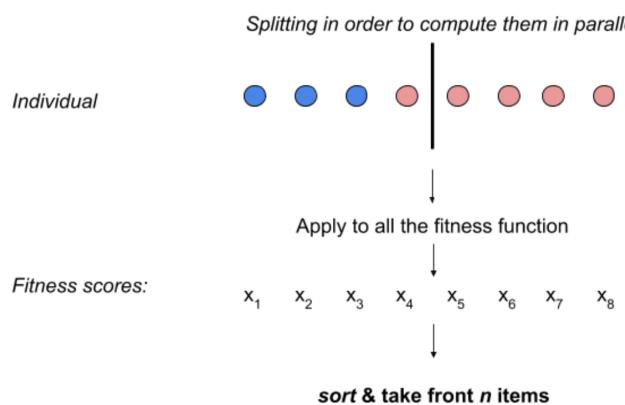
Per quanto riguarda la parallelizzazione della seconda parte abbiamo $P = P \cup f(N, P)$.

Consideriamo per prima cosa la $P = f(N, P)$, il parametro N indica la popolazione che ha già avuto una evoluzione, P invece indica la popolazione originale più quella che ha avuto l'evoluzione.

$$N = \text{blues} = \text{the evolved population}$$
$$P_i = \text{blues} + \text{reds} = \text{the original population}$$

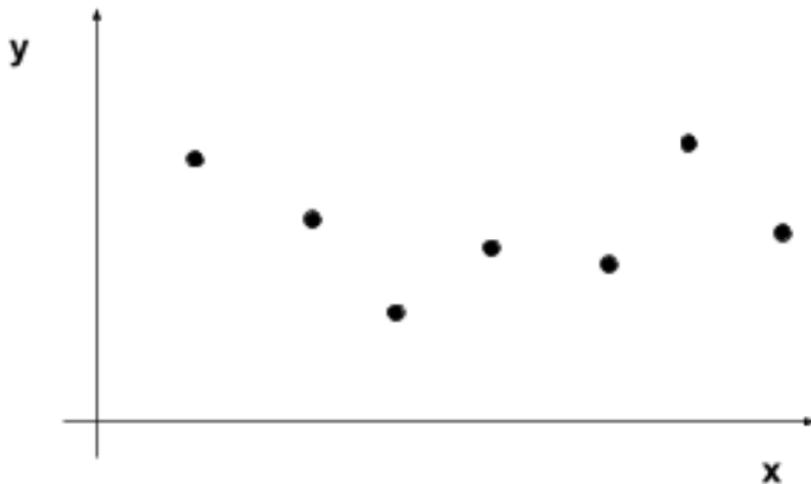


Anche in questo caso possiamo lavorare in parallelo dividendo i dati in due set andando poi ad eseguire la funzione per il calcolo del fitness. In questo caso però possiamo evitare di calcolare la funzione fitness sui nodi rossi perché la funzione fitness su questi l'abbiamo già calcolata al passo precedente.



Quindi selezioniamo solamente i nodi blu ed eseguiamo la funzione fitness su questi. Dato che vogliamo prendere solamente i migliori di questi elementi, uniamo i risultati dei rossi con i blu e li ordiniamo in base alla funzione fitness. Poi prendiamo i primi n elementi che sono i migliori.

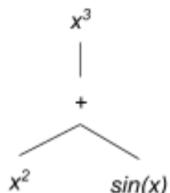
Un esempio: possiamo utilizzare il pool evolution pattern per calcolare la funzione che interpola un set di punti.



Supponiamo di avere un set di punti, questi sono l'input della funzione, poi possiamo immaginare gli individui come alberi formati da funzioni. Possiamo pensare ad un set di funzioni possibili, come ad esempio $\sin(x)$, $\cos(x)$, x^2 , x^3 , $x+c$, $x-c$.

Quello che posso fare è usare un set di individui, posso inizializzare il mio set di individui con un set di funzioni che vengono generate in modo random in un albero di una certa profondità.

Ad esempio possiamo creare un albero che ha 3 livelli e ogni volta per ognuna delle foglie devo scegliere una certa funzione che deve essere eseguita.

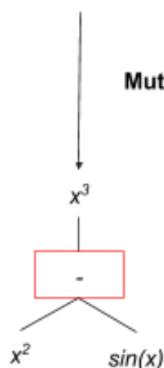
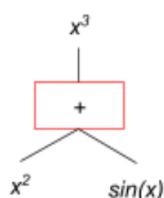


$$(\sin(x) + x^2)^3$$

Quando devo calcolare la funzione fitness, dovrò calcolare la funzione $(\sin(x) + x^2)^3$ per ognuno dei valori del piano cercando di compararla con ognuno dei valori esistenti, più sono simili e meglio è.

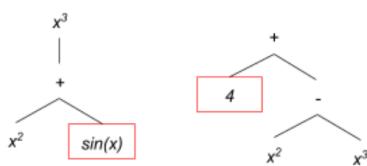
Cosa posso fare per selezionare e modificare gli individui?

La selezione è una cosa random, per farli evolvere devo cambiarli, quali sono i meccanismi in natura per farli cambiare? Potrei prendere uno degli alberi e ad esempio un possibile cambiamento è far diventare un + un -.

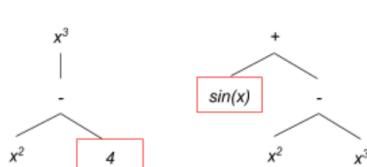


Mutation

Posso anche fare un crossover tra due alberi, nel senso che prendo un subtree di un albero (individuo) e ci metto l'altro tree.



Cross back
picked randomly



Orbit Pattern

Nella nostra terminologia lo possiamo indicare come “transitive closure of a grammar”.

Abbiamo una grammatica che mi definisce dei termini, prendiamo un set iniziale di termini, poi applichiamo le rule della grammatica fino al punto in cui applicando le rule non viene prodotto alcun termine “nuovo”.

L'orbit pattern usa un set di generatori (generator functions) che prendono un termine, applicano la funzione `gi` e poi otteniamo un nuovo termine.

$$g_i(t') \rightarrow t''$$

Una volta che il nuovo termine è stato generato allora dobbiamo chiederci se questo è già presente all'interno della nostra popolazione o no. Se è già presente allora non lo prendiamo in considerazione, se invece non è presente allora lo dobbiamo includere nella nostra popolazione P .

Portiamo questa idea dell'orbit pattern all'interno dell'idea del Pool evolution pattern:

- Abbiamo la funzione di selezione che, indipendentemente dall'input restituirà sempre true: `_ -> true`.
- Abbiamo un numero di evolution function che sono un'applicazione della funzione generatore. Se abbiamo 10 generatori, 10 production rule allora applichiamo 10 evoluzioni a quell'individuo
- Fitness function: lascia l'individuo così come è. È la funzione identità
- La funzione `f` controlla che gli item generati non fossero già presenti in precedenza

Possiamo vedere una possibile applicazione dell'Orbit Pattern nella soluzione del sudoku.

Una prima soluzione banale e brute force:

- Posso usare 9 generatori come nell'orbit pattern che in ognuna delle celle che sono vuote provano a mettere il loro numero generato. Per ogni gruppo da 9 celle provo a mettere un 1 in ogni posizione ad esempio e solamente uno finisce davvero nella bord mentre gli altri non possono metterlo perchè già è presente un 1.
- In principio da 1 board con k celle vuote possiamo generarne altre k. Quindi abbiamo un numero molto alto di possibili nuove configurazioni.

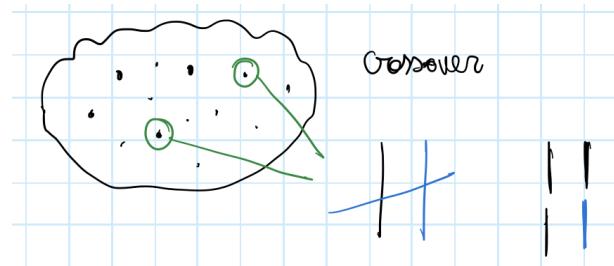
Questa sopra è una soluzione molto stupida e possiamo cercarne una migliore che mi termina in un numero decente di iterazioni.

Una possibile soluzione alternativa, invece di generare tutte le boards in ognuno degli step, prendiamo solamente le più promettenti come prossima evoluzione.

Possiamo anche avere altri algoritmi implementati con questo tipo di pattern, ad esempio gli algoritmi genetici.

Possiamo pensare un cromosoma come un albero e poi facciamo crossover o mutations, l'unica cosa importante è che la selection function deve selezionare due individui per il crossover e uno solo per la mutuation.

Quando consideriamo il crossover in realtà quello che facciamo è prendere due individui, prendere il genoma e ricombinarlo in un nuovo individuo. Questo vuol dire che nel nuovo individuo avrà parte del genoma di un individuo e parte dell'altro.



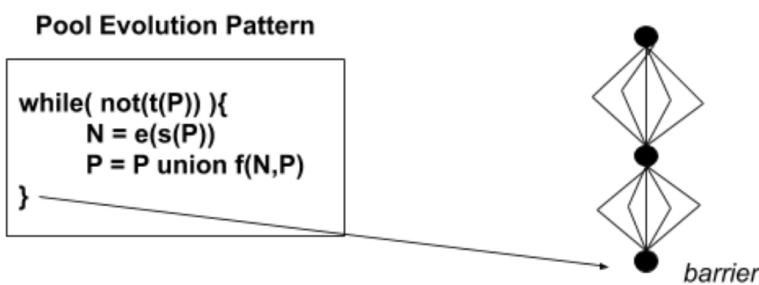
In natura in realtà succede qualcosa di differente, assumiamo che la popolazione in realtà è un set di popolazioni, ad esempio qua ne prendiamo due. Ognuno di questi set di individui evolve per un numero di iterazioni da solo.



Dopo le K iterazioni abbiamo le due popolazioni che si sono evolute e sono differenti rispetto a come erano all'inizio.

Poi quello che si fa è prendere alcuni individui da una delle due popolazioni e li mettiamo nell'altra popolazione, in questo modo abbiamo una evoluzione differente e più completa rispetto a quella vista fino ad ora.

RiconSIDERIAMO il codice visto in precedenza, per iniziare l'evoluzione successiva della popolazione devo per forza mettere una barrier alla fine perchè devo essere sicuro che ognuna delle popolazione abbia terminato la sua evoluzione.



Da un punto di vista matematico è reale ma nella realtà non è così perchè non è vero che prima di iniziare la successiva evoluzione in Australia devo aver terminato l'evoluzione in Giappone.

Quindi quello che si fa è rimuovere la barrier rendendo la popolazione più elastica, in questo modo la popolazione continuerà comunque ad evolversi.

Immaginiamo di avere una popolazione e un thread pool in cui ogni thread:

- Seleziona
- Evolve
- Filtra
- Manda i nuovi individui in P oppure no

Quindi alcuni individui magari ci metteranno meno tempo a fare l'evoluzione e altri di più, non ho più una sincronizzazione tra le varie iterazioni. Gli individui che evolvono più velocemente possono fare maggiori progressi nell'evoluzione.

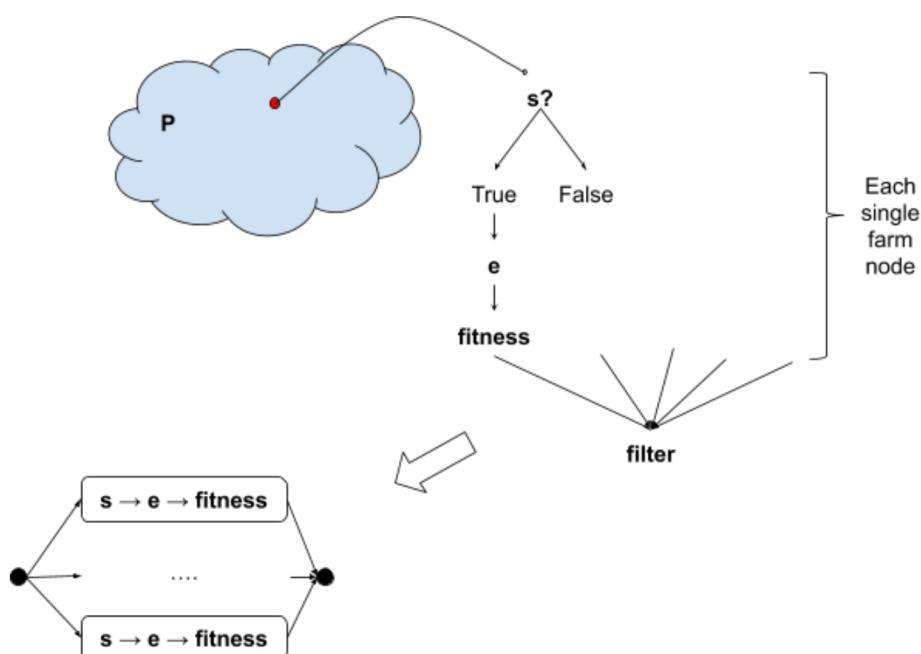
Pensando l'idea dell'evoluzione da un punto di vista dei pattern potrei utilizzare una pipeline(farm, farm), la prima farm mi fa l'operazione N e la seconda pipeline fa la parte P.

C'è un problema in questo caso, non abbiamo un parallelismo reale perchè ci sono delle dipendenze, questo vuol dire che se aggiungo la popolazione, userò magari 8 core per la prima parte ma i 5 che uso per la P sono inutilizzati, quando uso i 5 della P invece sono inutilizzati gli 8 thread della N.

Quindi questa soluzione non è utile per niente, l'unica cosa utile è: comp(farm, farm), prima fai la N e poi fai la P.

Oppure posso mergiare select, evolve e fitness, prima mi chiedo se l'individuo è stato selezionato, poi faccio l'evolve e poi la fitness. Alla fine faccio la filter ma solamente quando ho le funzioni fitness di tutti gli individui, quindi la filter è una vera e propria barrier.

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 163



Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 163

Lezione 22

Come possiamo fare ad adattare a runtime il comportamento della nostra applicazione parallela nel momento in cui la struttura parallela del nostro programma è già conosciuta?

L'idea è che differentemente da quando usiamo un pattern “non structured programming environments”, utilizzando i pattern andiamo ad esporre al compilatore la struttura della nostra applicazione parallela. Questa è una buona opportunità per fare delle ottimizzazioni.

Le ottimizzazioni che possiamo fare in questo caso sono solamente ottimizzazioni a runtime, questo comporta l'esecuzione di altro codice, differente da quello dell'applicazione.

Questo codice che viene eseguito è anche sequenziale e mi comporta un overhead per la mia applicazione. Questo overhead che abbiamo potrebbe però essere utile se abbiamo delle condizioni che ci permettono di adattare la nostra applicazione.

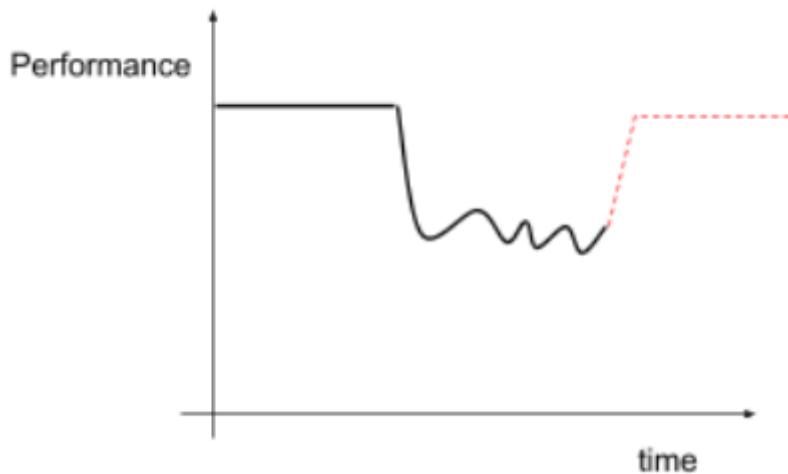
Un primo esempio:

Supponiamo di avere una non exclusive usage machine (come ad esempio quella multicore a cui abbiamo accesso), si tratta quindi di una macchina in cui lavorano molte persone in contemporanea.

In questo caso abbiamo un problema specialmente se ci troviamo ad avere un'applicazione parallela che lavora per molto tempo (minuti). Con un'applicazione di questo genere infatti abbiamo che se ci sono altri processi in corso nella macchina potrei avere un calo delle prestazioni della nostra applicazione. Quello che possiamo fare però è organizzare la nostra applicazione in modo che sia in grado di adattare il suo parallelism degree mantenendo quelle risorse che gli permettono comunque di raggiungere il goal che aveva precedentemente scelto. Supponiamo poi di avere anche una parallelism degree nw iniziale.

Possiamo vedere in un grafico la variazione delle performance della nostra applicazione nel tempo.

Inizialmente le performance sono alte, poi abbiamo un calo delle performance, se però la macchina ha abbastanza risorse possiamo pensare di riportare le performance dell'applicazione al livello che mi aspetto.



Ad esempio potrei avere una farm con 4 worker, ognuno di questi worker lavora su un core differente, supponiamo che venga aggiunta una nuova applicazione e che venga assegnata ad uno di questi 4 core. Il worker sul core in questione lavorerà meno di quanto lavorava in precedenza. Se però ho un altro core libero quello che posso fare è l'unstick del thread da quel core in modo da portarlo su un altro core che in quel momento non è utilizzato.

L'unstick è una system call che prende del tempo introducendo anche overhead, la cache di un core dovrà essere spostata nell'altro core quando viene svolta questa operazione di unstick.

Consideriamo un secondo esempio, abbiamo delle applicazioni che devono funzionare in modo differente a seconda della "fase" che devono gestire. Un esempio può essere la connessione WiFi che in alcuni momenti della giornata sarà più utilizzata e in altri meno.

Cosa possiamo fare per adattare la nostra applicazione al carico di lavoro?

- Possiamo variare il parallelism degree, questa ottimizzazione si chiama “concurrency throttling”.
Il “concurrency throttling” è diverso da “frequency throttling”, con il secondo intendiamo la diminuzione o l'aumento del clock del processore per avere maggiori performance o minori performance. Il consumo di energia sarà proporzionale al cubo della frequenza, più è alta la frequenza e maggiore è il consumo. La frequency throttling è utilizzata nelle CPU Intel con il turbo boost in cui il processore capisce che sto facendo dei lavori complicati e quindi modifica la frequenza per permettermi di avere performance migliori.

Supponiamo di avere una farm, vogliamo modificare il numero dei worker per avere un miglior Service Time, ad esempio potremmo voler passare da una farm con 3 worker ad una con 5 worker per fare in modo di lavorare ancora di più in parallelo.

$\text{farm}(f, nw) \rightarrow \text{farm}(f, nw')$

Dove $nw' > nw$, quello che devo fare è far sapere all'mitter che ora ci sono più worker di prima, questo vuol dire che quando il task verrà schedulato potremo assegnarlo ad un thread che prima non era presente.

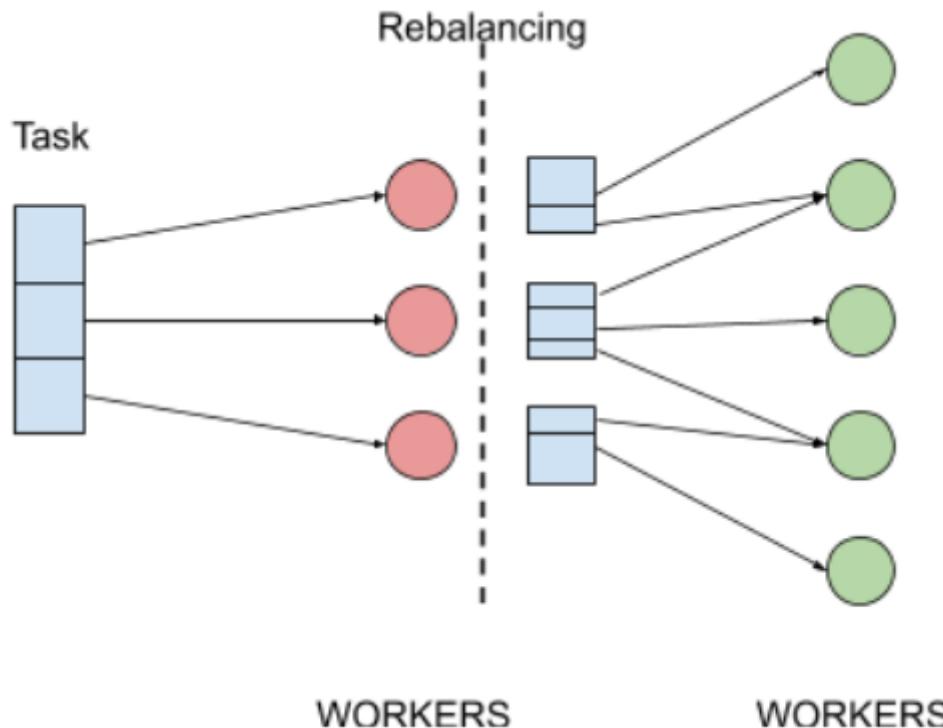
Se invece abbiamo una map, possiamo far variare il numero di worker che lavorano in contemporanea per fare in modo che la latenza sia migliore.

$\text{map}(f, nw) \rightarrow \text{map}(f, nw')$ con $nw' > nw$

Con la map la situazione è molto più complessa della farm, qua infatti abbiamo un array iniziale che viene suddiviso tra i vari worker, il problema è che se aggiungo nuovi worker dovremo andare a dividere nuovamente l'array in modo da avere più parti per assegnarle ai vari worker.

Questa ridistribuzione dei dati può essere molto costosa e non semplice perchè devo fermare i thread e permettere ai thread di

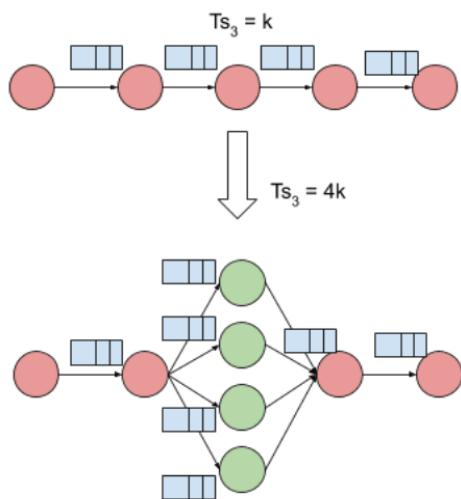
splittere i loro dati per assegnarli ad altri thread.



In tutto ciò non dovrò redistribuire gli elementi già calcolati e mi servirà anche un codice che mi permette di lavorare per un certo periodo come se la riconfigurazione non ci fosse stata.

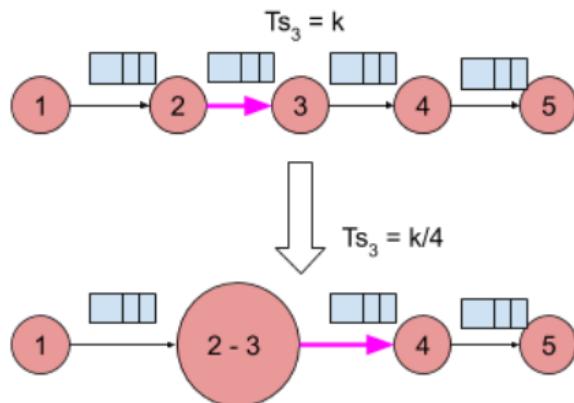
Tutti i worker che lavorano devono anche essere d'accordo con la ridistribuzione che viene fatta.

- Consideriamo il caso in cui abbiamo una pipeline, questa ad un certo punto può diventare più lenta e abbiamo bisogno di adattarla in modo che non si creino dei colli di bottiglia

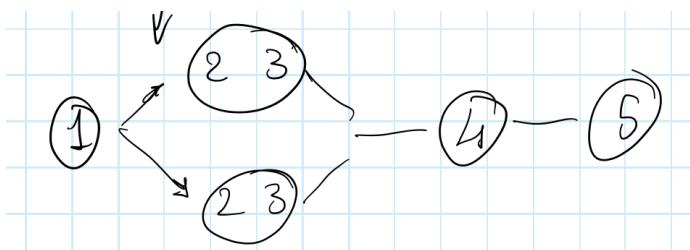


Per fare questa ottimizzazione dobbiamo fare alcune modifiche e fermare il worker precedente alla farm per dirgli che ora è cambiato il modo in cui deve ridirezionare l'output, ora non ha più un solo nodo in output ma ne ha più di uno oppure deve ridirezionare l'output verso un emitter.

Se invece siamo nella situazione opposta, con il service time che diminuisce allora dovremo fare in modo di unire i vari worker in uno unico che fa il lavoro che prima veniva svolto da entrambi i due worker.



Se l'unione dei due stage comporta la creazione di un nodo che è troppo grande rispetto agli altri è anche possibile organizzare questo nodo all'interno di una farm:

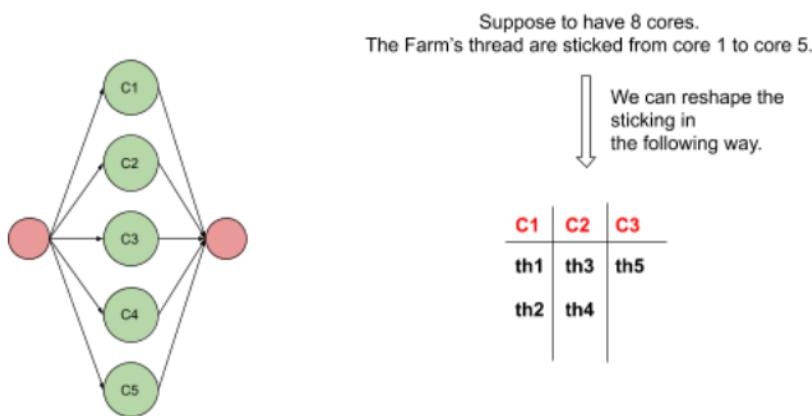


Consideriamo un altro problema, abbiamo 5 thread che lavorano in parallelo, ognuno in un core differente. Ci accorgiamo che in realtà ci bastano solamente tre thread dei 5 e quindi abbiamo varie possibilità, questo è utile specialmente se siamo in un'applicazione con fasi in cui sappiamo che prima o poi il numero di thread aumenterà di nuovo:

- Potremmo uccidere i thread che non ci servono ed eventualmente crearli nuovamente quando sarà necessario utilizzarli. Questo

procedimento è dispendioso e non conviene lavorare in questo modo.

- Possiamo fare in modo che i 5 thread continuino ad essere 5 comportandosi però come se in realtà fossero solamente 3. Quindi, inizialmente potremmo aver assegnato ogni thread ad un core, poi possiamo spostare i thread che non ci servono assegnandoli ad uno dei core che continuiamo ad utilizzare.



Questo ci porta ad avere un degrado delle performance dei thread che non saranno il doppio dei singoli thread perchè condividono delle risorse.

Ad esempio se abbiamo 8 core e nei primi 5 core mappiamo i 5 thread possiamo fare una modifica e spostare i thread dai core 4 e 5 nei core 1 e 2.

Fare un'operazione di questo genere richiede una system call per ogni thread che va spostato da un core ad un altro.

Se devo poi espandere la farm potremo riportare i thread nella situazione precedente in cui sono legati ad un core specifico. Con questa ottimizzazione si lavora su mapping del grafo delle risorse fisiche.

Consideriamo ora il Frequency Throttling

Supponiamo di avere 8 core e di voler migliorare le prestazioni di due thread che vengono eseguiti su due di questi core, potremmo aumentare la frequenza di questi due core.

Per fare queste modifiche abbiamo bisogno di alcuni permessi e di alcune “entità” nella nostra macchina:

- Ci serve l'accesso al governor che permette di modificare la frequenza dei vari core. Non tutti i sistemi operativi mi permettono l'esecuzione di modifiche di questo genere perchè è lo stesso sistema operativo a scegliere la frequenza dei core.
- Nella maggior parte delle CPU, le frequenze possono essere modificate solamente per socket, questo vuol dire che se ho 8 core, 4 su un socket e 4 su un altro posso modificare la frequenza dei due socket ma non posso modificare le frequenze dei singoli core. Questo è per motivi elettronici e fisici, la frequenza deve essere cambiata insieme al voltaggio che normalmente dipende dalla distribuzione dell'energia nel chip e la distribuzione dell'energia è fatta in modo che l'energia, così come il clock, abbia percorso di lunghezza uguale verso ognuna delle parti del chip. Altrimenti avremmo disallineamenti dal punto di vista delle performance.

Il trend ora è quello di spostarci in una situazione dove energia e frequenza sono per core e non per socket, questo vuol dire che potremmo cambiare solamente la frequenza del core X e quindi la frequenza dei contesti di quel core.

- Cambiare la frequenza richiede i diritti root, in questo caso facciamo una singola system call. Un'alternativa, se non siamo root è sviluppare un server che rimane attivo in attesa di particolari richieste e quando arriva una certa richiesta modifica la frequenza. Questa seconda soluzione è meno performante perchè dobbiamo fare più operazioni per modificare la frequenza.
- Se cambiamo la frequenza di un core, tutti i threads che vengono eseguiti su quel core vedranno la modifica delle performance (a

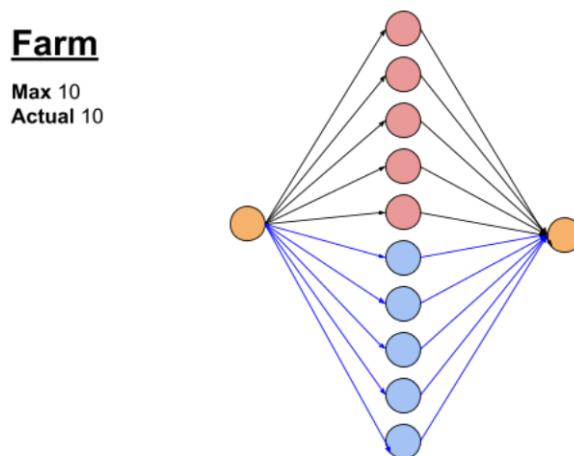
differenza dello sticking dei thread perchè in questo caso posso fare lo stick solamente dei miei thread). In una macchina condivisa questa non è una buona politica.

- Cambiare la frequenza è una tecnica molto utilizzata ed è chiamata DVFS ovvero Dynamic Volt Frequency Scaling.
- Se prendiamo una macchina con una frequenza da 1 GHz che può essere portata fino a 2,4GHz vuol dire che ci sono degli step per la modifica della frequenza del processore, non possiamo scegliere un qualsiasi valore, ci sono degli step per aumentare o diminuire la frequenza

Se noi conosciamo la struttura della computazione parallela possiamo utilizzare i meccanismi visti in precedenza per adattare l'applicazione e aumentare le performance, se invece non conosciamo la struttura dell'applicazione, l'unica cosa che possiamo fare è il DVFS.

Come possiamo sfruttare queste tecniche?

Quando dichiariamo un certo pattern, ad esempio una farm, possiamo definire un massimo parallelism degree che voglio avere per quel pattern. Ad esempio posso dichiarare una farm che ha internamente 10 worker. Questi 10 worker li lancio tutti insieme ma può capitare che alcuni di essi non siano necessari all'inizio, quindi potrei avere alcuni worker (thread) in stato ready mentre altri in stato idle.



Quando poi risvegliamo un thread dobbiamo avvisare l'mitter dicendogli che il numero dei thread disponibili è aumentato.

Possiamo però fare una cosa differente da quella appena descritta e utilizzare il “Control Theory” ovvero una teoria in cui viene osservato il sistema e in base a queste osservazioni si cerca di prendere delle decisioni che possano migliorare il comportamento del sistema.

L'idea è che abbiamo un sistema che viene testato, abbiamo alcuni “sensori” nel sistema e questi mi fanno alcune misurazioni di quello che succede nel sistema, poi ci sono gli attuatori che, dati i dati dei sensori possono fare modifiche al sistema. La connessione tra i sensori e gli attuatori è fatta tramite il Feedback loop, in questo canale passano le informazioni che dicono agli attuatori le varie misurazioni effettuate e se c'è un piano per modificare il sistema allora gli attuatori lo metteranno in atto adattando il sistema alle richieste ricevute.

In questo caso viene utilizzato il MAPE loop ovvero Monitoring Analyze Plan Execute.

Collegiamo alla nostra applicazione parallela un'attività parallela indipendente dai thread e dai processi che uso per calcolare i risultati che utilizzo per implementare il MAPE Loop.

Prendiamo il caso della farm, abbiamo detto che vogliamo modificare il numero di workers presenti all'interno della farm, posso associare alla farm un thread indipendente che fa da sensore e che mi fa una misurazione della farm, in particolare quello che viene misurato è il Service Time.

Dovrei essere in grado di chiedere all'mitter e al collector l'arrival rate e il delivery rate del risultato. Potrei osservare che l'arrival rate è alto ma il delivery non è troppo alto, ad esempio se osservo che arriva un task al secondo ma poi mando in output un risultato ogni due secondi, vuol dire che perdo un task per ogni risultato. Un modo per risolvere è aumentare il numero dei worker.

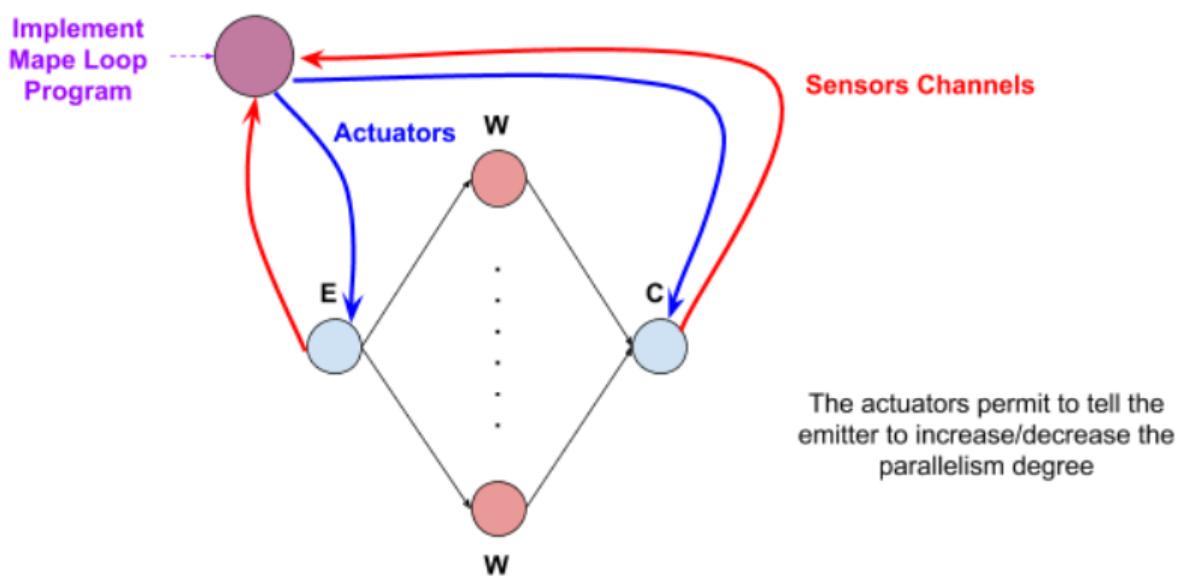
Quello che si occupa di fare questa misurazione è il Monitor e nel caso della farm mi basta metterne uno solo e questo mi fornisce un'idea di quello che devo analizzare.

Se invece avessi avuto una pipeline avrei dovuto mettere un monitor in ognuno dei vari stage, con la reduce strutturata ad albero avrei messo un monitor in ciascun nodo dell'albero.

Questo monitor lo posso utilizzare se conosciamo bene la struttura del pattern che stiamo utilizzando.

Quindi ad esempio nella farm devo conoscere quanti worker ho e come è organizzata, devo considerare il service time della farm che è il massimo tra emitter, collector e Tempo di ogni worker/Numero workers. Quando capisco chi tra questi tempi mi rallenta l'esecuzione complessiva allora dovrò fare una modifica in modo che le performance possano migliorare.

Come posso programmare un sistema di questo genere?

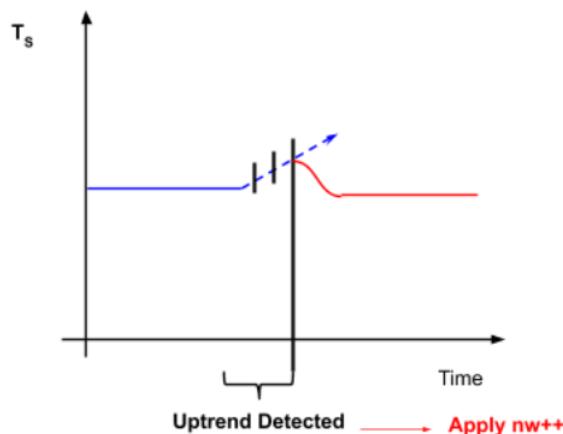


Per creare un sistema di questo genere dobbiamo creare un thread aggiuntivo che si occupa di controllare se la farm si sta comportando nel modo corretto. Quindi abbiamo un sensor channel connesso al collector e al thread esterno, in questo channel passano le varie indicazioni che

otteniamo monitorando la situazione del collector. Poi il thread esterno comunica con l'mitter e gli indica come variare il numero di workers attivi.

Idealmente devo reagire quando vedo qualcosa che non sta facendo funzionare bene la farm, il Mape Loop però può essere usato anche per azioni pro attive. Consideriamo il caso in cui vediamo qualche piccola anomalia nella farm, se vediamo un trend che prima o poi modificherà il corretto funzionamento della farm, possiamo fare subito una modifica per essere pronti già da subito a far tornare la farm al funzionamento corretto.

Possiamo vedere questo nel grafico presente qua sotto, supponiamo di avere un tempo sequenziale, con due thread più o meno lo dimezzo. Ad un certo punto mi accorgo che il tempo necessario per l'esecuzione della mia applicazione parallela sta salendo, quindi procedo subito con la modifica del numero di thread che stanno lavorando per fare in modo che il tempo necessario per l'esecuzione dell'applicazione possa tornare a scendere di nuovo.



Per fare questo ho bisogno di una certa granularità che mi permetta di comprendere cosa succederà prima che avvenga qualcosa di drammatico.

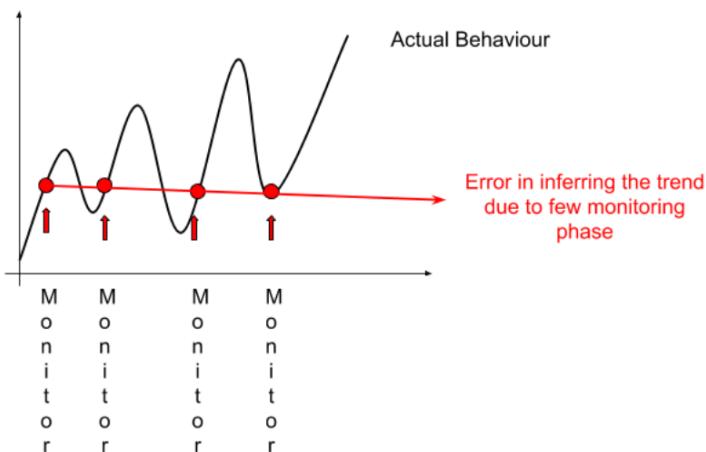
C'è un trade off tra il tempo che spendo per prendere le decisioni e il tempo che spendo per calcolare. Se chiedo al collector e all'mitter di indicarmi il service Time, questo tempo non è legato al tempo che serve per calcolare, eseguo del codice che non è legato al business logic code

e neanche al template code ovvero al codice di emitter e collector, si tratta solamente di template code.

Più codice metto nella fase di monitoring e adattamento delle performance e peggio sono le performance ma se questo mi permette di prendere decisioni intelligenti il tempo che perdo può servire per farmi guadagnare del tempo in futuro.

È sempre un trade off tra il codice che uso per il template, il tempo di lettura dei dati e il tempo per modificare l'applicazione parallela.

Per essere proactive devo avere una grande granularità nella fase di monitoring e devo continuare a monitorare di continuo per riuscire ad identificare il trend, questo perchè altrimenti non riuscirò mai ad identificare un trend che possa risultare interessante.



Problema della stabilità

C'è da considerare anche un altro problema ovvero la stabilità.

Se abbiamo un sistema che ha cambiamenti piccoli ma frequenti dobbiamo evitare di cadere in una situazione in cui per seguire tutte queste modifiche non raggiungiamo mai uno steady state.

Ad esempio prendiamo il caso di una farm con 10 workers, vediamo che il tempo di inter arrivo è maggiore del Ts dei worker = $Tw/10$.

Prendiamo quindi una contromisura e portiamo a 9 il numero di workers, in questo modo però inviamo meno dati di quanti ne possiamo calcolare quindi avremo che $Ta < Tw/10$. Quindi torniamo nuovamente ad avere 10 workers.

Il sistema in questo caso non è stabile e continua a passare da 10 a 9 workers, questo è un problema, non vogliamo prendere delle contromisure troppo consecutive che mi modificano continuamente lo stato della farm.

Possiamo modificare il Mape Loop in modo che mantenga una certa conoscenza di ciò che ha fatto in precedenza in modo che possa prendere le decisioni successive considerando anche il passato.

Nota: possiamo dire che se nella farm abbiamo una bottleneck nei worker la possiamo risolvere aumentando il numero di workers. Questo vuol dire che emitter e collector sono sotto utilizzati, quello che si può fare è inserire il mape loop direttamente all'interno dell'mitter in modo da evitare una comunicazione tra thread (Mape loop ed emitter).

Lezione 23

Introduzione a Cuda

Vediamo le differenze tra CUDA e OpenCL:

CUDA	OpenCL
Cuda è un linguaggio (una estensione di C++) perchè introduce alcune feature nuove. Per compilarlo non usiamo g++ ma un compilatore apposito, nvcc.	È una libreria, all'interno della libreria possiamo fare le chiamate e questa farà tutto il resto.
Usato per programmare le GPU	Usato per programmare le GPU e le FPGA

Cuda introduce due estensioni al linguaggio C++:

- Keyword `__global__` serve per segnare i routing che possiamo utilizzare per implementare il kernel nella GPU
- `<<< arg1, arg2 >>>` `nameF(parametri)`: sono usati per dire al compilatore di eseguire quello specifico kernel code sulla GPU. I parametri che passiamo sono le dimensioni della grid sulla GPU (`arg1`) e la dimensione del thread block (`arg2`).

Passando solamente questi due parametri abbiamo un'idea di spazio bidimensionale, quindi abbiamo gruppi di thread che formano una grid che influenzano l'esecuzione del kernel sulla GPU perchè:

- Possiamo sincronizzare cose che sono sullo stesso thread.
- Possiamo sincronizzare cose che sono sullo stesso gruppo di thread.
- Non possiamo sincronizzare cose che sono in gruppi diversi.
- I thread dello stesso gruppo devono eseguire lo stesso

codice.

NameF è la funzione che vogliamo eseguire sulla GPU con i suoi parametri. La funzione nameF va definita usando la keyword __global__.

Esempio di codice Cuda, la funzione che somma i due vettori viene definita mettendo davanti alla definizione la keyword __global__.

Se vogliamo allocare la memoria nella GPU dobbiamo usare il comando “malloc” di Cuda.

Dobbiamo anche gestire gli eventi che avvengono nella GPU. Ad esempio possiamo creare 2 eventi, start e stop, prima di avviare i thread e dopo averli avviati poi registriamo il tempo in questi due eventi.

Questi due eventi possono essere utilizzati per misurare il tempo di esecuzione del kernel, gli eventi sono memorizzati nella GPU e in alcuni casi dobbiamo sincronizzarli con la CPU per essere sicuri che la memoria relativa allo storage venga spostata nella memoria della CPU, per fare questo spostamento è presente la chiamata

CudaEventSynchronize.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;

    // Host input vectors
    double *h_a;
    double *h_b;
```

```
//Host output vector
double *h_c;

// Device input vectors
double *d_a;
double *d_b;
//Device output vector
double *d_c;

// Size, in bytes, of each vector
size_t bytes = n*sizeof(double);

// Allocate memory for each vector on host
h_a = (double*)malloc(bytes);
h_b = (double*)malloc(bytes);
h_c = (double*)malloc(bytes);

// Allocate memory for each vector on GPU
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

int i;
// Initialize vectors on host
for( i = 0; i < n; i++ ) {
    h_a[i] = sin(i)*sin(i);
    h_b[i] = cos(i)*cos(i);
}

// Copy host vectors to device
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

int blockSize, gridSize;

// Number of threads in each thread block
blockSize = 1024;

// Number of thread blocks in grid
gridSize = (int)ceil((float)n/blockSize);

// Execute the kernel
vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy array back to host
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

// Sum up vector c and print result divided by n, this should equal 1
within error
```

```
double sum = 0;
for(i=0; i<n; i++)
sum += h_c[i];
printf("final result: %f\n", sum/n);

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

// Release host memory
free(h_a);
free(h_b);
free(h_c);

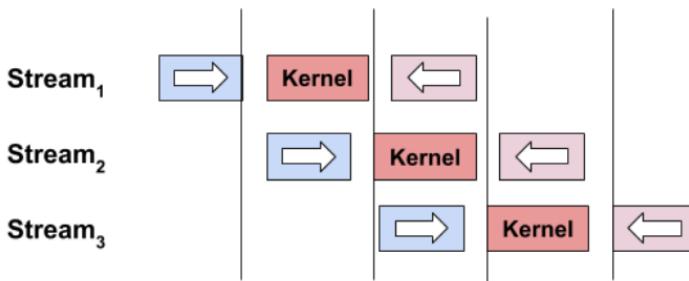
return 0;
}
```

Anche con Cuda dobbiamo dichiarare dei buffer per la memoria sulla global memory della GPU e poi muoviamo i dati dalla host memory alla memoria della CPU.

In particolare è presente un comando nelle ultime versioni di Cuda che è `cudaMallocManaged`, questo comando ci permette di dichiarare ad esempio un vettore che alloco con le regole di Cuda e per questo motivo lo possiamo utilizzare sia nella CPU sia nella GPU.

Lo spostamento dei dati dalla CPU alla GPU e viceversa viene gestito completamente dal `cudaMallocManager` e quindi non deve essere gestito manualmente dal programmatore.

È possibile definire degli stream che vengono sincronizzati direttamente da Cuda, con sincronizzazione si intende lo spostamento dei dati da GPU a CPU o viceversa. Lo spostamento dei dati sarà parallelo rispetto all'esecuzione del kernel.



Lezione 24

****Lezione importante per il report del progetto****

Supponiamo di dover risolvere un certo problema, la prima cosa che dobbiamo cercare di capire è il business logic code.

In particolare ci interessano tre parti al livello della logica del codice:

- Timings: dovremmo avere un'idea del tempo necessario per l'esecuzione delle varie fasi del programma
- Control: dobbiamo sapere quali sono le varie fasi e come si evolve l'applicazione nel corso del tempo
- Data (o state) Access: si tratta dell'accesso ai dati, in particolare l'accesso allo stato. Vogliamo distinguere i dati che vengono acceduti da una singola attività parallela da altri dati che invece vengono acceduti da varie attività parallele (in modi concorrente o sequenziale).

Affrontando il problema possiamo seguire il seguente processo, scegliendo per prima cosa il pattern corretto che vogliamo utilizzare:

- Per prima cosa scegliamo un pattern tra i vari che abbiamo a disposizione
- Se non troviamo un pattern che si adatta bene alle nostre esigenze abbiamo due possibilità
 - La prima possibilità è utilizzare degli strumenti a basso livello come ad esempio i thread e le mutex classiche. Si tratta di una scelta abbastanza complessa e quindi solitamente non si fa così.
 - La seconda possibilità consiste nell'utilizzo di building block, quindi nella creazione di un nuovo pattern che non è offerto in modo esplicito dalla libreria ma che viene costruito da noi.
- Può capitare che abbiamo un pattern che si adatta ad una certa situazione e uno che si adatta a risolvere un'altra parte del problema. La soluzione può essere la composizione dei vari

pattern che abbiamo a disposizione in modo da ottenere una soluzione efficiente per il problema.

In particolare quando si parla di pattern composition, abbiamo un albero di pattern che ha nelle foglie il codice sequenziale.

Possiamo cercare alternative e possiamo valutarle. Ad esempio se vogliamo valutare alcune parti “data parallel” della nostra applicazione e queste parti sono ad esempio un for con iterazioni indipendenti, possiamo usare un parallel for. Un’alternativa può essere lo spostamento di questo codice nella GPU o in un acceleratore simile. In alcuni casi possiamo anche modificare il pattern che utilizziamo, il map pattern ad esempio può essere sostituito con qualcosa che genera le sub partizioni, uno stream parallel pattern che calcola i risultati parziali e poi uno stage che crea il risultato finale.

Possiamo anche cambiare la struttura del singolo pattern con una composizione di altri pattern oppure con un pattern modificato, abbiamo infatti varie alternative a questo livello:

- Possiamo considerare una differente implementazione di questo pattern
- Possiamo considerare un sostituto per il pattern che vorremmo utilizzare
- Possiamo eseguire un refactoring dei subtree del mio albero.
Considerando i data parallel computations se ho una map seguita da una map posso considerare che potrebbe essere uno spreco spostare i dati anche tra una map e la successiva e non soltanto in ingresso alla prima map e in uscita dalla seconda. Quindi una possibile soluzione a questo problema potrebbe essere una composizione delle due funzioni che vengono svolte nelle due farm in modo da creare una singola farm.

Queste varie alternative devono essere valutate utilizzando il performance model, tutte queste alternative richiedono un utilizzo del performance model perchè quando valutiamo le alternative, se le alternative sono riconosciute come tali vuol dire che calcolano la stessa

cosa (non mi interessa se calcolano cose differenti) ma possibilmente con performance differenti e un uso differente delle risorse.

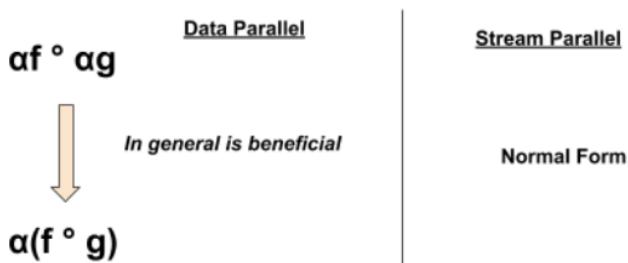
Prendiamo un esempio, abbiamo un'applicazione A con un codice e usiamo FastFlow come libreria e l'architettura dello Phi KNL per eseguire il codice. Se siamo nella condizione fortunata in cui l'applicazione è scritta in modo che facilmente posso inserire il codice all'interno di un FF_Node di fastflow posso trasformare l'applicazione in un codice di fastflow e il performance model che ho in mente dovrà considerare il modo in cui viene allocata la memoria e il modo in cui avvengono le comunicazioni.

Devo considerare tutta questa catena, questo performance model però può essere usato in modo “light” se cerchiamo di ragionare in termini della qualità che la mia applicazione parallela dovrà avere.

L'esempio è la map fusion, ho la possibilità di eseguire qualcosa che è un set di attività concorrenti seguite da una barrier e poi altre attività che fanno altre computazioni.

Potrei rendermi conto che la barrier non è necessaria per l'applicazione ma è presente solamente per un motivo “implementativo” che mi impone di averla, se riesco a mettere insieme più cose che calcolano lo stesso numero di attività concorrenti, questo sicuramente mi aumenterà il grain della computazione.

Questo è un beneficio in tutte le architetture a meno che tu hai un'applicazione che ha una fase in cui ogni sottoproblema è risolto da una stessa funzione seguita poi da un altro sottoproblema in cui il codice può divergere (if true... if false...). Quindi se alla fine prendi una GPU per risolvere questo problema ad un certo punto avrai i thread che saranno serializzati perché le funzioni divergono.



In generale se siamo nel caso del disegno e riusciamo ad arrivare alla forma normale, riusciamo anche a migliorare le performance perchè

l'overhead che è dovuto al setup delle componenti parallele sarà in percentuale minore rispetto a tutta l'esecuzione della computazione nella seconda soluzione.

Vediamo un esempio:

Consideriamo una pipeline con tre stage che sono delle farm e internamente le farm hanno del codice sequenziale, se prendo i thread t1, t2 e t3 per eseguire ognuno dei worker, idealmente quando eseguo questo come pipeline di una farm avrò da bilanciare i tre stage della pipeline.

Se metto n1 worker nella prima farm avrò un service time che sarà t_1/n_1 , io voglio che questo sia simile a t_2/n_2 e a t_3/n_3 .

Quando devo cercare di ottenere la forma normale vogliamo arrivare ad avere una singola farm che esegue la composizione degli stage sequenziali S1, S2 e S3.

Dato che $t_1+t_2+t_3$ è maggiore dei singoli tempi dovremo aumentare anche il numero di worker che devono lavorare in contemporanea.

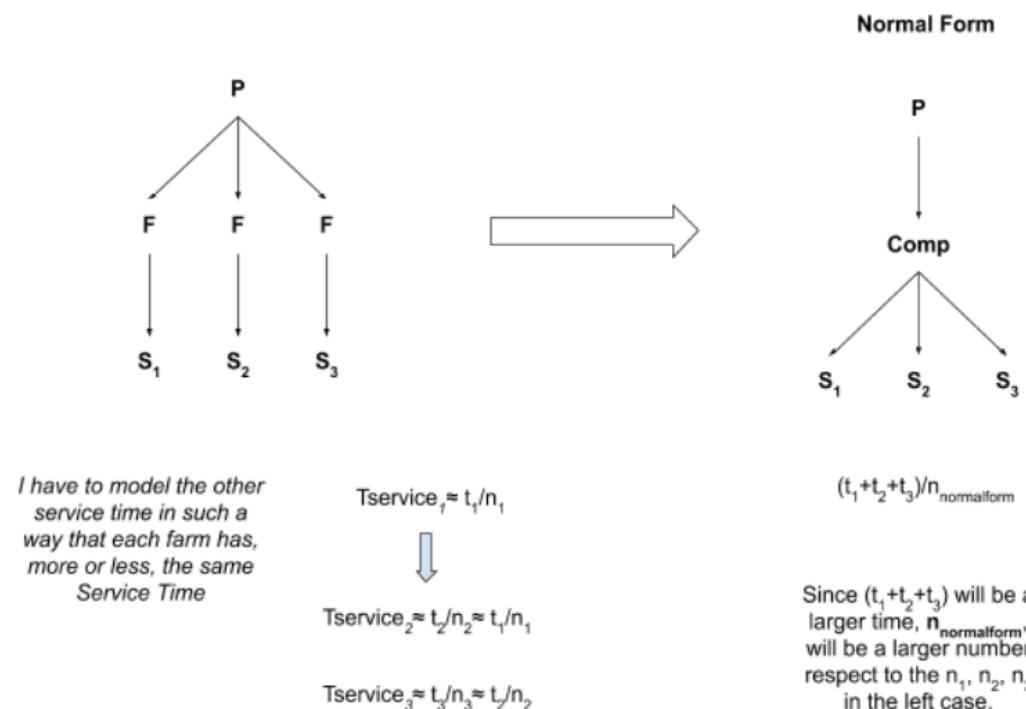
Consideriamo quello che succede quando creiamo un thread, vediamo il caso dello Xeon Phi, il tempo aumenta a poco a poco con il numero di thread perchè accediamo più strutture dati, modifichiamo più valori e quindi l'overhead per il singolo thread aumenta un po', non tantissimo.

Se il tempo che tu spendi in ognuno degli stage è significativo allora come conseguenza facendo la somma dei vari tempi la proporzione rimarrà più o meno la stessa.

Quindi attualmente i ragionamenti che facciamo sulla Normal form sono gli stessi che facciamo per la Map Fusion Rule.

Possiamo dire che utilizzando il performance model riusciamo a stabilire se ci sono delle composizioni di pattern che si comportano meglio di altre senza ancora aver scritto il codice, quindi per ora queste analisi sono solamente ad un livello di astrazione che mi consentono di eseguire un'analisi efficace.

S_i = sequential node
 n_i = worker for the i -th farm



Ci sono due cose da considerare a questo livello:

- L'overhead, che può arrivare, ad esempio, anche dalla lettura dal disco. Si deve cercare di abbassare questo tempo
- Serial Fraction: si tratta della parte di codice non parallelizzabile, vogliamo aumentare il numero di dati per renderla costante perché altrimenti più si parallelizza e più questa parte sequenziale diventa importante come tempo. Questa serial fraction va analizzata anche dal punto di vista della Amdahl Law e del Work/Span Model.

Targeting Hardware

Abbiamo in mente una certa soluzione e una macchina su cui vogliamo eseguire il nostro algoritmo parallelo.

Il ragionamento sull'hardware è fondamentale perché la soluzione che troviamo potrebbe non essere adeguata per l'architettura che abbiamo a

disposizione, non vogliamo modificare l'algoritmo ma considerare le varie implementazioni e capire quella che si adatta di più all'architettura.

Per fare questa fase di targeting hardware possiamo considerare vari parametri, abbiamo ad esempio due alternative per l'albero e vogliamo capire il migliore:

- Per prima cosa possiamo prendere il primo albero e consideriamo i meccanismi di comunicazione tra le attività concorrenti.
Questo costo, anche quando non è trascurabile, può essere mascherato con alcune strategie come ad esempio il double o il triple buffering, in presenza di un hardware che è capace di eseguire le comunicazioni indipendentemente dal processore.
Se ho una pipeline e devo far comunicare i due stage quello che posso fare è usare il triple buffering sopra agli stage della pipeline per velocizzare la comunicazione, questo funziona se ho l'hardware adatto ma anche se considero che la computazione prende un certo tempo che deve essere lungo rispetto al tempo della comunicazione.
- La maggior parte degli esempi li consideriamo in una shared architecture dove usiamo socket e pipe per la comunicazione

Se invece volessimo considerare un Cluster di Workstation (COW)?

Il COW è un insieme di macchine multicore che sono connesse attraverso una rete, la comunicazione avviene in pochi millisecondi e quindi qua il tempo diventa importante e dobbiamo cercare in ogni modo di nascondere il tempo della comunicazione.

Dopo questa fase iniziale di analisi possiamo spostarci alla scrittura del codice della nostra applicazione parallela.

Distinguiamo tre fasi nella scrittura di un'applicazione parallela:

- **Fase di prototipo dell'applicazione:** creiamo un'applicazione che più o meno funziona ma c'è comunque qualcosa che deve essere rifinito per ottenere un funzionamento migliore.
- **Fase di Refine:** quando eseguiamo la fase di refine, facciamo più o meno le stesse cose che abbiamo fatto in precedenza solamente che ora dobbiamo considerare e valutare meglio l'overhead per capire se possiamo migliorare e ottimizzare qualcosa.
Ad esempio consideriamo l'utilizzo delle cache, vogliamo ottenere la proprietà di località della cache. Se succede che una linea di cache viene acceduta da due thread differenti avremo il problema del false sharing. Per evitare questo problema è meglio riempire la linea della cache con dati inutili, questo mi spreca spazio ma riusciamo ad ottenere performance migliori. Ad esempio se voglio fare la map su una matrice vogliamo poter utilizzare un algoritmo che lavora preservando la locality.
- **Tuning dell'applicazione:** è un'attività minore quando si sviluppa un'applicazione parallela, si occupa di fare delle ottimizzazioni a livello di OS, sono comunque ottimizzazioni molto particolari, le ottimizzazioni migliori le abbiamo nelle prime due fasi.

Se dobbiamo implementare una soluzione parallela per un certo problema, dovremo seguire più o meno i passaggi visti sopra.

All'inizio avremo una versione sequenziale del codice che però dobbiamo scrivere noi perchè di base non ce l'abbiamo.

Il codice sequenziale anche se dobbiamo poi scrivere un'applicazione parallela sarà comunque necessario perchè poi dovremo confrontarlo con il codice parallelo.

Quando scriviamo il codice sequenziale è anche buona norma cercare di fare tante astrazioni del funzionamento definendo delle funzioni che poi torneranno utili in seguito.

Lezione 25

Hadoop

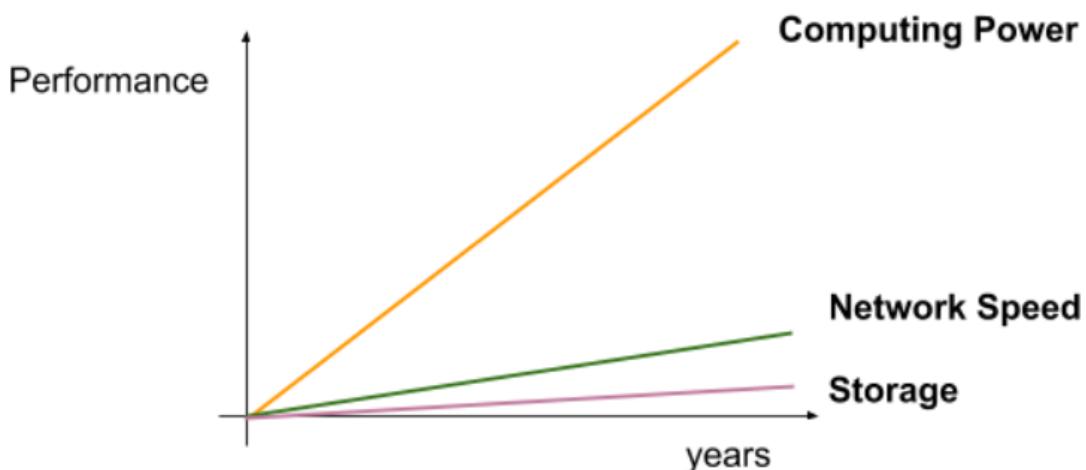
Hadoop è un framework molto comune, è la versione gratuita della Map Reduce di Google ed è sviluppato da Apache.

Questo framework ci permette di lavorare sui Big Data, questi dati sono definiti da tre attributi:

- Volume: abbiamo una grande quantità di dati
- Velocity: i dati aumentano molto velocemente
- Variety: i dati possono essere molto differenti tra di loro, possono essere di differenti tipi, strutturati o no

Quando lavoriamo sui big data, stiamo lavorando su dati che possono aumentare nel tempo e noi vogliamo per esempio fare una computazione che mi permette di trovare delle correlazioni tra i dati.

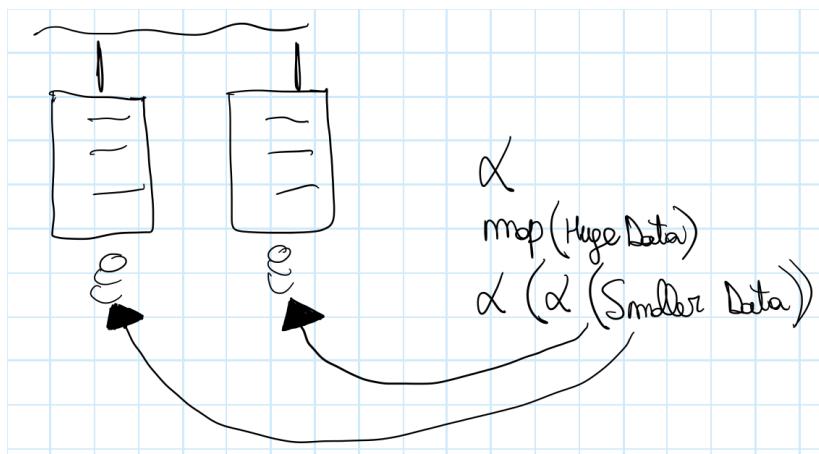
Il Volume di questi dati quindi è la caratteristica fondamentale, in breve tempo possiamo trovarci ad avere grandi quantità di dati da processare. Avendo molti dati si preferisce suddividerli in tanti SSD o in COW, negli ultimi anni abbiamo avuto un aumento della potenza computazionale, un aumento della velocità della rete ma rimangono problemi con l'accesso ai dati nel disco, quindi abbiamo che la vera bottleneck è l'accesso al disco.



Quindi, per evitare di avere una bottleneck per l'accesso ai dati abbiamo la possibilità di utilizzare Cluster di reti di processing elements dove internamente abbiamo le classiche macchine con shared memory e accelerators:

- Possiamo distribuire i dati sui dischi delle varie macchine in modo da poter parallelizzare l'accesso ai dati e quindi accedere in contemporanea a tanti più dati.
- Usiamo la rete per unire poi i dati che devono essere processati su un'unica macchina.

Abbiamo quindi sistemi differenti che sono interconnessi con una rete molto veloce, se facciamo una map su questo grande set di dati, quello che posso fare è astrarre un po' di più e dire che in realtà la map è una map fatta su una map di dati più piccoli (i dati locali alla macchina). Poi dopo dovrò riunire tutti i dati per avere il risultato tutto insieme.



Quindi questo sistema è utile se devo calcolare qualcosa come la map-reduce.

La reduce che viene applicata dopo la map può essere fatta localmente in ogni nodo in cui facciamo la map, quello che trasmettiamo agli altri nodi non è qualcosa che ha la stessa dimensione dell'input data set ma è qualcosa che viene ridotto perché facciamo già una reduce locale.

Quindi potrei avere vari TB di dati ma poi se per esempio ho 3000 parole diverse in questi dati, posso considerare che dovrò inviare solamente 3000 tuple <stringa, intero> e non tutti i dati iniziali.

La Map Reduce è molto utile, possiamo usarla per risolvere vari problemi ma non tutti i problemi possono essere risolti con la map reduce, non è una sorta di panacea.

Inizialmente era stata pensata per eseguire il rank dei risultati delle query su Google, questa è un'applicazione perfetta della Map Reduce. La cosa importante è che la computazione tra la prima fase e la seconda fase della Map Reduce deve essere indipendente.

La Map Reduce può essere implementata facilmente anche su un singolo computer, il problema in questo caso è che non posso lavorare su grandi quantità di dati per le limitazioni dello storage system.

Ci sono delle caratteristiche che rendono Hadoop ottimo per eseguire la Map Reduce su grandi quantità di dati, non è una semplice implementazione di questo sistema ma è un ecosistema di componenti.

Il programmatore che lavora con Hadoop deve fornire una funzione che prende dati di un certo tipo *Tin* e restituisce tuple del tipo *<Key, Tout>*. Questa funzione deve avere al suo interno la possibilità di prendere i dati in input e trasformarli in tuple, quindi sommando i *Tout* restituendoli come risultati.

Queste computazioni devono essere svolte interagendo con i seguenti componenti:

- HDFS (Hierarchical Distributed File System): è un file system dove dobbiamo caricare dei dati all'inizio dal disco da altrove poi questo implementa alcune strategie che usano:
 - Repliche dei dati
 - Distribuzione di dati

Se memorizziamo un file nel parallel file system quello che succede è che chunks di dati verranno distribuiti su dischi che appartengono a differenti macchine. Inoltre il file system si occupa di creare un numero di repliche dei chunks in modo che se succede qualcosa al sistema sei sempre in grado di recuperare i dati.

La distribuzione è per avere una migliore performance mentre la

replication è per la fault tolerant, questo fault tolerant è importante perchè le varie macchine hanno dei cicli di vita che sono disallineati tra di loro quindi potrebbe accadere che in alcuni casi dobbiamo fare delle modifiche su una macchina sostituendo dei pezzi ma vogliamo comunque avere dei meccanismi che mi permettono di recuperare i dati.

Le repliche mi permettono anche di avere una maggiore velocità perchè posso leggere una parte del chunk da una macchina e un'altra parte da un'altra macchina.

- Control Layer: in questi sistemi devo essere in grado di dare un nome ai vari nodi per essere in grado di identificare dove i file sono memorizzati. Quindi ci sono vari meccanismi che possiamo usare per gestire i dati e rappresentare i dati indipendentemente dalla macchina dove sono stati salvati.

Hadoop implementa la Map Reduce ma sopra ad Hadoop ci sono altri tools che possono essere utili:

- Flume: è un sistema per ottimizzare sequenze di Map Reduce calcolate sugli stessi dati. Ci sono varie ottimizzazioni, una di questa è la Map Fusion, quando devo calcolare due map sui dati, accedo una prima volta e calcolo la map, poi calcolare la seconda è gratis quindi la faccio direttamente perchè la parte che costa di più è l'accesso ai dati.
Ci sono anche altre ottimizzazioni simili, Flume è stato sviluppato sopra ad Hadoop e mi permette anche di avere esecuzioni più veloci o più esecuzioni della map reduce in contemporanea
- Altri sistemi mi permettono di lavorare su grandi grafi con miliardi di nodi. Questi sistemi sono costruiti sopra ad Hadoop, non usano la map reduce ma alcuni algoritmi su grafi ma comunque rimane l'idea che i dati vengono memorizzati su file che sono distribuiti in una rete con repliche e chunks.

Hadoop lo possiamo anche testare su una macchina singola, quello che possiamo fare è scaricare l'eseguibile e provarlo.

Abbiamo bisogno di alcuni prerequisiti per provare Hadoop:

- Serve un meccanismo per accedere ad un nodo remoto (o un nodo “fake” sulla nostra macchina, avremo insieme il client, installato di default e il server, che deve essere installato a mano). Questo meccanismo è SSH.
- Poi serve un meccanismo per evitare che le chiamate con SSH abbiano bisogno di una password, quindi usiamo un sistema con public key e privata nel .ssh.
- Bisogna installare una versione abbastanza recente della JDK, serve una versione di Java che sia superiore alla 7

Poi avviamo Hadoop e possiamo iniziare a copiare file all'interno dell'Hadoop File system sperimentando con il codice.

Il codice che possiamo eseguire è un codice scritto in C++ oppure in Java.

Nel codice dobbiamo definire una map function che prende uno degli item presente nei dati (ad esempio una stringa) e per ognuno di questi emette una tuple.

Poi va definita anche una funzione reduce, quello che devo fornire è una funzione che sia in grado di calcolare la somma di valori.

L'ultima cosa che va fornita nel main è una serie di configurazioni che mi dicono dove sono i file, quali classi vogliamo utilizzare per la reduce e per la map. Va dato un contesto che permetta al sistema di trasformare map e reduce in un'applicazione in grado di essere eseguita.

Se usiamo C++ possiamo scrivere programmi come in Java, la struttura sarà più o meno uguale ma serve una estensione chiamata pipes, quello che facciamo è implementare anche qua una map e una reduce, la map è dentro una classe che deve estenderne un'altra. La map prende un contesto, da questo la stringa e poi emette una tuple, come prima. È più semplice scriverlo in C++ che in Java.

Quindi, noi forniamo le funzioni basilari, poi a tutto il resto ci pensa Hadoop, quindi non dobbiamo pensare a dividere i dati in chunk oppure a condividerli nella rete.

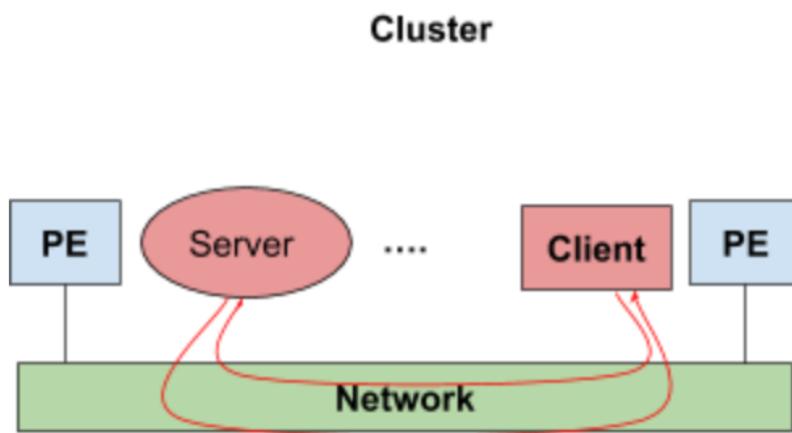
Lezione 26

Ancora su Hadoop

Immaginiamo di avere un cluster di server, abbiamo varie processing element che sono connesse tramite rete.



La maggior parte di questi sistemi utilizzano lo stack TCP/IP quindi è necessario che le varie processing elements conoscano l'ip del nodo con cui vogliono comunicare. Tutti i vari PE devono avere un server in grado di ricevere richieste, queste sono state inviate o in broadcast o in multicast. C'è una fase di configurazione in cui viene deciso un well known address (WKA) che poi verrà utilizzato successivamente da nuovi nodi che per esempio vogliono entrare a far parte del cluster.



Alla fine di questo processo i server conoscono l'ip della macchina che è stata aggiunta all'interno del cluster e possono comunicare con una comunicazione point to point (e non più broadcast) le informazioni direttamente al nuovo nodo che è stato aggiunto nel cluster.

Ci sono tanti tool per fare questo in automatico, Hadoop utilizza il suo tool, questo componente è qualcosa che poi è stato anche utilizzato in altri progetti (Spark). Questi componenti solitamente vengono riutilizzati anche in altri progetti, non solo in quello per cui sono inizialmente pensati, ad esempio abbiamo HDFS che viene utilizzato anche in GraphX, la libreria usata per gestire i grafi sviluppata da Spark.

Tra una libreria e l'altra cambia solamente il modo in cui vengono poi esposte all'utente e quello per cui effettivamente vengono utilizzate.

Abbiamo già visto Hadoop nella precedente lezione, è una implementazione distribuita della Map Reduce.

Abbiamo tanti dati, che aumentano in dimensione nel corso del tempo e vogliamo lavorare con questi dati.

Oltre ad Hadoop vedremo anche altre librerie:

- Storm e Spark: si differenziano da Hadoop perchè qua abbiamo una implementazione distribuita dello stream processing. Non abbiamo direttamente tutti i dati su cui vogliamo lavorare, i dati arrivano da uno stream e questo può anche essere infinito. Storm è un sistema vecchio mentre Spark è più recente.
- GraphX: permette di fare delle computazioni parallele sui grafi, fa svolgere delle analisi tipiche dei grafi in modo parallelo.

Storm:

Storm è stato il primo progetto rilasciato che ha permesso di eseguire computazioni in parallelo su uno stream.

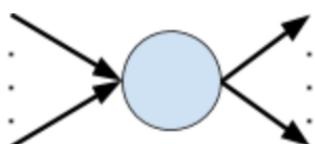
L'ultima versione è stata rilasciata nel 2018, questo tool si basa su alcuni concetti fondamentali:

- Spouts: sono i nodi che creano lo stream, quindi sono nodi che non hanno nulla in input e poi invece mandano in output uno

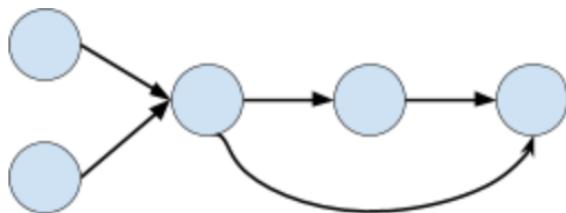
stream.



- Bolts: sono nodi che possono processare un qualsiasi numero di input stream producendo un qualsiasi numero di output stream. I bolt possono essere una farm ad esempio, tutto quello che prende un input uno stream e produce uno stream in output è un bolt.



- Topologies: Sono un insieme di nodi Spouts e Bolts che servono per processare i dati prodotti dagli spouts.



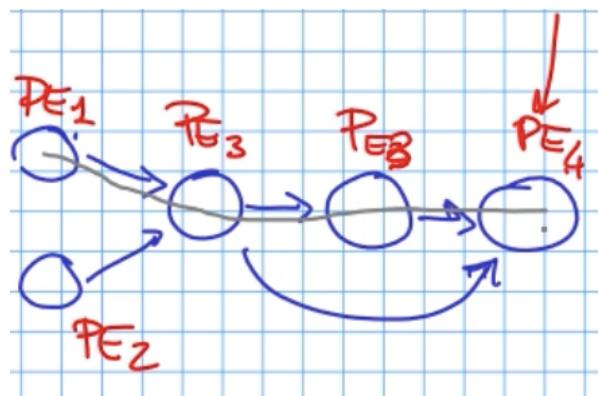
Storm funziona in modo tale da garantirmi:

- Alte performance: perchè il sistema usato da Storm permette di processare grandi quantità di dati nel sistema, normalmente possiamo processare 1 milione di tuples al secondo inserite in uno stream. Non è un numero assurdo ma è un numero decente.
- Fault tolerance: in particolare indichiamo due cose con fault tolerance:
 - Se uccidiamo un nodo nello storm, il sistema è in grado di tornare a funzionare senza perdere in alcun modo i dati su cui stavo lavorando. Questo lo possiamo fare perchè usiamo dei nodi stateless, replicati che garantiscono che i dati che

stiamo processando su un nodo potranno essere riprodotti nuovamente su un altro nodo nel caso di fallimenti.

Ogni nodo nella rete ha un ID e quindi noi possiamo sempre capire cosa è successo al nodo con quello specifico ID, dall'inizio alla fine del processo. Se qualcosa capita nel mezzo possiamo sempre recuperare quel nodo con quello specifico ID.

- Tutto questo permette di implementare un processo chiamato at least one che mi dice che ogni tuple che viene inserita all'interno del sistema, per almeno una volta farà tutto il tragitto dalla PE1 alla PE4.



Gli spout supportano stream di differente tipo, socket o altro tipo, possono essere definiti usando qualsiasi linguaggio in pratica e la comunicazione avviene con un protocollo JSON-based.

Inoltre è facile iniziare ad usare Storm perchè basta avere installato Java o Python e qualche tool di Hadoop.

Contro di usare Storm:

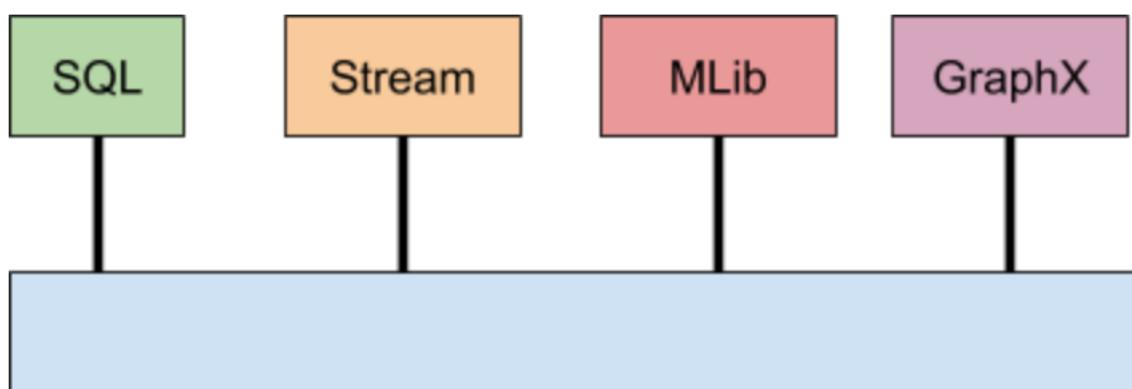
Storm non è adatto se vogliamo fare le cose semplici, è ottimo se vogliamo implementare cose più complesse. Se vogliamo implementare semplicemente degli stream processor dobbiamo fare tutto quanto a mano e questo risulta scomodo.

Spark

Anche Spark è sviluppato da Apache, il punto chiave qua è che Spark è più complesso perchè è organizzato in 4 moduli che sono eseguiti su un base system che ha molte cose in comune con Hadoop, ad esempio usa HDFS.

Queste 4 parti sono:

- Spark SQL: un modo per implementare database distribuiti garantendo la possibilità di accedere a questo database usando il classico SQL. Le tabelle sono distribuite e replicate, è garantito che non perdiamo dati ma comunque usano SQL.
- Stream: è quello più usato, garantisce delle funzionalità simili a quelle offerte da Storm con una interfaccia più moderna e più funzionalità perchè alcune delle cose che abbiamo messo nei contro di Stom vengono risolte qua.
Se voglio applicare, ad esempio, una filter, invece di scrivere una bolt potrò usare qualcosa che, fornita la funzione, mi costruisce l'intero processo che mappa la funzione su tutte le coppie dello stream.
- MLib: la libreria del machine learning fornita da Spark
- GraphX: libreria di Spark per processare grafi di grandi dimensioni. Distribuisce la computazione su tante macchine.



Spark può essere provato anche su una singola macchina ma non hai le stesse performance che hai con le architetture distribuite.

Questo viene fornito insieme a HDFS e Yarn, accetta varie lingue (Scala, Java, Python, R).

Spark lo possiamo eseguire su qualsiasi piattaforma, in particolare su quelle che supportano Hadoop ma anche su una architettura Cloud, ad esempio quelle che supportano Kubernetes.

Stream

Per quanto riguarda lo Stream, Spark vuole funzionare come Storm, vogliamo qualcosa che processi lo stream senza perdere i dati e che sia scalabile in modo decente.

Spark aggiunge qualcosa di strano, gli elementi dello stream vengono interpretati come coppie ma qua vengono anche salvati in un database, quindi lo stream viene interpretato come una tabella di un database che può solamente crescere di dimensione.

Quello che possiamo chiedere come stream processor sono tre differenti tipi di risultati che sono:

- **Complete:** Ogni elemento nello stream è come se ricreasse l'intero output, possiamo pensare al nuovo elemento che entra nello stream, una nuova riga viene inserita nella tabella, lo stream processor viene calcolato sulla tabella e poi il risultato finale viene calcolato di nuovo da 0.
- **Append:** per ogni elemento possiamo produrre solamente la differenza di risultato rispetto al precedente risultato. Produciamo un nuovo risultato che viene aggiunto al vecchio
- **Update:** il nuovo elemento dello stream genera cambiamenti nel risultato, viene prodotto un delta rispetto al precedente risultato. Immaginiamo di avere degli elementi X1, X2, X3, X4, dopo ognuno di questi viene prodotto un certo risultato che può o essere aggiunto ai precedenti, oppure può sostituire il precedente oppure modificare il precedente. In generale quello che viene fatto è guardare una piccola sezione dei dati dell'input stream e poi viene generato un risultato che riguarda quella specifica piccola parte di dati. Queste parti di dati sono chiamate MicroBatch.

Quando la computazione viene terminata e si deve emettere l'output, abbiamo vari tipi di output possibili e dobbiamo dichiarare la modalità che vogliamo per l'output.

Questo output è prodotto in modo che con questa modalità di funzionamento è garantito un response time di circa 100 millisecondi ovvero 10 elementi per secondo, non è eccessivamente veloce, per garantire velocità di output hanno diminuito il tipo di operazioni che è possibile svolgere su uno stream.

Cosa possiamo fare sullo stream? Possiamo definire varie operazioni che sono in varie classi:

- **Gli stream** possono essere forniti con varie interfacce, ci sono varie possibili origini per lo stream, un caso particolare è quello relativo allo stream che arriva da un file.
- C'è un modo per definire l'**output mode** (append, complete, update)
- Possiamo definire un **windows**, vuol dire che possiamo definire quali sono gli item che voglio considerare per essere processati. Per esempio posso dire che voglio processare solamente quelli arrivati negli ultimi 5 minuti. Può essere utile e sicuro per il mio algoritmo considerare delle windows perché se guardo una finestra abbastanza lunga posso prendere decisioni migliori.
- **Operazioni:** includono le cose tipiche che vengono fatte in uno stream, ad esempio la selezione (filter), l'aggregazione (reduce) e la projection (map).

Ci sono anche operazioni più complesse che arrivano dal mondo dei database, stream joints ad esempio serve per prendere vari stream differenti per poi unirli in un unico stream. La stream deduplication invece cerca duplicati e poi li elimina.

Tutto questo avviene in un sistema in cui le tuple hanno un ID univoco e sono garantite di essere processate con una semantica che è simile a

quella di at least one, quindi è garantito che almeno una volta vengono processati.

Questo sistema garantisce performance e garantisce la possibilità di implementare il tutto in modo distribuito.

Tutte queste cose sono sviluppate con il modello di programmazione “Fluent”. Fluent è un pattern di programmazione che viene utilizzato nella programmazione Object oriented. Si tratta di un metodo che permette di eseguire più method calls sull'oggetto che è stato creato già da una precedente method call. Si tratta di un modello che è stato usato da Java 8 con gli stream di Java.

In java dichiariamo uno stream, poi chiamiamo un certo metodo che mi fa fare un certo lavoro su quello stream. Il metodo in se per se è fatto in modo che implementa la semantica del metodo ma restituisce un oggetto che è qualcosa che accetta il prossimo metodo che dobbiamo calcolare.

Quindi in Spark dobbiamo scrivere un programma in cui dichiariamo uno stream e poi abbiamo un metodo che ad esempio prende lo stream da un socket, poi chiamo su questo un metodo, prendo un risultato finale in uno stream e posso poi chiamare un altro metodo e così via.

Alla fine possiamo avere qualcosa che sembra un programma sequenziale con vari step separati da punti in cui chiamiamo i vari metodi che vengono eseguiti sullo stream.

Possiamo anche scrivere queste cose in modo che siano molto rappresentative di quello che abbiamo in mente, è molto differente rispetto a quello che avevamo in Storm dove dovevamo pensare ad una topology per poter fare qualcosa.

GraphX

È l'altro componente di Spark, qua vogliamo processare grafi.

Abbiamo una serie di operazioni per lavorare sul grafo, si tratta di algoritmi per grafi. I grafi qua sono formati da nodi e archi ma entrambi possono avere dei label.

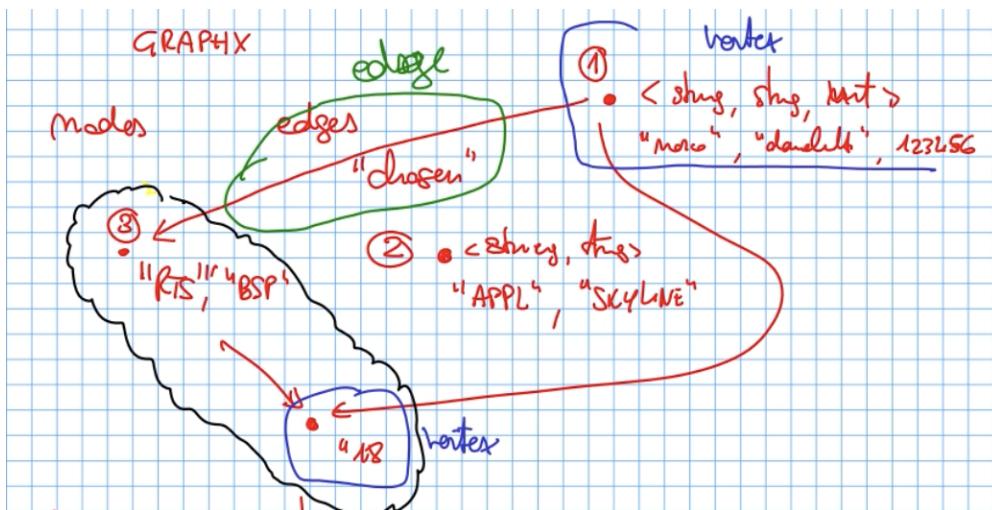
I nodi hanno ID univoci, ad ogni nodo sono associate delle tuple `<string, string, int>`, ad esempio al nodo 1 viene associata la tupla `<marco, danelutto, 1234>`.

Poi posso definire un altro tipo di nodo che però ha una tupla differente.
Poi posso prendere i vari nodi e collegarli con un arco.

Posso definire grafi molto grandi e il punto chiave è che qualsiasi cosa che ho dentro al grafo è una trasformazione funzionale del grafo, vuol dire che non viene modificato il grafo originale, ad esempio se aggiungo un nuovo nodo viene prodotto uno nuovo e l'originale non viene modificato.

Cosa posso fare sul grafo? Viene usato sempre il pattern Fluent.

- Posso accedere alle componenti del grafo. Posso accedere ai vertici del grafo oppure agli archi oppure alle triple del grafo. Ad esempio quello verde è un arco, quello blu un vertice e quello nero è una tripla.



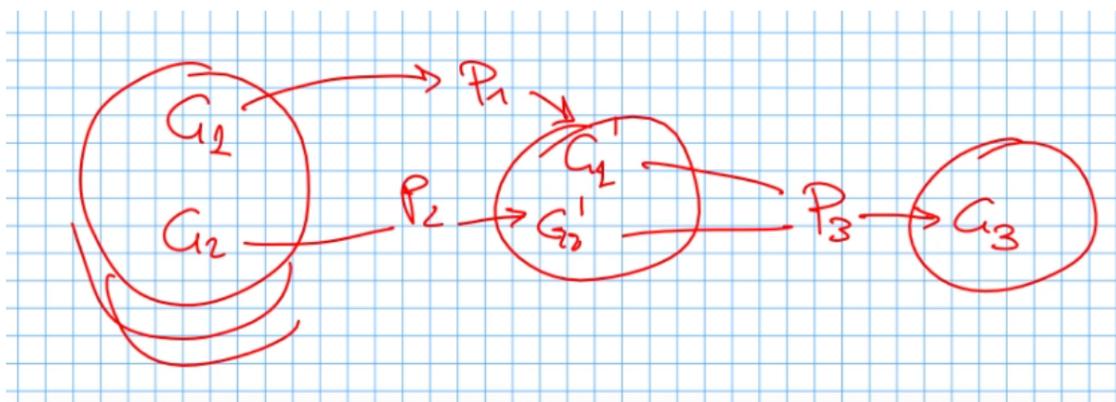
- Si possono **avere informazioni** sul grafo, ad esempio il numero di nodi e archi, l'average degree.
- Puoi **trasformare** il grafo, nella trasformazione abbiamo ad esempio una Map function su vertici, archi e triple che viene applicata e restituisce un nuovo grafo
- **Operatori strutturali:** ad esempio il reverse che prende un arco che ha una certa direzione e la modifica cambiandola nell'altra

direzione.

Poi abbiamo l'operatore subgraph che mi fa estrarre una parte del grafo in base ad alcuni parametri.

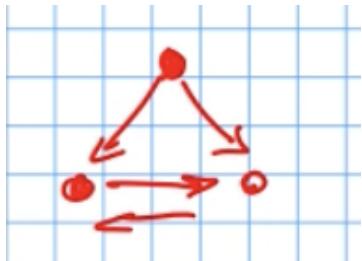
Possiamo avere l'operatore masks che permette di scegliere parti del grafo che rispecchiano le parti di un altro grafo che è dato.

- **Join operator:** prende due grafi e ne produce uno nuovo
- **Aggregate message operator:** per ogni componente del grafo definiamo cosa dovrebbe essere inviato da quella componente lungo gli archi che escono da quel nodo e un'altra funzione che dice che se il messaggio che arriva in quell'arco è appartenente ad un certo set avrai un effetto in quel nodo locale. È una sorta di map reduce, fai la map con la funzione send e la reduce con i messaggi che arrivano sul nodo.
- **Caching and uncaching method:** gli archi e i nodi del grafo sono distribuiti in un file system distribuito in modo permanente. Quando processiamo un grafo, ad esempio G_1 o G_2 produciamo un secondo grafo che è G_1' (o G_2'). Se non dico niente i dati relativi ai grafi vengono letti dal disco, modificati e poi riscritti sul disco e questo è un po' un problema perché ho varie letture e scritture. Possiamo risolvere il problema se andiamo a dire a GraphX di memorizzare i risultati intermedi nella memoria principale.



- **Algoritmi per il page rank:** Andiamo nel grafo e cerchiamo di interpretare gli archi in input per capire quali nodi all'interno del grafo sono più importanti. Questo è qualcosa che non deve essere programmato a mano, chiamiamo semplicemente la funzione.

- Anche per la ricerca delle componenti connesse è presente un algoritmo che mi fa cercare cluster all'interno del grafo
- Un altro algoritmo è il triangle count, conta i nodi che sono organizzati con questo pattern a triangolo, è importante per trovare particolari relazioni tra i vari nodi.



Questo è tutto quello che viene offerto da Spark, ci vuole un po' di tempo per capire come funzionano queste cose ma comunque sono utili. Nel background ci permettono di memorizzare e distribuire i dati, garantiscono che le tuple siano processate con la proprietà del fault tolerance e così via. Abbiamo un set di cose indipendenti che quindi sono molto utili.

Lezione 27

Vectorization

La vettorizzazione è un caso speciale di parallelizzazione automatica in cui un programma viene convertito da una implementazione scalare a un'implementazione vettorizzata.

Se abbiamo qualcosa che è data parallel e siamo su un processore moderno abbiamo sicuramente delle vector unit in grado di fare un gran numero di applicazioni. Per usare queste vector unit dobbiamo fare un po' di attenzione quando scriviamo il codice.

La teoria della vettorizzazione nasce intorno agli anni 70 ma ancora si sta evolvendo.

I principi per la vettorizzazione, in generale sono alcune proprietà che devono essere rispettate per avere un programma che sia vettorizzato correttamente dal compilatore:

- **Countable:** i loop che devono essere vettorizzati devono essere "contabili". Questo vuol dire che devono esserci dei confini definiti, ad esempio devo avere un limite quando faccio un certo while.
- **Single Entry/Exit point:** il blocco di codice che vogliamo usare come unità per la vettorizzazione deve avere un single exit e single entry point. Ad esempio per la vettorizzazione non va bene avere un loop che ha al suo interno un break.
- **Non diverging code:** vogliamo un codice che non diverge, se abbiamo un jump che si comporta diversamente a seconda del risultato di un IF allora non va bene e la vettorizzazione non funziona.
- **Innermost Loop:** quando ci capita di avere vari loop, uno dentro l'altro, il più interno è quello che viene vettorizzato.
- **No function calls:** Non abbiamo function calls perchè questo vorrebbe dire fare un salto altrove, se però tutti saltano alla stessa funzione può andare bene. Se facciamo il jump verso una funzione che non è vettorizzabile arriviamo al problema che i dati sono nel

vector register e la libreria invece pensa ad un “general purpose register”.

- **No dipendenze:** non dobbiamo avere dipendenze nelle parti differenti del loop perchè le operazioni sono svolte in parallelo dalle vector unit e quindi se abbiamo delle dipendenze, il sistema non funziona nel modo corretto.

Ci sono delle situazioni in cui c’è da stare attenti:

- Alignment: è legato agli accessi in memoria e in particolare alla forma degli accessi in memoria.

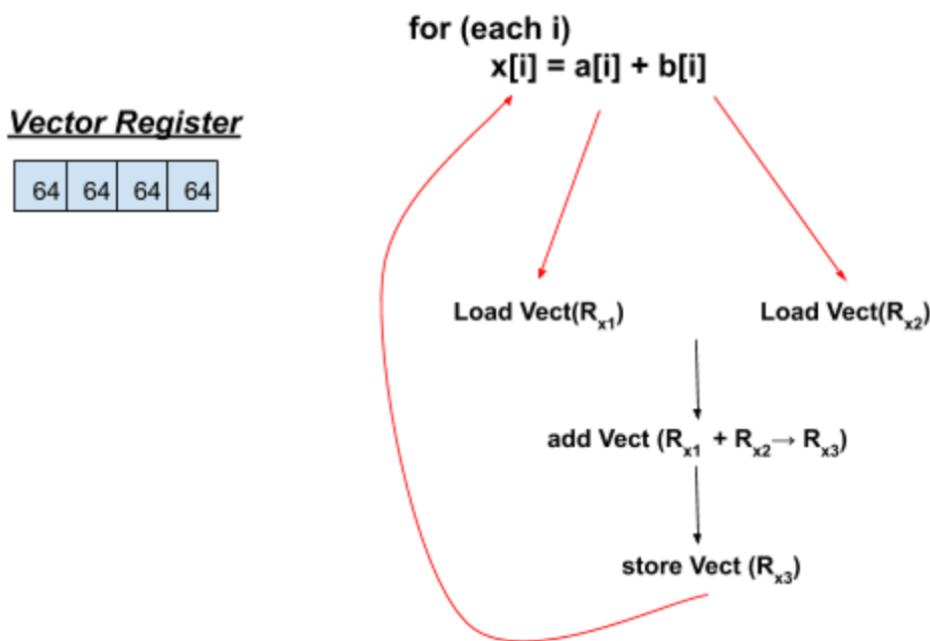
Ad esempio se voglio parallelizzare il loop:

```
for (each i)
    x[i] = a[i] + b[i]
```

Questa è una computazione in cui cerco di caricare aree contigue di memoria di x per memorizzare e di a e di b per leggere.

Come viene vettorizzato? Se lo faccio in un vector mode supponiamo di avere un vettore di registri di 64 bit. Quello che faccio è compilare questo codice in:

- un load vector che mette in un vettore la A
- Un secondo load vector che mette B in un secondo vettore
- Un add vector che serve per memorizzare la somma dei dati contenuti nei due registri.
- Uno store vector che prende quello che ho nell’ultimo vettore in modo che sia memorizzato in X.



Quindi, prendo un'area contigua di memoria e poi salvo su un unico vector register.

Prendiamo invece il caso in cui da una iterazione all'altra devo vedere i valori diversi del vettore saltando da una posizione ad un'altra.

Nel registro di B devo salvare in ogni iterazione qualcosa che non è continuo ma ogni volta salto una posizione, devo prendere solo gli elementi pari del vettore.

for (each i) $x[i] = a[i] + b[i+2]$

Quindi se uso il load vector che prende un valore base e cerca di capire quante parole servono per riempire quel registro non funzionerà perchè devo prendere la parola 0 poi la 2 poi la 4 ma devo saltare quelle che stanno nel mezzo. Solitamente è un problema ed è simile al problema che abbiamo quando consideriamo la cache e carichiamo dei dati saltando però alcuni valori che quindi sono inutilizzati. In questo secondo caso però il compilatore riesce a fare qualcosa.

C'è un altro grande problema da considerare con il parallel data access. Se accediamo un vettore in modo indiretto possiamo avere dei problemi, ad esempio:

for (each i)
a[i] = b[c[i]]

Qua idealmente dovrei prendere chunk contigui di C ma qua carica un indirizzo che in realtà viene usato per caricare un altro indirizzo.

Accessi non contigui in memoria vogliono dire che bisogna stare attenti agli strides e agli indirections.

Dobbiamo anche stare attenti alle dipendenze tra i dati, tra tutte le dipendenze:

- Read after write, questa è la più pericolosa perchè se l'iterazione i scrive la posizione che poi è letta alla iterazione $i+1$, queste due iterazioni non possono essere eseguite in due parti diverse dello stesso vector register
- write after read
- write after write

Se ad esempio provo a parallelizzare uno stencil, è sbagliato assegnare tutto il risultato a $x[i]$ e quindi posso risolvere assegnandolo alla y :

for (int i = 1; i < n-1; i++)
y[i] = (x[i] + x[i-1] + x[i+1]) / 3

Un altro problema, simile a questo, è il problema dell'alignment: La load vector operation funziona su boundaries relativi alla quantità di dati che dobbiamo memorizzare nei vettori di registri.

La load e la store funzionano meglio se, dato un register di 256 bit, ciascuna delle celle ha una dimensione che è divisore di 256.

Oltre al problema dell'alignment c'è anche il problema dell'aliasing, supponiamo di voler scrivere una procedura come la memcp. Quello che posso fare è scrivere una procedura che prende come parametri due puntatori (origine e destinazione) e la dimensione dei dati che vogliamo trasferire. Vogliamo copiare il primo puntatore nel secondo puntatore (incrementati entrambi).

```
memcpy(T* p1, T* p2, size){
    for (int i =0; 1; i < size; i++)
        *(p1)++ = *(p2)++;
}
```

Questo loop è ottimo per la vettorizzazione perchè prendo zone di memoria che sono vicine tra loro, dato che però stiamo lavorando con dei puntatori il compilatore non è capace di capire se ci sono degli overlap tra i due puntatori. Se c'è un overlap può essere un problema relativo alle dipendenze.

In questo caso non c'è niente da fare, il compilatore mi segnala un errore e l'unico modo per risolvere è aggiungere un IF in più che mi controlla che i due puntatori non si sovrappongano.

Se garantiamo che non ci sono overlap possiamo dire al compilatore che non abbiamo overlap, lo possiamo fare aggiungendo la keyword `__restrict__` che mi permette di dire che i due puntatori sono completamente differenti l'uno dall'altro.

```
memcpy(T* p1 __restrict__, T* p2 __restrict__ , size){
    for (int i =0; 1; i < size; i++)
        *(p1)++ = *(p2)++;
}
```

In alternativa possiamo fare così:

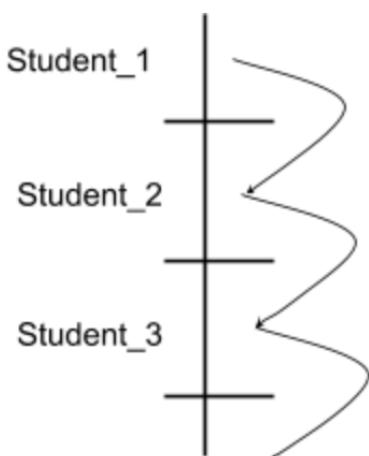
```
memcpy(T* p1, T* p, size){
    #pragma GCC ivdep
    for (int i =0; 1; i < size; i++)
        *(p1)++ = *(p2)++;
}
```

Questo pragma mi permette di dire che il loop ha delle iterazioni che sono indipendenti tra di loro.

È diverso dal restrict, perchè il restrict mi dice solamente che i due puntatori non si sovrappongono, così invece diciamo proprio che quello che c'è nel for è sicuro.

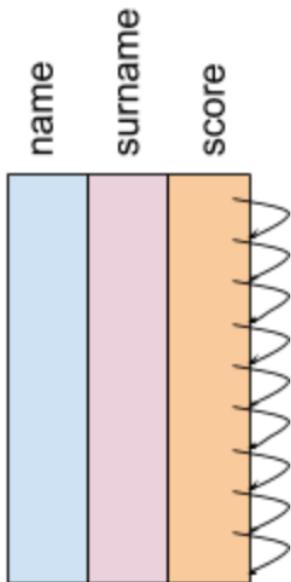
Vector of struct VS Struct of Vect

Se dobbiamo memorizzare un record per ogni studente possiamo pensare ogni record come un insieme di informazioni (nome, cognome, scores) e possiamo pensare ad un vettore di strutture di questo genere. In alternativa possiamo avere una struttura di vettori dove il primo vettore contiene lo score, il secondo il nome e il terzo il cognome. Se dobbiamo fare una data parallel computation sul vettore di strutture, quello che abbiamo in memoria è questo:



Quindi possiamo fare un for per scorrere tutti gli studenti ma questo vuol dire saltare in memoria da un punto all'altro. Tutte le posizioni sono vicine tra loro ma in realtà in memoria non sono vicine.

Se invece ho un vettore per ognuna delle informazioni posso scorrere il vettore corrispondente che è migliore rispetto all'altra strategia.

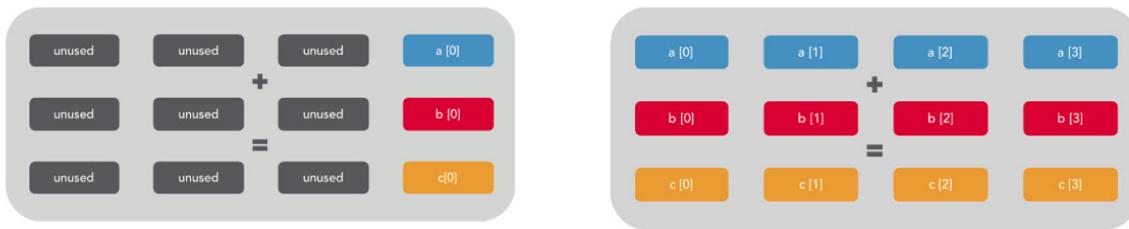


Data Sizes

Anche la dimensione dei dati è importante, se pensiamo a quanto spazio serve per rappresentare un dato, normalmente non vogliamo sprecare più spazio di quanto effettivamente ne serve per quel tipo di dato.

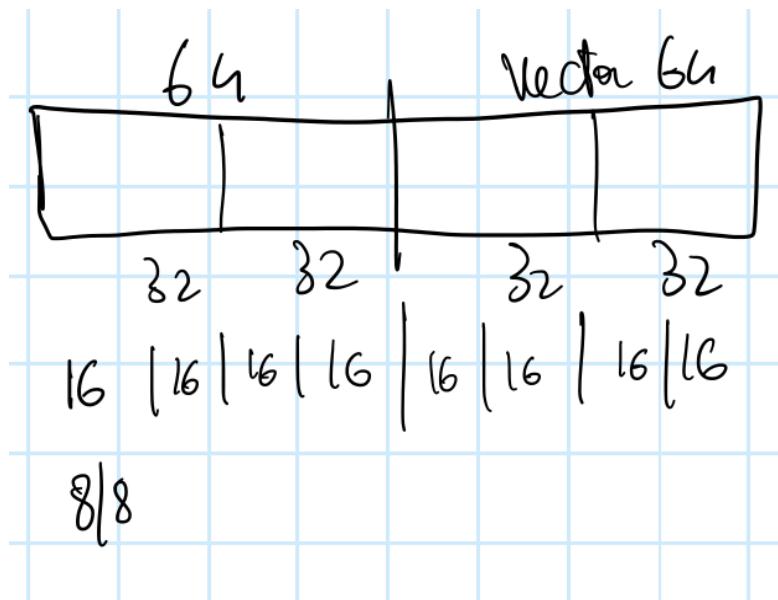
Vogliamo rappresentare ad esempio la matricola di uno studente, per esempio possiamo rappresentarlo con 64 bit in memoria.

Questo vuol dire che se vogliamo vettorizzare prendiamo il vettore da 256 bit e ci possiamo mettere al più 4 matricole ottenendo una speedup che è 4 volte il caso sequenziale.



Se invece rappresentiamo con 32 bit possiamo avere una speedup di 8 volte, quindi è importante prendere la dimensione corretta per i dati che vogliamo vettorizzare.

Quando usiamo un vettore di registri ci sono vari modi per fare lo split, solitamente dividiamo in blocchi da 64 chunks, 32 oppure 16 e spesso arriviamo anche a chunk di 8 bit.



C'è anche da considerare che alle vector unit abbiamo associate delle ALU che eseguono le operazioni più semplici e più comuni. Solitamente non sono supportate le operazioni su dati molto grandi oppure l'operazione modulo (abbiamo la divisione ma non il resto della divisione).

Bisogna anche fare attenzione al target che si sceglie per l'implementazione perchè dobbiamo fare attenzione all'istruzione set generato dal compilatore.

Per esempio l'ultima generazione di processori Intel supporta le vector unit AVX e AVX2 e quello che possiamo fare è dire al compilatore tramite il flag -mavx2 che il programma va compilato per quelle specifiche vector unit, se non specifichiamo nulla invece viene compilato con l'istruzione set base ovvero l'SSE.

Se facciamo attenzione a tutto questo possiamo utilizzare la vectorization con il flag -O3 oppure con il flag -fno-vectorize.

Se poi vogliamo vedere anche quali sono i flag che ho vettorizzato e quali no allora possiamo usare il flag -fopenmp-info-vec-all.

Esempio senza ottimizzazioni, qua non possiamo fare niente nei loop perchè in un caso abbiamo una funzione f che è definita in un altro file e in un altro caso invece abbiamo un pattern di accesso troppo complicato:

c

Esempio con un solo loop vettorizzato, qua il primo lo vettorizza perchè abbiamo la funzione f che è definita all'interno dello stesso file, il secondo non viene vettorizzato a causa del pattern di accesso troppo complicato. Il primo loop invece viene vettorizzato usando un vettore di 32 byte:

```
#include <iostream>
#include <utimer.cpp>

int f(int x){
    return (x % 2);
}

int main(int argc, char * argv[]){
```

```
const int n = 128;
int x[n];
int x = 0;
{
    utimer t("all");

    for(int i = 0; i < n; i++)
        x[i] = f(i);
    s = x[0];
    for(int i = 0; i < n; i++)
        s += x[i];
}
return 0;
}
```

Esempio in cui entrambi i loop vengono vettorizzati perché non incrementiamo la prima posizione del vettore ma salviamo il risultato in un'altra variabile :

```
#include <iostream>
#include <utimer.cpp>

int main(int argc, char * argv[]){
    const int n = 128;
    int x[n];
    int s = 0;
    {
        utimer t("all");

        for(int i = 0; i < n; i++)
            x[i] = i%2;
        s = x[0];
```

```
    for(int i=0; i<n; i++)
        s += x[i];
}
return 0;
}
```

Quando si eseguono questi tre con un vettore X di dimensioni abbastanza grandi possiamo notare come il tempo di esecuzione sia decisamente minore nella versione con i due loop vettorizzati.

Lezione 28

Abbiamo menzionato più volte OpenMP, questo è stato sviluppato da una community di ricercatori ma è anche usato da tante aziende e quindi anche le aziende che producono architetture parallele forniscono delle librerie per interagire con OpenMP.

Questi sono standard che sono nati nella ricerca, solitamente quello che accade è che le aziende poi prendono i risultati migliori per renderli commerciali, questo è accaduto ad esempio con Microsoft che ha acquistato quello che poi è diventato dos e poi windows o anche con C#.

Microsoft fornisce anche dei tool che possono essere utili per interagire con le architetture multi core:

- PPL: Parallel Pattern Library, questa libreria funziona sia con C++ che con C#
- TPL: Task Parallel Library, questa funziona solamente con C#

La maggior parte delle feature della PPL sono incluse anche nella TPL, nella TPL c'è anche una data flow library che non troviamo nella PPL.

Indipendentemente dal tipo di implementazione che vogliamo utilizzare (quella in C++ o quella in C#), queste due librerie garantiscono le seguenti features:

- **Automatic Vectorization**: ci sono pragma supportate
- **Task**: metodi per forkare l'esecuzione di processi che sono indipendenti tra loro
- **Concurrency Mechanism**: permettono l'esecuzione concorrente di processi

Poi ci sono delle feature che sono più legate con le librerie standard:

- **Parallel Algorithms**
- **Parallel Containers**: ad esempio alcune strutture dati, le possiamo usare in C++ e supportano anche la concorrenza, cosa

che non abbiamo con le classiche strutture dati che abbiamo in C++.

Ci sono anche altre feature che sono più strane:

- **Agents**: modello differente di programmazione, è una sorta di process model, ci sono attori che sono astrazioni che possono interagire con altri attori solamente inviando dati (quindi è un message passing model e non uno shared memory).
- **Data Flow**: modello in cui definiamo dei blocchi, come nel grafo data flow, abbiamo input e output, blocchi che sono sorgente per i dati, blocchi che trasformano i dati dividendo gli stream.

I meccanismi che sono usati per avere il controllo dei task sono simili a quelli che abbiamo per controllare gli agents o la computation, quindi rimane anche l'idea di avere dei token che possiamo utilizzare per fermare le attività di un thread o di un task di un blocco data flow. Quello che dobbiamo capire è quale di queste feature sono più adatte alle nostre esigenze.

Per quanto riguarda la vettorizzazione

Possiamo usare delle pragma per avere la vettorizzazione:

- `#pragma loop(int parallel(nw))`: questo è un pragma in cui indichiamo che il loop che segue il pragma dovrebbe essere parallelizzato con quella parallel degree.
Se provo a compilare un programma indicando nw=8 e ho 4 core allora avrò un extra parallelism mentre invece se ho 16 core lavorerò di meno.
- Possiamo usare lo stesso pragma loop per dire che non deve essere vettorizzato:

```
#pragma loop(no vector)
```

AMP (Accelerator Massive parallelism): è un modo per sfruttare delle strutture dati in modo che parte degli algoritmi e dei container paralleli non generino del codice che poi viene eseguito sui core della shared memory machine ma sui core che abbiamo nelle GPU collegate alla macchina.

Task

È una parallel library, il concetto di task è simile a quello di thread e normalmente dobbiamo fornire il body del task. Questo body viene eseguito concorrentemente al codice del main.

Abbiamo una call per creare il task, dopo che l'abbiamo creato possiamo avviarlo (o possiamo avviarlo in automatico), possiamo attendere per un task oppure possiamo attendere tutti i task che abbiamo creato fino ad ora. La differenza rispetto al fork-join model è che qui il task non è eseguito sicuramente con un nuovo thread ma con un thread pool, è più simile al task di OpenMP (quando inserisci il pragma task riesci ad eseguire quel codice in modo asincrono).

È più sofisticato di OpenMP, esiste un modo per sincronizzare i task con le dipendenze, per esempio:

- Continuation: possiamo creare un task e indicare che quando finisce l'esecuzione di quel thread potrà essere avviato il thread 2.
t1.create.then(t2).

Ci sono anche altri costrutti, ad esempio il when_all.

Posso forkare un numero di task che vengono eseguiti in modo concorrente e poi scrivendo .when_all(t2) posso dire che il task t2 verrà eseguito solamente dopo che tutti i task precedenti sono stati già eseguiti.

Esempio:

```
x{  
    create_task([] { __ });  
    create_locks};  
auto results = when_all(x.begin()  
                        x.end())
```

C'è anche un modo più complicato per creare gruppi di task che possono essere rappresentati in alberi.

Questi meccanismi sono più complessi di quelli che abbiamo con OpenMP.

Parallel Algorithms

Qua ci sono cose che sono più classiche ed altre meno, abbiamo due costrutti differenti per fare il parallel for:

- Parallel_For: parallelizza un for che non fa necessariamente una iterazione su tutti gli elementi della struttura dati. Possiamo avere un for che solamente per alcune iterazioni applica un metodo. Le iterazioni del for devono essere indipendenti e vengono eseguite in parallelo. Possiamo usare varie strategie per eseguire in parallelo.
- Parallel_forEach: qui abbiamo una collezione (vector, map) e per ognuno degli elementi svolgiamo una certa computazione.

Entrambi accettano come parametro una sorta di partitioner ovvero un metodo che mi indica in che modo devono essere suddivise le collezioni.

Abbiamo vari metodi per creare il partitioner:

- Statico: abbiamo un set di dati e un numero di thread, quello che facciamo è dividere la dimensione dei dati per il numero di thread e poi assegnamo questo range come lavoro ad ognuno dei thread.
- Affinity Partition: vengono considerate alcune ottimizzazioni, in particolare si presta attenzione alla cache, prima di spostare i dati si cerca di capire se questo spostamento sarà utile dal punto di vista della locality.
- Simple partitioner: abbiamo una chunk size e assegnamo il lavoro ad ognuno dei thread.
- Auto partitioner: iniziamo con un partizionamento statico, poi se ci accorgiamo che alcuni thread finiscono prima di altri e che quindi il lavoro non è bilanciato spostiamo il lavoro da un thread ad un altro in modo da bilanciarlo.

Ci sono anche altre cose interessanti:

- **Parallel_invoke**: avvia la computazione. Avviamo in parallelo due computazioni, prima viene invocata la prima e poi la seconda, quando finiscono entrambi il controllo passa di nuovo all'utente.

```
parallel.invoke({[a](){cond1}
                 {[b] (){cond2}
                  ....
                }});
```

- **Parallel_transform**: è equivalente alla map, prende una collezione e la funzione che vogliamo applicare, ci sono tre differenti varianti, data una collezione:
 - `Transform(v.begin(), v.end(), [](int x){return (x+x)})`
Prendo una collezione e per ogni elemento ottengo un nuovo valore in quella specifica posizione
 - Oppure posso fare la stessa cosa di sopra scrivendo però in un'altra collezione
 - Oppure posso prendere due collezioni in input e posso scrivere in una collezione di destinazione
- **Parallel_reduce**: `parallel_reduce(v.begin(), v.end, operation)`
Abbiamo la possibilità di usare una funzione binaria che somma una collezione di dati
- **Parallel_sort()**: data una collezione di dati abbiamo vari possibili algoritmi di sort e viene scelto quello che meglio si adatta al tipo di collezione e al tipo di dati che vogliamo ordinare.

Parallel Containers

Abbiamo le classiche collezioni che usiamo in C++ e vogliamo creare una versione concorrente che mi permetta di lavorarci in anche in parallelo.

Sono le seguenti strutture:

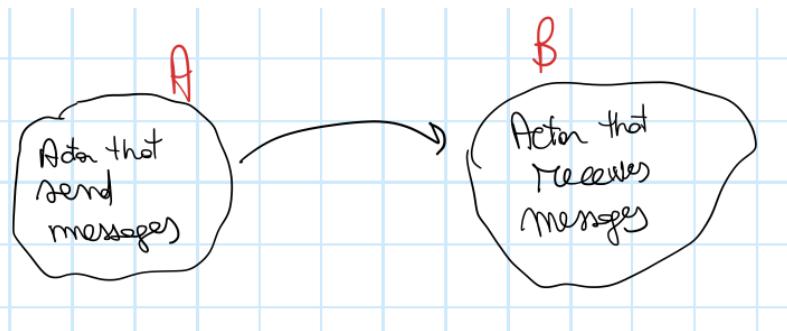
- **Concurrent_vector**: quello che vogliamo fare è avere la possibilità di aggiungere elementi o eliminarli durante l'esecuzione. Possiamo poi leggere la posizione di un elemento o settarla.
Tutte queste operazioni devono essere fatte in modo concorrente in modo "safe".
- **Concurrent_queue**: garantisce che la coda, gestita in modo LIFO o in modo FIFO, possa permettere l'inserimento o l'eliminazione di un elemento dalla coda in modo concorrente.

Abbiamo a disposizione:

- Concurrent map e multimap: la concurrent map è di base una hash table, abbiamo chiavi uniche e non sono ordinate. Nella multimap invece possiamo avere vari elementi che hanno la stessa chiave.
- Concurrent set e multiset: il set è un classico set con i valori, nel set devono essere unici, nel multiset possono anche non essere unici.
- Combinable: se dichiariamo una variabile combinable otteniamo quello che mi serve per lavorare in parallelo come se stessimo calcolando una map reduce. Se abbiamo una a combinable allora possiamo riferirci ad:
 - Una a locale per il thread
 - Abbiamo un metodo che è il combine method che mi permette di prendere il risultato finale da tutti i risultati locali.Con il combinable si parla più di un accumulator state, a differenza di OpenMP in cui invece dichiariamo una variabile in cui fare la reduce con il pragma parallel_for.

Agent Model

L'agent model è qualcosa in cui si ragiona in base al numero di attività indipendenti che comunicano con altre attività indipendenti. Quello che non è permesso è la condivisione dello stato interno dell'agente con un altro agente.



Un attore manda il messaggio e l'altro invece riceve il messaggio, l'idea è che gli agenti sono piccole entità che lavorano in parallelo, sono entità simili ma anche differenti dai task. Il task infatti quando viene forkato può condividere delle variabili con altri task e quindi può usare queste variabili per la comunicazione con gli altri task.

Al contrario un agent viene forkato e poi può solamente inviare e ricevere dati.

Le operazioni dell'agent sono le seguenti:

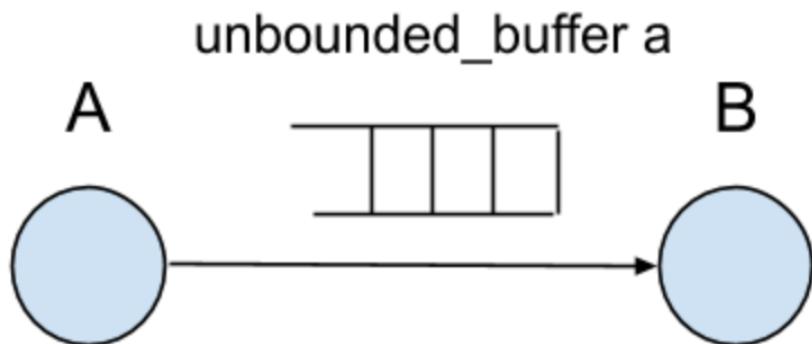
- Send()
- Receive()
- Done(): per indicare che ho finito

Per ogni agent deve essere implementato un metodo run che sarà quello che viene eseguito quando l'agent viene avviato.

Nel costruttore dell'agent possiamo passare un qualsiasi parametro, quello che possiamo fare è dichiarare un buffer (unbounded) di tipo int, ad esempio, e poi possiamo usare la creazione di questo agente in modo che non sappiamo precisamente il nome dell'agente ma sappiamo quale buffer dobbiamo utilizzare.

Ad esempio sappiamo che A invierà sul buffer a e poi B sa che dovrà

leggere dal buffer a.

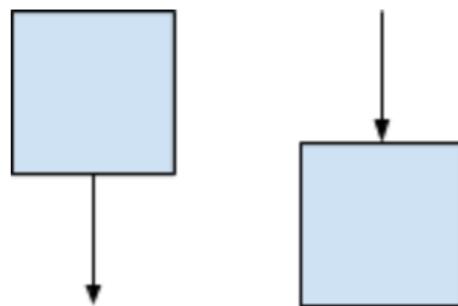


In questo modo con il message passing model evito anche che un thread vada accidentalmente a sovrascrivere una variabile che uso in un altro thread.

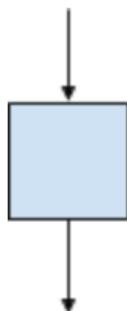
Data Flow

Nel data flow, che è solamente in C# perchè è nella libreria TPL, abbiamo tre tipi di blocchi:

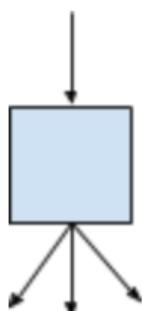
- Buffer blocks: i buffer blocks possono fare una post o una receive di dati in una coda. Hanno tipi differenti:
 - Normal buffers: post e receive
 - Broadcast block: post di un dato e poi più di un blocco può estrarre quel valore dalla coda
 - Write one block: posso fare la post di un valore, poi prendo il primo valore dalla coda da un blocco, poi se metto un altro valore all'interno della coda non lo considero dallo stesso blocco in cui ho già estratto l'altro. In pratica è come avere una variabile costante che non può quindi cambiare il suo valore nel tempo.
- Compute blocks: questo è più interessante, abbiamo vari tipi:
 - Action block: qualcosa che può produrre o consumare uno stream



- Transform block: hanno sia l'input che l'output, possono consumare i dati, crearli e trasformarli.



- Transform many block: in questo caso l'output è ridirezionato verso varie destinazioni.

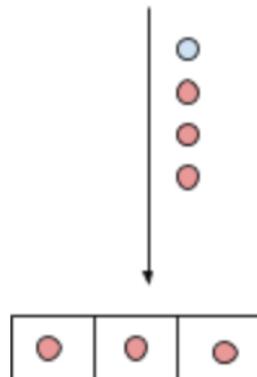


- Group blocks: abbiamo anche qua vari tipi:
 - Join block: prende i dati da più di una fonte e poi emette i dati in una singola coda in output.

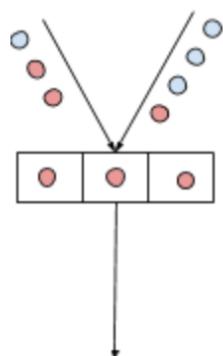


- Batch block: è un modo di accumulare i task per poi consumarli. Prima attendiamo da un singolo canale un certo numero di elementi e poi restituiamo il vettore composto da quegli elementi.

Batch(3)

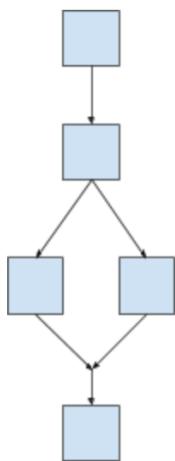


- Batch_Join block: ricombinazione dei tipi precedenti, prendo ad esempio 3 elementi da un canale e altri elementi da un altro. Poi li mando in output in un solo canale attendendo però che tutti gli elementi siano arrivati.



Con questi puoi definire tutto quello che vuoi, ad esempio puoi anche implementare un grafo data flow.

Se il data flow da implementare è molto semplice, possiamo anche farlo con i task dovendo però gestire a mano lo scheduling e le dipendenze. Con i metodi descritti sopra la situazione è più semplice perché non devo affrontare tutti i problemi legati alla concorrenza, allo scheduling e alle dipendenze.



C++ AMP

Vogliamo accelerare le computazioni in parallelo utilizzando le GPU. Questo C++ AMP ci fornisce la possibilità di convertire le strutture dati come i vettori attraverso delle viste in strutture dati che vengono gestite dai core della GPU.

Su queste strutture potremo anche applicare un `parallel_for_each` e poi dopo che nella GPU sono stati svolti i calcoli potremo portare i risultati nella memoria principale.

Qua l'unica cosa che va fatta a mano è dire che una certa struttura dati dovrà essere processata dall'accelerator, poi tutto il resto sarà automatico e trasparente all'utente.

Fondamentalmente possiamo fare ciò che ci serve usando agents, task e data flow sfruttando delle strutture dati parallele ed usando AMP come accelerator.

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 226

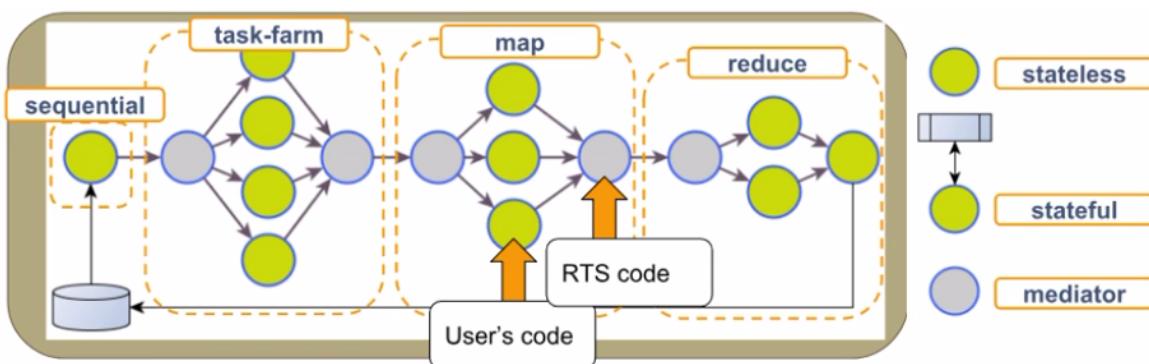
Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 226

Appunti su FastFlow

Lezione 1

Il modello di programmazione offerto da FastFlow:

Una applicazione FastFlow è un grafo diretto in cui i nodi sono entità che fanno la computazione e gli archi sono canali di comunicazione attraverso i quali vengono scambiati dati.



Siamo in una macchina con shared memory, i cerchi verdi (building block o nodes) nella figura sopra sono thread che eseguono il codice scelto dall'utente, i building block grigi sono nodi che eseguono del codice che viene fornito da fastflow.

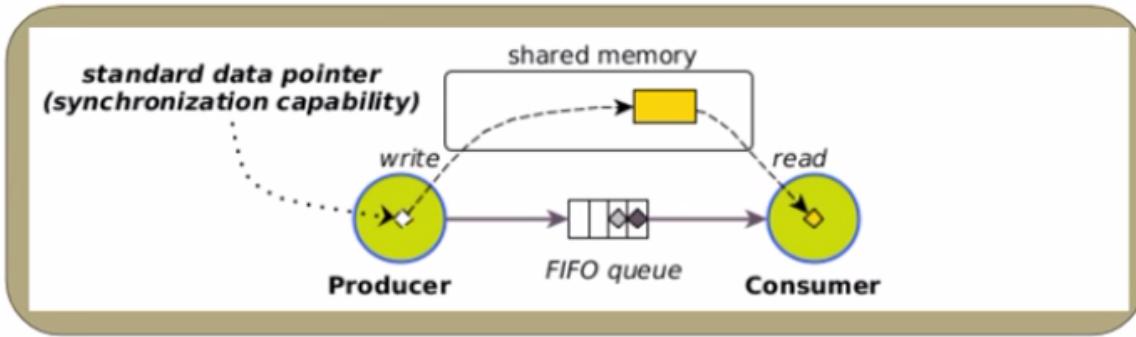
Questi nodi sono implementati 1:1 con un thread, questa è una feature e una limitazione perchè se abbiamo un grafo con tantissimi nodi non è una cosa positiva, se invece abbiamo un numero basso di nodi che vengono eseguiti su una macchina con un numero alto di core possiamo sfruttare al massimo la potenza della macchina assegnando un thread ad ognuno dei core.

I nodi possono essere senza stato (stateless) o con lo stato (stateful).

I nodi che sono all'interno del grafo si sincronizzano inviandosi dei messaggi che poi permettono l'accesso ai dati condivisi.

Comunicazione dei nodi

Per quanto riguarda la comunicazione tra i nodi, l'idea è che qua abbiamo una shared memory multi core.



Abbiamo un producer che manda i dati all'interno della memoria condivisa, poi prende una reference a questi dati che sono nella memoria condivisa e la mette nella coda FIFO che lo collega al consumer.

Si tratta di un message passing model, non muoviamo i dati, i dati stanno sempre nello stesso punto e quindi questo ci permette di ridurre il numero di copie dei dati che dobbiamo fare.

Muovere solamente la reference è un'idea migliore perchè spostiamo meno dati. Se il producer continua ad usare la reference potrei avere un problema, in realtà qua diciamo che quando il producer mette la reference nella coda, non accede più a quella reference e il consumer può accedere. Quindi non mi serve un meccanismo di lock che mi garantisca il passaggio dei dati dal producer al consumer.

Questo è il principio che sta sotto all'idea di Fastflow.

Stream Concept

Un altro concetto importante è l'idea dello stream concept, è importante perchè l'idea dello stream è un concetto di prima classe in FastFlow.

Uno stream è una sequenza di valori che provengono da uno o più fonti, se provengono tutti da una fonte hanno tutti lo stesso tipo.

Lo stream può essere creato da una fonte esterna a Fastflow, ad esempio un sensore o dall'applicazione stessa.

Tipicamente il generatore della fonte di dati è chiamato "Source" mentre chi prende i dati dallo stream è chiamato "Sink".

L'idea dello stream è molto interessante, posso avere uno stream già pronto e lo posso utilizzare per lavorare sui dati ma possiamo anche produrne uno dai dati che abbiamo.

Ad esempio potrei avere una matrice di grandi dimensioni e vogliamo produrre uno stream di dati partendo dai dati presenti all'interno della matrice, quindi allochiamo lo spazio necessario e poi mandiamo i dati nello stream.

Canali di comunicazione di FastFlow

Viene utilizzato un pattern producer-consumer e viene inviata nell'queue condivisa una reference al dato che vogliamo inviare. Quello che ci serve è una coda concorrente con un singolo produttore e un singolo consumatore.

Questa coda è chiamata SPSC, non usa operazioni atomiche o lock (sarebbe la cosa più semplice usare lock o mutex). Usare le lock è costoso e implica che faccio un'attesa passiva, quando la coda è vuota il consumatore attende che qualcosa venga aggiunto nella coda.

Questo vuol dire che il thread viene messo in sleep dal sistema e quindi abbiamo un costo anche per questa ragione.

Quando poi il thread verrà riavviato avremo anche un altro costo per risveglierlo.

La coda SPSC può essere implementata in modo non blocking (wait free e lock free algorithm).

Lock free non vuol dire che non ci sono lock, vuol dire che il sistema se c'è un fallimento per il producer o il consumer permette ad altri thread di andare avanti.

In un modello con le lock classiche se mettiamo la lock sulla struttura e poi il thread producer viene ucciso, il consumer non può fare niente perchè abbiamo la lock bloccata con il thread che è stato ucciso.

In un modello lock free algorithm invece non c'è niente che non mi permette di far lavorare il consumer anche se fallisce il producer.

In particolare non blocking vuol dire che il thread non si ferma mai, se la coda è piena o è vuota il thread continua a provare a fare la push o la pop dalla coda e non va a dormire. Anche questo ha un costo, anche per quanto riguarda il consumo della corrente.

Pattern di FastFlow (alto livello)

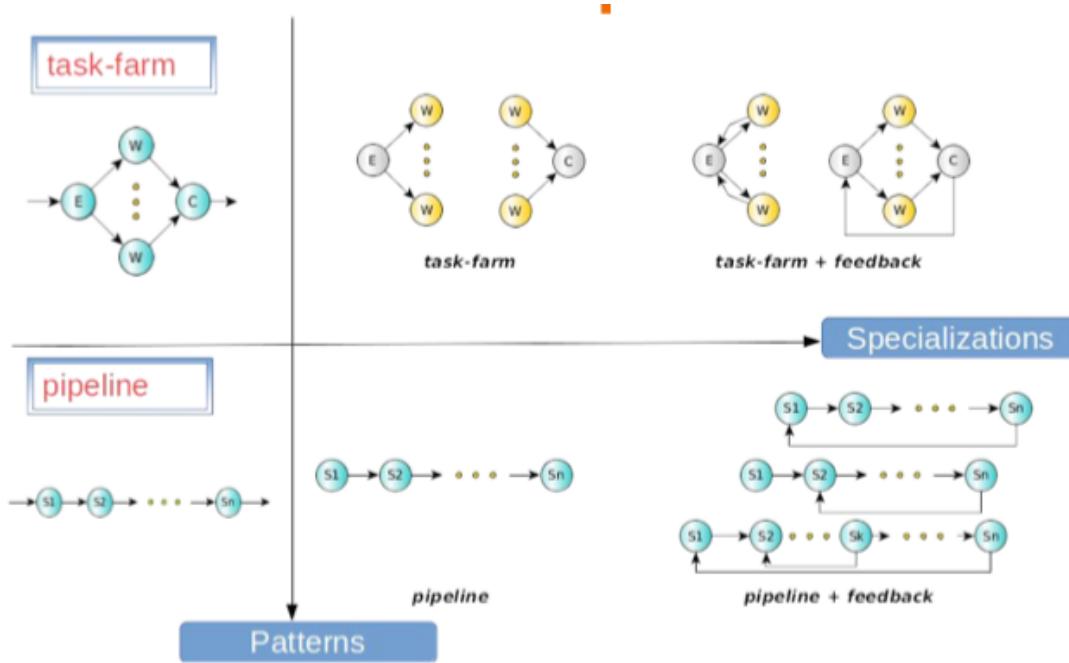
- **Stream Parallel:** ff_Pipe, ff_Farm, ff_OFarm
- **Data Parallel:** ff_Map, ParallelFor, poolEvolution, ff_stencilReduced. La ff_Map in particolare applica la stessa funzione a tutti gli elementi della collezione, non richiede alcuna sincronizzazione o comunicazione ad eccezione della barrier che mettiamo alla fine.

È implementata partendo dal building block della farm.

Nel caso del parallelFor e del parallelForReduce abbiamo una parallelizzazione dei loop in iterazioni indipendenti e inoltre utilizziamo delle variabili per eseguire la riduzione.

- **Task Parallel:** ff_DC, ff_mdf

Pipeline e Farm



Nel caso della farm abbiamo un master node chiamato Emitter che fa lo scheduling degli elementi della coda e li manda nei vari nodi, tipicamente nella farm i nodi sono omogenei e fanno tutti la stessa cosa. Poi abbiamo i nodi interni alla farm che eseguono la funzione e un collector che prende gli output.

Ognuno di questi nodi (emitter, collector e nodi interni) è un thread. È possibile modificare questo pattern della farm, possiamo per esempio rimuovere l'Emitter oppure il collector, è anche possibile creare dei feedback channel che hanno una direzione differente rispetto al data flow standard. Questi channel sono usati per mandare indietro risultati oppure qualche tipo di informazione, possiamo collegare collector ed emitter oppure un worker con l'Emitter oppure possiamo avere una combinazione di questi due.

Una pipeline è più semplicemente una sequenza lineare di ff_node, la stessa pipeline è un node ma è formata da vari ff_node. I vari nodi sono connessi con un channel.

La pipeline è eseguita prendendo come input i dati di uno stream di dati e questo ci permette di avere una reale parallelizzazione dei vari task.

Anche in questo caso possiamo avere dei feedback channel che collegano i vari nodi.

Implementazione concreta di nodi, grafi...

Uno stage sequenziale (un nodo del grafo) è implementato nella class ff_node che è la classe basilare di fastflow.

Esiste anche un classe ff_node_t che è la classe tipata della tt_node e prende in input il tipo dei dati in input e il tipo dei dati in output.

Questo ff_node il building block basilare per l'applicazione Fastflow

Quello che devo fare è utilizzare la classe ff_node e definire almeno uno dei metodi svc che sta per service.

L'implementazione minima di un ff_node **deve ridefinire almeno svc()**, ci sono anche altri due metodi che sono svc_init() e svc_end().

svc_init() viene chiamato solamente una volta quando il thread viene avviato e serve per l'inizializzazione, ad esempio se devo connettermi ad un DB o se devo aprire un file.

svc_end() è un metodo che posso definire e che viene chiamato prima che il nodo termini, ad esempio se abbiamo aperto un file in svc_init() magari possiamo chiuderli qua.

Queste due funzioni sono opzionali e non siamo obbligati a ridefinirli.

Quando definiamo il metodo **svc()** è necessario definire il tipo dei dati in input (IN_t) dalla queue e il tipo dei dati che vengono prodotti (OUT_t).

Nella maggior parte dei casi il tipo in input e il tipo in output è lo stesso.

Poi dentro alla funzione svc() definiamo il codice che vogliamo che venga eseguito, alla fine possiamo restituire qualcosa per il prossimo stage o per il prossimo pattern.

Quindi alla fine questo ff_node è solamente un thread che ha una coda in input, una in output e può eseguire un codice definito.

Definendo questi tre metodi siamo in grado di controllare il ciclo di vita del thread.

Esempio: come definisco un nodo source e un nodo sink

Per definire un nodo di questo tipo non usiamo la parola class ma usiamo la parola struct, in C++ una struct è una classe dove tutto è pubblico.

Source node	Sink node
<pre>struct Source: ff_node_t<Task> { Task *svc(Task *); // generates N tasks and then EOS for(long i=0;i<N; ++i) { ff_send_out(new Task); } return EOS; };</pre>	<pre>struct Sink: ff_node_t<Task> { Task *svc(Task * task); // do something with the task do_Work(task); delete task; return GO_ON; // it does not send out task };</pre>

Nella prima riga di Source definiamo ff_node_t con il tipo del dato in input che è “task”. Implemento il metodo svc, che produce in output un Task e in input prende un Task (in questo caso non prende niente perché questo è il producer).

Poi per ogni elemento del loop alloco un puntatore con il new Task e genero lo stream che poi viene mandato in output e viene preso dal sink. Abbiamo due modi per produrre qualcosa che viene mandato in output, uno è il return e uno è il ff_send_out.

La differenza tra i due: se devo produrre un solo elemento non ci sono differenze tra i due metodi.

Se invece devo generare più elementi in un loop non voglio uscire e rientrare ogni volta dal metodo svc cosa che succederebbe con return, quindi usiamo ff_send_out che non esce da questo metodo e quindi è più semplice da utilizzare [poco chiaro].

La cosa particolare del source node è che viene usato ff_send_out e poi viene generato l'EOS. Senza generare l'EOS non potrei terminare il nodo source e neanche il nodo Sink.

Il simmetrico è il nodo Sink che prende qualcosa in input e non restituisce niente al prossimo. Per non restituire niente si restituisce un puntatore particolare chiamato GO_ON che è un tipo particolare di EOS che viene usato per terminare e dire che non ho niente da restituire al prossimo nodo ma che voglio rimanere comunque vivo.

Le feature usate all'interno di questi nodi possono essere all'interno di un qualsiasi nodo di fastflow.

Come creare una 3 stage pipeline:

```
#include <ff/ff.hpp>
#include <ff/pipeline.hpp>

using namespace ff;
typedef long MyTask

/*
Creo tre nodi della pipeline, Source, Stage e Sink. Poi devo creare la pipeline
vera e propria che mi unisce i vari stage. La creazione della pipeline la faccio
nel main dove istanzio i vari nodi.
*/
struct Source: ff_node<MyTask>{
    MyTask* svc(MyTask *){
        for(long i=0; i<10; ++i){
            ff_send_out(new MyTask(i));
            // per terminare devo per forza restituire EOS, gli altri rimangono vivi
            // fino a quando non vedono l'EOS
            return EOS;
        };
    };

    struct Stage: ff_node<MyTask>{
        MyTask* svc(MyTask * in){
            return in;
        }
    };

    struct Sink: ff_node<MyTask>{
        MyTask* svc(MyTask * in){
            printf("Sink received %ld\n", *in);
            delete in;
            return GO_ON;
        }
    };

    int main(){
        // Istanzia gli stage della pipeline.
        Source S1;
        Stage S2;
```

```

Sink S3;

// Creo la pipeline
ff_Pipe<MyTask> pipe(S1, S2, S3);
// uso questa funzione perchè devo runnarla in modo      asincrono, posso
anche controllare gli errori.
if(pipe.run_and_wait_end() < 0){
    error("running pipeline \n");
    return -1
}
return 0;
}

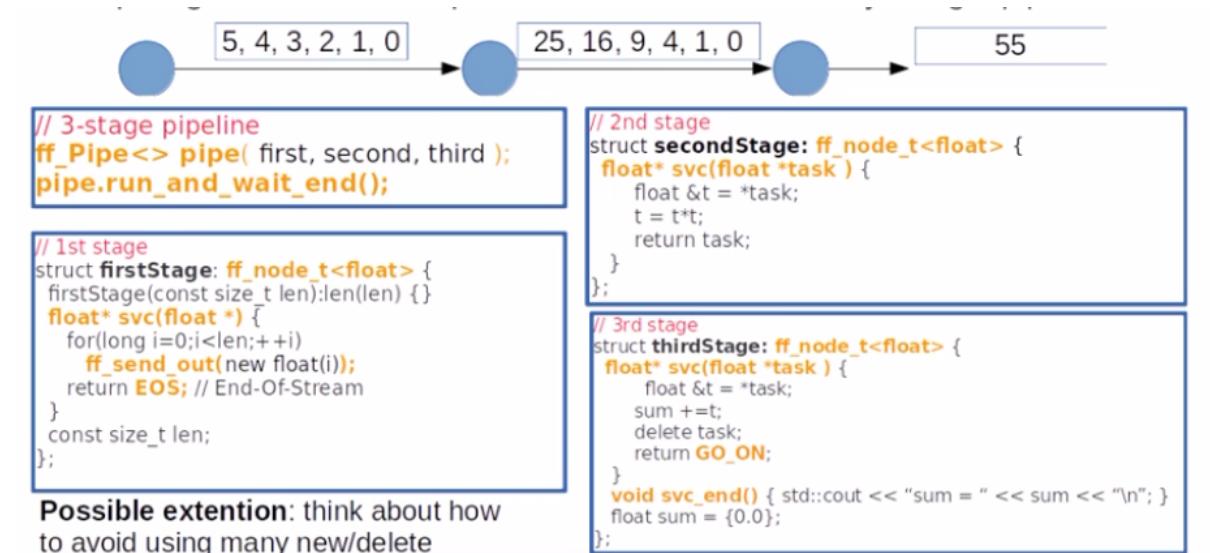
```

Per compilare:

g++ -std=C++17 -Wall -g -I ./Documents First.cpp -o first -pthread.

Il -I serve per specificare il path degli include

Un altro esempio di Pipeline:



Lezione 2

Un altro metodo che può essere ridefinito quando creiamo un nodo di fastflow è eosnotify().

Questo metodo viene chiamato ogni volta che un EOS è ricevuto dal nodo, quando ricevo un EOS il nodo deve terminare ma se ad esempio ho tanti nodi in input, quel nodo deve terminare solamente se ricevo un EOS da tutti gli input channel. In questo caso ogni volta che un EOS viene ricevuto da una delle input queue viene chiamato questo metodo e decidi cosa vuoi fare (ad esempio potrei fare il flush del buffer se mantengo qualcosa salvato nel buffer).

Consideriamo ora il caso di una pipeline ma in questo caso vogliamo considerare anche la possibilità di avere anche dei feedback channel che connettano i nodi nella direzione opposta rispetto al data flow classico.

Questi feedback channel possono essere utili perchè posso mandare indietro delle informazioni che in alcuni casi possono essere importanti.

```
#include <ff/ff.hpp>
using namespace ff;
ff_Pipe<> pipeIn(S_A,S_B);
pipeIn.wrap_around();
ff_Pipe<> pipeOut(S_0,pipeIn,S_1);
pipeOut.wrap_around();
if (pipeOut.run_and_wait_end()<0)
    error("running pipe");
```

Nel codice sopra creiamo una pipeline con due stage, A e B, quando viene creata la pipeline viene creato in automatico un forward channel che va da A a B, poi però quando chiamiamo il metodo wrap_around() viene creato il feedback channel.

Poi creo una seconda pipeline che ha come stage 0, 1 e poi la pipeline creata in precedenza, le pipeline possono essere organizzate come vogliamo tipo matrioska.

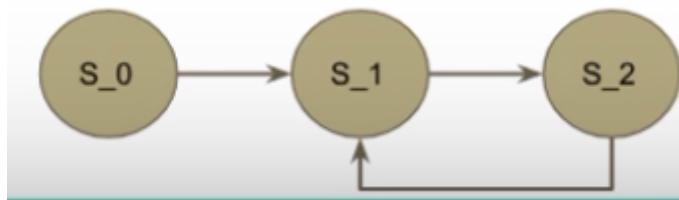
Anche per questa seconda pipeline chiamiamo wrap_around() e introduciamo il feedback channel.

Il nodo S_A ha due input e un output, è diverso da quello visto fino ad ora, in FastFlow è anche possibile avere nodi che hanno più di un input e un singolo output oppure anche più di un output.

Il problema con i feedback channel è che la terminazione è più complessa perchè un nodo termina quando riceve tutti gli EOS da tutti gli input channel. Una volta che ha ricevuto l'EOS da tutti i feedback channel può inviare l'EOS a tutti gli output channel e poi chiama svc_end().

Se ho un feedback channel posso avere un problema, il nodo S_1 potrebbe non ricevere l'EOS dal feedback channel se non ho inoltrato l'EOS dal nodo S_1 al nodo S_2.

È una specie di deadlock, perchè S_1 riceve l'EOS dal nodo S_0 però non può inoltrarlo se non riceve l'EOS da S_2, solo che S_2 mi manda l'EOS solamente se lo ha già ricevuto da S_1.



Ci serve qualcosa che possa permetterci di controllare il numero di EOS ricevuti e questo qualcosa è l'eosnotify().

L'eosnotify viene chiamato quando ricevo il primo EOS, nella funzione svc devo contare il numero di elementi che ricevo e invio al prossimo stage senza però che questi tornino indietro nel feedback channel.

Un esempio per vedere meglio la terminazione dei ff_node:

```
#include <ff/ff.hpp>
#include <ff/pipeline.hpp>
using namespace ff;

/*
     Loop che genera lo stream degli elementi
*/
struct Stage_0: ff_node_t<long>{
    long* svc(long*) {
        for(long i=1;i<=100;++i)
// qua mi basta castare il nuovo intero ad
// un puntatore, è un trick per evitare di allocare dati, in questo caso
// funziona perfettamente perchè non viene usato in realtà questo i
        ff_send_out((long*)i);
        //sleep(100);

        return EOS;
    }
};

struct Stage_A: ff_minode_t<long> { // multi-input node
    long* svc(long* in) {
        /*
        Devo controllare se i dati arrivano dall'input channel o no, se arrivano
        dall'input channel incremento onthefly, altrimenti devo decrementarlo. Se
        onTheFly è 0 posso generare l'EOS.
        */
        if (fromInput()) {
            ff_send_out(in);
            ++onthefly;
        } else
            if ((--onthefly<=0) && eosreceived) return EOS;
        return GO_ON;
    }
    /* definiamo il metodo eosnotify per capire se ho ricevuto l'EOS, quando ricevo
    l'EOS sono sicuro che arriva dal nodo precedente perchè non può arrivare dal
    successivo perchè l'EOS per ora non è ancora stato propagato.
    */
    void eosnotify(ssize_t) {
        eosreceived=true;
        // Dovrei avere anche questo controllo qua perchè potrei ricevere
        // l'ultimo elemento dello stream prima di ricevere l'EOS dallo
        // stage precedente.
        // Questo controllo non posso metterlo in Svc_end perchè quando
        // viene chiamato svc_end la coda è già chiusa e non posso
        // mandare niente al prossimo stage.
    }
}
```

```
    if (onthefly==0) ff_send_out(EOS);

}

bool eosreceived=false;
size_t onthefly=0;
};

struct Stage_B: ff_monode_t<long> {
    long* svc(long* in) {
        //usleep((Long)in);
        ff_send_out_to(in, 0);
        ff_send_out_to(in, 1);
        return GO_ON;
    }
};

struct Stage_1: ff_node_t<long>{
    long* svc(long* in) {
        printf("S_1 received %ld\n", (long)in);
        return GO_ON;
    }
};

int main() {
    Stage_0 S_0;
    Stage_1 S_1;
    Stage_A S_A;
    Stage_B S_B;
    ff_Pipe<> pipeIn(S_A, S_B); // pipeline interna
    pipeIn.wrap_around();      // feedback
    ff_Pipe<> pipeOut(S_0, pipeIn, S_1); // pipeline esterna
    if (pipeOut.run_and_wait_end()<0)
        error("running pipeOut\n");
    return 0;
}
```

Farm in FastFlow:

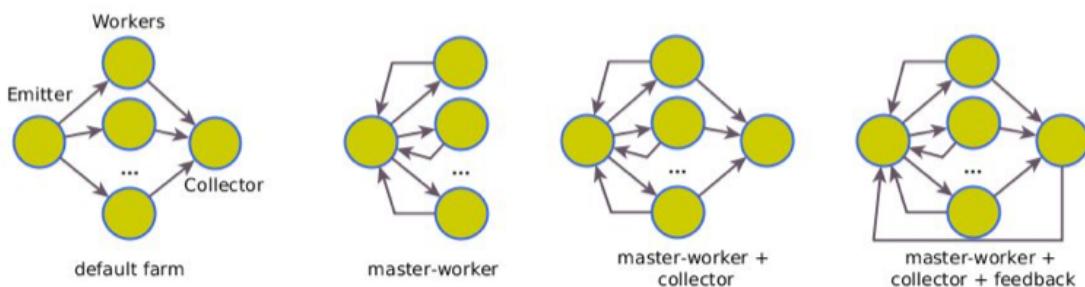
In una farm tipicamente abbiamo dei worker che sono stateless, abbiamo un emitter che fa da scheduler dei task ai vari worker.

In fastflow la farm è implementata in modo da avere una coda davanti di input e una di output per ognuno dei nodi worker (per il motivo delle lock spiegato la volta scorsa).

Abbiamo varie possibili implementazioni della farm:

- Farm classica

- Master-worker: è la farm senza il collector che usa il feedback channel per mandare indietro il risultato all'mitter senza usare il collector.
- Master, worker e collector con la possibilità di inviare dati indietro all'mitter dai worker
- Master, worker e collector con la possibilità di inviare dati indietro all'mitter dai worker e dal collector



Tutti questi tipi di farm possono essere implementati, si sceglie quello che meglio si adatta alle nostre richieste.

In generale ogni worker è un nodo, possiamo generare un worker come una pipeline o anche come un'altra farm perchè anche la farm a sua volta è un nodo.

Non abbiamo limitazioni nel modo in cui possiamo comporre i vari componenti.

Nella farm di fastflow per specificare il numero di worker utilizziamo un vettore, la dimensione del vettore mi indica il parallelism degree.

Di default la farm ha un emitter e un collector, il collector e l'mitter possiamo rimuoverli, allo stesso modo possiamo anche specializzarli e riscrivere il loro codice.

Lo scheduling dei dati che vanno dall'mitter ai worker di default è round robin, questo vuol dire che i canali in fastflow possono essere sia di dimensione fissata sia di dimensione non fissata, di default non è fissata la dimensione.

Se le queue **non hanno dimensione fissata** allora sono davvero round robin, se invece hanno una dimensione fissata **non è garantito** che lo scheduling sia realmente round robin. Quindi possiamo dire che la politica è pseudo-round-robin.

Codice per creare la farm:

```
struct myNode: ff_node_t<myTask> {
    myTask *svc(myTask * t) {
        F(t);
        return GO_ON;
    }
};

std::vector<std::unique_ptr<ff_node>> W;
W.push_back(make_unique<myNode>());
W.push_back(make_unique<myNode>());

ff_Farm<myTask>
    myFarm(std::move(W));

ff_Pipe<myTask>
    pipe(_1, myFarm, <...other stages...>);

pipe.run_and_wait_end();
```

Per prima cosa creo il nodo myNode con il codice che voglio eseguire all'interno dei vari worker della farm.

Poi devo creare un vettore e riempirlo con i nodi che ho creato e che svolgono il task nei vari worker.

Poi devo creare la farm vera e propria passando come parametro il vettore dei worker.

Nella penultima riga posso creare una pipeline in cui passo come stage la farm che ho creato.

Un esempio di Farm: passiamo da una pipeline in cui vengono emessi dei numeri, poi vengono elevati al quadrato e poi vengono sommati ad

una farm in cui nel secondo stage è presente una farm che svolge il calcolo in modo parallelo.

```
#include <iostream>
#include <ff/ff.hpp>
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;

struct firstStage: ff_node_t<float> {
    firstStage(const size_t length):length(length) {}
    float* svc(float *) {
        for(size_t i=0; i<length; ++i) {
            ff_send_out(new float(i));
        }
        return EOS;
    }
    const size_t length;
};

struct secondStage: ff_node_t<float> {
    float* svc(float * task) {
        float &t = *task;

        printf("secondStage%ld received %d\n", get_my_id(), t);

        t = t*t;
        return task;
    }
};

struct thirdStage: ff_node_t<float> {
    float* svc(float * task) {
        float &t = *task;
        //std::cout<< "thirdStage received " << t << "\n";
        sum += t;
        delete task;
        return GO_ON;
    }
    void svc_end() { std::cout << "sum = " << sum << "\n"; }
    float sum = 0.0;
};

int main(int argc, char *argv[]) {
    if (argc<3) {
        std::cerr << "use: " << argv[0] << " nworkers stream-length\n";
        return -1;
    }
}
```

```
}

const size_t nworkers = std::stol(argv[1]);

firstStage first(std::stol(argv[2]));
thirdStage third;

#ifndef __CINT__
// Creazione del vettore con le repliche dei nodi della farm
std::vector<std::unique_ptr<ff_node>> W;
for(size_t i=0;i<nworkers;++i)
    W.push_back(make_unique<secondStage>());

ff_Farm<float> farm(std::move(W)); // pay attention to the move, otherwise
you can use a Lambda

#else
// a different way to build a farm (in-place)

ff_Farm<float> farm(
    [nworkers]() {
        std::vector<std::unique_ptr<ff_node>> W;
        for(size_t i=0;i<nworkers;++i)
            W.push_back(make_unique<secondStage>());
        return W;
    } ());
#endif

ff_Pipe<> pipe(first, farm, third);

ffTime(START_TIME);
if (pipe.run_and_wait_end()<0) {
    error("running pipe");
    return -1;
}
ffTime(STOP_TIME);
std::cout << "Time: " << ffTime(GET_TIME) << "\n";
return 0;
}
```

In questa prima implementazione abbiamo sia il collector che l'mitter, se vogliamo possiamo passare ad una implementazione in cui rimuoviamo il collector. Se rimuoviamo il collector il prossimo stage diventa un multiple input stage.

Quindi dobbiamo fare una modifica nel codice del terzo stage che ora diventa un multi input node.

Nella farm devo anche rimuovere il collector.

```
#include <iostream>
#include <ff/ff.hpp>
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;

struct firstStage: ff_node_t<float> {
    firstStage(const size_t length):length(length) {}
    float* svc(float *) {
        for(size_t i=0; i<length; ++i) {
            ff_send_out(new float(i));
        }
        return EOS;
    }
    const size_t length;
};

struct secondStage: ff_node_t<float> {
    float* svc(float * task) {
        float &t = *task;
        std::cout << "secondStage received " << t << "\n";
        t = t*t;
        return task;
    }
};

struct thirdStage: ff_minode_t<float> { // <-- NOTE: multi-input node
    float* svc(float * task) {
        float &t = *task;
        std::cout << "thirdStage received " << t << " from " << get_channel_id()
<< "\n";
        sum += t;
        delete task;
        return GO_ON;
    }
    void svc_end() { std::cout << "sum = " << sum << "\n"; }
    float sum = 0.0;
};

int main(int argc, char *argv[]) {
    if (argc<3) {
        std::cerr << "use: " << argv[0] << " nworkers stream-length\n";
        return -1;
    }
    const size_t nworkers = std::stol(argv[1]);
    firstStage first(std::stol(argv[2]));
    thirdStage third;
```

```
//std::vector<std::unique_ptr<ff_node> > W;
//for(size_t i=0;i<nworkers;++i) W.push_back(make_unique<secondStage>());
//ff_Farm<fFloat> farm(std::move(W));
ff_Farm<float> farm( [&]() {
    std::vector<std::unique_ptr<ff_node> > W;
    for(size_t i=0;i<nworkers;++i)
W.push_back(make_unique<secondStage>());
    return W;
} () );

farm.remove_collector(); // <-----

ff_Pipe<> pipe(first, farm, third);
ffTime(START_TIME);
if (pipe.run_and_wait_end()<0) {
    error("running pipe");
    return -1;
}
ffTime(STOP_TIME);
std::cout << "Time: " << ffTime(GET_TIME) << "\n";
return 0;
}
```

In un esempio di questo tipo perchè stiamo utilizzando una pipeline? Non potrei semplicemente avere una farm con un emitter ed un collector che siano specializzati?

Se pensiamo alla farm implementata in questo modo, la farm diventa una 3 stage pipeline, dato che in FastFlow abbiamo la possibilità di customizzare emitter e collector possiamo spostarci in una implementazione di questo genere riutilizzando il codice che abbiamo già scritto.

In pratica quando creo la farm utilizzo quello che prima era il primo stage della pipeline come emitter, poi userò quello che era il terzo stage come collector della farm.

Un'altra versione per questo esempio è quella in cui abbiamo un emitter che fa anche da collector, quindi combiniamo il primo stage con il terzo e aggiungiamo un feedback channel.

In questo caso devo fare varie modifiche al codice, quando sono nell'mitter, per capire se sono nel primo o nel terzo stage devo controllare se in input abbiamo qualcosa o no.

```
#include <iostream>
#include <ff/ff.hpp>
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;

struct firstThirdStage: ff_node_t<float> {
    firstThirdStage(const size_t length):length(length) {}
    float* svc(float *task) {
        if (task == nullptr) {
            for(size_t i=0; i<length; ++i) {
                ff_send_out(new float(i));
            }
            return GO_ON;
        }
        float &t = *task;
        std::cout << "thirdStage received " << t << "\n";
        sum += t;
        delete task;
        if (++ntasks == length) return EOS;
        return GO_ON;
    }
    void svc_end() { std::cout << "sum = " << sum << "\n"; }

    const size_t length;
    size_t ntasks=0;
    float sum = 0.0;
};

struct secondStage: ff_node_t<float> {
    float* svc(float * task) {
        float &t = *task;
        std::cout << "secondStage received " << t << "\n";
        t = t*t;
        return task;
    }
};

int main(int argc, char *argv[]) {
    if (argc<3) {
        std::cerr << "use: " << argv[0] << " nworkers stream-length\n";
        return -1;
    }
}
```

```
const size_t nworkers = std::stol(argv[1]);
firstThirdStage firstthird(std::stol(argv[2]));

std::vector<std::unique_ptr<ff_node>> W;
for(size_t i=0;i<nworkers;++i) W.push_back(make_unique<secondStage>());

ff_Farm<float> farm(std::move(W), firstthird);

farm.remove_collector(); // needed because the collector is present by
default
farm.wrap_around();
//farm.set_scheduling_onDemand(); // optional

ffTime(START_TIME);
if (farm.run_and_wait_end()<0) {
    error("running farm");
    return -1;
}
ffTime(STOP_TIME);
std::cout << "Time: " << ffTime(GET_TIME) << "\n";
return 0;
}
```

Quindi, possiamo creare tanti codici differenti che fanno la stessa cosa e di solito non cambiamo nemmeno troppo il codice della nostra applicazione, solamente nell'ultimo esempio dobbiamo fare un cambiamento ma solo perchè non c'è un modo automatico di unire due nodi.

Problemi di ordinamento prodotti dalla farm: quando abbiamo replicato il secondo stage della farm, ognuno dei worker è implementato con un thread, questi competono per le risorse della macchina quindi possiamo pensare che alcuni saranno più veloci di altri e che alcuni worker produrranno risultati più velocemente di altri.

Quindi potremmo ricevere i risultati in un modo differente rispetto all'input. In questo caso dell'esempio non c'è problema perchè comunque la somma è commutativa, se pensiamo però ad un video e devo calcolare il frame di ogni video in ogni worker, l'ordine in questo

caso è importante e non possiamo mixarlo, quindi in alcune situazioni è molto importante mantenere lo stesso ordine degli input.

Ci sono due modi per mantenere l'ordine:

- Possiamo ignorare l'ordine e solamente nell'ultimo stage riorganizzare i frame nel modo corretto
- Possiamo utilizzare una variante della farm chiamata ordered farm (OFarm) che garantisce l'ordine in output dato l'ordine dei dati in input. In questo caso ci sono delle limitazioni per quanto riguarda la possibilità di utilizzare i feedback channel. Però se abbiamo una sequenza in input e vogliamo la stessa sequenza in output possiamo utilizzare la OFarm.

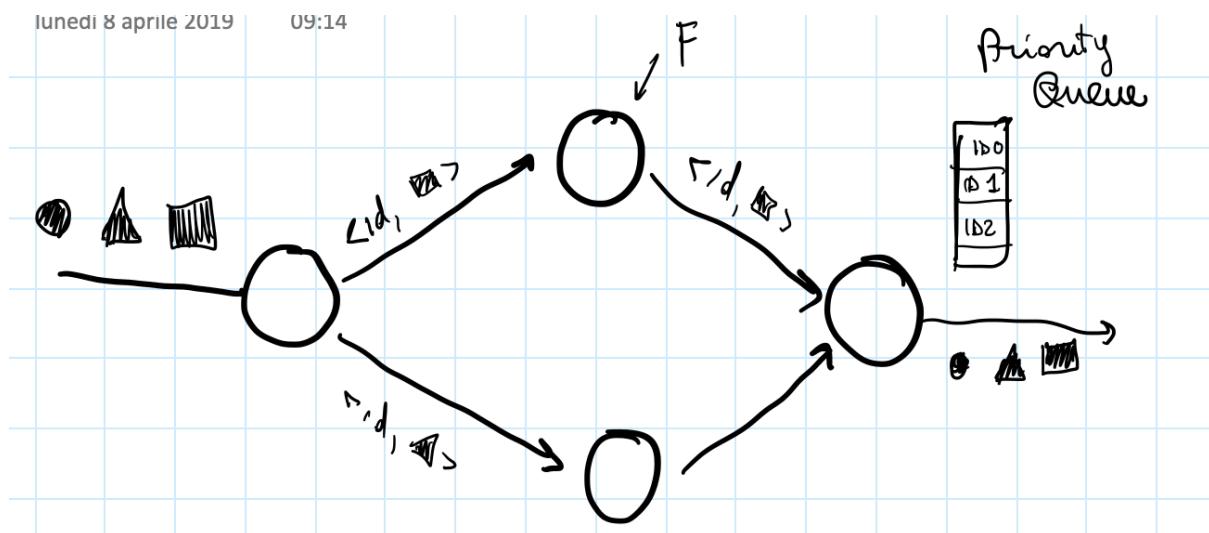
Come viene mantenuto l'ordine all'interno della farm?

In una ordered farm non possiamo rimuovere il collector perchè viene usato per mantenere l'ordine inoltre quando definiamo una OFarm, l'implementazione reale è come una farm ma con una politica fissata per lo scheduling.

La politica per lo scheduling è veramente round robin, non pseudo round robin. L'mitter invia element ai worker in modo round robin e il collector riceve in modo round robin, questo garantisce che l'ordine viene mantenuto. Per l'utente questo non cambia niente. C'è un problema, se uso questa politica di round robin nel caso della parallelizzazione di un video avremo alcune parti del video che richiedono più tempo di altre quindi avremo alcuni worker che costano di più di altri, lo scheduler e il collector devono aspettare che termini anche il worker che ci mette più tempo e questo introduce un problema di "load balancing".

Lezione 3

Per quanto riguarda la Ordered Farm, questa è implementata più o meno in questo modo, abbiamo un emitter che riceve in input i dati che devono essere poi inviati ai vari worker, quindi quello che si fa è assegnare un ID ad ognuno dei vari elementi poi quando il collector riceve il pair $\langle \text{Id}, \text{elemento} \rangle$ manterrà una priority queue in modo da mantenere l'ordine e produrre poi in ordine i risultati basandosi sui dati in ingresso.



Un esempio del funzionamento ordinato della farm utilizzando il solito esempio dei numeri elevati al quadrato e poi sommati.

Per simulare meglio l'esempio viene introdotto del delay nella computazione dei worker 0 e 1, questi due avranno più lavoro da fare.

```
#include <iostream>
#include <ff/ff.hpp>
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>

using namespace ff;

struct firstStage: ff_node_t<float> {
```

```
firstStage(const size_t length):length(length) {}
float* svc(float *) {
    for(size_t i=0; i<length; ++i) {
        ff_send_out(new float(i));
    }
    return EOS;
}
const size_t length;
};

struct secondStage: ff_node_t<float> {
    float* svc(float * task) {
        float &t = *task;
        t = t*t;

        ssize_t myid = get_my_id();
        switch(myid) {
        case 0:
        case 1:
            for(volatile long k=0; k<t;++k);
        default:;
    };

    return task;
}
};

struct thirdStage: ff_node_t<float> {
    float* svc(float * task) {
        float &t = *task;
        //std::cout<< "thirdStage received " << t << "\n";
        sum += t;
        delete task;
        return GO_ON;
    }
    void svc_end() { std::cout << "sum = " << sum << "\n"; }
    float sum = 0.0;
};

int main(int argc, char *argv[]) {
    if (argc<3) {
        std::cerr << "use: " << argv[0] << " nworkers stream-length\n";
        return -1;
    }
    const size_t nworkers = std::stol(argv[1]);

    firstStage first(std::stol(argv[2]));
    thirdStage third;
    std::vector<std::unique_ptr<ff_node> > W;
    for(size_t i=0;i<nworkers;++i) W.push_back(make_unique<secondStage>());
}
```

```
ff_OFarm<float> farm(std::move(w));

#if 0
    farm.set_scheduling_ondemand();
#endif

ff_Pipe<> pipe(first, farm, third);

ffTime(START_TIME);
if (pipe.run_and_wait_end()<0) {
    error("running pipe");
    return -1;
}
ffTime(STOP_TIME);
std::cout << "Time: " << ffTime(GET_TIME) << "\n";
pipe.ffStats(std::cout);

return 0;
}
```

Come possiamo fare ad utilizzare l'on_demand scheduling (metodo per mantenere l'ordine degli output basato sull'ordine degli input) sia nella OFarm che nella Farm?

L'On_Demand_scheduling funziona in modo tale che un worker chiede un altro lavoro da effettuare solamente dopo che ha terminato quello che stava facendo, quindi c'è un feedback channel tra worker ed emitter. Implementare un meccanismo on demand scheduling può non essere semplice, quindi il metodo set_scheduling_ondemand(n) prova a simulare un reale funzionamento dell'on demand scheduling.

Quando settiamo set_scheduling_ondemand le code tra emitter e worker sono bounded e di default la dimensione è 1. L'Emitter emette i dati in modo round robin poi la coda viene svuotata dal worker, l'Emitter la riempie di nuovo e quindi la coda ora è piena. Se arriva altro non ho posto per mettere il nuovo task, provo fino a che non trovo posto.

Supponiamo che il worker 1 finisce prima degli altri, questo manda in output il risultato, prende il prossimo task e poi l'Emitter mette in questa coda il task che andava eseguito.

Non abbiamo bisogno di un feedback channel reale, possiamo interpretare come un ready message il fatto che troviamo posto all'interno della coda.

È una buona approssimazione della politica on demand e funziona abbastanza bene.

Consideriamo il problema di implementare un on demand scheduling reale, non vogliamo utilizzare quello che viene fornito con l'implementazione della farm.

Abbiamo bisogno in questo caso di un feedback channel in cui il worker manda indietro all'mitter un segnale quando finisce di lavorare, a quel punto l'mitter fornirà al worker qualcosa da fare.

Poi anche il terzo stage dovrà riuscire a preservare l'ordine dei dati.

In questo caso dobbiamo scriverci il modo in cui funziona lo scheduling implementando da 0 il funzionamento dell'mitter.

In questo caso il worker è un multi output node e svolge la stessa computazione vista prima poi alla fine deve inviare qualcosa all'mitter (in questo caso un puntatore al task che ho ricevuto) e poi devo mandare un task al prossimo stage. Alla fine il worker restituisce il Go_on perchè comunque vuole rimanere vivo.

L'mitter invece può ricevere sia dall'input sia dal worker, quindi ci serve qualcosa che mi indica da quale canale ricevo, posso usare il numero del canale per distinguerli.

Se arriva l'input dall'input queue devo creare un nuovo task che è un pair dove mettiamo un identificatore e poi il task che ricevo in input.

Devo anche mantenere un vettore che mi dice quali dei worker è pronto a lavorare, quando ricevo il feedback da uno dei worker segno nel vettore che quel worker è pronto per lavorare.

Poi devo anche gestire l'EOS perchè quando abbiamo i feedback channel c'è il problema visto nella lezione precedente.

Per gestire gli EOS possiamo contare il numero di elementi che sono stati inviati e i rdy che ho ricevuto indietro. Ogni volta che qualcosa torna

indietro dobbiamo vedere se l'EOS è stato ricevuto e se tutto è stato calcolato nel modo corretto e in tal caso dobbiamo propagare l'EOS. Quindi la parte più complicata in questo caso è la gestione dell'EOS e l'implementazione di una politica di scheduling reale. L'esempio della implementazione dello scheduling on demand reale con feedback channel:

```
#include <map>
#include <iostream>
#include <ff/ff.hpp>
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;

using mypair_t = std::pair<size_t, float*>

struct firstStage: ff_node_t<float> {
    firstStage(const size_t length):length(length) {}
    float* svc(float * ) {
        for(size_t i=0; i<length; ++i) {
            ff_send_out(new float(i));
        }
        return EOS;
    }
    const size_t length;
};

// multi-output node
struct secondStage: ff_monode_t<mypair_t> {
    mypair_t* svc(mypair_t * task) {
        float &t = *(task->second);
        t = t*t;
        ssize_t myid = get_my_id();
        switch(myid) {
        case 0:
        case 1:
            for(volatile long k=0; k<t;++k);
        default:
        };
        ff_send_out_to(task, 0); // to the Emitter
        ff_send_out_to(task, 1); // to the next stage
        return GO_ON;
    }
};

// multi-input node
```

```
struct thirdStage: ff_minode_t<mypair_t, float > {
    thirdStage(const size_t nworkers):nworkers(nworkers) {}

    float* svc(mypair_t* in) {
        if (counter == in->first) { // it's the next to send out
            counter++;
            float &t = *(in->second);
            delete in;

            sum += t;

            std::map<size_t,mypair_t*>::iterator b = M.begin(), e = M.end();
            while(b != e) {
                if (b->first==counter) {

                    sum += *(b->second->second);
                    delete b->second;
                    counter++;
                    b = M.erase(b);
                } else break;
            }
            return GO_ON;
        }
        M[in->first] = in;
        return GO_ON;
    }

    void svc_end() {
        // check if something is still in the internal state!
        std::map<size_t,mypair_t*>::iterator b = M.begin(), e = M.end();
        while(b != e) {
            if (b->first==counter) {
                sum += *(b->second->second);
                delete b->second;
                counter++;
                b = M.erase(b);
            } else abort(); // something went wrong!
        }
        std::cout << "sum = " << sum << "\n";
    }

    size_t nworkers;
    size_t neos = 0;
    size_t counter = 0;
    std::map<size_t, mypair_t*> M; // internal state used for keeping the
    ordering of data

    float sum = 0.0;
```

```
};

struct Emitter: ff_node_t<float, mypair_t> {
    Emitter(ff_loadbalancer *lb):lb(lb) {}

    int svc_init() {
        last = lb->getNWorkers();
        ready.resize(lb->getNWorkers());
        for(size_t i=0; i<ready.size(); ++i) ready[i] = true;
        nready=ready.size();

        return 0;
    }

    mypair_t* svc(float *in) {
        int wid = lb->get_channel_id();
        if (wid == -1) { // task coming from the first stage, if wid>=0 then it
comes from a worker

            mypair_t* p = new mypair_t;
            p->first = counter;
            counter++;
            p->second = in;

            int victim = selectReadyWorker(); // get a ready worker
            if (victim<0) data.push_back(p);
            else {
                lb->ff_send_out_to(p, victim);
                ready[victim]=false;
                --nready;
            }
            return GO_ON;
        }
        assert(ready[wid] == false);
        ready[wid] = true;
        ++nready;
        if (data.size()>0) {
            lb->ff_send_out_to(data.back(), wid);
            data.pop_back();
            ready[wid]=false;
            --nready;
        } else
            if (eos_received && nready == ready.size()) return EOS;
        return GO_ON;
    }

    void svc_end() {
        // just for debugging
    }
}
```

```
        assert(data.size()==0);
    }
    void eosnotify(ssize_t id) {
        if (id == -1) { // we have to receive all EOS from the previous stage
            // EOS is coming from the input channel

            eos_received=true;
            if (eos_received          &&
                nready == ready.size()    &&
                data.size() == 0) {
                lb->broadcast_task(EOS);
            }
        }
    }

    int selectReadyWorker() {
        for (unsigned i=last+1;i<ready.size();++i) {
            if (ready[i]) {
                last = i;
                return i;
            }
        }
        for (unsigned i=0;i<=last;++i) {
            if (ready[i]) {
                last = i;
                return i;
            }
        }
        return -1;
    }

    bool eos_received = false;
    size_t counter=0;
    unsigned last;
    unsigned nready;
    ff_loadbalancer *lb;
    std::vector<bool> ready;
    std::vector<mypair_t*> data;
};

int main(int argc, char *argv[]) {
    if (argc<3) {
        std::cerr << "use: " << argv[0]  << " nworkers stream-length\n";
        return -1;
    }
    const size_t nworkers = std::stol(argv[1]);

    firstStage first(std::stol(argv[2]));
}
```

```
thirdStage third(nworkers);      // this is a multi-input node
std::vector<std::unique_ptr<ff_node>> W;
// workers are multi-output nodes
for(size_t i=0;i<nworkers;++i) W.push_back(make_unique<secondStage>());

ff_Farm<float> farm(std::move(W));
// the Emitter needs the internal Load-balancer
Emitter e(farm.getlb());
farm.add_emitter(e);
farm.remove_collector(); // removing the default collector
farm.wrap_around();      // creating a feedback channel between workers and
emitter

ff_Pipe<> pipe(first, farm, third); // build the pipeline

ffTime(START_TIME);
if (pipe.run_and_wait_end()<0) {
    error("running pipe");
    return -1;
}
ffTime(STOP_TIME);
std::cout << "Time: " << ffTime(GET_TIME) << "\n";

return 0;
}
```

Quindi, per quanto riguarda lo scheduling e i pattern visti fino ad ora abbiamo tre possibilità:

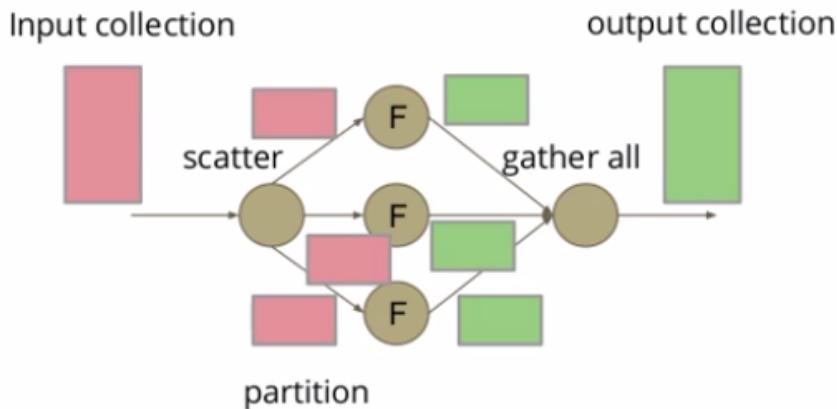
- Pipeline: non ha uno scheduling;
- Farm e OFarm: possiamo scegliere quale scheduling utilizzare, di default è round robin ma possiamo scegliere anche l'on-demand.
In alternativa possiamo definirci una politica di scheduling che però deve essere riscritta dall'utente implementando l'Emitter, il Collector e i Worker in modo da ottenere tutto quello di cui abbiamo bisogno.

Data Parallel Computation With FastFlow

Vogliamo lavorare in parallelo su grandi quantità di dati che vengono partizionati tra i vari worker, tipicamente la computazione è in place quando abbiamo una map dove gli elementi sono indipendenti uno dall'altro.

I pattern tipici sono: Map, Reduce, Map + Reduce, Stencil e Stencil + reduce.

Tipicamente il funzionamento di un pattern data parallel è il seguente:



C'è una barrier prima del gather perchè devo attendere che tutte le partizioni siano state terminate. Non è come una farm.

In fastflow la map è implementata sulla base del parallel for. Il parallel for funziona tipo quello di OpenMP, con una sintassi differente parallelizza un loop su parti differenti dell'array.

Ad esempio potrei avere un loop che mi fa la somma di due array, in modo sequenziale è così:

```
// A and B are 2 arrays of size N  
  
for(long i=0; i<N; ++i)  
    A[i] = A[i] + B[i];
```

Posso poi definire la versione sequenziale con il parallel for passando come lambda function quello che vogliamo fare:

```
#include <ff/parallel_for.hpp>
using namespace ff;

ParallelFor pf(8); // defining the object

pf.parallel_for(0, N, 1, 0, [&A,B](const long i) {
    A[i] = A[i] + B[i];
}, 4);
```

La sintassi:

- Prima definisco il parallelFor passando il numero massimo di worker che vogliamo avere
- Nella chiamata che esegue il for in parallelo abbiamo i seguenti parametri:
 - Starting point del vettore
 - Ending point del vettore
 - Lambda function che deve essere eseguita
 - Come ultimo parametro possiamo anche specificare il numero di worker massimo che vogliamo
 - Alla fine della chiamata abbiamo una barrier implicita

Un parametro importante è la dimensione dei chunk (grain), se il grain è 0 allora l'array viene diviso in parti uguali e assegnato ai vari worker, se è maggiore di 0 o minore di 0 cambia il funzionamento.

Un altro parametro possibile è la dimensione degli step che facciamo quando ci muoviamo nell'array. Per usare bene il parallel for vedere meglio la documentazione in cui vengono spiegati i vari parametri utilizzati.

In alcuni casi può essere utilizzata una chiamata differente del parallel for che è parallel_for_thid che prende un altro parametro che è il thread identifier, questo thread id mi dice quale thread esegue quella specifica

funzione, mi serve per capire chi è quel thread per poi assegnargli il task.

In alcuni casi questo thread id è utile.

Un'altra variante è parallel_for_idx che non prende solamente il thread id ma anche un range. (Vedere documentazione).

C'è un altro pattern che è la combinazione della map con la reduce, in questo caso il pattern è chiamato ParallelForReduce.

In questo caso vogliamo fare il calcolo della somma degli elementi di un vettore.

```
// A is an array of long integers of size N
long sum = 0;
for(long i=0; i<N; ++i)
    sum += A[i];
```

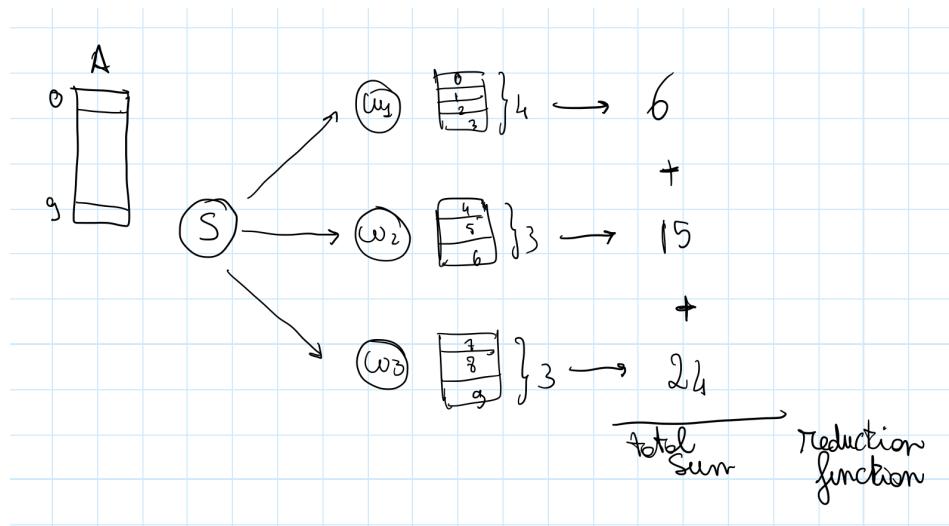
Qua non mi basta un parallel for ma mi serve un parallel for reduce. Devo istanziare il parallel for reduce e devo preparare una reduction variable.

```
#include <ff/parallel_for.hpp>
using namespace ff;

ParallelForReduce<long> pfr;
long sum=0;
pfr.parallel_reduce(sum, 0,
    0,N,1,0, [](const long i, long &mysum) {
        mysum += A[i] + B[i];
    },
    [](long &s, const long e) { s += e; }
);
```

Ogni worker calcola la funzione map che viene specificata come lambda quando viene chiamata parallel_reduce poi viene calcolata anche la reduction (sempre definita con una lambda).

Ogni worker calcola map e reduce sulla sua parte di array, poi alla fine dobbiamo calcolare la reduce completa di tutti i dati.



Il primo worker produce 6, il secondo 15 e il terzo 24, poi però devo sommarli tutti.

La riduzione finale viene calcolata in modo sequenziale.

Esempio di `parallel_for_reduce` per il calcolo del prodotto di due matrici.

```
ParallelForReduce<double> pfr(nworkers, (nworkers < ff_numCores())); // spinwait
is set to true if ...

    //pfr.parallel_for_static(θ, arraySize, 1, CHUNKSIZE, [&](const long j) {
A[j]=j*3.14; B[j]=2.1*j;});
    pfr.parallel_for(θ, arraySize, 1, CHUNKSIZE, [&](const long j) {
A[j]=j*3.14; B[j]=2.1*j;});
        auto Fsum = [] (double& v, const double& elem) { v += elem; };

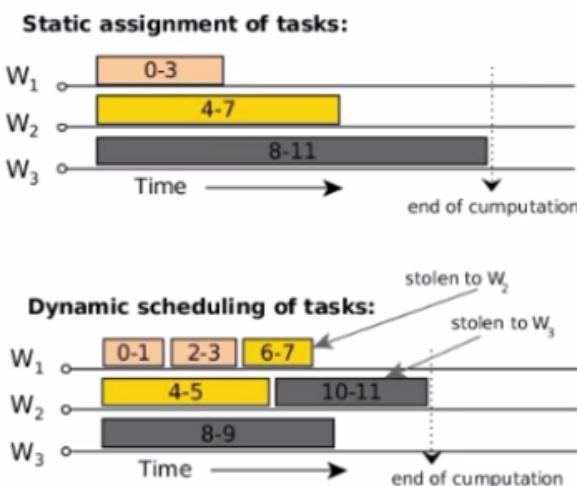
    ff::ffTime(ff::START_TIME);
    for(int z=0;z<NTIMES;++z) {
        //pfr.parallel_reduce_static(sum, 0.0,
        pfr.parallel_reduce(sum, 0.0,
                           0, arraySize, 1, CHUNKSIZE,
                           [&](const long i, double& sum) {sum +=
A[i]*B[i];},
                           Fsum); // FIX: , std::min(nworkers, (z+1)));
    }
}
```

Chunk size: una delle cose più importanti nel data parallel computation è decidere come dividere l'array iniziale:

- Assegnamento statico: dividiamo semplicemente l'array in N parti e ognuna di queste parti viene assegnata ad uno dei vari worker.

Problema in questo caso: se una parte dell'array necessita di più tempo per lavorare ci troviamo a dover attendere che il worker che lavora di più termini il suo lavoro.

- Assegnamento dinamico: in questo caso dividiamo comunque in parti uguali ma non assegnamo subito tutto il task ad ognuno dei worker, gli assegnamo solamente una parte del lavoro che deve essere svolto, in questo modo ci rendiamo conto quando ognuno dei vari worker termina il task e gli forniamo altro lavoro da fare, in questo modo riusciamo a bilanciare il lavoro che viene eseguito. Quando assegno un nuovo chunk ad un worker non è detto che gli venga assegnata una partizione del suo primo task, potrei anche dargli una partizione che prima era stata assegnata al secondo worker.



La divisione del chunk size dipende da un parametro che usiamo quando chiamiamo la funzione del parallel for:

- Se il parametro è 0 allora usiamo lo scheduling statico;
- Se il parametro è > 0 allora è dinamico;
- Se il parametro è < 0 abbiamo una variante dello static scheduling, è comunque statico ma il valore che specifichiamo è usato per

partizionare in modo round robin, ad esempio se mettiamo -2 lo scheduling è fatto così:

- 0 e 1 al worker 0
- 2 e 3 al worker 1
- 4 e 5 al worker 2
- 6 e 7 al worker 0 e così via.

Lezione 4

I pattern del data parallel e dello stream parallel possono essere combinati.

Ci sono due modi per usare il parallel for all'interno di una farm o di una pipeline:

- Possiamo usare direttamente il parallel for nell'svc di un nodo
- Possiamo creare un wrap all'esterno del parallel for che è chiamato ff_Map, i thread della map vengono creati quando il nodo nasce e poi rimangono vivi, invece se usiamo il parallel for nell'svc method i thread vengono creati/stoppati ogni volta che il metodo viene chiamato.

Il parallel for nell'ff_map pattern lo scheduler è disabilitato, quindi alcune piccole ottimizzazioni sono abilitate di default.

Un esempio di questa composizione di pattern, dobbiamo applicare un filtro ad una foto e lo facciamo con una pipeline formata da due stage. Il primo stage legge l'immagine e il secondo stage applica il filtro utilizzando il parallel for.

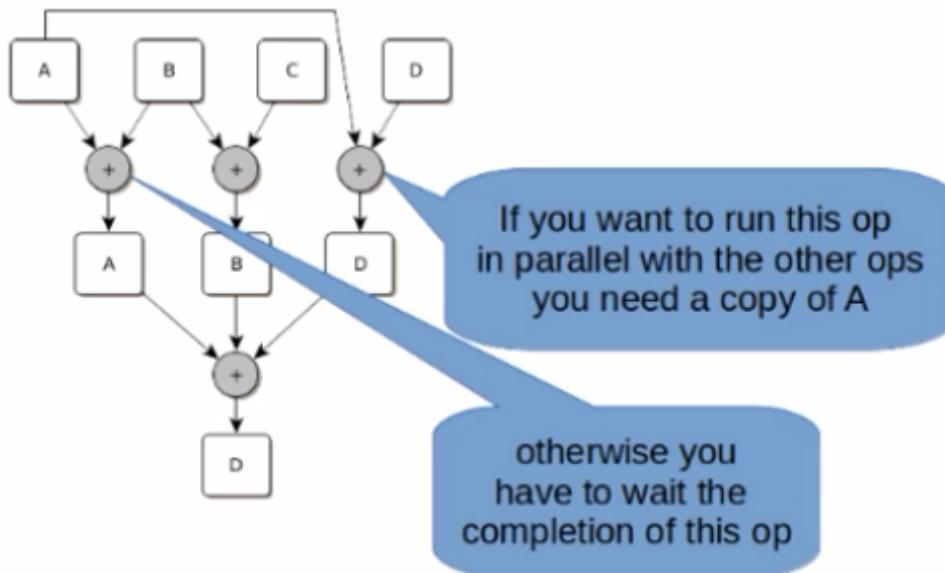
Possiamo provare ad immaginare altre versioni di questa applicazione, potremmo utilizzare direttamente una farm in cui l'mitter è specializzato e può leggere dal disco e fare lo scheduling ad uno dei worker e poi ognuno dei worker può applicare il parallel for ad ognuno dei task che riceve.

Macro Data Flow

Data Flow: la parallelizzazione arriva dal data flow graph, i nodi sono operazioni e gli archi dipendono dai dati, non abbiamo una singola istruzione ma un blocco di istruzioni.

Il Macro Data Flow: è un data flow dove i nodi sono funzioni.

Ad esempio possiamo avere questo grafo di data flow che mi dice quali operazioni devono essere svolte e in che ordine.



Tipicamente in questo macro data flow vogliamo riutilizzare le stesse locazioni di memoria per ridurre il consumo di memoria e questo in alcuni casi può essere un problema.

Supponiamo di avere il grafo della figura sopra, in questo caso dato che vogliamo fare delle computazioni in place dobbiamo stare attenti all'anti dependancy e alla sincronizzazione, se faccio alcune operazioni in alcuni casi devo attendere oppure dobbiamo avere una copia temporanea del dato in modo che comunque posso accedere all'informazione che mi serve.

Quindi queste dipendenze sono importanti e dobbiamo provare a calcolare tutti i task che possono essere calcolati in parallelo prima possibile.

In fastflow il macro data flow è implementato con una pipeline 2 stage, il primo stage genera il grafo poi il secondo stage è un pattern master worker in cui il worker calcola l'operazione che devo fare mentre lo scheduler prende in considerazione le dipendenze e schedula le varie operazioni.

Esempio:

```
// X = X+Y
void SUM(long *X, long *Y, size_t size);
// Z = X*Y
void MUL(long *X, long *Y, long *Z, size_t size);

{ // A = A+B
    const param_info _1={&A, INPUT};
    const param_info _2={&B, INPUT};
    const param_info _3={&A, OUTPUT};
    std::vector<param_info> P={_1,_2,_3};
    mdf->AddTask(P, SUM, &A, &B, size);
}
{ // C = A*B
    const param_info _1={&A, INPUT};
    const param_info _2={&B, INPUT};
    const param_info _3={&C, OUTPUT};
    std::vector<param_info> P={_1,_2,_3};
    mdf->AddTask(P, MUL, &A, &B, &C, size);
}
```

Quando aggiungiamo il task nel macro data flow stiamo specificando che deve essere eseguita la funzione SUM, poi indichiamo quali sono le dipendenze che abbiamo e i dati in input (A e B).

Nel secondo addTask specifichiamo anche la C che è la locazione in cui vogliamo memorizzare il risultato.

Un esempio di esecuzione di questo Macro Data Flow pattern è l'algoritmo di Strassen.

Tutto quello detto fino ad ora è implementato con una pipeline formata da due stage, come già detto.

I workers sono anonimi, ricevono in input i dati da calcolare e la funzione che devono calcolare, poi usano i dati all'interno della funzione.

Il business logic code è abbastanza intricato anche se il funzionamento generale del pattern non è troppo complicato.

Divide & Conquer

In questo caso vogliamo parallelizzare la ricorsione, qua dobbiamo fornire.

- Una funzione che dato un input produce 2 o più output
- Una funzione che è in grado di combinare ovvero dati due dati è in grado di unirli

L'interfaccia della libreria FastFlow, il pattern è chiamato `dived_f_t`.

Questo pattern è implementato come una computazione master worker, abbiamo un thread pool con uno scheduler, ognuno dei worker controlla se vale una certa condizione, poi va avanti o si ferma.

Esempio di esecuzione di Fibonacci con il Divide e Conquer di FastFlow. Abbiamo una funzione che mi controlla la condizione, una che fa la somma finale, una per la divisione dei dati e una per il caso base.

```
#include <iostream>
#include <functional>
#include <vector>
#include <ff/ff.hpp>
#include <ff/dc.hpp>
using namespace ff;

using namespace std;

/*
 * Operand and Result are just integers
 */

/*
 * Divide Function: recursively compute n-1 and n-2
 */
void divide(const unsigned int &op, std::vector<unsigned int> &subops)
{
    subops.push_back(op-1);
    subops.push_back(op-2);
}

/*
 * Base Case
 */
void seq(const unsigned int &op, unsigned int &res)
```

```
{  
    res=1;  
}  
  
/*  
 * Combine function  
 */  
void combine(vector<unsigned int>& res, unsigned int &ret)  
{  
    ret=res[0]+res[1];  
}  
  
/*  
 * Condition for base case  
 */  
bool cond(const unsigned int &op)  
{  
    return (op<=2);  
}  
  
  
int main(int argc, char *argv[]){  
  
    if(argc<3){  
        fprintf(stderr,"Usage: %s <N> <pardegree>\n", argv[0]);  
        exit(-1);  
    }  
    unsigned int start=atoi(argv[1]);  
    int nwork=atoi(argv[2]);  
  
    unsigned int res;  
    //Lambda version just for testing it  
    ff_DC<unsigned int, unsigned int> dac(  
        [](const unsigned int &op, std::vector<unsigned int>  
&subops){  
            subops.push_back(op-1);  
            subops.push_back(op-2);  
        },  
        [](vector<unsigned int>& res, unsigned int &ret){  
            ret=res[0]+res[1];  
        },  
        [](const unsigned int &op, unsigned int &res){  
            res=1;  
        },  
        [](const unsigned int &op){  
            return (op<=2);  
        }  
    );  
    dac(start, nwork, res);  
    cout<<res<<endl;  
}
```

```
        },
        start,
        res,
        nwork
    );

    if (dac.run_and_wait_end()<0) {
        error("running dac\n");
        return -1;
    }
    printf("Result: %d\n",res);
}
```

Un esempio più complicato è il Merge Sort, anche in questo caso viene definito il pattern divide e conquer con le varie funzioni necessarie.

La parte sequenziale del Merge Sort è eseguita utilizzando una funzione specializzata. In questo caso nella fase di unione dei risultati facciamo il merge degli array che sono stati ordinati in modo da creare un array finale ordinato.

In questo caso utilizziamo anche un CUTOFF, ogni volta che dividiamo spendiamo del tempo senza fare niente, poi avviamo il sort.

Il cutoff serve per bloccare la ricorsione arrivati ad un certo punto, l'albero viene tagliato ad un certo livello e poi dopo quel livello ci fermiamo con la ricorsione e applichiamo la ricorsione in modo sequenziale.

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
#include <cstring>
#include <ff/ff.hpp>
#include <ff/dc.hpp>
using namespace ff;
using namespace std;
#define CUTOFF 2000

/* -----
//maximum value for arrays elements
const int MAX_NUM=99999.9;

static bool isArraySorted(int *a, int n) {
```

```
for(int i=1;i<n;i++)
    if(a[i]<a[i-1])
        return false;
return true;
}

static int *generateRandomArray(int n) {
    srand ((time(0)));
    int *numbers=new int[n];
    for(int i=0;i<n;i++)
        numbers[i]=(int) (rand()) / ((RAND_MAX/MAX_NUM));
    return numbers;
}
/* ----- */

// Operand and Results share the same format
struct ops{
    int *array=nullptr;           //array (to sort/sorted)
    int left=0;                  //Left index
    int right=0;                 //right index
};

typedef struct ops Operand;
typedef struct ops Result;

/*
 * The divide simply 'split' the array in two: the splitting is only Logical.
 * The recursion occur on the left and on the right part
 */
void divide(const Operand &op,std::vector<Operand> &subops)
{
    int mid=(op.left+op.right)/2;
    Operand a;
    a.array=op.array;
    a.left=op.left;
    a.right=mid;
    subops.push_back(a);

    Operand b;
    b.array=op.array;
    b.left=mid+1;
    b.right=op.right;
    subops.push_back(b);
}
```

```
/*
 * For the base case we resort to std::sort
 */
void seq(const Operand &op, Result &ret)
{
    std::sort(&(op.array[op.left]),&(op.array[op.right+1]));

    //the result is essentially the same of the operand
    ret=op;
}

/*
 * The Merge (Combine) function start from two ordered sub array and construct
the original one
 * It uses additional memory
 */
void mergeMS(vector<Result>&ress, Result &ret)
{
    //compute what is needed: array pointer, mid, ...
    int *a=ress[0].array;                      //get the array
    int mid=ress[0].right;                     //by construction
    int left=ress[0].left, right=ress[1].right;
    int size=right-left+1;
    int *tmp=new int[size];
    int i=left,j=mid+1;

    //merge in order
    for(int k=0;k<size;k++)
    {
        if(i<=mid && (j>right || a[i]<= a[j]))
        {
            tmp[k]=a[i];
            i++;
        }
        else
        {
            tmp[k]=a[j];
            j++;
        }
    }

    //copy back
    memcpy(a+left,tmp,size*sizeof(int));

    delete[] tmp;

    //build the result
}
```

```
    ret.array=a;
    ret.left=left;
    ret.right=right;
}

/*
 * Base case condition
 */
bool cond(const Operand &op)
{
    return (op.right-op.left<=CUTOFF);
}

int main(int argc, char *argv[])
{
    if(argc<2)
    {
        cerr << "Usage: "<<argv[0]<< " <num_elements> <num_workers>"<<endl;
        exit(-1);
    }
    std::function<void(const Operand&,vector<Operand>&)> div(divide);
    std::function <void(const Operand &,Result &)> sq(seq);
    std::function <void(vector<Result>&,Result &)> mergef(mergeMS);
    std::function<bool(const Operand &) > cf(cond);

    int num_elem=atoi(argv[1]);
    int nwork=atoi(argv[2]);
    //generate a random array
    int *numbers=generateRandomArray(num_elem);

    //build the operand
    Operand op;
    op.array=numbers;
    op.left=0;
    op.right=num_elem-1;
    Result res;
    ff_DC<Operand, Result> dac(div,mergef,sq,cf,op,res,nwork);
    long start_t=ffTime(START_TIME);
    if (dac.run_and_wait_end()<0) {
        error("running dac\n");
        return -1;
    }
    long end_t=ffTime(STOP_TIME);

    if(!isArraySorted(numbers,num_elem))
    {
        fprintf(stderr,"Error: array is not sorted!!\n");
        exit(-1);
    }
}
```

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 273

```
    }
    printf("Time (usecs): %ld\n",end_t-start_t);
    return 0;
}
```

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 273

Lezione 5

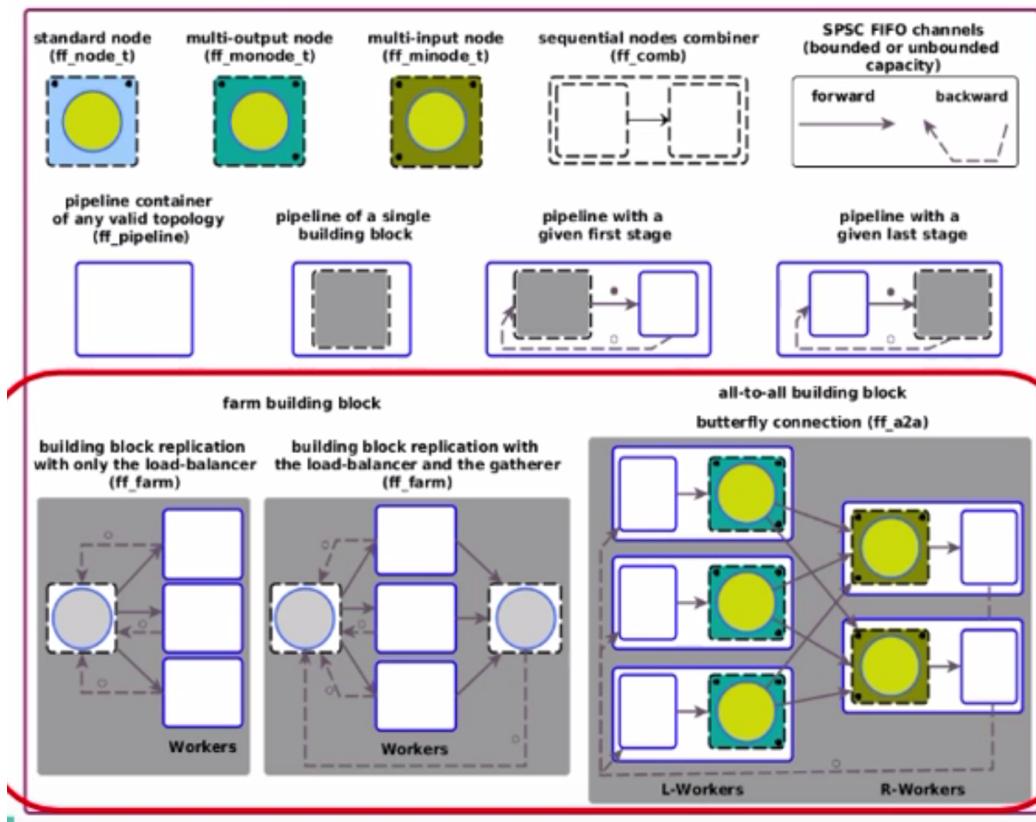
I data parallel pattern e gli stream pattern sono abbastanza per produrre una applicazione con FastFlow?

Possiamo creare combinazioni di pattern e usare delle metodologie differenti.

In Fastflow abbiamo il concetto di Building block che sono una astrazione ad un livello più alto delle idee basilari di concorrenza.

Abbiamo varie tipologie di building block:

- Building block sequenziali: sono dei nodi classici di FastFlow
- Channels: sono canali che possiamo utilizzare per mandare indietro le informazioni oppure in avanti
- Pipeline composition: il concetto di pipeline qua diventa un container che contiene anche altri building block. Staticamente possiamo vederla come un container in cui inserire vari building block connessi in modo data flow.
- Parallel nodes: in generale abbiamo un emitter e poi abbiamo altri worker che possono essere blocchi di un qualsiasi altro tipo, ad esempio una pipeline. Qua i vari nodi sono connessi con un channel che va in avanti ma in modo opzionale possiamo anche averne uno che porta i dati indietro.



Dentro ad una pipeline possiamo avere sia nodi sequenziali sia un building block parallelo.

Channel Building block: distinguiamo tra un channel “backward” e un channel “forward”. Tutti questi sono implementati con una single producer single consumer queue.

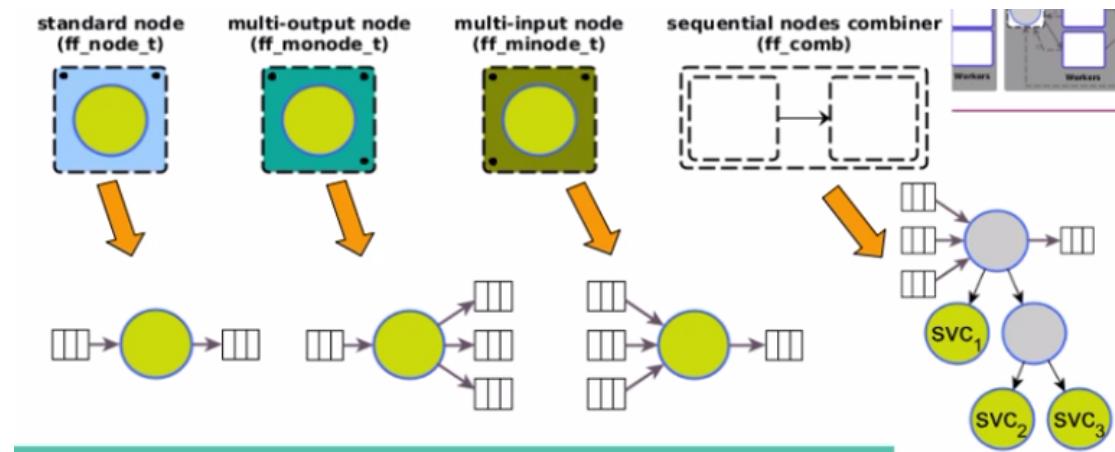
Questi canali possono essere:

- Non blocking: se provo a togliere qualcosa dal canale e il canale è vuoto il consumatore non aspetta
- Blocking: attesa attiva in un loop, consumo più energia e attendo che qualcosa compaia all'interno della coda.

Node Building block: abbiamo un singolo nodo che può avere più di un input, più di un output oppure può essere 1 input, 1 output.

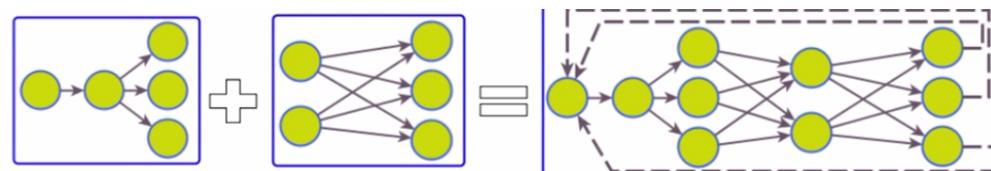
L'ultimo building block è nuovo è il combine e ci permette di mergiare due nodi sequenziali, senza riscrivere il codice li combiniamo insieme.

Alla fine viene prodotto un solo thread che contiene il codice dei due nodi che contiene.



Pipeline Building block: il building block lo chiamiamo usando **ff_pipeline**, questa è l'unica cosa che lo distingue da **ff_Pipe**. Qua abbiamo anche qualche metodo in più che ci permette di fare qualcosa in più.

L'idea di usare la pipeline come un container è che in generale puoi avere qualcosa che è stato creato in una 2 stage pipeline, poi abbiamo un'altra pipeline e alla fine vogliamo mergiarle insieme.



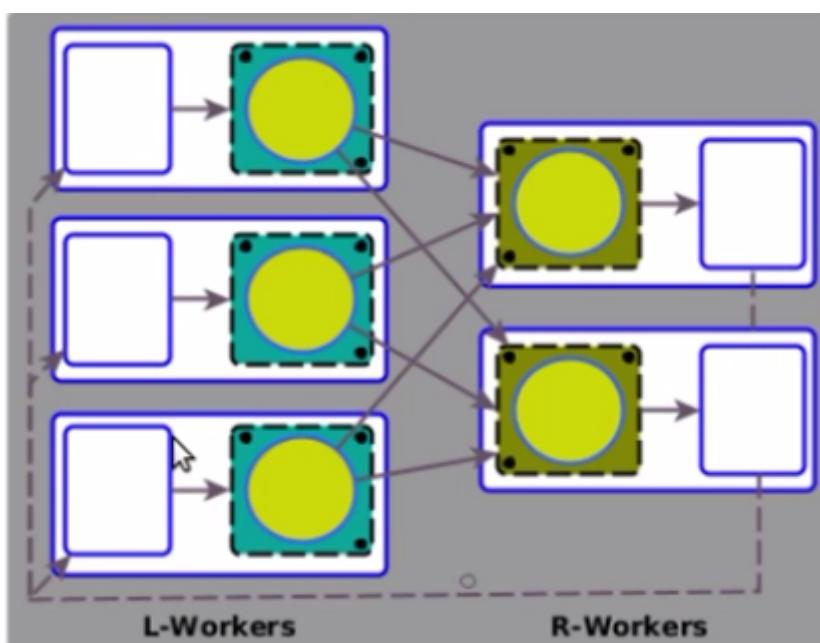
L'idea è di avere varie parti e combinarle insieme usando il building block della pipeline.

Staticamente possiamo aggiungere qualcosa alla pipeline, quando viene avviata viene eseguita in un modo data flow.

Farm Building block: niente di differente rispetto alla farm vista fino ad ora. C'è un master, uno scheduler, un pool di worker e può esserci un collector. Se vogliamo possiamo manualmente rimuovere il collector e possiamo aggiungere manualmente il feedback channel.

All to All building block: la cosa veramente nuova rispetto a quanto visto fino ad ora è questa e il combine. L'all to all è una multi functional replication, abbiamo una cosa che chiamiamo L-worker e una R-worker che possono contenere una qualsiasi altra cosa (pipeline, farm, un altro all to all).

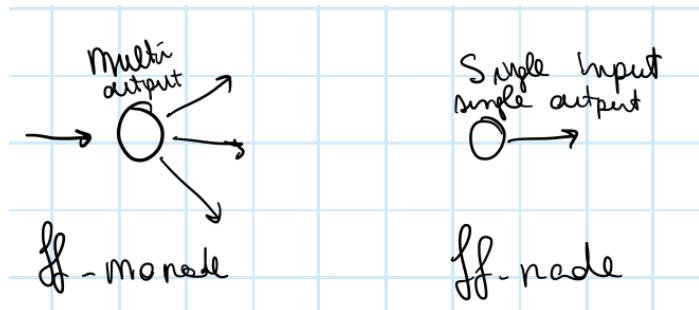
Nell'L-worker dobbiamo per forza avere come ultimo stage un multi output mentre invece nell'R-worker dobbiamo avere per forza come primo stage un multi input.



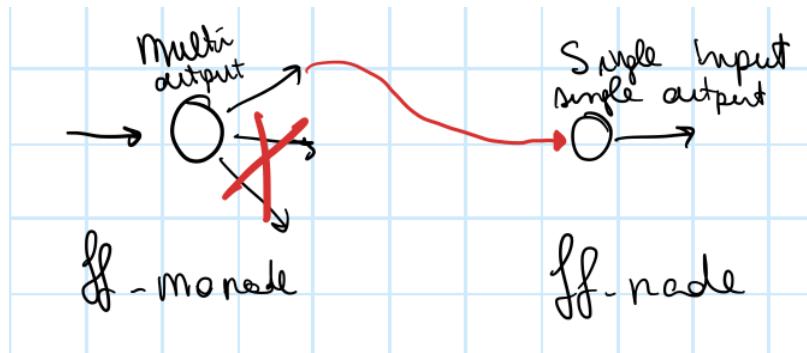
Possiamo anche avere dei feedback channel, quanti ne vogliamo, ma devono comunque andare dallo stage R a tutti gli altri stage, quindi devo avere un multi output nella R per implementare questo.

Per combinare tutti questi building block dobbiamo seguire tre regole:

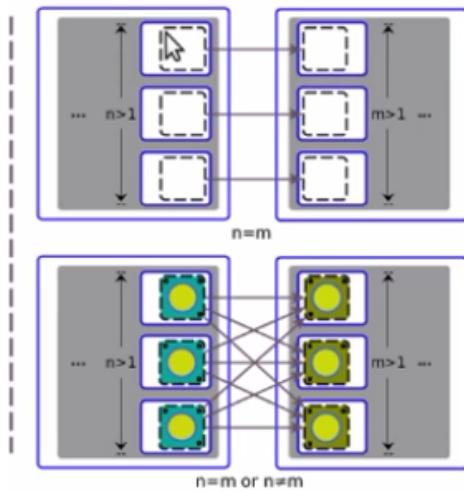
- Composition rule: due building block sequenziali possono essere connessi in pipeline senza considerare la loro cardinalità input/output, ad esempio posso connettere un multi output ad un single input, questo vuol dire che potrei avere una situazione come questa:



Lo possiamo avere perchè li mettiamo in una pipeline, quindi un canale sarà connesso al secondo stage mentre gli altri non verranno presi in considerazione.



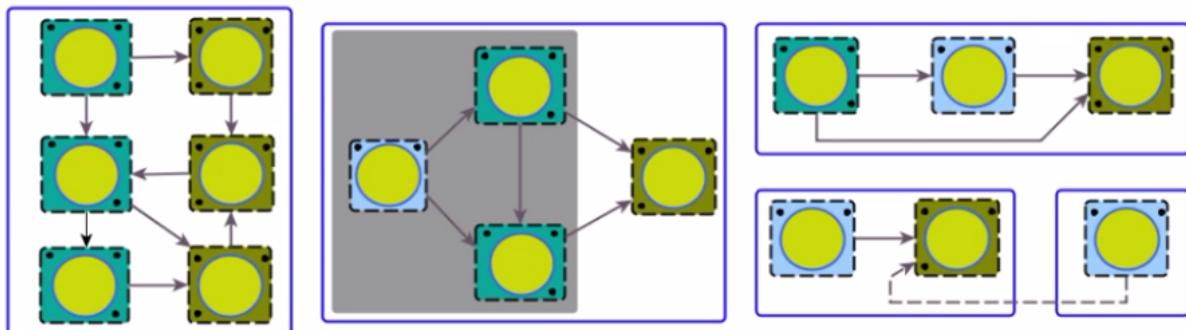
- Se vogliamo connettere un building block parallelo con uno sequenziale abbiamo bisogno di un multi output e di un multi input.
- Per connettere due building block paralleli: se abbiamo lo stesso numero di nodi sia a destra che a sinistra sono connessi direttamente. Quindi l'ultimo stage della pipeline può essere una qualsiasi cosa ma comunque sarà connesso con il primo stage della pipeline dell'altro building block.



Se il numero degli stage è n da una parte e dall'altra m , per connetterli abbiamo bisogno di un multi output da una parte e da un multi input dall'altra.

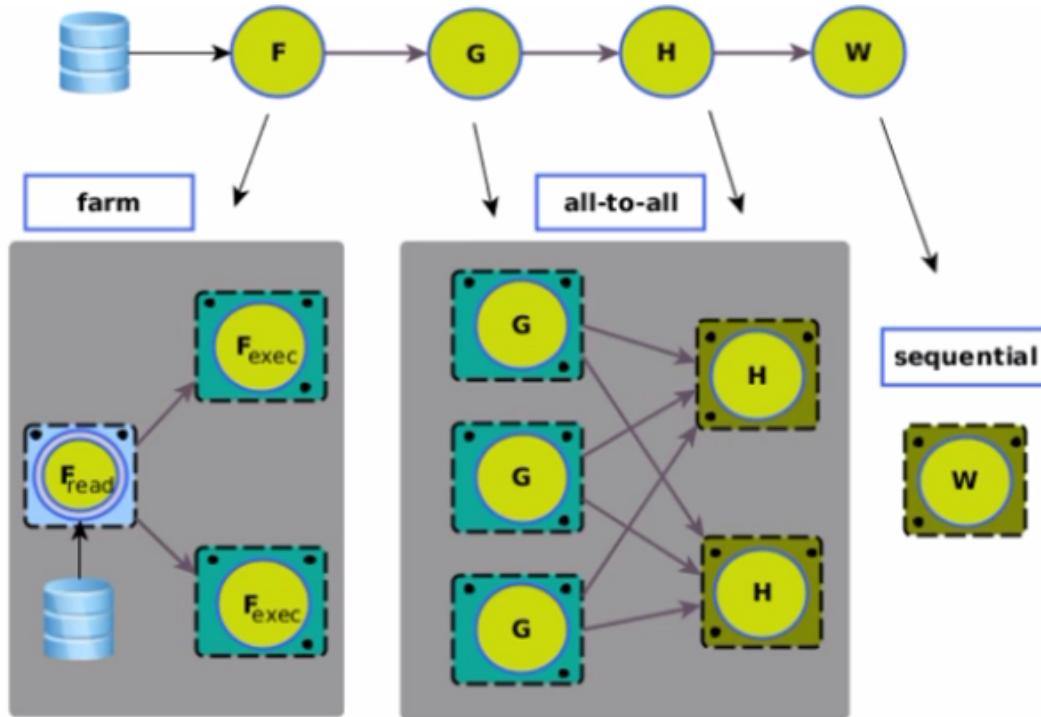
È possibile implementare tutti i grafi possibili usando i building block? No non è possibile.

Ad esempio i seguenti grafi non possono essere implementati perchè non ho regole che mi permettono la creazione e quindi dovrei farli a mano:



Tutti quelli più interessanti possono essere creati, con fastflow non posso fare qualsiasi cosa ma quello che probabilmente è più utile ed interessante.

Esempio di composizione dei building block:



Questa è una composizione di pipeline, in ogni nodo faccio una certa funzione, ogni nodo è un building block di fastflow, in uno è implementata una farm, in uno un all-to-all e l'ultimo è un nodo sequenziale.

Esempio di codice per creare un building block per la farm:

Vediamo che si usa ff_farm e non ff_Farm perchè creiamo un building block. Se voglio scrivere il mio emitter e il mio collector li devo passare quando creo la farm, emitter e collector possono essere multi input o anche multi output.

```
#include <ff/ff.hpp>
using namespace ff;
// user-defined ...
Emitter E; // ... emitter
Collector C; // ... collector
Worker W;
// creating the pool of workers
std::vector<ff_node*> V;
for(int i=0;i<nworkers;++i)
    V.push_back(new Worker(W));
ff_farm farm(V,E,C);
farm.set_scheduling_on-demand();
farm.cleanup_workers();
if (farm.run_and_wait_end()<0)
    error("running farm");
```

Codice per la creazione di un all to all, devo generare sia l'L-Worker che l'R-worker. La sintassi è simile a quella vista fino ad ora ma c'è da prestare attenzione perchè c'è qualche dettaglio da considerare ogni volta.

```
#include <ff/ff.hpp>
using namespace ff;
// user-defined workers
Worker1    W1; // standard node
Worker2    W2; // multi-output node
MultiInputHelper1 H1; // helper node
MultiInputHelper2 H2; // helper node

// creating the L-Workers
std::vector<ff_node*> V1;
for(int i=0;i<nworkers1;++i)
    V1.push_back(new ff_comb(H1,W1));

// creating the R-Workers
std::vector<ff_node*> V2;
for(int i=0;i<nworkers2;++i)
    V2.push_back(new ff_comb(H2,W2));

ff_a2a a2a;
a2a.add_firstset(V1, 0, true);
a2a.add_secondset(V2, true);
a2a.wrap_around();
if (a2a.run_and_wait_end()<0)
    error("running a2a");
```

Concurrency Graph Transformer

L'idea è di fornire un set di funzioni e funzionalità che permettano di modificare il concurrency graph dell'applicazione.

Creiamo la nostra applicazione utilizzando i building block, li connettiamo basandoci sulle regole, poi magari vogliamo modificare il concurrency graph, che di base è una pipeline o una multi pipeline, e lo vogliamo fare senza cambiare troppo il codice e la semantica della nostra applicazione e possibilmente riducendo il numero di nodi presenti all'interno dell'applicazione applicando combinazioni e ottimizzazioni.

Ci sono due metodi per fare questo:

- Usare una interface function chiamata `optimize_static()` che però ci permette di fare solamente una parte delle modifiche e delle ottimizzazioni che possiamo avere
- Usare una funzione che permette di modificare la forma del concurrency graph.

Se sei abbastanza esperto puoi provare a fare delle ottimizzazioni usando le funzioni che vengono offerte dal layer concurrency graph transformation.

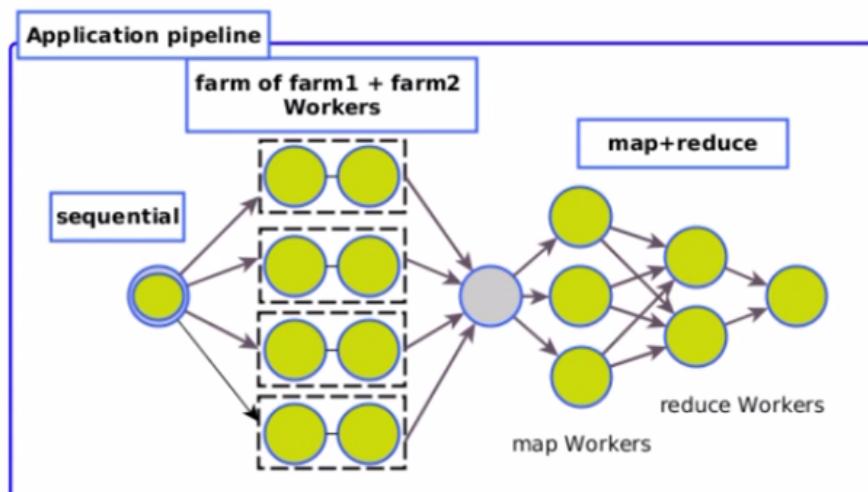
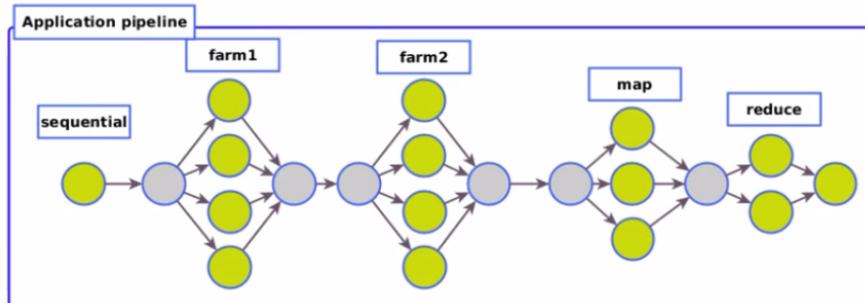
Le trasformazioni che vengono fornite riguardano:

- Fusione di varie map in una singola
- Riduzione del numero di nodi
- Combinare varie farm
- Introduzione dell'All to all building block

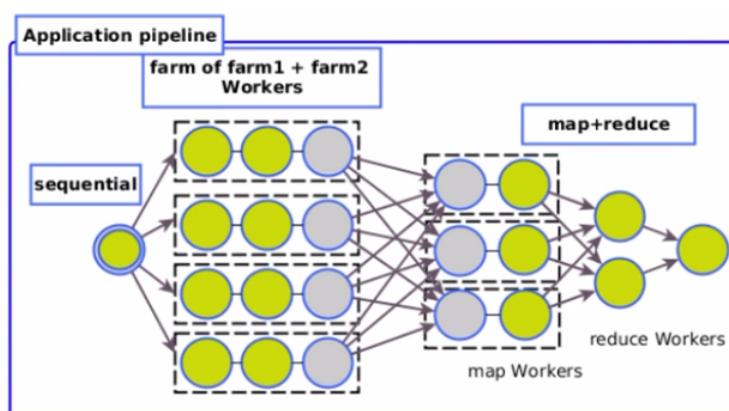
Ci sono anche altre cose che possono essere aggiunte in modo automatico per ottimizzare.

Ovviamente una ottimizzazione è più di una semplice trasformazione perchè si suppone che una ottimizzazione vada a migliorare le performance o il consumo di energia, ad esempio.

Supponiamo di avere una pipeline e vogliamo, senza cambiare il codice, applicare una trasformazione in modo che i worker delle farm vengano uniti insieme. Il codice delle farm non deve essere modificato.



Poi magari vorrei poter fare un'altra trasformazione e rimuovere le possibili bottleneck.



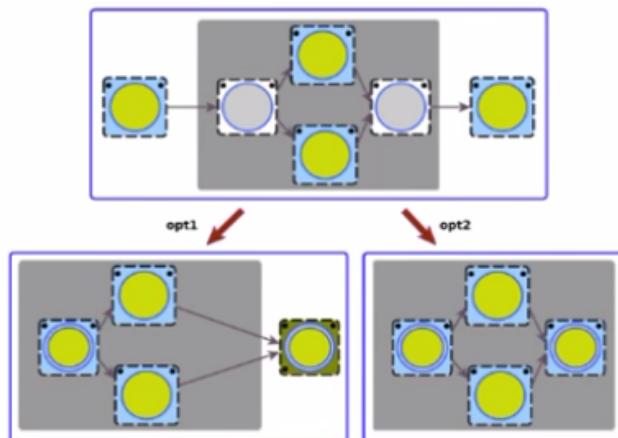
Queste sono trasformazioni che possono essere fatte anche in automatico, perchè sappiamo quali sono i nodi che sono solamente per il routing e quindi possiamo eliminarli.

Se vogliamo ridurre il numero di nodi all'interno dell'applicazione possiamo ridurre l'optimize_static.

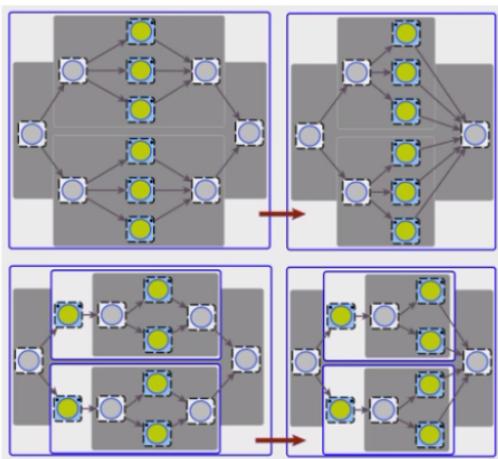
Una cosa importante è che un farm building block come minimo ha almeno un emitter e un pool di worker.

È impossibile rimuovere l'Emitter, possiamo fare un trick e fondere l'Emitter con un altro nodo ma non è possibile rimuoverlo del tutto.

Possiamo per esempio unire l'Emitter con il collector perchè sto facendo un merge con un altro nodo.



Un'altra riduzione dei nodi l'abbiamo quando abbiamo una farm in un'altra farm, abbiamo all'inizio molti nodi che in principio possono essere rimossi:

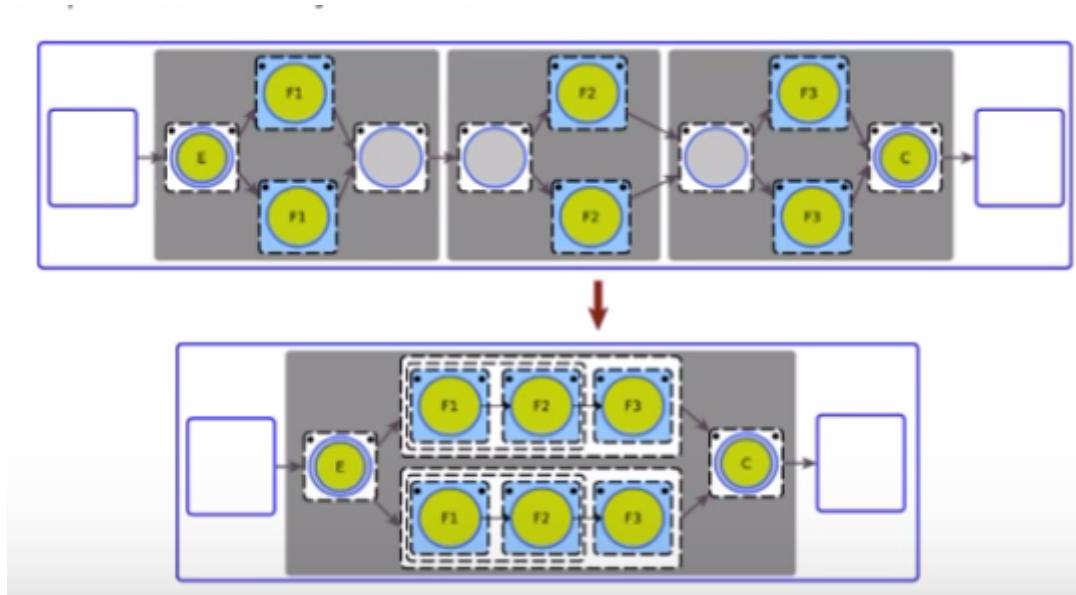


In alcuni casi i collector non posso rimuoverli perché sono connessi con altri, alcuni però li posso rimuovere perché sono messi di default e quindi non sono sempre necessari.

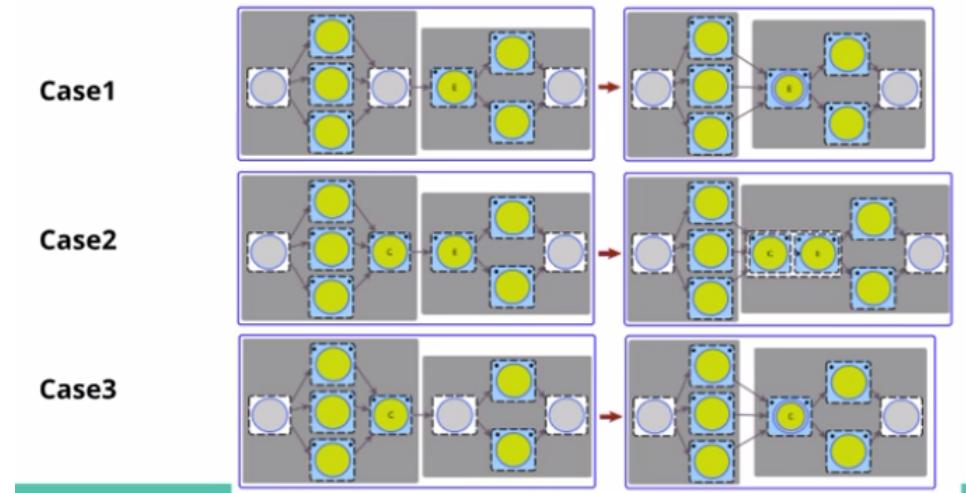
Il problema è che se il collector è qualcosa che all'interno è “verde”, quindi utile, ovviamente non deve essere eliminato dall'ottimizzatore, questo è possibile perché capiamo se quello specifico collector è stato ridefinito o no.

Un'altra ottimizzazione possibile è la **“Farm Fusion”**, in questo caso abbiamo più di una farm in pipeline e ognuna ha lo stesso numero di worker.

Dato che hanno lo stesso numero di worker e dato che il collector e l'mitter sono di default allora possiamo ridurre questo ad una sola farm in cui i worker vengono uniti in una sola funzione.



Un altro caso che può essere trasformato in automatico è il “Farm Combination” in cui abbiamo due farm in pipeline senza però avere regole particolari sul parallelism degree e sui nodi emitter/collector.



Sempre parlando di Farm Combine c’è da considerare anche il caso in cui abbiamo una ordered farm (quelle fino ad ora sono classiche), cosa succede quando abbiamo una ordered farm e dobbiamo mantenere l’ordine?

Il caso più complesso ce l’abbiamo quando non possiamo rimuovere il collector della prima farm perché preserva l’ordine.

Altra ottimizzazione possibile: **introduzione dell’All to All**.

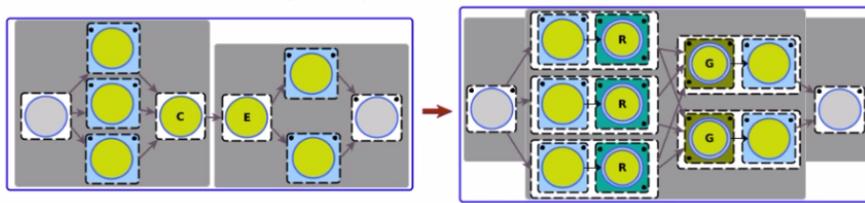
Abbiamo una pipeline di farm, in ognuna abbiamo un numero differente di task, quindi in principio non possiamo fonderla ma possiamo creare un all-to-all. Possiamo fare in modo che ogni volta il sistema mi trasforma queste due farm in una sola farm.

Dato che il numero di worker sono differenti dobbiamo trasformare la quantità di output di ognuno dei worker.

La cosa più complessa con l'all to all: abbiamo due farm e vogliamo rimuovere alcuni dei thread, C ed E.

Sappiamo che nella prima soluzione abbiamo un problema e vogliamo rimuovere C ed E, non possiamo rimuoverle semplicemente replicandole perché magari i dati devono arrivare in ordine o basandosi su una chiave.

Quindi questa funzione mi permette di introdurre questa trasformazione e l'utente però deve indicare cosa va eseguito in ognuno dei punti da eliminare in modo che poi il risultato finale venga prodotto in automatico. (Poco chiaro ma pure lui l'ha detto).



Ci sono alcune trasformazioni, tipo questa sopra che non possono essere applicate in modo automatico, c'è bisogno di chiamare una funzione specifica, ad esempio ff_pipeline combine_ farms (ce ne sono anche altre ma la logica e la semantica è più o meno la stessa).

Modificare il grafo è complicato e richiede delle analisi quindi fastflow ci fornisce i tool per consentire le trasformazioni, alcune in automatico altre no.

Lezione 6

Recap

I building block possono essere visti come mattoncini lego che possono essere composti in modo da creare codici sia semplici che più complessi. È un concetto abbastanza potente perché con pochi pezzi possiamo creare più o meno tutto.

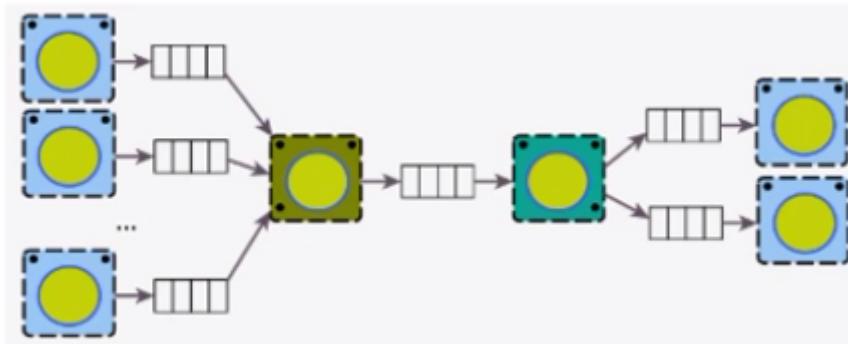
L'obiettivo è permettere all'utente di creare nuove cose perchè i pattern ci sono e sono utili ma in alcuni casi non ho un match diretto tra il pattern disponibile e quello che effettivamente vogliamo fare.

Non possiamo creare tutto ma cose complesse comunque le possiamo fare.

La cosa nuova nel building block è il combiner, nella libreria fastflow i nodi sono mappati 1:1 con i thread quindi in alcune situazioni possiamo avere più thread che core e questo in generale non è buono quindi il combiner mi permette di combinare nodi differenti insieme in modo da ridurre il numero dei nodi, non solo per ridurre il numero di thread ma anche per evitare che venga riscritto il codice di un nodo.

I parallel brick sono parallel farm in due modi differenti, nel building block abbiamo solamente un emitter mentre non abbiamo un collector di default.

Una cosa importante è il discorso sui nodi sequenziali, sono connessi utilizzando solamente FIFO Single producer single consumer queue, alla fine abbiamo un grafo di questo genere:



Poi abbiamo il combiner che ci fa mergiare i nodi che hanno multi output multi input node, per ogni input leggiamo il svc method.

Esempi di building block con il combine:

Nel main: a livello di building block dobbiamo definire per prima cosa la ff_pipeline, in questo caso eseguo i 5 stage in una pipeline reale.

Ora possiamo scoprire che un paio di questi stage sono molto leggeri e spendere tre thread per questi è troppo.

Quindi vogliamo una funzione che prende i due nodi sequenziali e li combina. In questo caso nel main combiniamo tre nodi e poi creiamo una nuova pipeline.

Riusciamo anche a raggiungere l'obiettivo di riutilizzare il codice.

In alcuni casi vogliamo giocare con la nostra applicazione quindi non sappiamo se è bene o no avere più concorrenza o meno concorrenza.

La funzione combine_nodes è un modo semplice per combinare i vari nodi. La funzione che mi crea il building block vero e proprio è la ff_comb che combina le cose in modo differente aggiungendo al combiner due istanze degli stage che vogliamo combinare.

```
#include <iostream>
#include <ff/ff.hpp>
using namespace ff;

long ntasks = 100000;
long worktime = 1*2400;

struct firstStage: ff_node_t<long> {
    long *svc(long*) {
        for(long i=1;i<=ntasks;++i) {
            ff_send_out((long*)i);
        }
        return EOS; // End-Of-Stream
    }
};

struct secondStage: ff_node_t<long> { // 2nd stage
    long *svc(long *t) {
        ticks_wait(worktime);
        return t;
    }
};
```

```
struct secondStage2: ff_node_t<long> { // 2nd stage
    long *svc(long *t) {
        ticks_wait(worktime);
        return t;
    }
};

struct secondStage3: ff_node_t<long> { // 2nd stage
    long *svc(long *t) {
        ticks_wait(worktime);
        return t;
    }
};

struct thirdStage: ff_node_t<long> { // 3rd stage
    long *svc(long *task) {
        ticks_wait(worktime);
        return GO_ON;
    }
};

int main() {
    firstStage _1;
    secondStage _2;
    secondStage2 _3;
    secondStage3 _4;
    thirdStage _5;

    unsigned long start=getusec();
    long *r;
    for(long i=1;i<=ntasks;++i) {
        r = _2.svc((long*)i);
        r = _3.svc(r);
        r = _4.svc(r);
        r = _5.svc(r);
    }
    unsigned long stop=getusec();
    std::cout << "TEST FOR Time = " << (stop-start) / 1000.0 << " ms\n";

    ff_pipeline pipe;
    pipe.add_stage(_1);
    pipe.add_stage(_2);
    pipe.add_stage(_3);
    pipe.add_stage(_4);
    pipe.add_stage(_5);
    pipe.run_and_wait_end();
    std::cout << "TEST PIPE Time = " << pipe.ffwTime() << " ms\n";

    ff_pipeline pipe2;
    auto comb = combine_nodes(_2, combine_nodes(_3, _4));
    pipe2.add_stage(_1);
    pipe2.add_stage(comb);
```

```
pipe2.add_stage(_5);
pipe2.run_and_wait_end();
std::cout << "TEST PIPE2 Time = " << pipe2.ffwTime() << " ms\n";

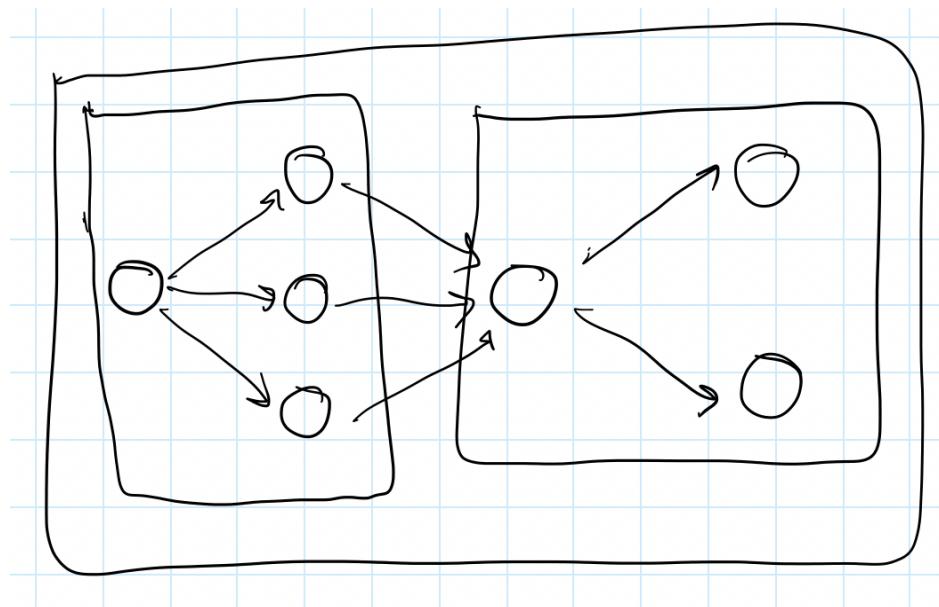
ff_pipeline pipe3;
pipe3.add_stage(_1);
pipe3.add_stage(new ff_comb(new secondStage, new secondStage2, true, true),
true);
pipe3.add_stage(new ff_comb(new secondStage3, new thirdStage, true, true),
true);
pipe3.run_and_wait_end();
std::cout << "TEST PIPE3 Time = " << pipe3.ffwTime() << " ms\n";

return 0;
}
```

La sintassi per la pipeline e per la farm è molto simile, la pipeline non è solamente un pattern data flow, i building block contenuti nella pipeline, vengono poi eseguiti in modo data flow ma la pipeline in realtà è un container, un blocco che contiene altri blocchi e che a sua volta può essere connessa con altri blocchi in pipeline.

L'all-to-all definisce due set di workers, abbiamo quello che chiamiamo L-Worker e l'R-Worker. Questi workers sono container a loro volta quindi posso metterci quello che voglio all'interno, l'ultimo stage della L però deve essere un multi output e il primo stage della R deve essere un multi input channel.

Dal punto di vista della semantica, un all to all è simile ad avere due farm in pipeline:



Esempio di utilizzo di un building block all to all per risolvere il conteggio dei numeri primi in un range.

Per creare un all to all bisogna usare il costruttore ff_a2a.

Allocchiamo i vari LWorkers e i vari RWorkers, in principio puoi aggiungere uno standard node e il runtime system lo trasformerà in un multi input e in un multi output. In questo caso viene specificato che sono multi output e multi input.

Il codice è simile a quello che abbiamo già scritto tranne per il fatto che in questo caso quando riceviamo range differenti e creiamo il risultato parziale, prima di inviare i risultati, devo terminare solamente se ricevo da un certo canale ma prima di terminare devo ordinare i dati che ho. Quindi in questo caso eseguo una computazione finale per avere i dati ordinati.

```
#include <cmath>
#include <string>
#include <vector>
#include <iostream>
#include <ff/ff.hpp>
#include <ff/all2all.hpp>
using namespace ff;

using ull = unsigned long long;
```

```
// see http://en.wikipedia.org/wiki/Primality_test
static inline bool is_prime(ull n) {
    if (n <= 3) return n > 1; // 1 is not prime !

    if (n % 2 == 0 || n % 3 == 0) return false;

    for (ull i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
    }
    return true;
}

struct Task_t {
    Task_t(ull n1, ull n2):n1(n1),n2(n2) {}
    const ull n1, n2;
};

// generates the numbers
struct L_Worker: ff_monode_t<Task_t> {

    L_Worker(ull n1, ull n2)
        : n1(n1),n2(n2) {}

    Task_t *svc(Task_t *) {

        const int nw = get_num_outchannels();
        const size_t size = (n2 - n1) / nw;
        ssize_t more = (n2-n1) % nw;
        ull start = n1, stop = n1;

        for(int i=0; i<nw; ++i) {
            start = stop;
            stop = start + size + (more>0 ? 1:0);
            --more;

            Task_t *task = new Task_t(start, stop);
            ff_send_out_to(task, i);
        }
        return EOS;
    }
    ull n1,n2;
};

struct R_Worker: ff_minode_t<Task_t> {
    Task_t *svc(Task_t *in) {
        ull n1 = in->n1, n2 = in->n2;
        ull prime;
        while( (prime=n1++) < n2 ) if (is_prime(prime))
results.push_back(prime);
    }
}
```

```
//std::cout << "Worker" << get_my_id() << " found " << V.size() << " primes\n";
    return GO_ON;
}
void svc_end() {
    std::sort(results.begin(), results.end());

#if 0
    printf("Worker%d:\n", get_my_id());
    for(size_t i=0;i<results.size();++i)
        printf("%lld ", results[i]);
    printf("\n");
#endif
}
std::vector<ull> results;
};

int main(int argc, char *argv[]) {
if (argc<5) {
    std::cerr << "use: " << argv[0] << " number1 number2 Lworkers RWorkers [print=off|on]\n";
    return -1;
}
ull n1          = std::stoll(argv[1]);
ull n2          = std::stoll(argv[2]);
const size_t Lw = std::stol(argv[3]);
const size_t Rw = std::stol(argv[4]);
bool print_primes = false;
if (argc >= 6) print_primes = (std::string(argv[5]) == "on");

ffTime(START_TIME);

const size_t size = (n2 - n1) / Lw;
ssize_t more = (n2-n1) % Lw;
ull start = n1, stop = n1;

std::vector<ff_node*> LW;
std::vector<ff_node*> RW;
for(size_t i=0; i<Lw; ++i) {
    start = stop;
    stop  = start + size + (more>0 ? 1:0);
    --more;
    LW.push_back(new L_Worker(start, stop));
}
for(size_t i=0;i<Rw;++i)
    RW.push_back(new R_Worker);

ff_a2a a2a;
a2a.add_firstset(LW);
```

```
a2a.add_secondset(RW);

if (a2a.run_and_wait_end()<0) {
    error("running a2a\n");
    return -1;
}

std::vector<ull> results;
results.reserve( (size_t)(n2-n1)/log(n1) );
for(size_t i=0;i<Rw;++i) {
    R_Worker* r = reinterpret_cast<R_Worker*>(RW[i]);
    if (r->results.size())
        results.insert(std::upper_bound(results.begin(), results.end(),
r->results[0]),
                       r->results.begin(), r->results.end());
}
ffTime(STOP_TIME);

// printing obtained results
const size_t n = results.size();
std::cout << "Found " << n << " primes\n";
if (print_primes) {
    for(size_t i=0;i<n;++i)
        std::cout << results[i] << " ";
    std::cout << "\n\n";
}
std::cout << "Time: " << ffTime(GET_TIME) << " (ms)\n";
std::cout << "A2A Time: " << a2a.ffTime() << " (ms)\n";

return 0;
}
```

Come vengono usati i building block per produrre uno streaming DSL?

Windflow è una libreria per il Data stream processing sviluppata utilizzando fastflow.

Questa libreria permette l'utilizzo di vari pattern basilari e alcuni parallel pattern specializzati per processare dati in streaming.

Windflow usa tantissimo i building block di fastflow, in particolare Pipeline, farm e A2A.

Cosa fa questa libreria?

Permette di generare uno stream leggendo da disco/memoria/qualsiasi altra cosa, tutti gli elementi dello stream hanno lo stesso tipo.

Poi ci sono un numero di operatori basilari che possono essere visti come un nodo singolo, ad esempio abbiamo l'operatore Map che prende il dato T1 e produce il dato T2, poi la Filter in cui abbiamo qualcosa in input e poi in output possiamo anche non avere niente perchè devo controllare una certa condizione che deve verificarsi.

Abbiamo la Flat Map che è simile alla Map ma l'output può essere più di uno, ad esempio per un elemento in input posso produrre un array o vari elementi in output.

Poi c'è l'accumulator che svolge la reduce sulla tuple e produce il risultato.

Questi sono gli operatori più semplici, poi ci sono quelli più complessi che permettono di fare operazioni più particolari.

Questi operatori (map, filter, flatMap, accumulator e sink) sono implementati come building block All to All.

In questo caso l'all to all è utilizzato come un container di building block per creare la rete.

Per creare questa rete dobbiamo connettere i vari building block in pipeline, a differenza di quanto visto fino ad ora, visto che stiamo utilizzando un all to all, abbiamo varie pipeline, il grafo è un multi pipe dove le varie pipeline lavorano in parallelo e non sono indipendenti tra loro perchè in alcuni punti ci sono delle connessioni tra le varie pipeline. Le connessioni, che possono essere lineari o shuffled, dipendono dal tipo di pattern, se il parallelism degree è lo stesso possiamo forzare lo shuffle e possiamo avere una pipeline lineare, questo dipende dalle richieste che abbiamo e da cosa dobbiamo fare (ad esempio del nodo A abbiamo due istanze quindi due repliche, per B abbiamo tre repliche quindi non matcha con il numero di A).

Come viene implementato il runtime?

Utilizziamo un All-To-All che di base è una matrioska, queste matrioska sono trasparenti all'utente perchè questo è solamente il runtime.

Quando c'è un match della cardinality abbiamo un match lineare altrimenti uno shuffle.

Un esempio:

Abbiamo una pipeline, poi creiamo una source con un parallelism degree pari a 2.



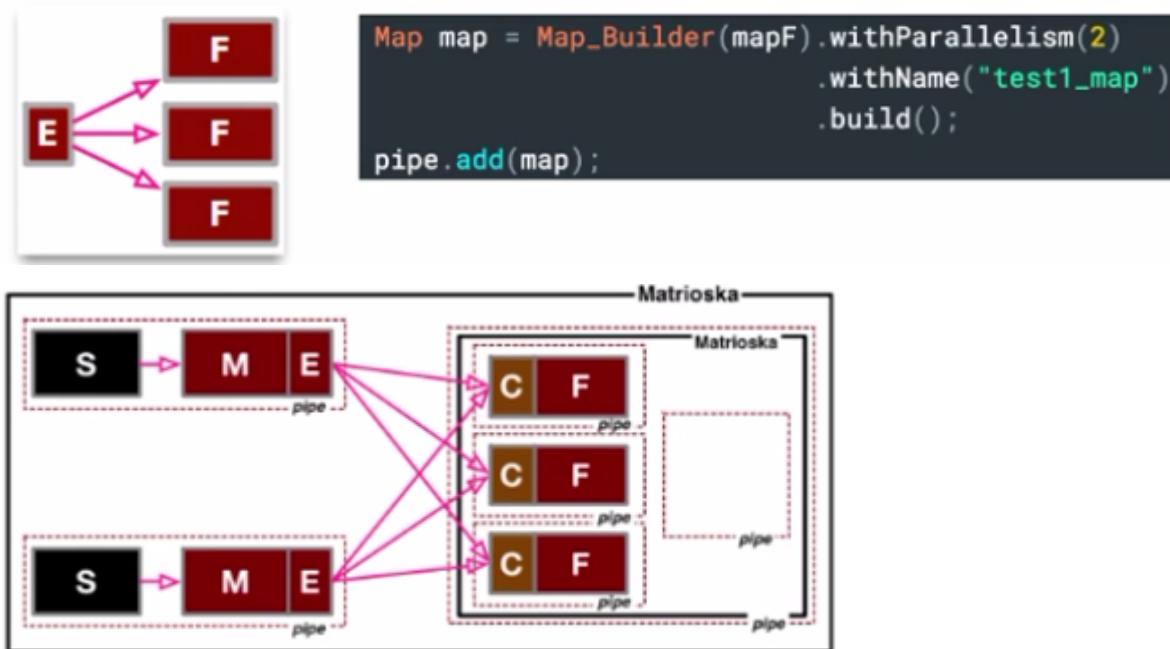
Poi vogliamo aggiungere una Map Filter, anche in questo caso con un parallelism degree pari a 2:

C'è un match e quindi la map viene connessa subito con il "source", rimane la singola matrioska e la Map viene aggiunta nella parte a sinistra della matrioska. Come viene aggiunta? Il runtime della pipeline crea la pipeline e la replica due volte.



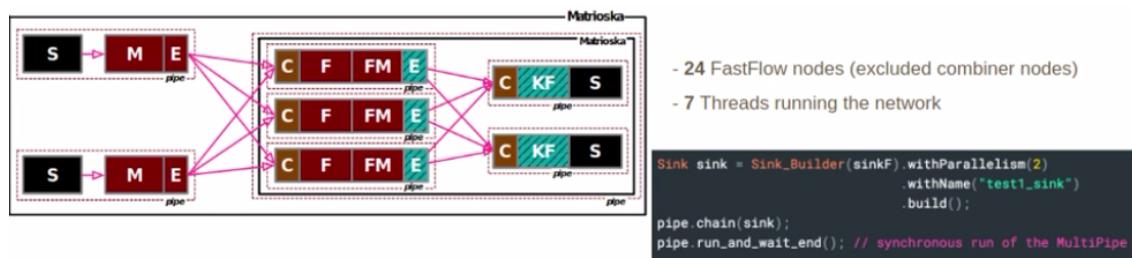
Ora supponiamo di voler aggiungere anche una filter, per farlo dobbiamo introdurre il fatto che questo nuovo operatore è eseguito nella seconda parte della map.

Devo anche aggiungere un emitter e un collector perchè mi serve che la parte sinistra diventi un multi output e la parte con la filter un multi input.



Quando voglio aggiungere qualcosa lo metto nella parte dell'R-Worker, ci sono però dei casi in cui devo fare il chaining per ridurre il numero di worker. Se ho lo stesso parallelism degree posso fare un merge in modo che possa avere una pipeline non di tre stage ma una pipeline con uno stage con tre cose combinate all'interno. Potrei voler fare questo perchè magari non voglio usare un thread per ognuno degli stage.

Quindi posso fare il merge di qualcosa nella Left side o nella right side. Alla fine possiamo ottenere una cosa del genere:



Alla fine dobbiamo aggiungere un Sink perchè la matrioska è completa solamente se abbiamo almeno un source e almeno un Sink.

La cosa interessante è che creo un complex network ma non un network standard che posso usare con una singola pipeline.

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 300

Usando il livello dei building block posso produrre un network così complicato senza che l'utente possa conoscere tutto il livello delle matrioska e delle difficoltà dei blocchi.

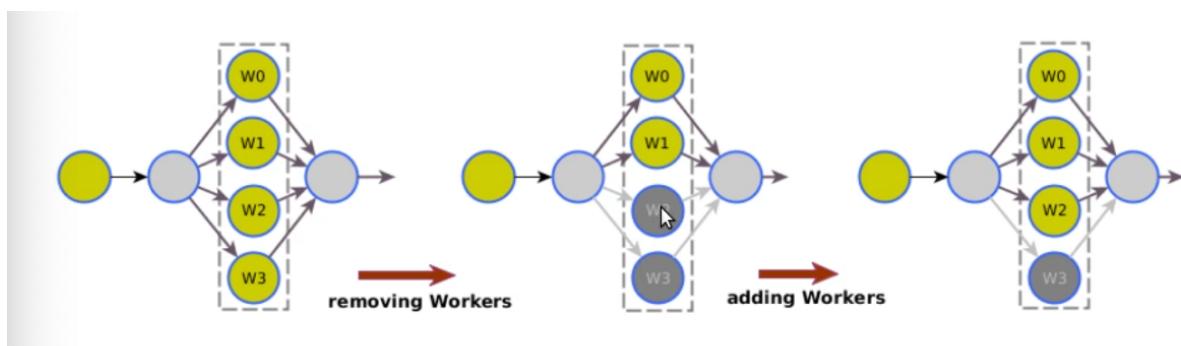
Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 300

Lezione 7

Throttling del numero di worker del building block (della farm): con throttling si intende la possibilità di cambiare dinamicamente il numero di thread di una applicazione parallela. Le motivazioni per cui modificare il numero di thread dinamicamente sono varie, tipicamente è per ridurre il consumo di energia perché se ho più thread ho delle performance migliori ma ho anche un consumo maggiore.

Oppure potrei voler aumentare il numero di thread quando le richieste verso un web server sono maggiori, ad esempio un sito che di giorno riceve molte richieste e la notte di meno, è inutile usare lo stesso numero di thread.

Quindi potrei far partire la mia applicazione con un certo numero di worker e poi aumentare o diminuire il numero di workers per arrivare ad una soluzione migliore.



In fastflow i worker possono essere aggiunti e rimossi dinamicamente, i nodi in realtà (e quindi i thread) non vengono davvero rimossi e poi aggiunti nuovamente. Questi nodi vengono messi in sleep (continuano ad avere un TID) con un segnale e poi quando qualcuno (l'mitter o un altro nodo) si accorge che serve più performance allora chiede di eseguire un restart dei nodi in sleep.

Emitter e collector sono sempre attivi, l'unica cosa che possiamo fare è aumentare o diminuire il numero di worker in una farm.

Dato che i worker non sono davvero creati e distrutti, all'inizio posso avviare il numero massimo di worker che posso creare (un numero maggiore dei core) e poi, dopo che li abbiamo avviati, posso metterne alcuni in sleep ed eventualmente riavviarli.

In FastFlow quando un nodo sequenziale riceve l'EOS, il thread è terminato e poi l'EOS è propagato come in una pipeline, questo è vero se il freezing flag non è settato, se invece è settato i thread non vengono terminati ma vengono solamente messi in sleep. Posso mandare un EOS al nodo, se il nodo è avviato con il freeze (avviare con run_then_freeze e poi usare wait_freezing) va in sleep e l'EOS viene mandato al prossimo stage, se il prossimo stage non ha il freezing flag abilitato allora questo stage viene terminato.

Dopo che il thread va in sleep posso poi risvegliarlo con il metodo thaw che è come una signal nella libreria pthread.

Gestire lo sleep del thread con l'EOS può essere difficile se vogliamo solamente bloccare un singolo worker.

C'è un altro valore che può essere mandato al prossimo stage che ha un significato simile a quello dell'EOS, si tratta di GO_OUT che ha la stessa semantica dell'EOS ma non è propagato al prossimo stage della pipeline o al collector della farm. Quindi quando GO_OUT viene ricevuto da un thread che ha il flag del freeze, il thread andrà in sleep e il messaggio GO_OUT viene distrutto.

Un esempio: abbiamo nel main la creazione del building block della farm, abbiamo un emitter ridefinito dall'utente, poi una pool di workers, l'Emitter deve provare a stoppare e a far ripartire i thread dei worker.

L'Emitter prende come parametro ff_loadbalancer che è una struttura interna che serve per schedulare i dati verso i workers.

L'Emitter nell'svc_init() mette il freezing flag a tutti i workers e poi manda a tutti i workers il messaggio GO_OUT. Vogliamo attendere in modo sincrono che vadano davvero a dormire.

Poi l'Emitter esegue il metodo svc che è la vera e propria applicazione e qua facciamo una iterazione in cui l'Emitter non fa nulla (è solo un test) e poi risveglia tutti i thread che erano in sleep usando la funzione thaw.

Poi vengono inviati dei task ai vari worker e poi i worker vengono mandati nuovamente in sleep.

```
#if !defined(FF_INITIAL_BARRIER)
// to run this test we need to be sure that the initial barrier is executed
#define FF_INITIAL_BARRIER
#endif
#include <vector>
#include <cstdio>
#include <ff/farm.hpp>

using namespace ff;

class Emitter: public ff_node {
protected:
    void stop_workers() {
        size_t nw = lb->getnworkers();
        for(size_t i=0; i<nw;++i) {
            lb->ff_send_out_to(GO_OUT, i);
        }
        for(size_t i=0;i<nw;++i) {
            lb->wait_freezing(i);
        }
    }
    void wakeup_workers(bool freeze=true) {
        for(size_t i=0;i<lb->getnworkers();++i)
            lb->thaw(i,freeze);
    }
public:
    Emitter(ff_loadbalancer *const lb):lb(lb) {}
```

```
int svc_init() {
    // set freezing flag to all workers
    for(size_t i=0;i<lb->getnworkers();++i)
        lb->freeze(i);
    stop_workers();

    return 0;
}
void *svc(void *) {

    for(int i=0;i<10; ++i) {
        printf("iter %d\n", i);
        usleep(i*10000);

        // restart all workers
        wakeup_workers();

        // do something here
        for(size_t j=0;j<lb->getnworkers();++j)
            lb->ff_send_out_to(new int(j), j);

        // put workers to sleep
        stop_workers();
    }
    // restart workers before sending EOS
    wakeup_workers(false);
    return EOS;
}

void svc_end() {
    printf("Emitter exiting\n");
}
private:
    ff_loadbalancer *const lb;
};

class Worker:public ff_node {
public:
    void *svc(void *t) {
        printf("worker %ld received %d\n", get_my_id(), *(int*)t);
        return GO_ON;
    }
    void svc_end() {
        printf("worker %ld going to sleep\n", get_my_id());
    }
};

int main(int argc, char *argv[]) {
```

```
int nworkers = 3;
if (argc>1) {
    if (argc!=2) {
        printf("use: %s nworkers\n", argv[0]);
        return -1;
    }

    nworkers = atoi(argv[1]);
}
ff_farm<> farm;
std::vector<ff_node*> w;
for(int i=0;i<nworkers;++i)
    w.push_back(new Worker);
farm.add_workers(w);
farm.add_emitter(new Emitter(farm.getlb()));

farm.run();
farm.getlb()->waitlb();

return 0;
}
```

Un altro esempio, in questo caso abbiamo una farm in cui l'Emitter genera una serie di numeri, poi vogliamo calcolare il quadrato di quei numeri e poi alla fine li sommiamo.

In questo caso l'Emitter funziona come un manager nel senso che quando l'elemento schedulato è 100 allora uno dei worker viene rimosso, quando ho schedulato 1000 elementi ne rimuovo un altro, quando ne ho schedulati 5000 faccio il restart dell'ultimo worker che ho bloccato e poi faccio la stessa cosa quando arrivo a schedulare il 10.000 elemento.

Ogni volta l'Emitter manda i dati ad uno dei nodi worker che è attualmente attivo.

```
#include <iostream>
#include <ff/ff.hpp>
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;

struct firstStage: ff_node_t<float> {
    firstStage(const size_t length):length(length) {}
```

```
    float* svc(float * ) {
        for(size_t i=0; i<length; ++i) {
            ff_send_out(new float(i));
        }
        return EOS;
    }
    const size_t length;
};

struct secondStage: ff_node_t<float> {
    float* svc(float * task) {
        float &t = *task;
        t = t*t;
        return task;
    }
};

struct thirdStage: ff_node_t<float> {
    float* svc(float * task) {
        float &t = *task;
        //std::cout<< "thirdStage received " << t << "\n";
        sum += t;
        delete task;
        return GO_ON;
    }
    void svc_end() { std::cout << "sum = " << sum << "\n"; }
    float sum = 0.0;
};

struct Emitter: ff_monode_t<float> {
    int svc_init() {
        active = get_num_outchannels();
        return 0;
    }
    float* svc(float* in) {
#if defined(THROTTLING)
        const unsigned nw = get_num_outchannels();

        if (nw == 1) return in;
        switch(cnt) {
        case 100: {
            std::cout << "REMOVING WORKER " << nw-1 << "\n";
            ff_send_out_to(GO_OUT, nw-1);
            --active;
        } break;
        case 1000: {
            if (nw>2) {
                std::cout << "REMOVING WORKER " << nw-2 << "\n";
                ff_send_out_to(GO_OUT, nw-2);
            }
        }
    }
}
```

```
        --active;
    }
} break;
case 5000: {
    if (nw > 2) {
        // to be sure the worker nw-2 went to sleep
        ff_monode::getlb()->wait_freezing(nw-2);
        std::cout << "RE-ADDING WORKER " << nw-2 << "\n";
        ff_monode::getlb()->thaw(nw-2, true);
        active++;
    }
} break;
case 10000: {
    // to be sure the worker nw-1 went to sleep
    ff_monode::getlb()->wait_freezing(nw-1);
    std::cout << "RE-ADDING WORKER " << nw-1 << "\n";
    ff_monode::getlb()->thaw(nw-1, true);
    active++;
} break;
default:
}
#endif
    ff_send_out_to(in, cnt % active);
    ++cnt;
    return GO_ON;
}

size_t cnt=0;
unsigned active=0;
};

int main(int argc, char *argv[]) {
    if (argc<3) {
        std::cerr << "use: " << argv[0] << " nworkers stream-length\n";
        return -1;
    }
    const size_t nworkers = std::stol(argv[1]);

    firstStage first(std::stol(argv[2]));
    thirdStage third;

    ff_farm farm;
    std::vector<ff_node*> W;
    for(size_t i=0;i<nworkers;++i)
        W.push_back(new secondStage);
    farm.add_workers(W);
    Emitter E;
    farm.add_emitter(&E);
    farm.add_collector(nullptr);
```

```
farm.cleanup_workers();

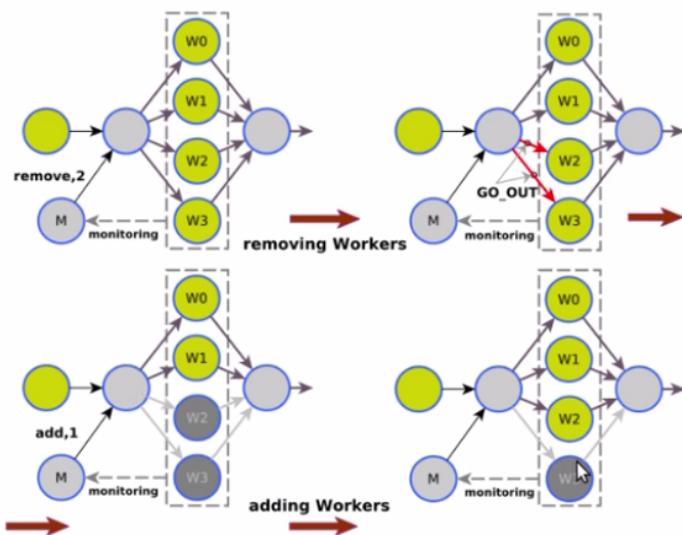
ff_pipeline pipe;
pipe.add_stage(first);
pipe.add_stage(farm);
pipe.add_stage(third);

ffTime(START_TIME);
if (pipe.run_then_freeze()<0) {
    error("running pipe");
    return -1;
}
pipe.wait();

ffTime(STOP_TIME);
std::cout << "Time: " << ffTime(GET_TIME) << "\n";
#if defined(TRACE_FASTFLOW)
    pipe.ffStats(std::cout);
#endif
return 0;
}
```

Possiamo complicare un po' il funzionamento di tutto questo metodo per la gestione dinamica del numero di worker attivi andando ad aggiungere un vero e proprio thread manager che sostituisce l'mitter nella gestione del numero di thread attivi o in sleep.

Questa nuova entità, chiamata Manager è collegata direttamente all'mitter e ha il compito di prendere delle informazioni dai worker (utilizzo cpu, tempo passato nel metodo SVC...). All'interno del manager poi viene generata una politica che mi indica in quali situazioni devo aggiungere o rimuovere dei worker dalla farm.

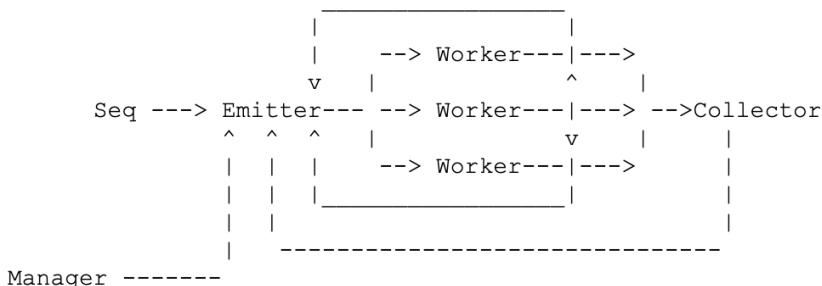


Quindi l'mitter non solo legge dall'input reale dello stage precedente ma anche dal manager, capisce cosa deve fare ed eventualmente manda un certo segnale ai vari worker.

In una applicazione possiamo avere più di una farm e poi un manager che prende le decisioni che poi l'mitter applicherà.

Un esempio che mostra il funzionamento di questo manager, ci troviamo in una rete di questo tipo:

```
pipe(seq, farm2)
```



In una situazione tipica il collector viene sostituito con il prossimo stage della pipeline.

Nel Main: creiamo la struttura che vogliamo utilizzare, il manager node (è un semplice nodo), la farm con un certo numero di worker, poi aggiungiamo il canale del manager verso l'mitter della farm.

Internamente il manager ha un channel, tipicamente è una cosa che viene fatta non dal programmatore ma da chi sviluppa il pattern, va solamente specificata una politica che mi indica quando aggiungere o rimuovere un nodo.

All'interno del metodo svc del manager dobbiamo avere la policy. L'mitter deve essere in grado di ascoltare il canale che arriva dal manager, quando so quello che devo fare mando il segnale ai worker. Il problema reale è l'mitter perchè deve leggere dal collector, dal manager, dal feedback e dall'input. Come distinguo i channel?

Dato che il manager non è parte dell'applicazione, il canale è "artificiale" quindi devo dargli un identificatore, questo è un numero e tramite questo numero riesco ad identificare il canale da cui ho letto i dati, in questo caso il canale del manager.

Se il messaggio arriva dal manager devo anche interpretare il tipo di messaggio che ho ricevuto e comportarmi di conseguenza inviando il messaggio corretto al worker.

Se ricevo qualcosa dai worker, posso mandargli un nuovo task da eseguire, quindi devo implementare uno scheduler.

Quando ricevo il messaggio dal collector devo memorizzare il numero di dati che sono on the fly, aumentando o diminuendo la variabile corrispondente. Quando ricevo un EOS, dal manager o dallo stage precedente, se è lo stage precedente e non c'è altro da fare posso riavviare tutti i worker e poi mandargli un EOS.

```
#include <iostream>
#include <ff/ff.hpp>

using namespace ff;

// this is an id greater than all ids
const int MANAGERID = MAX_NUM_THREADS+100;

typedef enum { ADD, REMOVE } reconf_op_t;
struct Command_t {
    Command_t(int id, reconf_op_t op): id(id), op(op) {}
    int id;
    reconf_op_t op;
```

```
};

// first stage
struct Seq: ff_node_t<long> {
    long ntasks=0;
    Seq(long ntasks):ntasks(ntasks) {}

    long *svc(long *) {
        for(long i=1;i<=ntasks; ++i) {
            ff_send_out((long*)i);

            struct timespec req = {0, static_cast<long>(5*1000L)};
            nanosleep(&req, NULL);
        }
        return EOS;
    }
};

// scheduler
class Emitter: public ff_monode_t<long> {
protected:
    int selectReadyWorker() {
        for (unsigned i=last+1;i<ready.size();++i) {
            if (ready[i]) {
                last = i;
                return i;
            }
        }
        for (unsigned i=0;i<=last;++i) {
            if (ready[i]) {
                last = i;
                return i;
            }
        }
        return -1;
    }
public:
    int svc_init() {
        last = get_num_outchannels(); // at the beginning, this is the number of workers
        ready.resize(last);
        sleeping.resize(last);
        for(size_t i=0; i<ready.size(); ++i) {
            ready[i] = true;
            sleeping[i] = false;
        }
        nready=ready.size();
        return 0;
    }
}
```

```
long* svc(long* t) {
    int wid = get_channel_id();

    // the id of the manager channel is greater than the maximum id of the
    workers

    if ((size_t)wid == MANAGERID) {
        Command_t *cmd = reinterpret_cast<Command_t*>(t);
        printf("EMITTER2 SENDING %s to WORKER %d\n",
    cmd->op==ADD?"ADD":"REMOVE", cmd->id);
        switch(cmd->op) {
            case ADD: {
                ff_monode::getlb()->thaw(cmd->id, true);
                assert(sleeping[cmd->id]);
                sleeping[cmd->id] = false;
                frozen--;
            } break;
            case REMOVE: {
                ff_send_out_to(GO_OUT, cmd->id);
                assert(!sleeping[cmd->id]);
                sleeping[cmd->id] = true;
                frozen++;
            } break;
            default: abort();
        }
        delete cmd;
        return GO_ON;
    }

    if (wid == -1) { // task coming from seq
        //printf("Emitter: TASK FROM INPUT %ld \n", (Long)t);
        int victim = selectReadyWorker();
        if (victim < 0) data.push_back(t);
        else {
            ff_send_out_to(t, victim);
            ready[victim]=false;
            --nready;
            onthefly++;
        }
        return GO_ON;
    }

    if ((size_t)wid < get_num_outchannels()) { // ack coming from the
    workers
        //printf("Emitter got %ld back from %d data.size=%ld,
        onthefly=%d\n", (Long)t, wid, data.size(), onthefly);
        assert(ready[wid] == false);
        ready[wid] = true;
        ++nready;
        if (data.size()>0) {

```

```
        ff_send_out_to(data.back(), wid);
        onthefly++;
        data.pop_back();
        ready[wid]=false;
        --nready;
    }
    return GO_ON;
}

// task coming from the Collector
--onthefly;
//printf("Emitter got %ld back from COLLECTOR data.size=%ld,
onthefly=%d\n", (Long)t, data.size(), onthefly);
if (eos_received && ((nready + frozen) == ready.size()) &&
(onthefly<=0)) {
    printf("Emitter EXITING\n");
    return EOS;
}
return GO_ON;
}
void svc_end() {
    // just for debugging
    assert(data.size()==0);
}
void eosnotify(ssize_t id) {
    if (id == -1) { // we have to receive all EOS from the previous stage
        eos_received = true;
        //printf("EOS received eos_received = %u nready = %u\n",
        eos_received, nready);
        if (((nready+frozen) == ready.size()) && (data.size() == 0) &&
onthefly<=0) {

            if (frozen>0) {
                for(size_t i=0;i<sleeping.size();++i)
                    if (sleeping[i]) ff_monode::getlb()->thaw(i, false);
            }
            printf("EMITTER2 BROADCASTING EOS\n");
            broadcast_task(EOS);
        }
    }
}
private:
    bool eos_received = 0;
    unsigned last, nready, frozen=0, onthefly=0;
    std::vector<bool> ready;           // which workers are ready
    std::vector<bool> sleeping;         // which workers are sleeping
    std::vector<long*> data;           // local storage
};
```

```
struct Worker: ff_monode_t<long> {
    int svc_init() {
        printf("Worker id=%ld starting\n", get_my_id());
        return 0;
    }

    long* svc(long* task) {
        //printf("Worker id=%ld got %ld\n", get_my_id(), (Long)task);
        ff_send_out_to(task, 1); // to the next stage
        ff_send_out_to(task, 0); // send the "ready msg" to the emitter
        return GO_ON;
    }

    void eosnotify(ssize_t) {
        printf("Worker2 id=%ld received EOS\n", get_my_id());
    }

    void svc_end() {
        //printf("Worker2 id=%ld going to sleep\n", get_my_id());
    }
};

// multi-input stage
struct Collector: ff_minode_t<long> {
    long* svc(long* task) {
        //printf("Collector received task = %ld, sending it back to the
Emitter\n", (Long)(task));
        return task;
    }
    void eosnotify(ssize_t) {
        printf("Collector received EOS\n");
    }
};

struct Manager: ff_node_t<Command_t> {
    Manager():
        channel(100, true, MANAGERID) {}

    Command_t* svc(Command_t *) {

        struct timespec req = {0, static_cast<long>(5*1000L)};
        nanosleep(&req, NULL);

        Command_t *cmd1 = new Command_t(0, REMOVE);
        channel.ff_send_out(cmd1);
    }
};
```

```
Command_t *cmd2 = new Command_t(1, REMOVE);
channel.ff_send_out(cmd2);

{
    struct timespec req = {0, static_cast<long>(5*1000L)};
    nanosleep(&req, NULL);
}

Command_t *cmd3 = new Command_t(1, ADD);
channel.ff_send_out(cmd3);

Command_t *cmd4 = new Command_t(0, ADD);
channel.ff_send_out(cmd4);

{
    struct timespec req = {0, static_cast<long>(5*1000L)};
    nanosleep(&req, NULL);
}

channel.ff_send_out(EOS);

return GO_OUT;
}

void svc_end() {
    printf("Manager ending\n");
}

int run(bool=false) { return ff_node_t<Command_t>::run(); }
int wait()           { return ff_node_t<Command_t>::wait(); }

ff_buffernode * const getChannel() { return &channel; }

ff_buffernode channel;
};

int main(int argc, char* argv[]) {
#ifndef BLOCKING_MODE
    //TODO: in blocking mode the manager channel does not work!!!
    return 0;
#endif

    unsigned nworkers = 3;
    int ntasks = 1000;
    if (argc>1) {


```

```
if (argc < 3) {
    std::cerr << "use:\n" << " " << argv[0] << " numworkers ntasks\n";
    return -1;
}

nworkers =atoi(argv[1]);
ntasks =atoi(argv[2]);
if (ntasks<500) ntasks = 500;
if (nworkers <3) nworkers = 3;
}

Seq seq(ntasks);
Manager manager;

std::vector<ff_node* > W;
for(size_t i=0;i<nworkers;++i)
    W.push_back(new Worker);
ff_farm farm(W);
farm.cleanup_workers();

// registering the manager channel as one extra input channel for the Load balancer
farm.getlb()->addManagerChannel(manager.getChannel());

Emitter E;

farm.remove_collector();
farm.add_emitter(&E);
farm.wrap_around();
// here the order of instruction is important. The collector must be added after the wrap_around otherwise the feedback channel will be between the Collector and the Emitter
Collector C;
farm.add_collector(&C);
farm.wrap_around();

ff_Pipe<> pipe(seq, farm);

if (pipe.run_then_freeze()<0) {
    error("running pipe\n");
    return -1;
}
manager.run();
pipe.wait_freezing();
pipe.wait();
manager.wait();

return 0;
}
```

Questi riassunti sono stati prodotti da Luca Corbucci. Le lezioni sono state sbobinate in parte da Alessandro Berti e alcune da Luca Corbucci. Gran parte dei disegni sono stati prodotti da Alessandro Berti. Questo materiale non deve essere venduto e non sostituisce il libro di testo o le lezioni. 317

Ringraziamenti

Ringraziamenti speciali vanno a Carlo Alessi che ha segnalato vari errori presenti all'interno di questi riassunti e ad Alessandro Berti che ha sbobinato gran parte delle lezioni producendo molte delle illustrazioni.