Appunti di Sistemi di Calcolo Paralleli e Distribuiti

A.A. 2020-2021

Matteo Brunello

Indice

1 Metriche di performance

Le metriche di performance permettono di dare una stima sulla performance dei sistemi di calcolo paralleli e distribuiti. La metrica di performance piu' importante per tale scopo e' lo **speedup factor**.

1.1 Speedup Factor (Strong Scaling)

Lo speedup factor e' una misura che indica quanto e' il miglioramento delle prestazioni (in termini di tempo) che si ottiene per risolvere un problema ben definito a priori, utilizzando p processori anziche' uno singolo.

$$S(N) = \frac{t_s}{t_p}$$

dove:

- t_s e' il tempo di esecuzione utilizzando un singolo processore (migliore esecuzione sequenziale)
- t_p e' il tempo di esecuzione utilizzando N processori

E' necessario specificare che "processori" deve essere inteso nella sua accezione piu' generale (e quindi come unita' di computazione a livello astratto). Non si specifica se tali sono core di un processore, processori differenti oppure addirittura macchine differenti. Fatta questa premessa, pero', vien da se che il confronto deve essere fatto utilizzando coerentemente la stessa unita' di computazione per entrambe le misurazioni. Ad esempio, se si confrontasse un singolo processore con un clock piu' alto rispetto ai p processori, la metrica risulterebbe inesatta. Questo perche' le condizioni degli esperimenti sono differenti tra loro, di fatto inserendo nella stima della metrica dei fattori che ne influenzano la qualita'. Un altro esempio potrebbe essere quello di testare l'agoritmo sul singolo processore su una CPU e successivamente quelli paralleli su una GPU. Per rendere quindi la metrica il piu' possibile precisa, l'unica condizione (o parametro) che viene resa variabile per l'esperimento e' l'algoritmo. Questo perche' l'implementazione dell'algoritmo che sfrutta il parallelismo e' spesso differente da un'implementazione sequenziale.

Nella maggior parte dei casi, lo speedup e' al massimo lineare, cioe' $S(N) \leq N$. Questo perche' il tempo parallelo migliore si ottiene nel caso in cui sia perfettamente divisibile per il numero di processori N, dal quale risulta che

$$S(N) \leq \frac{t_s}{t_s/N} = N$$

E' possibile anche ottenere casi in cui lo speedup e' superlineare, cioe' in cui $S(p) > p$, ma cio' accade spesso per ragioni come la presenza di memoria aggiuntiva (quale Cache e RAM) in sistemi multiprocessore e per l'impiego di algoritmi nondeterministici. Per esempio, in alcuni problemi il programma potrebbe essere troppo grosso da caricare interamente in memoria su una macchina

singola. Sfruttando il fatto che sia possibile caricarlo interamente nella RAM o nella Cache utilizzando piu' macchine, il fattore puo' diventare di conseguenza superlineare.

> Nell'informatica moderna si sfrutta questo fattore di superlinearita' nel training delle reti neurali. Poiche' il training delle reti neurali ha un fattore di convoluzione (moltiplicazione di matrici) per cui i processori moderni sfruttano i cosiddetti *tensor cores*, tenendo l'intero dataset su cui fare il training in una memoria piu' efficiente (nelle GPU chiamata *shell memory*) e permetto quindi di avere delle ottimizzazioni superlineari in termini di speedup. Si sfrutta quindi il fatto che i dati sono caricati interamente in una memoria molto veloce per ottenere uno speedup superlineare.

1.2 Speedup Massimo: Legge di Amdahl

Lo speedup di tipo lineare assume che gli algoritmi che stiamo valutando siano parallelizzabili interamente. Per questa ipotesi e' quindi possibile dividere in N parti il problema, per poi far risolvere gli N sottoproblemi ottenuti da N elementi di calcolo. Questa ipotesi pero' non e' generalmente vera poiche' non tutti gli algoritmi sono parallelizzabili. Si pensi ad esempio all'I/O: molti algoritmi leggono inizialmente uno o piu' file e terminano generando in output uno o piu' file. La legge di Amdahl tiene conto delle sezioni non parallelizzabili e che quindi devono essere eseguite per forza in modo sequenziale. La serial fraction comprende qualsiasi sezione che non sia parallelizzabile e che debba essere eseguita in modo sincrono.

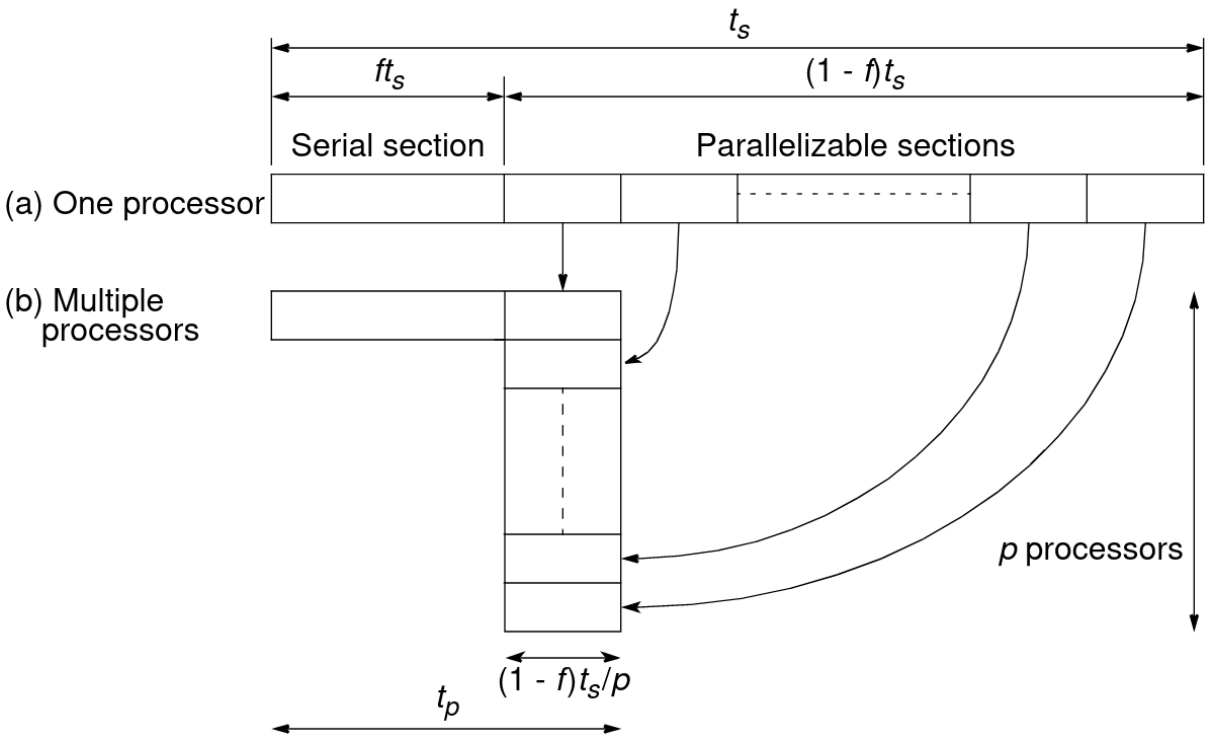

Figura 1: Rappresentazione grafica di un problema in cui e' presente una sezione non parallelizzabile (*serial section*)

La legge di Amdahl da un upper bound dello speedup massimo (teorico) raggiungibile S_{max}:

$$S_{max} = \frac{s}{(s + \frac{p}{N})}$$

In cui

- N e' il numero di processori in parallelo
- p e' la sezione che viene eseguita in parallelo
- s e' la sezione che viene eseguita in serie

Siccome l'intero programma e' composto dalla parte parallela e da quella seriale, allora si ha che $s + p = 1$, per cui si ottiene infine:

$$S_{max} = \frac{1}{s + p/N} = \frac{1}{(1 - p) + p/N}$$

Si nota fin da subito dalla relazione che tale speedup massimo non potra' mai essere N, per cui questa legge da una stima in un certo senso pessimistica.

1.3 Speedup Massimo: Legge di Gustafson (Weak Scaling)

La legge di Gustafson e' un raffinamento della legge di Amdahl, che tiene conto anche della scalabilita' della grandezza del problema. In generale, la grandezza del problema e' costante nella legge di Ahmdahl, mentre nella legge di Gufstafson viene tenuta in considerazione rendendola variabile come il numero di processori N. La grandezza del problema viene fatta aumentare all'aumentare della grandezza del sistema N (numero di processori). In questo modo il tempo di esecuzione parallelo e' costante, risultando di fatto numericamente differente dallo speedup massimo della legge di Ahmdal. La legge di Gufstafson e' anche chiamata *speedup factor scalato*. (speedup quando il problema viene fatto scalare). Ipotizziamo come fatto nel caso della legge di Amdahl, che un programma sia compsto da una parte parallelizzabile p e una seriale s (non parallelizzabile), e che la loro somma sia l'intero programma $s+p=1$. Al crescere della grandezza del problema e del numero di processori N, il tempo di esecuzione sequenziale e parallelo sono pari a

$$t_s = s + Np \quad t_p = s + \frac{Np}{N}$$

per cui se si calcola lo speedup

$$S_{scaled}(N) = \frac{t_s}{t_p} = \frac{s+Np}{s+p} = s + pN$$

ottenendo cosi' la legge di Gustafson. Notiamo che ci sono due assunzioni principali in questa legge:

- Il tempo di esecuzione parallelo e' costante
- Il tempo di esecuzione della parte seriale e' costante e non dipende da p

2 Sistemi di calcolo paralleli [2]

Le architetture parallele sono un'estensione naturale delle architetture dei computer convenzionali, in cui l'attenzione e' posta su problemi quali la comunicazione e la cooperazione tra diversi elementi di calcolo. In sostanza, quando si parla di architetture parallele, non si fa altro che parlare di un concetto piu' esteso di architettura di calcolo. Tale estensione e' proprio data dalla presenza di un'ulteriore architettura che si occupa dei problemi di comunicazione.

> Architettura parallela = Architettura di calcolo + Architettura di comunicazione

L'architettura di comunicazione estende anche l'organizzazione dell'architettura di calcolo andando ad aggiungere anche elementi hardware per il suppporto alla comunicazione. In generale quando si parla di architetture parallele si tende anche a descrivere un modello di un programmazione ad esse

associato. Esso rappresenta una concettualizzazione del modello di esecuzione, cioe' della macchina che esegue i programmi. I 3 modelli di programmazione parallela trattati in questo corso sono:

- *Shared Memories*
- *Message Passing*
- *Single Instruction, Multiple Threads*

Storicamente parlando, i modelli di programmazione corrispondevano ad un determinato modello di esecuzione, per cui era pratica comune associare con un mapping 1:1 modello di programmazione e di esecuzione. Tale approccio e' pero' meno appropriato al giorno d'oggi; in primo luogo perche' esistono ormai molte similitudini tra modelli di esecuzione paralleli, mentre in secondo luogo perche' i modelli di esecuzione hanno introdotto il supporto a differenti modelli di programmazione. In questo senso si puo' parlare di una convergenza verso un modello "misto" dei diversi modelli di programmazione.

2.1 Modelli di esecuzione

Con il termine *parallel computer* (sistema di calcolo parallelo) ci si riferisce arbitrariamente ad un singolo computer dotato di piu' processori o piu' computers interconnessi coerentemente tra di loro in modo da formare una piattaforma di calcolo ad alte prestazioni [2]. Riprendendo quanto descritto in precedenza, e' possibile far corrispondere ai 3 modelli di programmazione 3 tipologie differenti di sistema di calcolo parallelo:

- Sistemi multiprocessore a memoria condivisa
- Sistemi multicomputer a memoria distribuita
- **G**raphics **P**rocessing **U**nit

Ovviamente, come gia' detto in precedenza, la corrispondenza diretta e' solo di natura storica e non implica che determinati sistemi di calcolo non supportino altri modelli di programmazione. Vediamo ora nel dettaglio i sistemi di calcolo descritti.

2.1.1 Sistemi Multiprocessore a Memoria Condivisa

Sono sistemi che estendono il modello di esecuzione a processore singolo. L'idea e' quella di aggiungere altri processori, connettendoli a diversi moduli di memoria in modo che ogni processore possa accedere ad ogni altro modulo di memoria. La connessione tra la memoria e i processori e' mediata da una *rete di interconnessione*. In questi sistemi ogni locazione di memoria e' unica per ogni processore che la vuole utilizzare. In altri termini si dice che tali sistemi hanno uno *spazio di indirizzamento singolo*. Cio' implica che la memoria e' condivisa da tutti i processori connessi.

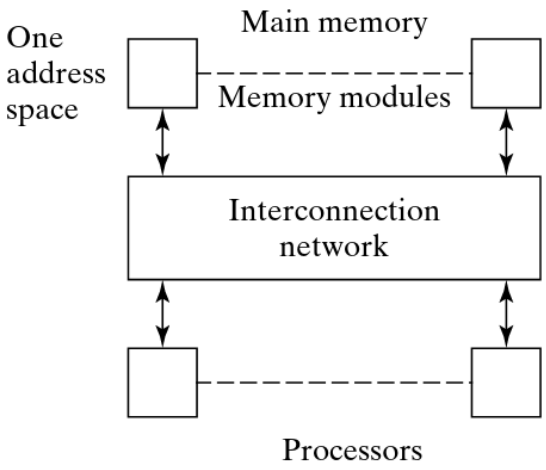

Figura 2: Modello multiprocessore a memoria condivisa

Tale caratteristica e' in un certo senso un'arma a doppio taglio: da una parte e' particolarmente convieniente il fatto che i dati siano condivisi tra tutti i processori, soprattutto dal un punto di vista di un programmatore. Tuttavia, pero', il fatto che la memoria sia condivisa e l'assenza di supporto hardware adeguato non permettono a tali sistemi di scalare in numero di processori. Per questa ragione quando si parla di questo tipo di sistemi, ci si riferisce spesso a computer singoli multiprocessore, i quali hanno spesso un numero limitato di processori. I sistemi di questo tipo possono inoltre essere classificati in base al tempo di accesso dei processori alla memoria principale. Secondo questo criterio si possono avere due tipologie di sistema a memoria condivisa:

- UMA (*Uniform Memory Access*): Il tempo di accesso alla memoria e' sempre uguale per ogni indirizzo e per ogni processore.
- NUMA (*Non-Uniform Memory Access*): E' l'oppposto di UMA.

2.1.2 Sistemi Multicomputer a Scambio di Messaggi

Con sistemi a multicomputer ci si riferisce ad un insieme di computer interconnessi tra di loro attraverso una rete di interconnessione, come mostrato in figura 3

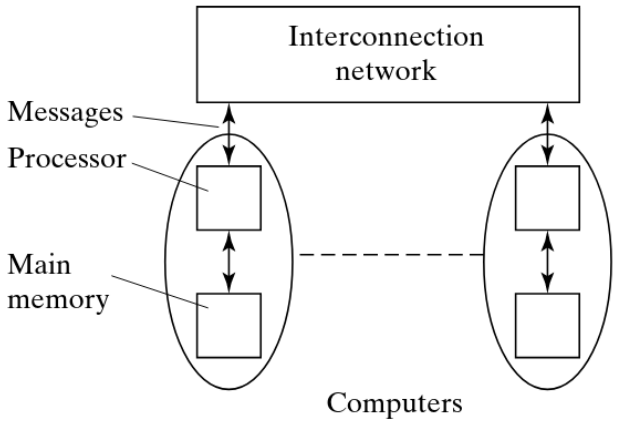

Figura 3: Modello multicomputer a passaggio di messaggi

Ogni computer connesso ha a sua volta la propria memoria, percio' ogni computer avra' uno spazio di indirizzamento privato. Per questa ragione i computer connessi non possono accedere direttamente alla memoria di altri computer, di fatto rendendo privato l'accesso alla memoria di ogni nodo. In opposizione al modello a memoria condivisa che utilizza la memoria come medium di comunicazione per l'informazione tra diversi processori, nei sistemi a multicomputer si utilizza lo scambio di messaggi per raggiungere lo stesso scopo. La rete di interconnessione serve quindi come mezzo attraverso il quale i messaggi passano. In questo senso quindi rappresenta un fattore che puo' influenzare l'efficienza con cui i messaggi raggiungono un determinato nodo. Dal punto di vista di programmazione si utilizzano linguaggi sequenziali che supportano procedure e strutture dati per lo scambio di messaggi. Un programma viene suddiviso in *processi* che possono essere eseguiti in parallelo da diversi computer, ma che possono essere anche eseguiti sullo stesso computer tramite una politica di tipo *time sharing*. Come anticipato, i processi comunicano dati e informazioni tra di loro tramite lo scambio di messaggi. Dal momento che la memoria condivisa e' deprecata in favore a un modello basato sullo scambio di messaggi asincroni, i sistemi a scambio di messaggi scalano molto piu' facilmente all'aumentare del numero di computer connessi.

2.2 Memoria Condivisa Distribuita (Distributed Shared Memory)

Il paradigma basato sullo scambio di messaggi presenta delle limitazioni dal punto di vista dei programmatori. Questo perche', codice scritto per questo modello di esecuzione e' in generale difficile da debuggare. Inoltre, il fatto che i dati non possano essere condivisi tra tutti i processori ma che possano essere solo in un certo senso copiati potrebbe essere problematico, soprattutto in applicazioni che richiedono di effettuare operazioni su grandi quantita' di dati. Dall'altra parte, pero', questo paradigma

offre la possibilita' ai programmatori di evitare operazioni di sincronizzazione che sarebbero invece necessarie per la programmazione di sistemi basati sul modello a memoria condivisa. Queste diverse motivazioni hanno favorito la concentrazione degli sforzi verso la ricerca di modello che sfruttasse i vantaggi di entrambi i modelli, che culmino' con la nascita del concetto di *sistema a memoria condivisa distribuita*. In questi sistemi la memoria e' *fisicamente* distribuita tra tutti i processori ma con la differenza che ogni processore ha accesso all'intera memoria utilizzando uno *spazio di indirizzamento singolo*. Un processore accede ad una locazione di memoria facente parte dello spazio ad esso riservato lo fa in modo usuale, mentre quando tale locazione fa parte dello spazio riservato ad un altro processore, allora avviene uno scambio di messaggi. Il funzionamento avviene in maniera totalmente nascosta dal punto di vista del programmatore, che in questo modo potra' sfruttare i vantaggi del modello a memoria condivisa.

2.3 Reti di interconnessione

Una rete di interconnessione e' un sistema di collegamenti che connette uno o piu' dispositivi tra di loro con lo scopo di connettere processori (computer) ad altri processori o di consentire a piu' processori di accedere a uno o piu' moduli di memoria condivisi [3]. Nei sistemi a multicomputer, una rete di interconnessione rappresenta il percorso fisico tramite il quale i messaggi vengono inviati da un computer ad un altro. In queste tipologie di reti ci sono diversi fattori da tener conto che possono influenzare le prestazioni quali:

- Bandwidth: numero di bits che possono essere trasmessi in un'unita' di tempo
- Latenza: tempo impiegato da un messaggio a raggiungere la destinazione
- Latenza di comunicazione: tempo totale impiegato per inviare il messaggio che comprende ritardi dovuti all'overhead del software, ed eventuali ritardi nelle interfacce hardware

Quando si parla di *diametro* di una rete di interconnessione ci si riferisce al numero minimo di connessioni tra i due nodi piu' distanti nella rete, mentre con *bisection width* ci si riferisce al numero minimo di connessioni che devono essere eliminate in modo da dividere la rete in due parti uguali. Vediamo ora nel dettaglio alcune tipologie di reti di interconnesione.

2.3.1 Mesh Network

Una mesh network a due dimensioni consiste in una matrice di dimensioni $\sqrt{p} \times \sqrt{p}$ in cui ogni nodo e' connesso a tutti i suoi 4 nodi adiacenti. I nodi situati su di una frontiera possono essere connessi agli estremi dei nodi situati alla frontiera opposta. Questa tipologia di reti e' detta anche *toroidale* o *torus network*. Secondo questo principio di connessione e' possibile ottenere anche mesh network a tre dimensioni, collegando ogni nodo ai suoi 6 nodi adiacenti. E' facile notare come sia possibile generalizzare la nozione a dimensioni generiche k.

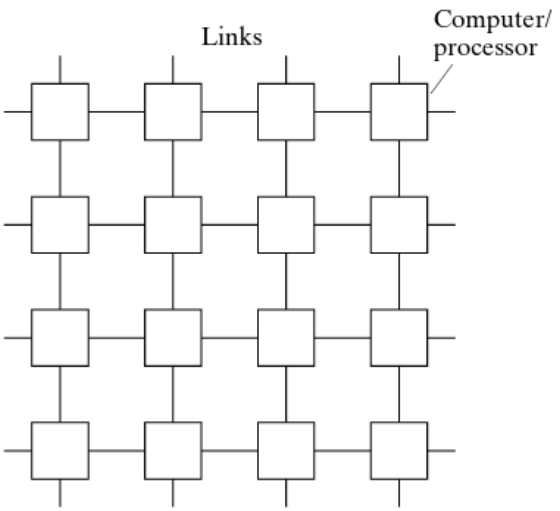

Figura 4: Rete mesh 2D

Per questa loro caratteristica, le reti mesh sono particolarmente utilizzate in applicazioni in cui i punti della soluzione sono organizzati in array a due o tre dimensioni. In generale vale che in una mesh network di k dimensioni i nodi hanno $2k$ vicini (e quindi connessioni). Degno di nota e' anche il diametro delle mesh network; per reti di dimensioni $\sqrt{p} \times \sqrt{p}$ il diametro e' $2(\sqrt{p} - 1)$. Mentre la bisection width e' pari a $\sqrt{p}$. Dove p e' il numero di processori all'interno della rete.

2.3.2 Hypercube Networks

In una hypercube network a d dimensioni, ogni nodo e' connesso ad un nodo in ogni dimensione della rete. Ogni nodo della rete possiede un indirizzo univoco formato da d bits. Per esempio, nel caso di una hypercube network a 3 dimensioni, ogni nodo sara' connesso ad altri 3 nodi e ogni indirizzo sara' composto da una sequenza di 3 bit.

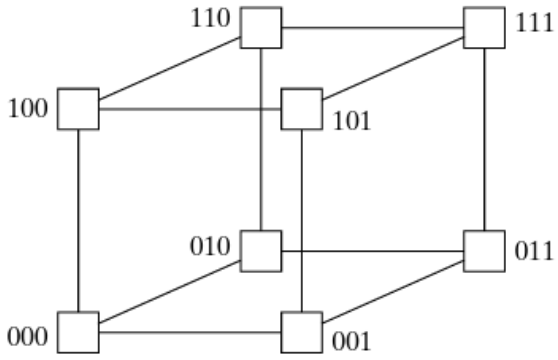

Figura 5: Hypercube a 3 dimensioni

Un vantaggio delle reti hypercube e' dato dal fatto che il diametro di tali reti risulta $log_2 p$, che indica una crescita ragionevolmente bassa all'aumentare del numero di processori p. La bisection width e' $p/2$. Un'altro vantaggio principale di questa tipologia di reti e' l'algoritmo di routing. Tale algoritmo si basa sull'idea che ogni bit di indirizzo corrisponde ad una dimensione nella rete. Sfruttando questo fatto, dato un indirizzo sorgente X e un indirizzo destinazione Y, ogni bit dell'indirizzo Y che differisce dall'indirizzo X rappresenta una dimensione nel cubo che deve essere scelta per raggiungere Y. Per fare cio' l'algoritmo calcola $Z = X \oplus Y$, il quale conterra' tutti i bit che differiscono da Y. Viene poi scelto il bit piu' significativo di Z come dimensione da scegliere per il reindirizzamento (il che corrisponde ad un bit flip nella stessa posizione dell'indirizzo X). E' bene notare che ad ogni passo il nodo sorgente X cambia, mentre Y rimane invariato. Per esempio, ipotizzando il caso in cui si voglia inviare un messaggio dal nodo $X = 001101$ al nodo $Y = 101010$, l'algoritmo compierebbe i seguenti passi:

Passo	X	$Z = X \oplus Y$	Next
1	001101	100111	101101
2	101101	000111	101001
3	101001	000011	101011
4	101011	000001	101010

2.3.3 Crossbar Switch

Una rete di interconnessione di tipo crossbar switch consiste in una collezione di switch connessi tra di loro in modo da formare una matrice [4]. Ogni nodo e' connesso ad uno switch che a sua volta e' connesso ad altri switch e cosi' via. Questa tipologia di reti e' piu' prevalente in sistemi a memoria condivisa che in sistemi a scambio di messaggi.

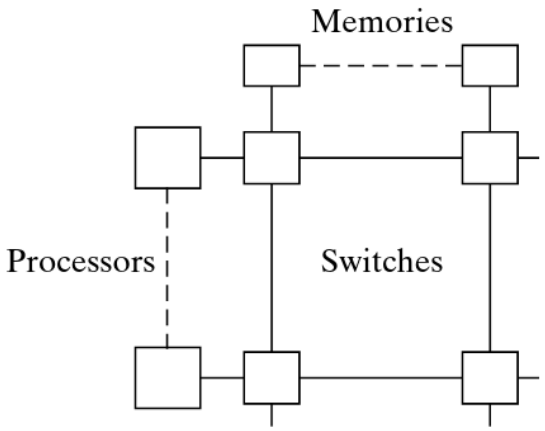

Figura 6: Layout di una rete di tipo crossbar switch

2.3.4 Tree Networks

Un'altra tipologia di rete che impiega l'utilizzo di switch e' proprio la tree network. In queste reti ogni switch e' connesso ad altri m switch sottostanti fino ad arrivare ai nodi foglia che sono i processori. Un aspetto chiave in queste reti e' che il numero di connessioni per arrivare dal nodo radice fino ai nodi foglia e' logaritmico ($log_m p$).

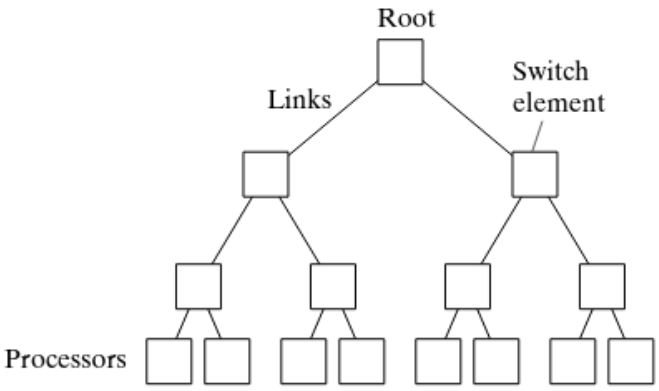

Figura 7: Tree network con *m=2*

Un problema noto di questo tipo di reti e' la congestione che si forma verso il nodo radice, rappresentando di fatto un collo di bottiglia.

2.3.4.1 Fat Trees Per ovviare al problema della congestione verso il nodo radice si possono impiegare i cosiddetti *fat trees*, in cui il numero di connessioni aumenta progressivamente all'avvicinarsi al nodo radice. Cosi' facendo, si aumenta esponenzialmente il numero di connessioni verso la radice, di fatto riducendo il collo di bottiglia.

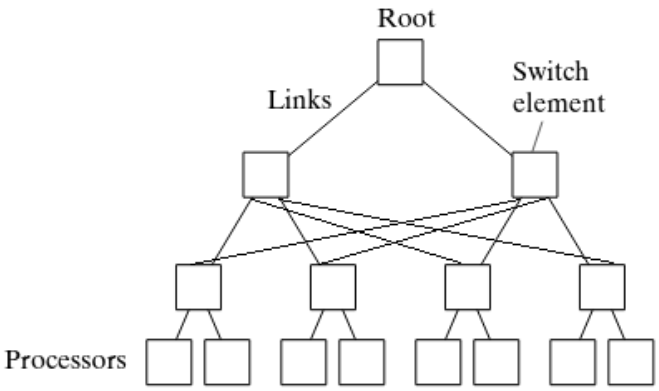

Figura 8: Fat Tree network con *m=2*

2.3.5 Multistage Interconnection Networks

Sotto questa tipologia di reti ricadono diverse configurazioni che si basano sulla connessione di diversi livelli di switch. Diversi livelli di switch sono connessi tra di loro in modo che un pacchetto possa viaggiare indipendentemente da una parte all'altra della rete mediante un percorso arbitrario. La rete omega e' un esempio che ricade in questa tipologia, particolarmente conveniente per la semplicita' dell'algoritmo di routing. Dato un indirizzo composto da n cifre $X_n..X_1$, se si passa la rete scegliendo la porta di output in base al valore del bit: alta se il bit e' 0, bassa se il bit e' 1. In questo modo ogni indirizzo contiene in modo implicito anche il percorso da fare all'interno della rete per essere raggiunto.

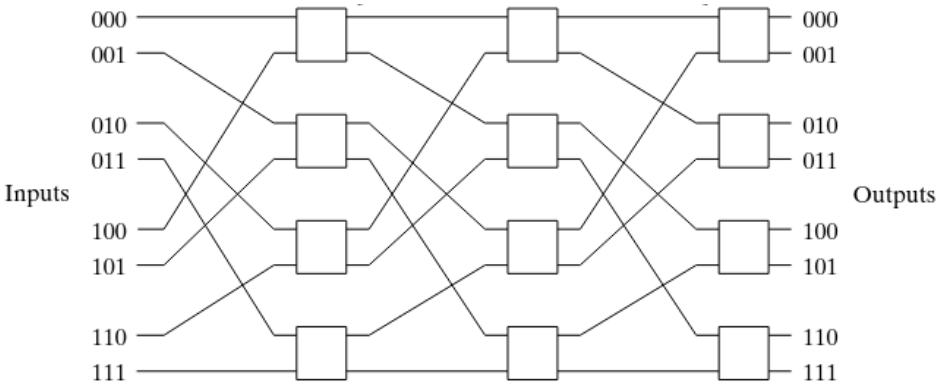

Figura 9: Rete MIN di tipo Omega

2.4 Tassonomia di Flynn

Nel 1966 Flynn invento' una tassonomia per classificare le diverse tipologie di sistema di calcolo. Le classificazioni sono basate sul numero concorrente di istruzioni eseguite e sulla disponibilita' del flusso di dati. In sostanza, si dividono in 4 classificazioni:

- Single Instruction Stream Single Data Stream (SISD): Sistema che non sfrutta nessun meccanismo di parallelismo, ne a livello di istruzioni ne a livello di flusso dati. Sono i tradizionali sistemi di calcolo sequenziali.
- Single Instruction Stream Multiple Data Streams (SIMD): Sistemi in cui una singola istruzione opera su piu' dati contemporaneamente. A livello di architettura sono composti da piu' processori che effettuano la stessa operazione contemporaneamente su dati differenti. Particolarmente utili per applicazioni di tipo multimediale o di processing grafico dove la stessa operazione deve essere effettuata su dati diversi (ad esempio applicare un kernel ad una immagine). I processori moderni offrono alcune istruzioni SIMD specifiche per questi utilizzi.
- Multiple Instruction Streams Single Data Stream (MISD): In questa tipologia di sistemi, i processori eseguono diverse operazioni sugli stessi dati.

- Multiple Instruction Streams Multiple Data Streams (MIMD): Sono i sistemi di calcolo descritti fin ora che comprendono i sistemi a memoria condivisa e a scambio di messaggi. In questi sistemi i processori eseguono indipendentemente istruzioni differenti su dati differenti. Tipicamente il software eseguito segue il modello di programmazione SPMD (*Single Program, Multiple Data*), che consiste nel far eseguire a tutti i processori lo stesso programma indipendentemente su dati diversi.

3 Programmazione Message-Passing

Come detto in precedenza, per programmare sistemi di calcolo paralleli (e quindi anche un multicomputer basato sullo scambio di messaggi) e' possibile utilizzare principalmente:

1. Un linguaggio di programmazione appositamente creato per la programmazione parallela
2. Un linguaggio di programmazione sequenziale la cui sintassi e' estesa da istruzioni specifiche per la programmazione parallela
3. linguaggio di programmazione sequenziale e una libreria apposita per accedere alle funzionalita' di procedure e primitive utili alla programmazione parallela

Oltre a queste modalita', potrebbe essere possibile anche utilizzare compilatori specifici atti a parallelizzare codice sequenziale. Tale opzione pero' non e' praticabile soprattutto su sistemi a scambio di messaggi, per il fatto che i linguaggi sequenziali non sono dotati della nozione di scambio di messaggi. Noi ci concentreremo principalmente sulla terza opzione, utilizzando come linguaggio di programmazione `C` e la libreria `Open MPI` (*Open Message Passing Interface*). Il paradigma message passing si basa su due meccanismi principali:

1. Un metodo per la creazione di processi separati per l'esecuzione su computer differenti
2. Un metodo per mandare e ricevere messaggi tra processi

3.1 Creazione di processi

Si divide essenzialmente in creazione dinamica e statica. Nella creazione statica, il numero di processi viene definito prima dell'esecuzione e il sistema eseguira' tale numero fissato di processi. Nella creazione dinamica un numero arbitrario di processi puo' essere eseguito. I processi possono essere creati anche durante l'esecuzione di altri processi. E' piu' flessibile ma introduce *overhead*. Un'altra distinzione viene fatta se i processi sono creati a partire dallo stesso codice o a partire da codici diversi. Nel primo caso si parla di SPMD, mentre nel secondo di MPMD. `MPI` segue il modello di programmazione SPMD, in cui lo stesso programma viene eseguito su piu' processori, operando su dati diversi. La diversificazione e' ottenuta attraverso diversi statement di controllo di flusso all'interno del programma. In questo modello la creazione dei processi e' statica.

3.2 Metodi per lo scambio di messaggi

Lo scambio di messaggi nella programmazione basata sul message-passing utilizza principalmente due istruzioni principali per lo scambio di messaggi: `send` e `receive`. La tipologia di queste istruzioni puo' a sua volta essere organizzata in modo tassonomico. Si parla quindi di `send`/`receive` *sincrona/asincrona*, *simmetrica/asimmetrica*: Per sincrono si indica un'operazione in cui si *attende* che venga conclusa, mentre con simmetrica si indica uno scambio di messaggi mediato tra soli due processi. Una send sincrona attende fino a quando il messaggio puo' essere accettato dal processo destinazione prima di spedire il messaggio. D'altra parte, una receive sincrona attende finche' il messaggio che ci si aspetta arrivi. Intrinsecamente queste primitive oltre che a *trasferire dati* servono anche a *sincronizzare processi*. Il termine *rendezvouz* e' usato per descrivere proprio l'azione con cui si sincronizzano dei processi utilizzando operazioni `send` e `receive` sincrone. E' inoltre importante menzionare che la versione sincrona di queste primitive non necessita di buffer interni come invece richiedono le varianti asincrone. L'implementazione e' spesso mediata tramite un protocollo 3-way handshake.

D'altro canto, le send e receive asincrone consistono nel non aspettare che le azioni completino prima di andare avanti con l'esecuzione. In generale, servono solo a scambiare dati e non hanno nessuna utilita' in termini di sincronizzazione di processi. Le primitive asincrone utilizzano dei buffer interni di messaggi per poter ottenere questo comportamento.

In questa tipologia di primitive possiamo distinguere altre due caratterizzazioni:

- Bloccanti (*locally blocking*): la primitiva completa (va alla prossima istruzione) dopo che l'azione locale completa. Nel caso della send, con azione locale si intende che il messaggio sia stato inserito nel buffer. Questo garantisce che il messaggio sara' inviato ma non che sara' necessariamente ricevuto. Nel caso della receive, invece, se il buffer dei messaggi e' vuoto l'azione non ritorna fin quando il messaggio non e' arrivato.

- Non Bloccanti: la primitiva completa immediatamente. Nell'esempio della send, non da la garanzia che il messaggio sia effettivamente stato spedito. Cio' significa che non e' nemmeno bloccante sullo stato del buffer dei messaggi.

Siccome l'asincronia e' legata direttamente allo spazio di bufferizzazione interno per i messaggi, nei casi delle send asincrone, si potrebbero verificare casi in cui l'invio dei messaggi sia molto piu' veloce rispetto al consumo dei tali. Questo potrebbe portare alla saturazione del buffer, con un conseguente comportamento sincrono della primitiva. In altre parole, quando il buffer e' saturo, la primitiva si blocca e aspetta che si svuoti.

3.3 Selezione e differenziazione dei messaggi

I messaggi sono differenziati tra di loro attraverso un `tag` che viene incapsulato nel messaggio stesso. In questo modo, le primitive di send e receive accetteranno anche un parametro addizionale `tag` per specificare un messaggio che che si vuole inviare o che ci si aspetta. Esistono anche delle *wildcards*, cioe' degli opportuni tag speciali predefiniti, che permettono di specificare l'intenzione di ricevere messaggi con tag qualsiasi da parte di un processo.

3.4 Primitive Collettive

Sono routines di natura asimmetrica per lo scambio di messaggi tra piu' processi. Lo scopo e' quello di mandare un messaggio a piu' processi nel caso della send, e viceversa nella receive per ricevere messaggi di piu' processi da un processo singolo. La presenza di queste routines e' legata al fatto che sfruttando particolari caratteristiche del sistema, si possano ottenere delle implementazioni piu' efficienti, rispetto a implementazioni basate ad esempio sulle primitive a scambio di messaggi singoli.

3.4.1 Scatter

A partire da un gruppo di n processi, tra cui uno di questi e' un processo *root* contenente un *array*, la primitiva scatter consiste nel inviare ogni *i-esimo* elemento di tale array ai corrispondenti *i-esimi* processi. L'idea e' quella di avere un mapping 1:1 tra elementi dell'array e messaggi inviati ai processi. Quando le due dimensioni non coincidono si ricorre a partizionamento.

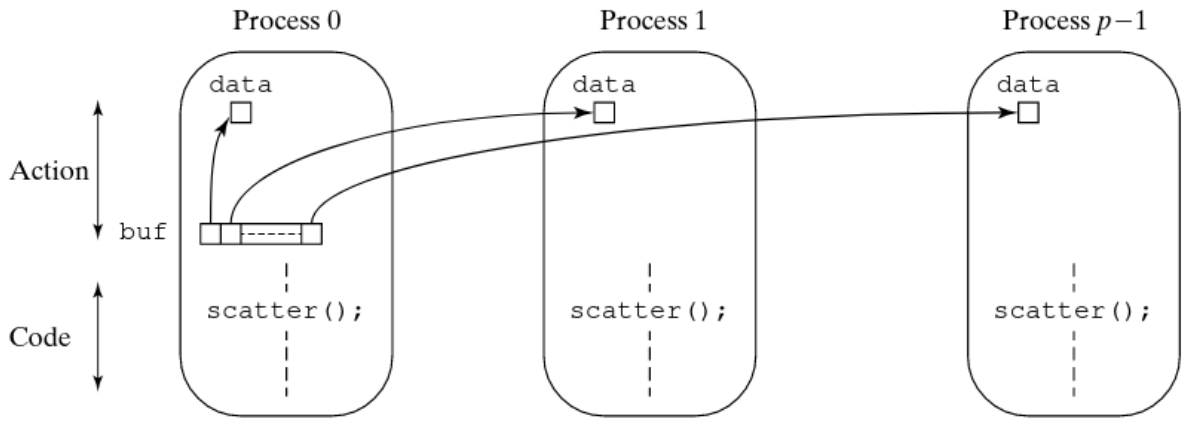

Figura 10: Operazione Scatter

3.4.2 Gather

E' l'inversa della scatter. Permette ad un processo root di collezionare i valori inviati attraverso messaggi da parte degli altri processi all'interno di un array. Il dato arrivato dall'*i-esimo* processo viene inserito nell'*i-esima* posizione dell'array del processo root.

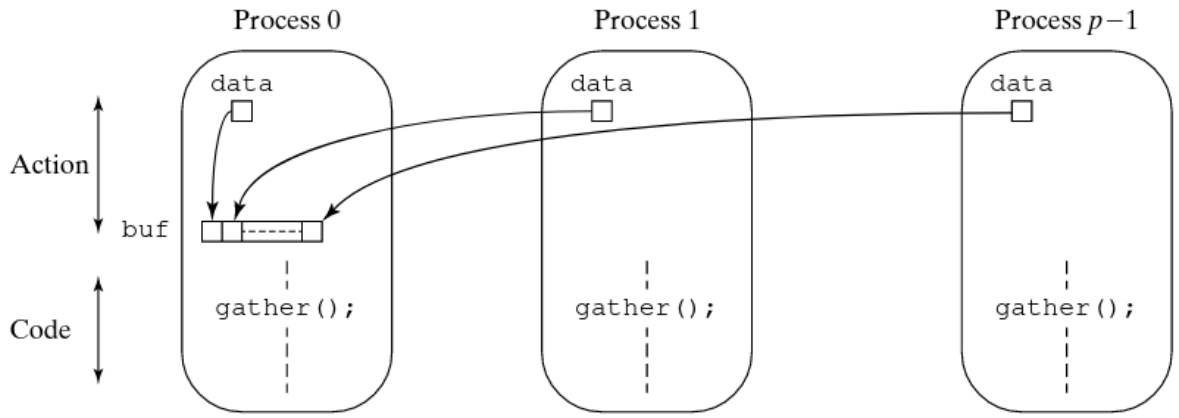

Figura 11: Operazione Gather

3.4.3 Reduce

E' una vaiante della gather. Al posto di salvare i dati all'interno di un array, i dati vengono combinati attraverso un'operazione associativa. E' come la `fold` dei linguaggi funzionali, dove lo stream di dati e' rappresentato dai messaggi (dati) provenienti dagli altri processi.

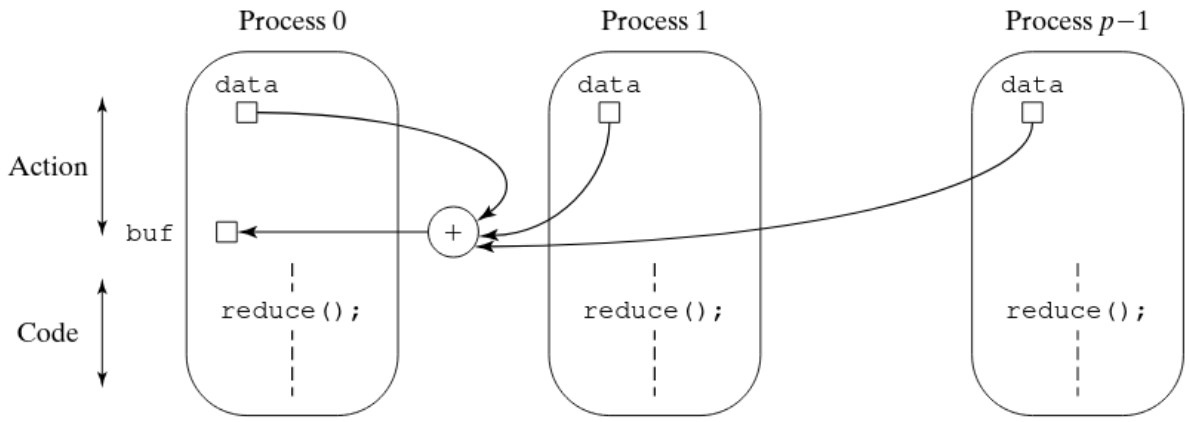

Figura 12: Operazione Reduce

3.5 MPI (Message Passing Interface)

E' uno *standard de-facto* che definisce le routines e le primitive (in sostanza un API) utili per la programmazione di sistemi message passing. E' importante notare che MPI non definisce l'implementazione di tale standard. Ad oggi ci sono diverse implementazioni di MPI, alcune free e open source e altre commerciali. Noi ci concentreremo su una versione *FOS* chiamata *OpenMPI*.

3.5.1 Creazione di processi ed esecuzione

La creazione e' volutamente non definita esplicitamente, lasciando aperta la flessibilita' dell'implementazione. Principalmente la creazione di processi e' statica, cioe' che il numero di processi che verranno eseguiti e creati (contemporaneamente) e' definita a priori (al momento dell'esecuzione del programma). OpenMPI permette nelle versioni piu' recenti la creazione dinamica di processi, ma in generale non se ne consiglia l'utilizzo per diverse ragioni e problematiche. OpenMPI segue inoltre il modello di programmazione SPMD, cio' significa che un singolo programma verra' eseguito su piu' processori e che il comportamento di tale programma si differenziera' a runtime.
Prima di utilizzare qualsiasi funzione, il sistema MPI deve essere inizializzato mediante l'istruzione `MPI_Init()` e terminato alla fine con `MPI_Finalize()`.

3.5.2 Comunicatori

Inizialmente, tutti i processi fanno parte dello stesso contesto comune di comunicazione, a cui ogni processo e' assegnato un *rank* univoco (un numero compreso tra 0 e $1 - p$). Tale contesto e' chiamato `MPI_COMM_WORLD`. I processi pero' possono appartenere a contesti di comunicazione (chiamati comunicatori) differenti. In questo modoo si separano i processi tra di loro in differenti contesti di comunicazione. Esistono due diversi tipi di comunicatori:

- Intracomunicatori: per la comunicazione all'interno di un gruppo di processi
- Intercomunicatori: per la comunicazione tra gruppi di processi

La presenza di contesti di comunicazione e' utile soprattutto per differenziare i messaggi di libreria (in questo caso di OpenMPI) dai messaggi del programma. E' importante inoltre notare che un processo possa appartenere a piu' contesti di comunicazione differenti. Grazie a questa caratteristica, tutte le primitive MPI per lo scambio di messaggi richiedono sempre di specificare il comunicatore.

3.5.3 Primitive per lo scambio di messaggi

Vediamo ora le diverse primitive fornite da MPI per lo scambio di messaggi tra processi.

3.5.3.1 Comunicazione Point to Point Come detto in precedenza, il comportamento delle routine send e receive in MPI e' di default asincrono bloccante (*locally blocking*). Cio' significa che l'operazione blocchera' solo il tempo necessario per poter completare l'azione locale. L'eccezione e' solo la receive, che nel caso in cui il buffer sia vuoto, si blocchera' fin quando non arrivera' il messaggio desiderato nel buffer. Il formato generale dei parametri di una send asincrona bloccante e' il seguente

```
1 MPI_Send(buf, count, datatype, dest, tag, comm);
```

In cui:

- buf: variabile che contiene il contenuto del messaggio
- count: numero di elementi da inviare del contenuto del messaggio
- datatype: tipo del singolo elemento di ogni array
- src: rank del processo a cui e' destinato il messaggio
- tag: tag del messaggio
- comm: comunicatore
- status: stato dopo l'operazione

Mentre la routine receive ha il seguente formato:

```
1 MPI_Recv(buf, count, datatype, src, tag, comm, status);
```

In cui i campi sono l'opposto della send. In caso in cui il numero di dati inviati da un altro processo sia piu' grande di quelli specificati dalla receive, si incorre in un errore di overflow. Esistono poi le varianti non bloccanti, che terminano a prescindere dal fatto che l'azione locale sia terminata o meno. Il nome di questa versione delle primitive ha come prefisso una `I` che indica "*immediate*".

```
1 MPI_Isend(buf, count, datatype, dest, tag, comm, request)
2 MPI_Irecv(buf, count, datatype, dest, tag, comm, request)
```

In caso di utilizzo di questa tipologia di primitive, si possono utilizzare altre operazioni per controllare se l'azione locale e' stata completata o meno, quali `MPI_Wait()` e `MPI_Test()`. In questo senso le varianti non bloccanti insieme alle primitive per controllare la terminazione delle azioni locali, sono delle versioni piu' generiche e flessibili di quelle bloccanti.

La primitiva send puo' avere tre modalita' di comunicazione che definiscono il protocollo di send/receive:

- Modalita' standard: la send non assume che la corrispondente routine di receive sia stata eseguita.
- Modalita' bufferizzata: serve in caso sia necessario specificare esplicitamente lo spazio di bufferizzazione riservato ai messaggi tramite `MPI_Buffer_attach()`.

- Modalita' sincrona: la send e la receive possono iniziare prima l'una rispetto all'altra in qualsiasi ordine ma completeranno solamente insieme.
- Modalita' ready: una send puo' iniziare solo se una receive corrispondente e' gia' stata iniziata, altrimenti ritorna un errore.

3.5.3.2 Comunicazione Collettiva

Le primitive di comunicazione collettiva sono quelle principali discusse in precedenza, piu' altre primitive meno utilizzati ma non meno importanti:

- `MPI_Bcast()`: Manda un messaggio in broadcast
- `MPI_Gather()`: Primitiva Gather
- `MPI_Scatter()`: Primitiva Scatter
- `MPI_Alltoall()`: Invia i dati da tutti i processi a tutti i processi
- `MPI_Reduce()`: Primitiva Reduce
- `MPI_Reduce_scatter()`: Combina i valori e utilizza scatter sul risultato della combinazione
- `MPI_Scan()`: Distribuisce ai vari processi tutti i prefissi accumulati

Tutte queste primitive di comunicazione hanno come effetto collaterale anche la necessita' di sincronizzazione. La barrier e' una primitiva atta alla sincronizzazione e non allo scambio di messaggi. Serve a far sincronizzare tutti i processi di un determinato comunicatore. La barrier e' una primitiva da usare con cura. Secondo alcuni autori ha utilita' solo per supportare il programmatore nello scrivere programmi paralleli, poiche' tutti i programmi che utilizzano la primitiva barrier possono essere scritti senza tale primitiva, essendo cosi' piu' efficienti, siccome la barrier deve aspettare il processo piu' lento per sincronizzarli tutti.

3.6 Valutazione di programmi paralleli

Nella valutazione di programmi sequenziali si usa spesso come metrica il numero di istruzioni che vengono eseguite. Per un algoritmo parallelo, le cose si complicano leggermente ed e' necessario tener conto anche del tempo che passa il programma a far comunicare i processi tra loro. In questo caso, quello che si misura e' il cosiddetto *work-clock* time, cioe' il tempo misurato *"a orologio"*. Come prima approssimazione e' possibile esprimere il tempo di esecuzione parallela t_p come la somma di due parti: il tempo di computazione e il tempo di comunicazione

$$t_p = t_{comp} + t_{comm}$$

In realta' il tempo di comunicazione sarebbe piu' propriamente chiamato tempo di *overhead*, in modo da comprendere tutte quelle parti (come la sincronizzazione, load balancing ecc..) che non verrebbero comprese altrimenti. Nel caso in cui piu' di un processo venga eseguito in parallelo, si tiene conto solo

del processo che ha tempo di esecuzione massimo. L'unita' di tempo di t_{comp} puo' essere misurata in numero di istruzioni eseguite ma molto spesso si tratta di vere e proprie "*fasi*" che il programma ha bisogno di eseguire per computare il risultato. Possiamo quindi eseprimere il tempo computazionale come la somma del tempo necessario ad eseguire le fasi che lo compongono, separate dai momenti in cui avvengono gli scambi di messaggi.

$$t_{comp} = t_{comp1} + t_{comp2} + \dots$$

> Per utilizzare queste relazioni, ovviamente, bisogna supporre che tutte le macchine/processori operino alla stessa velocita'.

Il tempo di comunicazione t_{comm}, invece, dipende direttamente dal numero di messaggi, dalla grandezza di un singolo messaggio, dalla tipologia di rete di interconnessione e dalla modalita' di trasmissione. Siccome l'indice e' influenzato da diversi fattori, e' difficile ottenere un modello molto preciso. Come prima approssimazione si potrebbe utilizzare la seguente relazione:

$$t_{comm} = t_{startup} + wt_{data}$$

Questa relazione indica che il tempo di comunicazione e' dato dal tempo di startup ($t_{startup}$) (essenzialmente il tempo necessario a mandare un messaggio senza nessun dato) piu' il tempo necessario a inviare un messaggio (t_{data}) moltiplicato per il numero di messaggi inviati (w). Anche in questo caso, il tempo di comunicazione complessivo e' dato dalla somma dei singoli tempi di comunicazione

$$t_{comm} = t_{comm1} + t_{comm2} + t_{comm3} + ...$$

Una volta ottenuti i valori di t_s, t_{comp} e di t_{comm} e' possibile calcolare il fattore di speedup (descritto in precedenza). Riscriviamo la relazione sostituendo il valore di t_p

$$speedup = \frac{t_s}{t_{comp} + t_{comm}}$$

Da questa relazione, risulta inoltre evidente come il tempo di comunicazione influenzi direttamente il fattore di speedup. In alcune implementazioni, ad esempio, si potrebbe verificare la condizione in cui il programma passi significativamente piu' tempo a comunicare che ad effettuare calcoli all'aumentare della grandezza del problema. In altri termini, se la complessita' computazionale del tempo di computazione e del tempo di comunicazione sono le stesse, allora difficilmente le performance aumenterebbero all'aumentare della grandezza del problema n. Il rateo computazione/comunicazione da' una stima di come questi due tempi cambiano.

$$comp/comm_{ratio} = \frac{t_{comp}}{t_{comm}}$$

Generalmente una buona implementazione parallela ha un rateo di computazione/comunicazione in cui la complessita' computazionale e' piu' grande di quella della comunicazione. Questo rapporto e' anche espresso spesso come G, che indica la *granularita'* (o grana) della computazione.

> Per grana computazionale si intende la quantita' di dati con la quale si e' suddiviso il problema generale. (Cioe' la grandezza di un task) Ad esempio, se decido di suddividere una griglia LxH per 4, la grana sara' (LxH/4).

3.6.1 Valutazione Empirica

Tutte le considerazioni fatte nella sezione precedente, valgono solo se il paradigma scelto per la computazione parallela e' noto. Nonostante una valutazione teorica sia in linea di massima preferibile, in alcuni casi e' necessario valutare empiricamente le prestazioni del programma parallelo.

4 Paradigmi di programmazione paralleli

Nella programmazione parallela ci sono due macrocategorie principali di paradigmi di programmazione:

- Stream Parallel (task parallel)
- Data Parallel

Lo scopo di tali paradigmi e' di indicare dato un problema, dove viene "*estratto*" il parallelismo. In realta' il libro di testo a cui fa riferimento il corso, segue una classificazione differente, ma comunque le considerazioni ad alto livello fatte in questa sezione rimangono valide.

- Stream Parallel
  - Task Farm
    - Embarassingly Parallel
  - Pipeline
    - Pipelined Computations
- Data Parallel
  - Synchronous Computations
    - Locally Synchronous
    - Globally Synchronous
  - Asynchronous Computations

4.1 Stream Parallel

Nel caso stream parallel, l'assunzione e' quella di avere uno stream (lista unbounded) di dati. Un modo per rappresentarlo essenzialmente e' pensare ad uno stream come ad una lista di dati che non sono tutti disponibili fin da subito. Per questa ragione, l'accesso ai dati di uno stream richiede che ogni elemento sia acceduto "in sequenza", per cui e' differente da un array in cui e' possibile accedere in modo random in qualsiasi posizione. Le forme di parallelismo piu' comuni nel caso stream parallel sono a loro volta *pipeline* e *farm*.

4.1.1 Pipeline

Nel caso della pipeline l'idea e' quella di calcolare una funzione f per uno stream di dati inbound. Tale funzione e' possibile scomporla in diverse sottofunzioni indipendenti tra di loro, formalmente e' possibile rappresentare f come composizione di n funzioni

$$f = f_1 \cdot f_2 \cdot \cdots \cdot f_n$$

In questo modo, le funzioni possono essere eseguite in parallelo tra di loro, stile "catena di montaggio".

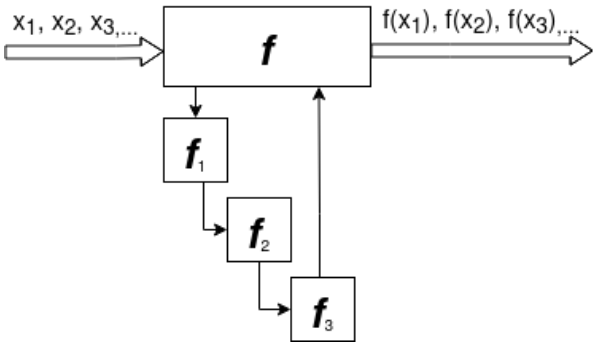

Figura 13: Modello di esecuzione a pipeline

Se nel caso di una pipeline sequenziale il tempo di esecuzione e'

$$T_{seq} = T_{f1} + T_{f1} + \cdots + T_{fn}$$

in una pipeline parallela, invece, si ha che il tempo di esecuzione e'

$$T_{par} = \max_{i \in 0,\ldots,n} (T_{fi})$$

In questa forma di parallelismo, il caso migliore e' evidentemente quello in cui ***tutti i tempi di esecuzione delle singole*** f ***sono uguali***, in modo che il tempo massimo non penalizzi gli altri tempi. Il problema del pipelining puo' essere essenzialmente evidenziato per mezzo di un esempio: Se supponiamo che il tempo piu' grande sia quello di T_{f3} e che tutti gli altri tempi siano minori e che si comunichi con primitive di comunicazione asincrone bloccanti, allora il buffer di ricezione del processo che gestisce f_3 molto probabilmente soffrira' di overflow per le troppe richieste provenienti dai livelli inferiori della pipeline. Dimensionare tale buffer e' inoltre impossibile.

4.1.2 Task Farm

Nel caso del paradigma task farm, l'idea e' invece quella di avere un processo iniziale E che si occupa di distribuire i differenti task ad elementi di calcolo diversi, ma in cui tutti processano la funzione f. In questo senso differisce dalla pipeline in cui i diversi processi calcolavano le sottofunzioni f_n. La Figura 14 riassume graficamente il paradigma.

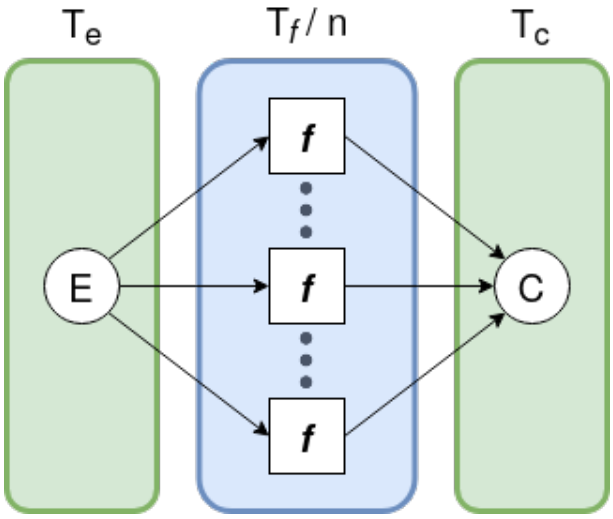

Figura 14: Modello di esecuzione task farm. L'elemento di computazione E(mitter) manda ai workers (f) gli elementi dello stream da computare. Una volta computati, il processo C(ollector) colleziona i risultati dai workers.

Il tempo di esecuzione sequenziale di un modello task farm e' pari al tempo di esecuzione di f stesso

$$T_{seq} = T_f$$

Nel caso parallelo, supponendo che si abbiano n elementi di calcolo (workers), come prima approssimazione si ha

$$T_{par} = \frac{T_f}{n}$$

In pratica, se vediamo l'intero sistema come una pipeline in cui $T_E, (T_f = T_f/n), T_C$ sono i tempi di esecuzione rispettivamente dell'emitter, della funzione e del collector, abbiamo che il tempo di esecuzione reale e' pari a quello di una pipeline

$$T_{par} = \max\{T_E, \frac{T_f}{n}, T_C\}$$

Se assumiamo ora a scopo di semplificazione che $T_E = T_C$(tempo di dispatching e collection uguali). Dal momento che abbiamo detto che per una pipeline le performance migliori si raggiungono quando i tempi sono tutti uguali, allora possiamo imporre che

$$n \leq \frac{T_f}{T_E}$$

Questa relazione evidenzia come le prestazioni di questo paradigma aumentino linearmente fino ad un threshold $\tilde{n}$. Questa situazione evidenzia un collo di bottiglia, dato appunto dai tempi di servizio dell'*emitter* e del *collector*. Un modo per aumentare $\tilde{n}$ il piu' possibile e' quello di fare batching di task (aumentare la grana computazionale).

4.2 Data Parallel

Nel data parallelism, l'impostazione e' diversa rispetto allo stream parallellism. I dati non sono piu' sottoforma di generico stream in cui i dati non erano gia' tutti disponibili, ma sono invece gia' tutti presenti all'interno di una struttura (lista, matrice, tensore ecc..) per cui e' possibile gia' accederli tutti insieme. Il paradigma data parallel si occupa di esplicitare il parallelismo dividendo i dati (gia' presenti) tra i diversi elementi di calcolo. Principalmente, ci sono due sottocategorie principali di data parallelism:

- Globally Synchronous: *tutti* i processing elements necessitano di sincronizzarsi tra di loro (tipicamente mediante una barrier)
- Locally Synchronous (o *stencil*): *alcuni* processing elements necessitano di sincronizzarsi tra di loro

Alternativamente possiamo caratterizzare le computazioni data parallel nel modo seguente (prendendo come esempio di struttura dati una *lista*):

- $mapf[a_0, \ldots, a_n] = [f(a_0), \ldots, f(a_n)]$
- $reduce \oplus [a_0, \ldots, a_n] = a_0 \oplus \cdots \oplus a_n$

- $stencil f[a_0, \ldots, a_n] = [\ldots, f(a_{i-1}, a_i, a_{i+1}), \ldots]$ (in questo caso si e' presa come esempio una funzione a 3 parametri, ma in generale ne ha n)

5 Embarassingly Parallels Computations

Nella parallelizzazione di programmi sequenziali, quello che si vuole ottenere idealmente e' una suddivisione del problema in parti diverse (idealmente p) che saranno eseguite in parallelo dai diversi p processori. Molti problemi sono facilmente divisibili mentre altri no. La divisibilita' deriva principalmente dalla presenza o meno di dipendenze tra operazioni che devono essere eseguite in un ordine preciso. Quando questa dipendenza e' assente, le operazioni possono essere eseguite indipendentemente su ogni dato, per cui e' possibile parallelizzare tali operazioni. Molti problemi hanno questa caratteristica, tanto che sono stati chiamati problemi "*embarassingly parallels*", proprio per la facilita' con cui si puo' ottenere una suddivisione del problema sequenziale per ottenere una versione parallela.

> Nelle computazioni embarassingly parallels le comunicazioni tra le unita' di computazione e' ridotta al minimo se non del tutto assente

Siccomes stiamo parlando di computazioni che ricadono nel paradigma *Task farm*, le computazioni embarassingly parallels sono anch'esse costituite da un `Master` e diversi `Workers`. Come gia' detto, una computazione embarassingly parallel e' riassumibile nei seguenti passi:

1. Il `Master` divide il lavoro per il numero di `Workers`
2. Il `Master` comunica il lavoro ad ogni `Worker`
3. Il `Master` colleziona i risultati dei `Workers`

I workers possono essere sia creati *staticamente* (e quindi partono insieme al master) oppure *dinamicamente* (per cui vengono inizializzati dal master). Analizziamo ora alcuni esempi di computazioni di questo genere.

5.1 Processing di immagini a basso livello

Nel caso di processing di immagini, l'idea e' quella di dividere una matrice di valori `RGB` per un certo numero di workers. Il concetto di grana qui e' molto semplice: si tratta appunto del "*quadrato*" di pixels che viene assegnato al singolo worker. Al posto di mandare un singolo pixel ad ogni worker, si aggregano i dati in modo da risparmiare sui tempi di inizializzazione della comunicazione t_0. Ogni worker poi calcolera' la funzione sulla sua porzione di immagine associata, ad esempio un Gaussian Blurring, un filtro mediano o qualsiasi altro filtro per image processing.

5.2 Calcolo dell'insieme di Mandelbrot

L'insieme di Mandelbrot e' essenzialmente un insieme di punti nel piano complesso per cui una funzione z_k non diverge oltre un certo limite *threshold*. Il generico passo $k+1$ e' calcolabile nel modo seguente

$$z_{k+1} = z_k^2 + c$$

Dove c e' il punto nel piano complesso che si sta considerando, mentre $z_0 = 0$. Il calcolo della funzione viene re-iterato fin quando una di queste due condizioni viene soddisfatta:

1. $z_{length} = \sqrt{a^2 + b^2} > 2$, cioe' la magnitudine di $z = a + bi$ e' piu' grande di 2
2. $k \geq t$, cioe' il numero di iterazioni k raggiunge un certo limite t

Per plottare l'insieme, si colora il pixel in base a quante iterazioni k sono necessarie per superare la norma (o il threshold). Siccome ogni punto e' calcolabile in maniera indipendente, possiamo applicare il paradigma, dividendo l'immagine in un numero di righe pari al numero di worker per poi far calcolare ad ognuno di essi la loro porzione. Un problema evidente di questo tipo di approccio, e' che alcune zone dell'insieme di mandelbrot hanno una densita' piu' alta di punti che non convergono rispetto ad altre, con la conseguenza che alcuni workers dovranno computare molto piu' di altri. Questo problema e' un problema di *bilanciamento del carico*, e puo' essere risolto con l'assegnazione dei task dinamica. Possiamo quindi distinguere due tipologie di assegnamento dei tasks:

- **Dinamica** (*on-demand*): le unita' di computazione richiedono la prossima parte da computare al master dopo aver computato la precedente
- **Statica**: dividi semplicemente la regione in un numero fisso di parti, ognuna assegnata ad un'unita' di computazione. La regola di assegnazione e' arbitraria (dettata dalla policy di schedulazione).

Nella maggior parte dei casi, l'assegnazione *on-demand* e' quella ottimale, proprio grazie alla sua caratteristica di bilanciamento automatico e naturale del carico. Come contro, l'assegnazione *on-demand* soffre nei problemi in cui la grana computazionale e' particolarmente "fine". Questo perche' i workers potrebbero stare tanto tempo ad aspettare che il master comunichi i task ai workers prima di lui. Quindi, all'aumentare della finezza della grana si accentua questo fenomeno in cui i workers passano sempre piu' tempo ad aspettare che il master gli comunichi il prossimo task da esguire.

5.3 Calcolo del Pi-Greco con il metodo Monte Carlo

Impiegare il metodo montecarlo per il di π consiste essenzialmente nella generazione random di un numero arbitrario di punti (x, y) (mediante due generatori di numeri casuali) all'interno di un quadrato. L'intuizione e' che contando i punti dentro al cerchio e i punti fuori dal cerchio e considerandone il

rapporto si ottiene un'aprrossimazione del valore di π. Tanto piu' il numero di punti $n \to \infty$ tanto sara' piu' precisa l'approssimazione. Questo poiche' il rapporto tra l'area del cerchio e l'area del quadrato inscritto in esso e' pari a π.

Un'altra applicazione particolarmente interessante del metodo e' per stimare il valore di un integrale in funzioni non integrabili o particolarmente complesse da integrare. Per calcolarlo basta sfruttare le somme di *Riemann*

$$\int_{x_1}^{x_2} f(x)dx = \lim_{N\to\infty} \sum_{i=1}^{N} f(x_r)(x_2 - x_1)$$

in cui x_r e' un punto generato casualmente.

5.3.1 Generazione di numeri casuali

Un'accortezza particolare per questo tipo di applicazioni deve essere prestata per la generazione di numeri casuali in applicazioni parallele.

> *Non c'e' modo di avere un'implementazione sequenziale che usi un determinato generatore di numeri sequenziali e un'implementazione equivalente parallela che usi lo stesso generatore. In generale, non possono mai produrre lo stesso risultato.*

Supponiamo di avere un programma sequenziale che utilizza un solo processore P. Tale programma inoltre utilizza un generatore s_0, che genera la sequenza $r_1, r_2, \ldots, r_n$ di numeri casuali. Il problema sorge quando si vuole parallelizzare tale programma. L'idea e' che se si volesse far generare dei numeri casuali a piu' processori P_i con lo stesso generatore di numero casuale, si finirebbe col generare n volte la stessa sequenza. La situazione e' descritta graficamente in Figura 15.

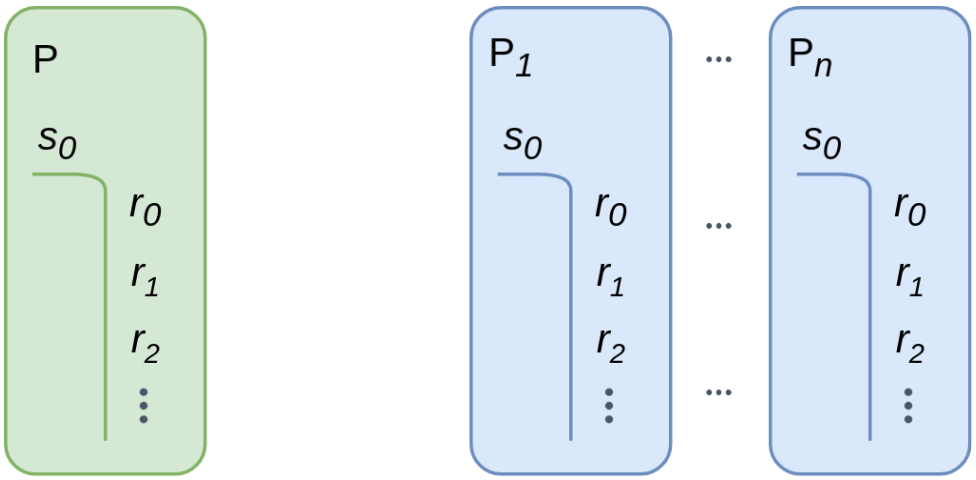

Figura 15: A sinistra (verde) l'implementazione sequenziale utilizza s_0 per generare la sequenza. A destra (blu), dal momento che ogni processore P_i utilizza s_0 genera la stessa sequenza di numeri casuali, di fatto portando a n duplicati della stessa sequenza

Per risolvere questo problema senza sacrificare la determinabilita' (data dal seed) si procede nel modo seguente:

1. Inizialmente un processo prediletto si occupa di generare randomicamente mediante un determinato algoritmo (ad esempio *Mersenne Twister*) un seed per ogni processo.
2. I processi, dopo aver ricevuto il seed dal processo prediletto, utilizzano un *algoritmo differente* (as esempio *Modulo* p) da quello utilizzato per generare i seeds, per generare la loro sequenza.

In questo modo non si sacrifica la determinabilita' ma si ottengono sequenze statisticamente corrette per ogni processo coinvolto nella computazione. L'unico problema e' che non c'e' modo di ottenere una controparte sequenziale a singolo generatore.

6 Strategie *Divide et Impera*

La strategia *divide et impera* e' una metodologia sistematica molto utile per la soluzione di diversi problemi molto ricorrente in computer science. Sostanzialmente l'idea e' quella di suddividere un grosso problema in diversi sottoproblemi di minor dimensione, ma della stessa forma del problema originale (tipicamente ottenuta mediante *ricorsione*). Ogni soluzione di questi sottoproblemi di portata piu' piccola viene poi combinata con le altre soluzioni, in modo da ottenere la soluzione del problema originale. Una strategia divide et impera e' in generale applicabile quando i problemi impongono una struttura di tipo gerarchico. Si pensi, ad esempio, agli algoritmi di ordinamento. Alcuni di essi, nonostante operino su una struttura dati di tipo lineare (*array*), hanno comunque una struttura imposta dalla computazione gerarchica, per cui e' possibile sfruttare questa strategia.

> *La strategia dividi e conquista e' diversa dal semplice partizionamento, che consiste invece in una semplice divisione del problema in parti.*

Nel contesto del parallel computing e' possibile trarre vantaggio da questa strategia di pensiero, siccome e' possibile sfruttare la suddivisione per estrarre il parallelismo. L'idea e' semplicemente quella di utilizzare diverse unita' di calcolo per risolvere i diversi sotto-problemi, aggregando infine i singoli risultati.

L'implementazione che puo' risultare banale dal punto di vista sequenziale, nasconde delle difficolta' quando si parla di implementazione parallela. Questo perche' la ricorsione necessita' di uno *stack*, che per essere implementato richiede l'allocazione dinamica della memoria, che e' un problema difficile in ambienti distribuiti. Addirittua nelle GPU e' sconsigliato, perche' non supportato e a causa di prestazioni spesso scadenti.

Vediamo ora alcuni esempi di problemi che possono sfruttare questo modo di risoluzione.

6.1 Operazioni su strutture dati lineari

Consideriamo delle operazioni di folding su delle strutture dati lineari. In linea di massima, data un'operazione di aggregazione che sia *associativa*, si vuole applicarla su un'intera sequenza di elementi per ottenere un'aggregazione finale. E' anche chiamata `foldr` o `reduce`. Formalmente, dato l'operatore associativo $\oplus$ e una sequenza di elementi $[e_1, e_2, \ldots, e_n]$, si vuole ottenere $e_1 \oplus e_2 \oplus \cdots \oplus e_n$.

In questo caso, l'idea e' quella di suddividere la sequenza progressivamente fino ad una grandezza stabilita (che rappresenta appunto la grana computazionale), per poi far eseguire l'operazione di associazione sulle sottosequenze ottenute da diverse unita' computazionali. I risultati di queste operazioni, a loro volta, saranno un'altra sequenza di elementi, che sara' combinata nello stesso modo per ottenere il risultato finale.

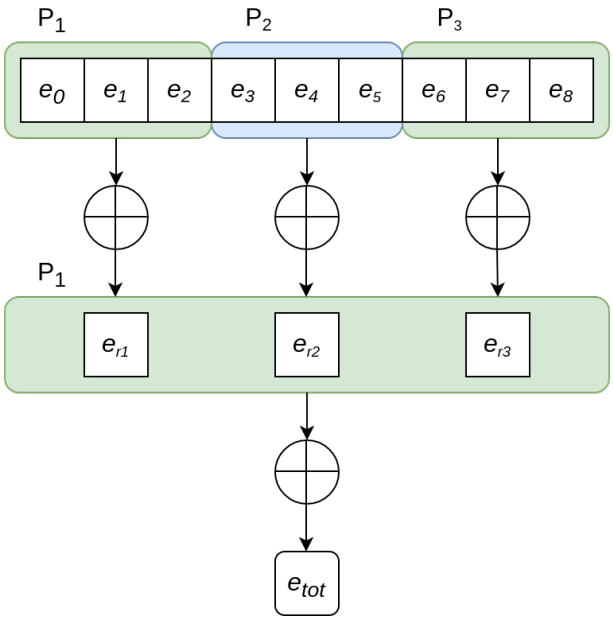

Figura 16: Operazione di reduce risolta con strategia dividi e conquista. La lista originale e' divisa in 3 sottoliste risolte da differenti unita' computazionali. I risultati formano a loro volta una lista, risolta infine da un'unita' computazionale predefinita in modo da ottenere il risultato finale.

6.2 Algoritmi di ordinamento

Un algoritmo di ordinamento che sfrutta particolarmente bene questo tipo di parallelizzazione e' il *bucket sort*. A questo punto, al lettore verrebbe da pensare perche' non siano stati scelti algoritmi noti per essere performanti come ad esempio il *quicksort*. La risposta e' che un algoritmo come il quicksort non e' di facile parallelizzazione, per evidenti problemi di bilanciamento del carico.

L'idea alla base della parallelizzazione del *bucket sort*, e' che ogni unita' di calcolo si scelga il proprio range operativo, cioe' il range di numeri da ordinare all'interno della lista. Una volta scelto il range, vengono presi tutti i numeri compresi in questo range ed organizzati in un "*bucket*" - che non sono altro che un insieme non ordinato di numeri. Successivamente, viene applicato un algoritmo di ordinamento sequenziale al bucket, ottenendo una lista che puo' essere fusa con le altre "liste-risultato" delle altre unita' di calcolo. E' bene notare che l'operazione di unione finale e' immediata, dal momento che ogni unita' di calcolo conosce il punto in cui deve stare la propria sottolista all'interno della lista risultato.

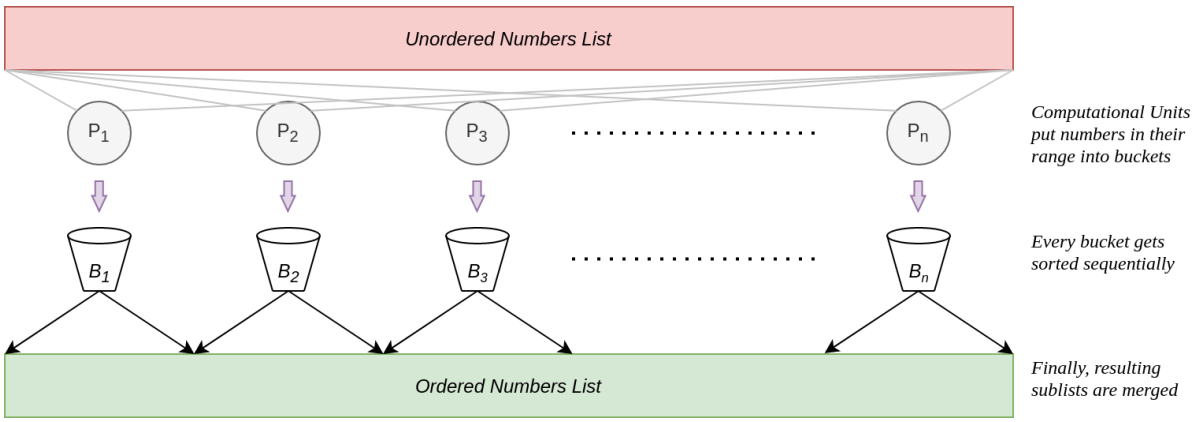

Figura 17: Bucketsort parallelo

Un evidente problema di questo tipo di soluzione, e' evidenziato dal fatto che ogni unita' computazionale necessita di scegliere il proprio range operativo. Per far cio', pero', deve essere in grado di leggere l'intera lista di numeri, per cui ne consegue che una delle seguenti condizioni deve essere vera:

- Ogni unita' di calcolo ha accesso all'intera struttura
- Ogni unita' di calcolo ha una copia dell'intera struttura

Una versione piu' raffinata di questo algoritmo e' quella che consiste nell'assegnare ad ogni UC una sottolista, da cui a sua volta verranno generati diversi *sub-buckets*. In questo modo, cio' che viene replicato in ogni UC non e' l'intera lista in input, ma i *buckets*. Nella fase successiva, ogni UC scambia i *sottobuckets* con gli altri processi, in modo che ogni UC collezioni tutti i sottobuckets corrispondenti al proprio range operativo. Alla fine di questa fase, ogni UC avra' ottenuto il bucket corrispondente al proprio range operativo, su cui potra' applicare un algoritmo di ordinamento sequenziale e infine un'operazione di merge con le altre liste.

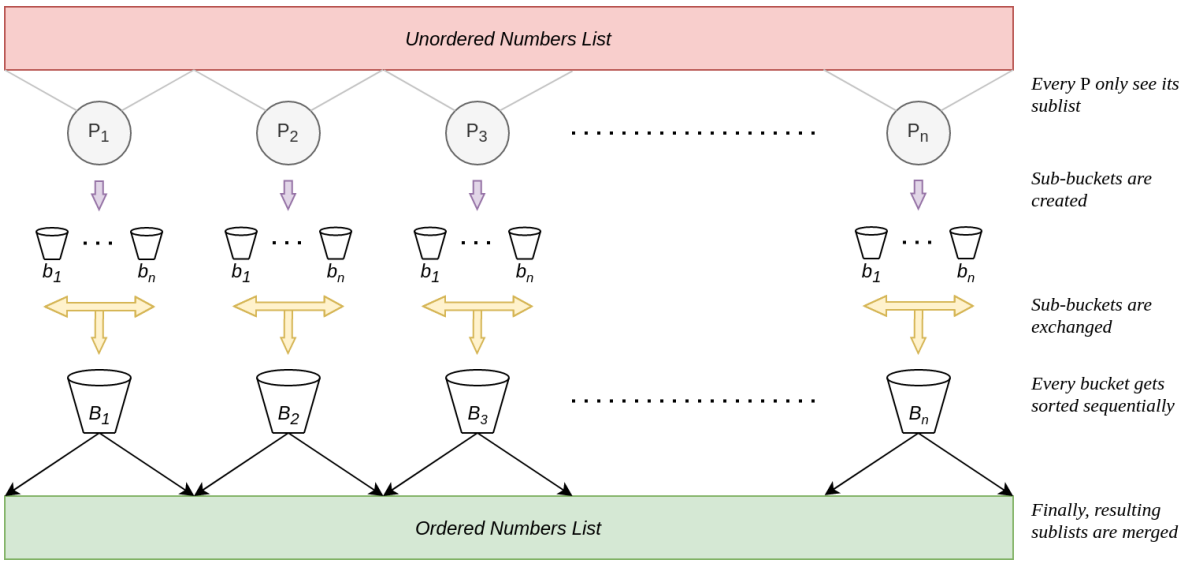

Figura 18: Bucketsort parallelo (versione finale)

Questa versione necessita di mettere tutte le unita' computazionali in comunicazione tra loro. Per far cio' esistono primitive messe a disposizione ad esempio da `MPI`, come la primitiva di tipo *all-to-all*. Tale primitiva corrisponde essenzialmente ad un esecuzione di un'operazione di `gather` + `broadcast`, ma e' ottimizzata per lo scopo, di fatto essendo piu' efficiente. L'efficienza risiede nel fatto che non c'e' una centralizzazione intermedia del risultato, per cui ogni processo manda la sua parte ad ogni altro processo, cosi' come gli altri.

6.3 Quadratura Adattiva

Precedentemente abbiamo visto nei metodi embarassingly parallel con assegnamento di task statico come computare l'AUC (*Area Under the Curve*) di una funzione f. In quel caso pero', si supponeva di dividere l'area in intervallli di lunghezza δ, rendendo di fatto impossibile impiegare l'algoritmo per raggiungere una precisione prestabilita. Un approccio possibile per risolvere il problema potrebbe essere quello di iniziare da un intervallo per ogni unita' computazionale, e ridurlo successivamente finquando non si raggiunge la precisione stabilita. Un modo per determinare se la precisione e' sufficiente, consisterebbe nel considerare 3 aree A, B e C, controllando successivamente se $C - (A + B)$ scende al di sotto di un threshold predefinito. Siccome il carico di lavoro varia in base alle zone della funzione, e' piu' appropriato impiegare tecniche di bilanciamento del carico viste in precedenza.

6.4 Problema *N-Body*

L'ultimo problema visto che puo' sfruttare al meglio il metodo di dividi e conquista e' proprio l'*N-body problem*. Il problema consiste essenzialmente nel calcolare le posizioni di *N* corpi celesti nello spazio, soggetti a a forze gravitazionali impresse dagli altri corpi. La forza gravitazionale di due corpi di massa m_a e m_b e' data dalla legge gravitazionale seguente:

$$F = \frac{Gm_a m_b}{r^2}$$

Un corpo soggetto alla forza gravitazionale degli altri corpi accellerera' di conseguenza secondo la seconda legge di Newton

$$F = ma$$

Il cambiamento della forze nel tempo e' spesso descritto in termini di equazioni differenziali

$$F = \frac{mdv}{dt}, \quad v = \frac{dx}{dt}$$

Dove v e' il vettore della *velocita'*. Per discretizzarle basta segliere un valore Δt appropriato, ottenendo di conseguenza

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}, \qquad v^{t+1} = v^t = \frac{F\Delta t}{m}$$

Inoltre, la posizione di ogni corpo e' data da

$$x^{t+1} - x^t = v\Delta t$$

Si hanno tutti gli elementi per ottenere un codice sequenziale per la soluzione del problema

```
1  for (t = 0; t < tmax; t++) {                /* for each time period */
2      for (i = 0; i < N; i++) {               /* for each body */
3          F = Force_routine(i);               /* compute force on the ith
               body */
4          v[i]_new = v[i] + F * dt / m;       /* compute ith v(t+1) */
5          x[i]_new = x[i] + v[i]_new * dt;    /* compute ith x(t+1) */
6      }
7      for (i = 0; i < N; i++) {               /* update velocities and
          positions */
8          x[i] = x[i]_new;
9          v[i] = v[i]_new;
10     }
11 }
```

Essenzialmente consiste in due cicli annidati che "*spazzolano*" le due dimensioni del problema sul tempo e nel numero di nodi. La presenza dei due cicli ci suggerisce che la complessita' computazionale e' $O(n^2)$. Il problema e' che nel dominio di riferimento, n e' spesso molto grande.

Per la versione parallela, si utilizza l'***algoritmo di Barnes-Hut*** che si basa appunto su un approccio *divide et impera*. Essenzialmente l'idea e' quella di assumere che se un corpo e' relativamente distante da un agglomerato di altri corpi (che sono tutti vicini tra loro), allora la forza puo' essere calcolata in relazione al *centro di massa* del cluster e non in relazione ad ogni corpo di cui e' composto. I passi principali dell'algoritmo possono essere riassunti come:

1. A partire da uno spazio in cui tutti i corpi sono contenuti in un cubo, dividi tale cubo in 8 sotto-cubi
2. Se un sottocubo non contiene nessun corpo, allora elimina il sottocubo
3. Se un sottocubo contiene un corpo viene mantenuto
4. Se un sottocubo contiene piu' di un corpo, allora viene diviso ricorsivamente fin quando ogni sottocubo contiene esattamente un corpo

Una di queste strategie di suddivisione e' chiamata *Orthogonal Recursive Bisection*. Essenzialmente consiste nel dividere lo spazio lungo gli assi principali, cambiando asse dopo ogni divisione. La divisione deve essere eseguita in modo che ogni partizione abbia lo stesso numero di corpi (ovviamente per numeri pari di corpi). Ad esempio, per suddividere uno spazio in 2D si divide preliminarmento lo spazio lungo l'asse y ottenendo due partizioni. Successivamente, ogni partizione ottenuta dalla divisione precedente viene divisa lungo l'asse x. Il procedimento viene poi ripetuto per ogni partizione.

Questa suddivisione dello spazio puo' essere codificata all'interno di una struttura dati chiamata *octree* (cioe' un albero in cui ogni nodo puo' avere fino ad 8 figli). Una volta costruito l'albero, il centro di massa totale di ogni sottocubo viene salvato nei nodi. In questo modo, se voglio sapere la forza rispetto ad un un determinato nodo e' sufficiente percorrere l'albero dalla radice al nodo stesso. Sia la costruzione di un *octreee* che la sua *visita* hanno complessita' $O(n \log n)$.

7 Computazioni Pipelined

Seppur gia' introdotte precedentemente come *stream parallelism*, riprendiamo la trattazione che segue il libro su questo tipo di parallelismo. Le computazioni pipelined possono essere impiegate per risolvere diverse classi di problemi. Essenzialmente l'idea e' quella di dividere un singolo problema in una serie di *step* che sono legati da una dipendenza di terminazione tra i precedenti (l'*n-esimo* step puo' essere eseguito solo dopo l'*(n-1)-esimo*).

> Nel contesto del parallel computing, il parallelismo viene sfruttato assegnando ad ogni step della pipeline un unita' computazionale differente.

La differenza essenziale tra gli altri metodi discussi fin'ora, e' che questo metodo si basa su una suddivisione ***funzionale*** del problema. Cio' significa che non vengono piu' suddivisi i ***dati***, ma e' lo stesso problema che viene suddiviso in piu' parti che possono essere eseguite in modo parallelo. L'idea importante e' che nelle pipelined computations si vuole *"spezzare"* una funzionalita' in tante funzionalita' piu' piccole che possono potenzialmente essere eseguite in parallelo.

Il pipelining puo' essere impiegato per ottenere uno speedup essenzialmente nei casi in cui ci si trovi in una delle tre possibili situazioni:

1. Se deve essere eseguita piu' di una istanza del problema completo
2. Se una serie di dati devono essere processati per cui vengono richieste diverse operazioni
3. Se le informazioni necessarie a far partire il prossimo processo possono essere passate prima che il processo abbia completato tutte le prorie operazioni interne

In linea di massima ad ognuna di queste situazioni corrisponde una tipologia di pipeline. L'idea della pipeline di tipo 1 (implementata da `Intel TBB` [5]) e' quella di far eseguire la stessa funzione su dati diversi da unita' computazionali differenti. Ad esempio, supponiamo di voler eseguire la funzione $F = f_1; f_2; f_3$ (dove ; indica una relazione di sequenzialita' delle funzioni) per ogni istanza di un dataset $X = \{x_1, \ldots, x_n\}$. In questo caso, quello che si fa semplicemente e' assegnare ogni x_i ad ogni unita' computazionale (*pipeline step*) che eseguira' $F(x_i)$. Ogni processing element computa *tutte le funzioni* (poiche' $F = f_1 \cdot \cdot \cdot \cdot \cdot f_n$) su un determinato dato. La pipeline di tipo 1 viene utilizzata molto spesso dai *Workflow Engines*.

La pipeline di tipo 2, invece, consiste nel far eseguire le *singole* funzioni f_i da ogni processing element. In altri termini, si fanno eseguire tutte le parti di F $(f_1, \ldots, f_n)$ da processing element differenti $P_1, \ldots, P_n$.

> La differenza sostanziale e' che nella pipeline di tipo 1 ogni processing element calcola **tutte**

> le funzioni su un certo dato, mentre nel tipo 2 ogni processing element calcola **una specifica** funzione su tutti i dati.

La pipeline di tipo 3 non e' molto diversa dalle precedenti. La differenza e' che nelle altre pipeline il risultato veniva passato al prossimo stage alla fine di ogni stage, cioe' quando tutte le operazioni di un determinato stage erano terminate. In questo caso, invece, il passaggio al prossimo stage avviene prima che tutte le operazioni di un determinato stage siano terminate. Se lo vediamo nel contesto di parallel computing, l'UC assegnata ad un particolare stage continua ad eseguire operazioni anche dopo aver fatto partire il prossimo stage.

TODO: Aggiungere immagini pipelines

Fino ad ora, in tutte le pipelines si e' ipotizzato che il numero di stadi fosse uguale al numero di processori. In generale quando questo non accade si tende a ridurre il parallelismo, raggruppando piu' stadi nella stessa UC. Il parallelismo della pipeline ha delle limitazioni legate inerentemente alla propria struttura: ll numero massimo di task che vengono eseguiti in parallelo e' pari al numero di stage della pipeline. Questo pone delle limitazioni non indifferenti, soprattutto se si vuole raggiungere un grado di parallelismo di ordine molto alto.

> In generale, e' possibile trasformare le pipelines di tipo 2 in pipelines di tipo 1. L'idea e' quella di comporre gli stage della pipeline di tipo 2 trasformandola in una pipeline di tipo 1, e utilizzare un paradigma di tipo *task farm* per eseguire gli stage.

Se ipotizziamo che le pipelines siano composte da p steps, che siano di tipo 1 o 2, e che debbano eseguire m istanze del problema, allora la pipeline eseguira' tutte le m istanze in $(p-1)+m$ steps. Un'altra considerazione puo' essere fatta sull'implementazione di pipelines. In generale le pipelines richiedono che le UC siano connesse direttamente tra loro. Una struttura di interconnessione molto ideale e' ad esempio quella ad anello o a linea.

Vediamo ora i diversi problemi che possiamo risolvere con le diverse strutture a pipeline viste.

7.1 Somma di una lista di numeri

Anche in questo caso possiamo sfruttare questa astrazione per sommare una lista di numeri in modo parallelo. In linea di massima e' possibile implementare una pipeline per lo scopo in due modi differenti. Il primo consiste nel mappare uno stage ad ogni posizione nella lista, per cui ogni stage avra' un numero della lista. Ogni stage riceve la somma accumulata dagli altri step, ne somma il proprio elemento, e inoltra il risultato al prossimo step della pipeline. Ogni step, "consumato" il proprio elemento utilizzera' poi l'elemento corrispondente della prossima lista. Siccome la pipeline funziona per diverse istanze (lista) dello stesso problema, stiamo parlando di una pipeline di tipo 1.

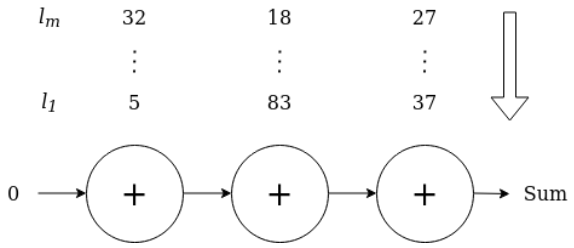

Figura 19: Esempio di pipeline per la somma di liste di 3 numeri della prima tipologia. Diverse liste vengono sommate dalla pipeline. Ogni processing element effettua la somma del numero assegnato alla propria posizione sulla lista con il risultato accumulato dallo stage precedente. La somma viene poi passata allo stage successivo.

7.2 Ordinamento di numeri

Un'altro impiego della pipeline e' quello dell'ordinamento di liste di numeri, implementando di fatto una versione parallela dell'*insertion sort*. Per semplicita', consideriamo una pipeline che abbia tanti steps quanti elementi abbia la lista da ordinare. Inizialmente, ogni step avra' al proprio interno il numero piu' piccolo possibile ($-\infty$). Successivamente, ogni elemento e_r ricevuto dal singolo step viene confrontato con l'elemento interno dello step e_i:

- Se $e_r > e_i$, allora passa e_i al prossimo stage e imposta $e_i = e_r$
- Altrimenti, passa e_r al prossimo stage, senza alterare e_i

Alla fine della pipeline, la lista ordinata sara' contenuta all'interno di ogni step della pipeline. Il metodo migliore per collezionare il risultato e' quello di far passare poi gli elementi all'indietro nella pipeline, in modo che il primo step contenga la lista ordinata. Questa struttura e' detta ad *anello*. La tipologia di pipeline utilizzata e' chiaramente la tipologia 1, siccome e' utilizzata per ordinare una sola lista (*singola istanza del problema*).

8 Computazioni Sincrone

Le computazioni sincrone sono delle computazioni *data parallel* che consistono nella regolare sincronizzazione di tutte le UC del programma. Possiamo identificare essenzialmente due tipologie di computazione sincrona:

- Globalmente sincrona
- Localmente sincrona

8.1 Computazioni Globalmente Sincrone

Nella loro forma piu' semplice le UC sono assegnate a computare parallelamente dei dati (spesso indipendenti), fino ad un punto prestabilito del programma. Tale punto e' detto *barrier*, e una volta raggiunto, ogni UC dovra' aspettare fino all'arrivo dell'ultima UC alla barrier (quella piu' lenta). In questo modo si sincronizzano tutte le UC allo stesso punto. A seguito di questa operazione, spesso le UC si scambiano anche dati in modo da poter far continuare la computazione globale. Infine, il ciclo si ripete da capo. Siccome i dati in cui operano le UC sono indipendenti, questa tipologia di computazione e' detta anche *data parallel*. Possiamo individuare diverse ragioni per cui questo tipo di computazioni e' particolarmente conveniente:

- Facilita' di implementazione (essenzialmente e' un singolo programma)
- Possibilita' di scalare facilmente a problemi di dimensioni maggiori
- Molti problemi numerici/non numerici possono essere riformulati in una forma di computazione *data parallel*

> Il paradigma *data parallel* puo' essere visto anche come la parallelizzazione dei loops

Se noi prendiamo in considerazione un programma SPMD che implementa il paradigma data parallel, notiamo che ogni UC dovra' avere accesso a tutti i dati dell'applicazione. In `MPI`, la primitiva `AllGather` serve proprio a sincronizzare i risultati derivanti dalle altre UC. Per spiegare questa primitiva, prendiamo per esempio un caso in cui n processi effettuano delle computazioni su un array. Ogni processo effettua dei calcoli sulla propria porzione dell'array, e successivamente aspetta il risultato di tutte le altre UC. La primitiva `AllGather` fa proprio questo: inserisce il risultato della computazione dell'UC corrente nell'array, e colleziona il risultato di tutte le altre nelle loro rispettive porzioni di array.

> Lo stochastic gradient descend distribuito utilizza questa primitiva per condividere con tutti i processi i gradienti ottenuti, in modo che ogni processo possa ripartire dall'average gradient ottenuto

La grana computazionale di solito non e' particolarmente fine in questi problemi, sopratutto nel caso in cui i dati su cui bisogna operare sono tanti. In altri termini, il numero delle unita' computazionali e' tipicamente piu' basso del numero di dati su cui tali unita' devono andare ad operare. In genere ci sono due strategie di partizionamento dei dati:

- **Block allocation**: consiste nell'allocamento di sezioni di dati consecutive alle unita' computazionali in ordine crescente
- **Cyclic allocation**: consiste nell'allocare ciclicamente diverse porzioni di dati alle diverse UC

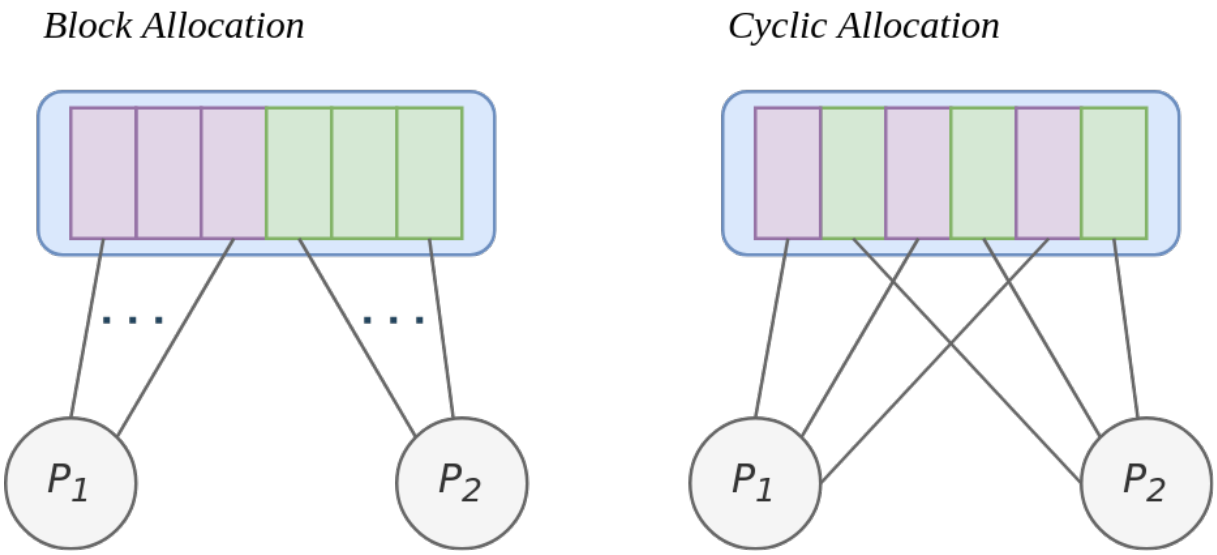

Figura 20: Strategie di partizionamento a confronto

E' stato visto empiricamente che piu' la grana computazionale diminuisce piu' il tempo di comunicazione aumenta e il tempo di computazione diminuisce. In questo caso non si puo' raggiungere oltre un certo upper bound, per cui le computazioni globally synchronous non hanno una scalabilita' molto grande. Le computazioni locally synchronous, sono invece in generale piu' scalabili.

8.1.1 Considerazioni sull'operazione di barrier

Ci sono essenzialmente due modi per poter sospendere l'esecuzione di una singola UC nella barrier:

- Tramite primitiva di sistema operativo (l'*SO* sospende il processo mettendolo nella coda di I/O)
- Tramite attesa attiva (ad esempio tramite *spinlock*)

In `MPI` abbiamo una primitiva che e' `MPI_Barrier()` con solo parametro `MPI_COMMUNICATOR`, che indica il gruppo di UC su cui fare l'operazione. Un'implementazione *"banale"* consiste nel designare

una UC a tenere un **contatore**. Ad ogni messaggio arrivato, questa UC va ad aumentare il contatore, fino a quando questo non sara' pari al numero di processi che partecipano alla barrier. Una volta raggiunto, tale UC mandera' un messaggio a tutti gli altri processi in modo che possano ripartire insieme.

> Una buona implementazione della barrier deve tenere conto del fatto che tale operazione puo' essere esegita piu' volte, soprattutto all'interno di un costrutto iterativo.

Le barrier del tipo descritto (basate su contatori) possiamo essenzialmente dividerla in due fasi principali:

- Una UC entra nella fase di arrivo e non la lascia fino a quando tutte le altre altre UC siano arrivate in questa fase
- Successivamente, le UC lasciano la fase di arrivo venendo rilasciate

Vediamo ora un'implementazione possibile della tipologia di barrier descritta in *pseudo C*. Iniziamo con il codice del **Master**, cioe' l'UC designata di tenere conto degli elementi presenti nella fase di arrivo.

```
1 for (i = 0; i < n; i++)
2     recv(P_any);
3 for (i = 0; i < n; i++)
4     send(P_i)
```

Invece, il codice delle singole UC che prendono parte alla barrier e' il seguente

```
1 send(P_master)
2 recv(P_master)
```

Questa implementazione a contatore, non e' la piu' efficiente siccome il *master* deve ricevere e mandare i messaggi in una maniera sequenziale (lineare). Con un numero molto grande di UC, potrebbe avere delle performances poco soddisfacenti. Un'alternativa a questa barrier consiste quindi nell'utilizzare una struttura ad albero al posto di una struttura lineare di blocco/rilascio. L'idea e' quella di mandare messaggi alle UC adiacenti come in figura.

TODO (img): Aggiungere Tree Barrier

Un'altra implementazione invece e' quella chiamata *Butterfly Barrier*, che sfrutta gli stessi principi della rete di interconnessione Butterfly discussa in precedenza, ma al livello software. Tramite questo processo e' possibile far conoscere ad ogni UC che una determinata UC e' entrata nella barrier.

TODO (img): Aggiungere Butterfly Barrier

In sostanza il ruolo che era assegnato ad una sola UC di accentrare tutte le sincronizzazioni e' invece distribuito su tutti le UC.

8.2 Computazioni Localmente Sincrone

In questo tipo di computazioni, le singole UC necessitano di sincronizzarsi solamente con un set di processi logicamente "vicini" e non con tutte le UC che sono coinvolte nella computazione. Sostanzialmente lo *"schema di vicinato"* definisce uno *"stencil"*, per cui questo tipo di computazioni e' anche chiamato *stencil computations*.

8.2.1 Simulazione di diffusione del calore

Consideriamo il caso in cui si voglia ottenere una simulazione del calore in due dimensioni. L'idea e' quella di considerare un quadrato di qualsiasi materiale la cui temperatura in ogni suo punto e' nota. Possiamo rappresentare il quadrato di metallo con una matrice `h[n][n]`. L'idea e' essenzialmente quella di ottenere i valori della temperature in ogni punto del quadrato mediante una media dei quattro punti adiacenti

$$\frac{h_{i,j} = h_{i+1,j} + h_{i-1,j} + h_{i,j+1} + h_{i,j-1}}{4}$$

> E' possibile ottenere l'equazione anche discretizzando l'equazione del calore di Laplace.

8.2.1.1 Versione Sequenziale Possiamo quindi utilizzare l'equazione precedente per calcolare la temperatura in ogni punto del quadrato.

```
1 void step(*h) {
2     for(i = 1; i < n; i++)
3         for (j = 1; j < n; j++)
4             h[i][j] = 0.25*(h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j
                  +1]
5 }
```

Per cui la formulazione finale del programma consistera' nell'iterazione della procedura `step(h)` piu' volte, fino alla convergenza di ogni punto.

8.2.1.2 Versione Parallela Nella versione parallela si introduce un secondo array bidimensionale `g[n][n]`. L'idea e' quella di utilizzare `g` per salvare i valori della prossima iterazione, e poi scambiare i puntatori di `g` ed `h` alla fine di ogni iterazione. In questo modo siamo sicuri che ogni UC stia cosiderando effettivamente i valori dell'iterazione precedente, evitando di fatto inconguenze. Possiamo idealmente "mappare" ogni UC ad ogni posizione, e indurre una sincronizzazione locale attraverso le ricezioni bloccanti.

```
1 void step() {
2     g = 0.25 * (w + x + y + z);
```

```
    send(&g, P_{i-1, j})
    send(&g, P_{i+1, j})
    send(&g, P_{i, j-1})
    send(&g, P_{i, j+1})
    recv(&w, P_{i-1, j})
    recv(&w, P_{i+1, j})
    recv(&w, P_{i, j-1})
    recv(&w, P_{i, j+1})
}
```

Ovviamente le send devono essere non bloccanti siccome potrebbero produrre un deadlock. Fino ad ora abbiamo supposto che il numero di UC sia uguale a $n \times n$, ma in generale sappiamo che non e' cosi'. Quando ci troviamo a dover partizionare i dati del problema ci sono essenzialmente due possibilita: La prima e' partizionare per **blocchi**, mentre l'altra e' partizionare per **strisce**. Il secondo metodo e' quello che minimizza il tempo di comunicazione totale, siccome richiede la meta' delle comunicazioni per *strip*.

Un'altro problema e' quello di gestire le *boundary conditions*, cioe' le celle che non hanno 4 vicini (in generale quelle ai bordi). Una soluzione che si impiega spesso per questo tipo di problema e' l'impiego di *ghost cells*. Si trasforma la matrice ad essere $n + 1 \times n + 1$, in modo da "normalizzare" anche le celle che precedentemente erano ai bordi. Le nuove celle o possono contenere dei valori di default, oppure i valori delle celle al lato opposto. La seconda soluzione e' chiamata anche *halo swap*.

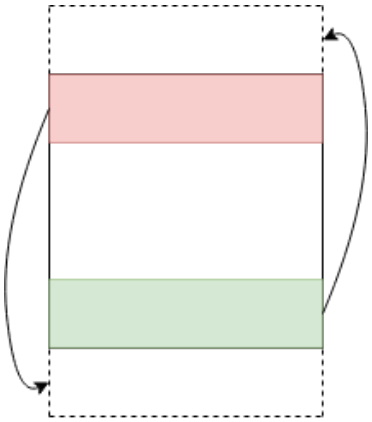

Figura 21: Illustrazione della tecnica di Halo swap sulla dimensione y

9 Bilanciamento del Carico e Terminazione

Il bilanciamento del carico e' utilizzato per distribuire il carico in maniera equa tra tutti i processi in modo da ottenere lo speedup migliore possibile. La terminazione invece e' un problema che concerne la determinazione di quando il programma debba effettivamente terminare (cioe' ha fatto cio' per cui e' stato programmato). Piu' nello specifico, si vuole che il protocollo di terminazione (come ad esempio quello di `MPI`) venga rispettato. Sia il *load balancing* che la *termination detection* sono dei probemi particolarmente difficili quando parliamo di un ambiente distribuito (tipicamente message passing)

Ci sono vari modi per garantire il bilanciamento del carico. Le due famiglie principali sono il *bilanciamento statico* e il *bilanciamento dinamico*. Essenzialmente il bilanciamento statico e' fatto a tempo di compilazione, per cui e' generalmente piu' efficiente della sua controparte, mentre quello dinamico e' calcolato a runtime. Nonostante il bilanciamento dinamico sia meno performante di quello statico, risulta comunque piu' flessibile. Alcune tecniche di load balancing statico sono ad esempio:

- *Round Robin*: assegna i tasks alle UC in maniera sequenziale, passando di nuovo alla prima UC quando tutte sono state assegnate
- *Randomized Algorithms*: assegna i tasks in modo randomico
- *Recursive Bisection*: divide ricorsivamente il problema in sottoproblemi di uguale effort computazionale
- *Simulated Annealing* o *Algoritmi Genetici*: tecniche di ottimizzazione per trovare l'assegnamento sub-ottimo

Ci sono un certo numero di problemi fondamentali che riguardano il load balancing, nonostante esistano delle soluzioni matematiche chiuse.

- E' molto difficile stimare accuratamente il tempo di esecuzione senza effettivamente eseguire le parti di programma
- I delay di comunicazione cambiano profondamente in base alle circostanze operative (es. la topologia di alcune reti e' inerentemente non uniforme, fattori legati al caching in sistemi non distribuiti ecc..)
- Alcuni problemi hanno un numero indeterminato di step per raggiungere la soluzione (es. convergenza)

Il bilanciamento dinamico risolve questi problemi poiche' essenzialmente sono dati dal fatto che il tempo di esecuzione non puo' essere noto, mentre nel bilanciamento dinamico implicitamente lo sono. Tutti i fattori precedenti vengono tenute in considerazione effettivamente dall'algoritmo di load balancing, che comunque introduce un overhead nel programma. Nonostante cio', la risoluzione di questi problemi e' solo in linea di principio e dipende da altre variabili, per cui si potrebbe incappare in un caso in cui si perda piu' di quello che si guadagni.

Possiamo classificare il bilanciamento dinamico in due ulteriori varianti:

- Centralizzata
- Decentralizzata

Quando parliamo di struttura centralizzata stiamo essenzialmente parlando di un paradigma *Master-Worker*, in cui il master e' il punto centralizzato incaricato di schedulare i tasks ai workers. Il problema di questo tipo di approccio e' che il *master* e' il *single point of failure*, per cui se esso per qualche ragione dovesse smettere di funzionare l'intero sistema cadrebbe.

> Per risolvere il problema del *single point of failure*, Kubernetes impiega 3 masters, in cui 2 di essi sono essenzialmente la copia di quello principale. L'idea e' che si massimizzi cosi' la *reliability* ma sacrificando il *thoughtput*.

La terminazione nel caso del load balancing centralizzato e' particolarmente semplice. Essa puo' terminare solo quando le seguenti condizioni sono soddisfatte insieme

- La coda dei task dle master e' vuota
- Ogni worker ha fatto una richiesta senza che nessun'altro task sia stato generato

L'idea del load balancing decentralizzato e' invece quella di distribuire il compito del master tra tutti i workers. In linea di principio ogni worker puo' ricevere dei tasks da altri workers e a sua volta puo' inviare tasks ad altri workers. Un modo per farlo e' organizzare una struttura gerarchica di masters. Ogni master gestira' un sottoinsieme di workers con la propria coda interna, che verra' riempita a sua volta dal master padre al livello successivo. Ovviamente una rete di master puo' avere qualsiasi struttura. Un'altra soluzione e' quella di distribuire totalmente la pool di task su tutti i workers, comunicando tra di loro come se fossero una rete *peer-to-peer*:

- **Receiver-Initiated method**: un worker richiede i tasks dagli altri workers quando non ha piu'/abbastanza tasks da eseguire. E' un metodo che si e' dimostrato funzionare bene con alti carichi e pochi processi. Il problema e' che in generale un worker non ha idea di chi e' messo nelle condizioni di inviarle dei tasks, per cui potrebbe fare molte richieste a processi che sono nelle sue stesse condizioni, per cui all'aumentare al numero di processi il numero di comunicazioni aumenta di conseguenza con magnitudine piu' grande.
- **Sender-Initiated method**: un worker manda i tasks ad altri processi che seleziona. Tipicamente un worker con un alto carico passa i suoi taks ad altri che si sono resi disponibili ad accettarli. Contrariamente al metodo receiver-initiated funziona bene solo per carichi molto bassi.

Questi due metodi vengono chiamati in letteratura come *work-stealing*. In questo caso la terminazione puo' avvenire solo se le seguenti condizioni di terminazione sono soddisfatte:

- Ogni worker non deve avere uno o piu' task non eseguiti completamente. (*condizione locale di terminazione*)

- Non ci devono essere piu' messaggi in transito nel sistema (*condizione globale di terminazione*)

9.1 Terminazione Distribuita

9.1.1 Algoritmi di terminazione ad Albero

E' possibile ricavare anche un algoritmo di terminazione distribuito molto generale. L'idea e' quella di ricavare una struttura a grafo, indipendente dalla struttura di comunicazione dei workers che descrive la relazione di attivazione tra task diversi. Essenzialmente l'algoritmo funziona nel modo seguente:

- Se un worker ha mandato un task ad un altro worker ne diventa il padre
- Quando un worker riceve un task, manda immediatamente indietro un messaggio di *acknowledge*, ma solo se il worker da cui lo riceve sta nella gerarchia piu' in alto di lui (e quindi e' un padre, nonno ecc..)

In questo modo possiamo determinare se un worker qualsiasi puo' terminare verificando che tutte le seguenti condizioni siano rispettate:

- La condizione di terminazione locale del singolo worker e' soddisfatta
- Il worker ha trasmesso tutti i suoi messaggi di acknowledgement per task che ha ricevuto
- Il worker ha ricevuto tutti i suoi messaggi di acknowledgement per i task che ha mandato

In questo modo il grafo di attivazione si espande in dimensioni quando i workers comunicano task tra di loro, mentre si riduce ad un solo punto quando devono terminare. Si noti come una conoscenza distribuita venga in questo modo accentrata su un solo punto (l'ultimo worker che terminera') Ovviamente ci sono versioni differenti di questo algoritmo che inducono delle strutture differenti di quelle a grafo come ad esempio degli anelli

9.1.2 Algoritmi di terminazione a Energia fissa

L'idea e' essenzialmente quella di impostare una quantita' fissa all'interno del sistema detta *energia*. Il sistema inizializza il worker iniziale con tutta l'energia dell'intero sistema. Ogni volta che un worker manda delle richieste di tasks ad altri workers, ne passa una porzione della sua energia. In caso un worker ricevesse delle richieste per dei task (e cosi' anche una porzione di energia), l'energia verra' suddivisa e passata ad altri workers. Un worker in idle, (che non ha task da eseguire) prima di richiedere nuovi tasks, passa la propria energia al worker da cui l'ha ricevuta. In generale, un worker non mandera' indietro la sua energia fin quando tutta l'energia che ha inviato ad altri workers non sara' tornata indietro (non avranno terminato i loro tasks). Quando tutta l'energia ritorna al worker iniziale (*root*) e diventa idele, allora si e' sicuri che tutti i workers sono idle e la computazione puo' terminare.

Il problema principale e’ che la divisione dell’energia dovra’ essere rappresentata su un numero a precisione finita, per cui quando si va a fare la somma delle energie parziali per poi mandarle indietro, la somma potrebbe non essere uguale alla quantita’ originale. Un’altro problema, sempre legato alla rappresentazione dell’energia, e’ dato dal fatto che non si possa dividere l’energia all’infinito, perche’ raggiungerebbe lo zero molto velocemente.

10 Modello di programmazione Shared Memory

Programmare sistemi a memoria condivisa e' molto piu' semplice di programmare sistemi message passing. Questo perche' il programmatore ha una visione globale della memoria in questo tipo di sistemi, mentre nel message passing no. Si pensi ad esempio il caso in cui si voglia sapere il valore di una determinata variabile. Nei sistemi shared memory sarebbe immediato dal momento che la variabile risiederebbe sicuramente in memoria, mentre nei sistemi message passing tale variabile deve essere ricevuta da qualche altro processo ignoto. Anche se piu' conveniente dal punto di vista di programmazione, tale paradigma necessita di controllare gli accessi alla memoria in modo esplicito dal programmatore tramite opportune sincronizzazioni.

> Il problema di questo tipo di sistemi e' che non hanno un'alta scalabilita' per delle questioni che verranno trattate successivamente. In generale, se si vuole vere un'alta scalabilita' e' preferibile il paradigma *message passing*, mentre se si si vuole avere un'applicativo con numero di UC basso si preferisce il paradigma *shared memory*. Questo perche' per un numero basso di UC il paradigma message passing e' molto piu' efficiente perche' riduce significativamente l'overhead della comunicazione.

Partiamo ora dalla definizione di un sistema multiprocessore a memoria condivisa. In generale un sistema a memoria condivisa e' qualsiasi sistema in cui ogni locazione di memoria possa essere acceduta da qualsiasi processore. Ogni locazione di memoria ha inoltre un indirizzo univoco all'interno del range possibile di indirizzi. In altri termini, hanno un *singolo spazio di indirizzamento*. Essenzialmente possiamo descrivere un sistema shared memory come un sistema di processori connessi tra di loro e a loro volta connessi alla memoria attraverso un sistema di interconnessione. Come gia' visto la rete di interconnessione puo' essere di varie tipologie. Tipicamente, in qeusto tipo si sistema si impiega una crossbar switch. I processori hanno a loro volta delle memorie di piccole dimensioni molto performanti chiamate memorie *cache*.

TODO: Inserire immagine Shared Memory

Ci sono diversi metodi per programmare un sistema a memoria condivisa multiprocessore, tra cui alcuni molti diversi tra loro:

- Utilizzando processi (*heavyweight*)
- Utilizzando threads (*lightweight*)
- Utilizzando un linguaggio completamente progettato per la programmazione parallela (es. Ada)
- Utilizzando routines di liberire di un linguaggio di programmazione sequenziale
- Modificando la sintassi di un linguaggio sequenziale creando di fatto un linguaggio parallelo
- Utilizzando un linguaggio sequenziale e "decorarlo" con delle direttive di compilazione (es. OpenMP)

Il primo approccio non e' molto utilizzato in parallel computing per il troppo overhead causato dallo scheduling dei processi. I processi hanno un programma completamente separato con le proprie variabili, il proprio stack e il proprio heap, mentre i threads necessitano solamente di uno stack e un istruction pointer. `pthreads` e' una libreria POSIX che fornisce dei costrutti di basso livello (livello del sistema operativo) per operare con i threads. Per eseguire un thread si utilizza la seguente chiamata, dove `&thread1` e' l'handle e `proc1` e' una funzione da far eseguire dal thread.

```
1 pthread_create(&thread1, NULL, proc1, &arg);
```

con la chiamata `pthread_join(&thread1, *status)` si puo' aspettare invece il completamento del thread. I threads che non sono joined vengono chiamati *detached*. Quando un thread termina, le sue risorse vengono di conseguenza rilasciate. Quando compiliamo un programma multitreaded, il compilatore potrebbe applicare delle ottimizzazioni a livello di istruzioni, riodinandone l'ordine di esecuzione. Ad esempio, lo statement

```
1 a = b + 5
2 x = y + 4
```

potrebbe essere compilato per essere eseguito nell'ordine inverso

```
1 x = y + 4
2 a = b + 5
```

ed essere comunque logicamente corretto. Questo tipo di ottimizzazioni viene fatto da quasi tutti i compilatori moderni.

> Un'operazione e' detta *thread safe* se puo' essere chiamata da piu' threads simultaneamente e produrre sempre risultati corretti. L'I/O standard e' thread safe, poiche' se `println` viene chiamata da piu' threads simultaneamente, i caratteri non sono interfogliati.

10.1 Accesso ai dati condivisi

Come detto in precedenza, nei sistemi a memoria condivisa bisogna prestare particolarmente attenzione agli accessi in memoria per poter evitare eventuali *data races*. Consideriamo due processi che vogliano aggiungere un'unita' alla stessa variabile contenuta in memoria. Ogni processo dovra' quindi preliminarmente leggere il contenuto della variabile, calcolarne il risultato e poi scriverlo all'interno della variabile. In base all'ordine in cui vengono eseguite le istruzioni di `read` si otterranno diversi risultati. Per poter risolvere questo problema si utilizza un meccanismo molto comune chiamato *sezione critica*

> La sezione critica e' un meccanismo che assicura che un solo processo (o thread) possa accedere ad una particolare risorsa alla volta

Il meccanismo di base per implementare una sezione critica e' attraverso dei *lock* (o piu' generalmente dei *semafori*). Un lock e' una variabile binaria che indica se un arbitrario processo/thread e' dentro la sezione critica o meno. Funziona concettualmente come una serratura di una porta che puo' essere chiusa/aperta. Per implementare la lock a sua volta e' necessario che sia implementata un'istruzione a livello hardware di lettura e scrittura chiamata `CAS` (*Compare And Swap*). Tale istruzione essenzialmente viene impiegata per scrivere e leggere nello stesso momento una locazione in memoria, cioe' in modo atomico.

> Se si volesse implementare un lock, un processo utilizzerebbe `CAS` andando a leggere e scrivere successivamente il valore 1 nella variabile lock, ignorando preliminarmente cio' che c'era scritto precedentemente. Una volta scritto, viene poi visto se la variabile era effettivamente a 0 o a 1. In caso fosse a 1 ricicla con la stessa operazione.

In pthread i locks vengono chiusi/aperti con le primitive `pthread_mutex_lock(&mutex1)` e `pthread_mutex_unlock(&mutex1)`. Se un thread raggiunge un lock e lo trova chiuso, aspettera' fino a quando il lock non sara' stato aperto. Se piu' di un thread aspetta il lock, il sistema scegliera' un thread tra quelli in attesa per poter continuare. Ovviamente solo il thread che chiude il lock puo' rilasciarlo aprendolo. L'implementazione della lock puo' essere fatta mediante attesa *attiva* o *passiva*. L'attesa attiva consiste nel ciclare fin quando il lock non e' aperto, mentre l'attesa passiva consiste nel sospendere il processo/thread e mandarlo in coda di sleep. Il problema di determinare una delle due implementazioni e' difficile perche' varia in base al contesto. In casi in cui i processi occupano i lock per poco tempo, conviene implementarli con attesa attiva, mentre la sospensione quando occupano molto tempo.

> Task fine grained $\rightarrow$ active waiting
>
> Task coarse grained $\rightarrow$ passive waiting

Il deadlock e' una condizione che si verifica quando un thread/processo che ha ottenuto una risorsa richiede una risorsa che e' bloccata da un'altro thread, nello stesso momento in cui tale thread necessita della risorsa bloccata.

Uno dei modi per *rompere* la dipendenza bloccante che puo' causare un deadlock, e' rimuovere l'aspetto bloccante del lock dei mutex. Ad esempio `pthread_mutex_trylock()` e' una primitiva che testa se il lock e' stato preso o meno, senza bloccare il codice. Ovviamente l'utilizzo di tale primitiva e' una soluzione non particolarmente buona che non risolve realmente il problema alla radice.

Le lock come detto derivano essenzialmente da un oggetto piu' generico che e' il semaforo. Un semaforo e' un oggetto che supporta due operazioni principali indivisibili `P` e `V`. Possiamo descrivere il

semaforo come una variabile arbitraria che viene inizializzata ad un valore. La semantica di `P` e' quella di decrementare il valore di uno, mentre quella di `V` e' quella di incrementarlo di uno. La peculiarita' e' che quando un processo/thread cerca di utilizzare `P` quando il valore e' a 0 (e il risultato sarebbe conseguentemente < 0) viene bloccato fino a quando il valore non e' > 0 (cioe' un'altro processo/thread ha incrementato la variabilec con `V`). Possiamo quindi notare come una lock sia semplicemente un semaforo impostato con valore iniziale a 1.

Un *monitor* e' un'altro costrutto di sincronizzazione che permette ai threads di avere mutua esclusione, con la possiblita' di sospendersi all'interno di essa per aspettare una determinata condizione. Un monitor consiste in una **lock** e una **condition variable**. Le condition variables sono essenzialmente dei contenitori di threads che aspettano una determinata condizione. I monitor provvedono dei meccanismi di sincronizzazione che permettono ai thread che hanno acquisito il lock di rilasciarlo temporaneamente in modo che una determinata condizione possa verificarsi. In Pthread questa operazione e' codificata dalla funzione `pthread_cond_wait(condition, lock)`, che mette il thread in stato *wait* fino a quando `condition` non e' vera, rilasciando il `lock` del monitor. Con la funzione `pthread_cond_signal(condition)` si sveglia invece un qualsiasi thread che e' bloccato su quella condizione. Il thread scelto dipende dalle policy del sistema, per cui non possiamo determinarlo a priori. La funzione `pthread_cond_broadcast(condition)` e' invece una variante della signal che permette di svegliare tutti i threads.

> Alternativamente possiamo definire un monitor come una classe, oggetto o modulo thread-safe con un lock (o mutex) che permette l'accesso sicuro ad un metodo o ad una variabile in modo mutualmente esclusivo. Semplicemente permette a dei metodi di avere una lock intrinseca in modo che solo un processo/thread possa eseguire il codice contenuto contemporaneamente. Inoltre, all'interno di ogni metodo e' possibile dare il controllo ad altri threads in modo da aspettare il verificarsi di una determinata condizione (tramite condition variables).

10.2 Analisi delle dipendenze

Il vantaggio della programmazione di sistemi a memoria condivisa e' la relativa portabilita' da codice sequenziale a codice parallelo. Una via alternativa ad un porting totale del codice sequenziale ad un codice parallelo, e' rappresentata dalla parallelizzazione *"automatica"* di alcuni costrutti iterativi, come ad esempio i *loops*. Siccome la parallelizzazione automatica e' eseguita da un compilatore parallelizzante, e' necessario trovare un metodo algoritmico per determinare le porzioni di codice che possano essere eseguite in parallelo. Facciamo un esempio e consideriamo il codice seguente

```
1 for (i = 0; i < 5; i++)
2     a[i] = 0
```

E' possibile notare che le operazioni eseguite nel corpo del loop siano *indipendenti* tra di loro e quindi possono essere eseguite in qualsiasi ordine. In altri termini, *cambiare l'ordine di esecuzione delle istruzioni dentro al corpo del ciclo non cambia la semantica dell'intero programma*. L'*analisi delle dipendenze* non e' nient'altro che la determinazione automatica delle porzioni di codice che hanno questa proprieta'.

Per fare cio', gli algoritmi si basano su dei risultati teorici, primi tra tutti le *Condizioni di Bernstein*.

> **Condizioni di Bernstein**: Insieme di condizioni *sufficienti* a determinare se due statements possono essere eseguiti in qualsiasi ordine. Dati i seguenti insiemi
>
> - I_i che indica l'insieme delle locazioni di memoria *lette* (input) dallo statement S_i
> - O_j che indica l'insieme delle locazioni di memoria *scritte* (output) dallo statement S_j
>
> diciamo che due processi P_1 e P_2 possono essere eseguiti in parallelo se le tre condizioni seguenti sono soddisfatte
>
> 1. $I_1 \cap O_2 = \emptyset$
> 2. $I_2 \cap O_1 = \emptyset$
> 3. $O_1 \cap O_2 = \emptyset$

Facciamo ora alcuni esempi per ogni condizione partendo dalla prima, che modella la *True Data Dependency*. Essenzialmente puo' essere riassunta dal seguente codice:

```
1  A = B + C;
2  D = A - 2;
```

in cui il problema e' la dipendenza della seconda istruzione rispetto alla prima. La seconda modella invece la *Anti-Dependency*, riassunta mediante:

```
1  A = B + C;
2  B = 3;
```

in questo caso non si ha una TDD, ma il problema risiede nel fatto che `B` possa essere scritta prima che la prima riga possa essere eseguita. Infine, la terza modella la *Output Dependency*:

```
1  C=3;
2  C=4;
```

in cui il problema risiede nel fatto che non sia possibile determinare il vero valore di `C`.

10.3 OpenMP

Parlando di compilatori parallelizzanti e' impossibile non parlare anche di **OpenMP**, uno standard che introduce delle direttive di compilazione per la parallelizzazione di statements. L'idea e' quindi utilizzare dei pragma di compilazione, mantenendo sempre un linguaggio sequenziale come base (quali Fortran e C/C++). OpenMP essenzialmente nasce per parallelizzare i loop, anche se le nuove versioni si discostano dall'obiettivo inziale. Tutte le direttive di OpenMP hanno la forma `#pragma omp directive_name`. Ad esempio la direttiva `#pragma omp parallel` indica di parallelizzare lo snippet che segue. Il codice viene parallelizzato poi seguendo un modello *"fork-join"*, ma basato sui threads. Cio' significa che le porzioni sequenziali vengono eseguite da il thread master (mentre gli altri threads non fanno nulla), mentre le porzioni parallele vengono naturalmente eseguite da tutti i threads. Il numero di threads di default e' pari al numero di threads della macchina (in modo da ottenere il throughtput massimo), ma e' possibile specificare un numero differente nelle pragmas.

TODO: Aggiungi immagine fork-join vs OpenMP

Alcune direttive che mette a disposizione OpenMP sono:

1. `Sections`: definisce blocchi che vengono condivisi dai thread
2. `For`: indica che il loop che segue la pragma deve essere eseguito in parallelo
3. `Single`: specifica che una sezione all'interno di una sezione parallela debba essere eseguita da un solo thread
4. `Atomic`/`Critical`: definisce una sezione che viene eseguita da un thread per volta. Nel caso di atomic se la sezione e' composta da un update di una sola variabile
5. `Flush`: rende la memoria consistente con lo stato dei thread fino a quel momento. Il flush viene eseguito automaticamente all'inizio e/o fine di alcune direttive

OpenMP e' particolarmente semplice da utilizzare, ma necessita di diverse accortezze nella programmazione, per cui e' fondamentale saper riconoscere alcuni *code smells*. Ad esempio, il seguente snippet e' considerato code smell

```
1 i=0
2 for(i = 0; i < N; i++) { ... }
```

perche' si sta utilizzando una variabile solamente per fare il conteggio del loop in uno scope globale. Inoltre, il problema e' evidenziato anche dal fatto che se si parallelizzasse il loop

```
1 i=0
2 #pragma OMP parallel
3 for(i = 0; i < N; i++) { ... }
4 println(i)
```

il valore di output di `i` non e' deterministico. E siccome lo scope di tale variabile e' globale, e' possibile di conseguenza utilizzarla anche fuori dal loop, per cui il compilatore OMP darebbe sicuramente errore. La soluzione e' data dal seguente codice

```
#pragma OMP parallel
for(int i = 0; i < N; i++) { ... }
```

Con questa piccola accortezza, otteniamo diversi vantaggi. Il primo e' che il compilatore (anche sequenziale) assegnerebbe la variabile ad essere un registro (e quindi velocizzando di conseguenza le operazioni di count). La seconda e' che la varibile rimane nello scope del loop per cui non e' possibile utilizzarla al di fuori di esso. Infine, la terza (conseguenza della seconda) e' che e' possibile parallelizzarlo automaticamente.

10.4 Modelli di Consistenza di Memoria

Come detto in precedenza, la programmazione di sistemi a memoria condivisa ci permette di parallelizzare in modo relativamente facile un programma sequenziale. In alcuni casi, pero', e' possibile che i programmi parallelizzati non abbiano delle buone prestazioni, proprio per problematiche legate al fatto che la memoria sia condivisa (si pensi ad esempio al caso in cui un grosso numero di threads debba scrivere nella stessa variabile). La prima cosa da capire bene per comprendere al meglio queste problematiche e' la *memoria cache*. Una memoria cache e' una memoria ad alte prestazioni molto vicina (fisicamente) al processore. Le memorie cache si basano su un principio detto *principio di localita'*, che a sua volta si suddivide in:

- **Localita' spaziale**: secondo la quale e' molto probabile che dopo un'istruzione che accede ad una determinata locazione di memoria, la successiva dovra' accedere ad una locazione vicina (nella stesssa *pagina* di memoria). Un esempio e' l'accesso agli array in memoria
- **Localita' temporale**: secondo la quale e' molto probabile che la prossima istruzione da eseguire sia la stessa di quella eseguita. Un esempio sono i loops

La probabilita' per cui una porzione di memoria richiesta stia gia' dentro alla cache e' detto *hit rate*. Se parliamo di tempo di accesso in memoria, possiamo formulare una legge abbastanza generica per il tempo di accesso. Supponiamo che il processore voglia accedere alla memoria e se ne voglia calcolare il tempo di accesso (t_a) in memoria. Se ipotizziamo inoltre che la cache abbia 2 livelli intermedi, allora possiamo dire che

$$t_a = p_{L_1} \cdot t_{L_1} + (1 - p_{L_2}) \cdot t_{L_2}$$

dove p_{L_i} e' la probabilita' di hit del livello i di cache. Se ipotizzassimo che ci sia un'ulteriore livello di cache (L3), otterremmo che

$$t_a = p_{L_1} \cdot t_{L_1} + (1 - p_{L_2}) \cdot p_{L_2} \cdot t_{L_2} + (1 - p_{L_3}) \cdot t_{L_3}$$

L'ottimizzazione di programmi sequenziali e paralleli su sistemi shared memory passa attraverso la minimizzazione di questa formula. In genere e' un problema molto difficile perche' le p_i dipendono da molti fattori. In un processore multicore e' ancora piu' difficile siccome ogni core ha la propria memoria cache. Questo fatto introduce inoltre una problematica legata alla *coerenza* dei dati nelle cache di ogni processore, il cui compito di garantirla e' data da un *protocollo di cache coherence*. Essenzialmente un protocollo del genere si basa su due strategie:

- *Update Policy* - strategia secondo la quale i dati nella cache vengono aggiornati rispetto al dato in memoria solo quando esso viene acceduto da un processore
- *Invalidate Policy* - strategia secondo la quale quando un dato viene alterato in memoria, lo stesso dato viene invalidato nelle cache

Nei sistemi moderni queste policy vengono implementate utilizzando un bus apposito detto di *snooping* (o *sniffing*), su cui tutti i processori ascoltano i cambiamenti dei dati. L'idea e' essenzialmente che qual'ora che un processore abbia un determinato dato nella cache che e' stato modificato da un altro processore (e tale evento e' stato notificato attraverso il bus), allora il processore applichera' l'invalidate policy. Ovviamente anche l'update policy e' implementata nello stesso modo.

> La cache coherence e' uno dei motivi per cui non esistono sistemi multicore a memoria condivisa con un numero molto alto di processori. In tali sistemi, infatti, l'aumento del numero di processori causerebbe di conseguenza un alto traffico sul bus di snooping, con conseguente tempo di servizio piu' lungo

Un effetto importante della coerenza e' il cosiddetto *false sharing*. Il false sharing e' un evento che accade quando diversi processori accedono diverse parti di una pagina, ma la stessa pagina e' comunque condivisa. Se un processore fa un update della propria parte (che non e' condivisa con gli altri processori), deve comunque essere fatto l'update dell'intera pagina dagli altri processori, nonostante la loro parte sia rimasta invariata.
Il problema viene risolto a tempo di compilazione, andando ad alterare il layour dei dati salvati nella memoria principale, raggruppando tutte le locazioni accedute da un processore in una pagina unica.

Oltre al problema della coerenza, nelle memorie cache abbiamo anche il problema della *consistenza* (*sequential consistency*).

> Un processore e' detto sequentially consistent se il risultato di ogni istruzione e' lo stesso di quello che si avrebbe se le operazioni di tutti i processori fossero eseguite in un qualche ordine sequenziale, e le operazioni di ogni processore individuale occorrono in questa sequenza nell'ordine specificato dal programma

Per spiegarlo in altri termini, supponiamo di avere due threads definiti da due porzioni differenti di codice. Allora questi due threads saranno *sequentially consistent* se ogni interfogliamento delle

istruzioni dei due threads non cambia la semantica del programma intero. Posto che le istruzioni di ogni thread ritengano lo stesso ordine interno.
L'effetto della consistenza sequenziale essenzialmente garantisce che tutte le operazioni che vengono fatte sulla memoria siano a loro volta sequenziali. Non possono esistere quindi due operazioni che vengono eseguite contemporaneamente. Si noti come la consistenza sequenziale non imponga un ordine specifico, ma solo che **esista** un ordine qualsiasi.

La consistenza sequenziale ci permette di ragionare piu' facilmente sui programmi paralleli. Per spiegarlo meglio, consideriamo un esempio di due threads:

T_1:

```
1 /* ... */
2 data = new;
3 flag = true;
4 /* ... */
```

T_2:

```
1 /* ... */
2 while(flag != true);
3 data_copy = data;
4 /* ... */
```

Grazie alla consistenza sequenziale possiamo essere certi che quando `flag=true`, sia stato sicuramente scritto `data = new`. Nonostante questa consistenza sia garantita dall'esecuzione del programma (imposta dal processore) nella memoria e' difficile ottenerla, sopratutto in un sistema a multiprocessore. Questo perche' se ipotizziamo che sia T_1 che T_2 vengono eseguiti in parallelo da due processori differenti, allora T_1 potrebbe scrivere `data=new` ma solo dentro la propria cache, senza che effettivamente la modifica abbia effetto sull'intera memoria, per cui `data_copy` non sara' uguale a `new` previsto.

Ad oggi, i multiprocessori **non** sono sequenzialmente consistenti. Anche se esiste l'opzione di renderli tali, le performance degradano troppo, per cui non ne vale la pena.

In generale, siccome la consistenza sequenziale e' molto difficile da raggiungere, si preferisce una sua versione piu' rilassata, che impone la consistenza sequenziale solo per le operazioni di read e write. Per garantire questa consistenza piu' rilassata, il processsore utilizza delle apposite istruzioni:

- *Memory Barrier* (`MB`) - aspetta che tutte le operazioni di memoria precedenti siano completate
- *Write Memory Barrier* (`WMB`) - come la prima ma solo sulle operazioni di memoria in modalita' *write*
- *Read Memory Barrier* (`RMB`) - come la prima ma solo sulle operazioni di memoria in modalita' *read*

A volte alcuni processori impongono questi vincoli solo per dei sottoinsiemi di operazioni, ad esempio i processori multicore Intel garantiscono la consistenza solo sulle operazioni di write (*Total Store Order*). I processori ARM utilizzano una versione ancora piu' rilassata chiamata *Weak Order*, in cui non ci sono sincronizzazioni su nessuna operazione in memoria, ma solo in dei punti prestabiliti del programma chiamate *fence*.

11 Calcolo parallelo su GPU

Fino ad ora abbiamo visto modelli che si basano sulle CPU. In generale, l'architettura di una CPU moderna a pipeline e' costituita da diversi componenti pricipali quali:

- Registri
- ALU
- Cache
- Componenti di OOO, Branch Prediction, Memory

Siccome i registri e l'ALU formano essenzialmente un core minimale per l'esecuzione effettiva delle istruzioni, una direzione possibile per aumentare la densita' di calcolo potrebbe essere quella di aumentare effettivamente il numero di questi core sullo stesso die di silicio. Possiamo inoltre notare come tutti questi componenti (a parte i registri e l'ALU), siano stati progettati appositamente per fare pipelining in questa tipologia di processori. Inoltre, di tutti questi componenti, l'unita' di calcolo effettiva richiede veramente poco spazio rispetto ad altri componenti come la cache. Questo perche' la cache si e' essenzialmente sviluppata per avere uno storage sempre piu' grande in modo da ridurre al minimo i cache misses.

Il pipelining e' una tecnica molto comoda e nascosta al programmatore, che ci permette di eseguire codice sequenziale il parallelo, ma per questa ragione ci impone anche dei vincoli che possono essere sorpassati solo cambiando modello di esecuzione in primo luogo.

Un'idea possibile, quindi, potrebbe essere quella di rimuovere completamente tutti i componenti adibiti al pipelining (compresa la cache) in favore di un maggiore numero di unita' di calcolo. Il problema, pero', e' che se metto un numero molto alto di queste unita' di calcolo, ogni core dovra' andare a leggere il proprio flusso di istruzion indipendente dagli altri. Poter usare tutti questi core contemporaneamente sarebbe impossibile, perche' il collo di bottiglia dato dal traffico in memoria sarebbe enorme (immagina ogni processore indipendente che va a leggere dalla memoria ogni volta dati indipendenti).

Per risolvere il problema, quindi, si puo' pensare di condividere con tutti i cores un solo program counter (*Instuction Stream Sharing*). In questo modo, tutti i threads eseguono *la stessa istruzione* contemporaneamente, ma su dati diversi. La conseguenza di questo punto di vista e' che si vanno a ridurre tutti i trasferimenti in memoria, andando a raggrupparli in "*blocchi*". Se prima ogni threads accedeva potenzialmente ad elementi di array differenti, con questo approccio ogni thread opera su una particolare cella dello stesso array, riducendo di conseguenza il numero di accessi totali. Per questa ragione, il modello di programmazione di questi sistemi e' detto **SIMD** (*Single Instruction Multiple Data*).

Questo modo di vedere le cose ha delle implicazioni profonde sulla programmazione di sistemi SIMD. Una di queste e' che il codice sequenziale non puo' essere utilizzato in nessun modo senza essere

ristrutturato per supportare questa tipologia di architettura. Un'altra problematica e' quella della *thread divergence*, che si verifica quando ogni unita' di calcolo deve fare un branching. Si consideri, ad esempio, il seguente codice:

```
1 for(i=0; i < N; i++)
2     if(A[i] < 0) A[i] = -A[i];
3     else A[i] = A[i] + B[i];
```

Quale delle due istruzioni dovranno eseguire tutti i threads? Ovviamente, non e' possibile farlo, per cui l'unico modo e' far eseguire il corpo dell'`if` da tutti i threads per cui e' vera mantenendo quelli per cui e' falsa in *idle*, e poi fare lo stesso per il branch `else`.

11.1 Da SMD a SMT

Per mitigare queste problematiche, le GPU utilizzano un modello **SIMT** (*Single Instruction Multiple Threads*). Il focus essenzialmente non e' quello di eseguire la stessa istruzione per tutti i cores, ma invece per tutti i threads. In questo modo e' possibile eseguire altri stream di istruzioni mentre si stanno aspettando i dati dalla memoria, una tecnica chiamata *latency hiding*. Per eseguire in modo efficiente centinaia di thread leggeri e concorrenti, il multiprocessore della GPU implementa il multithreading in hardware, ossia gestisce ed esegue centinaia di thread concorrenti in hardware, senza overheads dovuti pianificazione dell'esecuzione.

Una GPU e' essenzialmente un dispositivo progettato per questo tipo di programmazione. E' un dispositivo esterno (*periferica*), dotato di memoria propria, e connesso alla CPU e alla memoria del calcolatore attraverso il North Bridge.

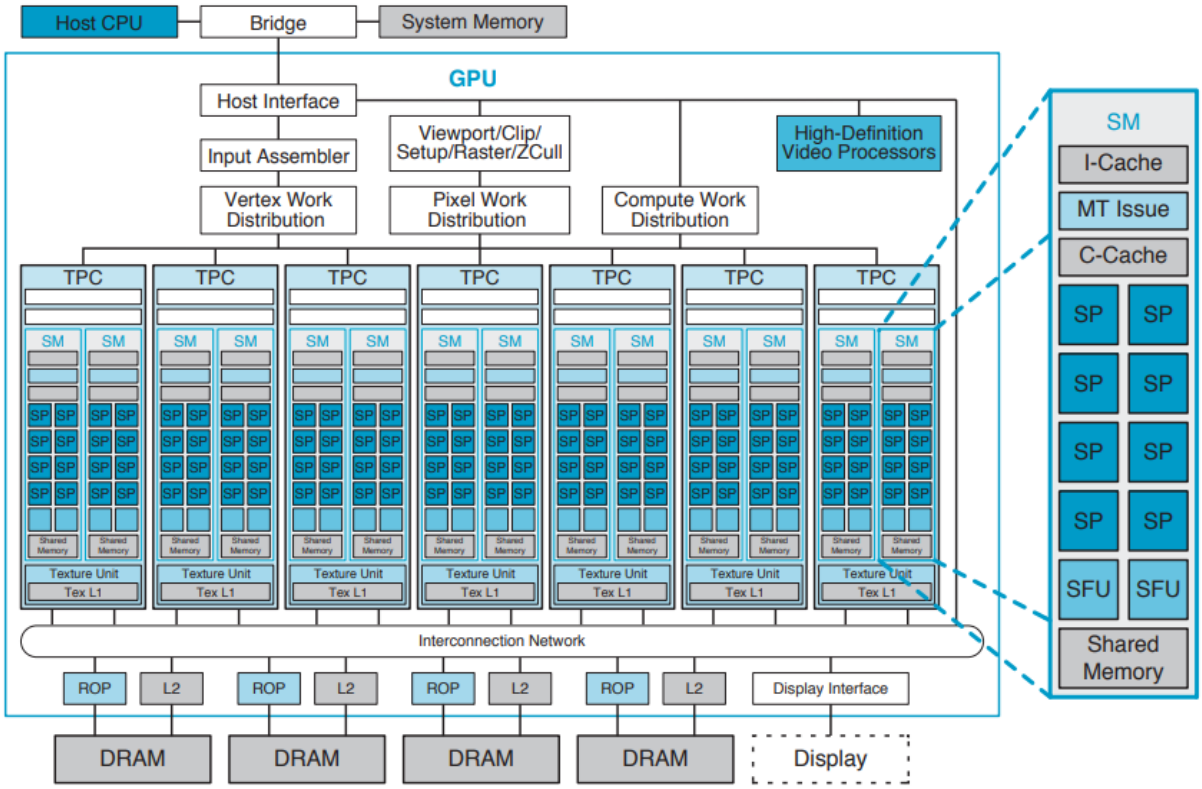

Figura 22: Architettura Tesla di base di una GeForce 8800 NVIDIA

I componenti essenziali di una GPU sono i cosiddetti *Streaming Multiprocessors* (SM). Ognuno di essi e' composto da:

- Diversi *Streaming Processors* (SP) - anche detti *CUDA cores* nella terminologia NVIDIA
- Un grosso insieme di registri multithread (*Register File*)
- Una memoria cache condivisa
- Un'unita' di cache per le costanti
- Un'unita' di lancio delle istruzioni multithread
- Diverse unita' adibite ad operatizoni matematiche speciali (SFU)

Per implementare il modello SIMT, lo SM esegue concorrentemente gruppi di threads paralleli, chiamati *warps*. I threads che compongono un warp partono tutti dalla stessa istruzione, ma poi sono liberi di divergere (*thread divergence*). Ovviamente anche in questo caso si ha una problematica come quella del modello SIMD, per cui se i threads di un warp divergo, alcuni threads potrebbero rimanere inattivi durante l'esecuzione di alcune istruzioni. In generale, si ottiene un livello di efficienza massima quando tutti i threads di un warp seguono lo stesso flusso di istruzioni. Un warp in generale e' composto al massimo di 32 threads, che vengono eseguiti dallo SM in 4 insiemi di 8 threads. Inoltre, ogni warp

condivide la memoria dello SM, per cui e' possibile scrivere programmi che sfruttino questa possiblita' per ridurre il numero di accessi in memoria.

> Ricapitolando: i threads sono divisi in gruppi di 32 chiamati *warps*. Lo streaming multiprocessor schedula i warps e li esegue, facendo eseguire la stessa istruzione di tutti i threads ai singoli streaming processors.

11.2 Modello di Programmazione CUDA

Avendo illustrato l'architettura generale, possiamo ora parlare del modello di programmazione effettivo di questa tipologia di dispositivi. Siccome diversi costruttori hanno impiegato il proprio modello di programmazione (nonostante si basino sulla stessa architettura, hanno delle dissimilarita'), ci limiteremo a considerare la programmazione di GPU CUDA.

La programmazione di acceleratori in CUDA ruota intorno al concetto di kernel: una funzione che indica cosa deve fare il thread generico in una batteria di threads. Tale batteria e' piu' propriamente chiamata *blocco* di threads. Ogni threads all'interno di un blocco condivide la memoria con gli altri. A loro volta, i blocchi sono poi organizzati in una *griglia*. Tra di loro, i blocchi condividono la memoria globale, che ha pero' un tempo di accesso molto piu' alto rispetto a quella condivisa tra blocchi.

I blocchi vengono implementati come (*Cooperative Thread Arrays*), cioe' gruppi di threads che eseguono lo stesso flusso di istruzioni e possono cooperare per arrivare al risultato. Tali CTA sono essenzialmente tradotti in dei warps a livello architetturale. Possiamo quindi dire che un blocco di threads rappresenta un insieme di threads concorrenti (siccome sono composti da warps) che possono sincronizzarsi localmente tra di loro per raggiungere un risultato. Tali blocchi sono tra di loro indipendenti e possono essere eseguiti in qualsiasi ordine.

Tecnicamente parlando, CUDA e' un'estensione del linguaggio C/C++, introdotta dal compilatore di NVIDIA `nvcc`. CUDA essenzialmente distingue due categorie di codice eseguibile differenti: quello eseguito dall'*host* (CPU) e quello eseguito dal *device* (GPU). Per indicare che il codice deve essere eseguito a livello di CPU o GPU, lo si annota con delle macro `__host__` e `__device__`. La direttiva `__global__`, invece, serve a indicare che una determinata funzione e' un *kernel*. Ad ogni chiamata di kernel da parte dell'host, si devono specificare le dimensioni della griglia e dei blocchi di threads su cui si vuole lanciare il kernel.

Bibliografia

[1] A. G. David Culler Jaswinder Pal Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1998.

[2] B. Allen C. Michael; Wilkinson, *Parallel programming: techniques and applications using networked workstations and parallel computers*, 2nd ed. Pearson/Prentice Hall, 2004; 2005.

[3] S. Weiss, *Lecture Notes on Parallel Computing*. Department of Computer Science, Hunter College.

[4] «Crossbar Switch». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Crossbar_switch.

[5] A. D. Robison, «Intel® Threading Building Blocks (TBB)», in *Encyclopedia of Parallel Computing*, D. Padua, A c. di Boston, MA: Springer US, 2011, pagg. 955–964.