Riassunto di High performance computing

# Perché sono necessarie le alte prestazioni?

La necessità di prestazioni sempre maggiori è dovuta al fatto che in molti domini applicativi tutto questo è necessario per ottenere risultati in tempi brevi.

Alcuni di questi domini applicativi sono:

* la simulazione numeri per prevedere il comportamento dei sistemi fisici, come ad esempio la previsione climatica dei prossimi cinquant’anni, qui si utilizzano modelli che hanno bisogna di tanta elaborazione;
* la visualizzazione e animazione grafica di scene complesse, indipendentemente dal fatto che sia in tempo reale o no;
* Big data analytics;
* la sintesi di molecole per la progettazione di farmaci e altre sostanze.

## Simulazione numerica

La simulazione numerica permette di simulare un sistema fisico attraverso modelli matematici, precisamente si valuta un modello per un dato insieme di input e da ciò risolvere le equazioni e calcolare gli output, mettendoli in questo modo in relazione.

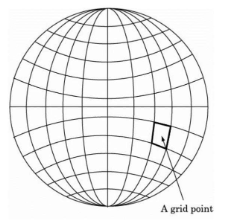
Il modello produce osservazioni che valgono però per un solo punto nello spazio e, dal momento che questi punti sono infiniti, servirebbe un tempo infinito.

A fronte di ciò, dal momento che il numero di punti è continuo nei sistemi fisici, si discretizza lo spazio suddividendolo in un insieme di regioni contigue, da ognuna si sceglie un punto su cui vengono calcolate le soluzioni del modello.

La precisione dei risultati dipende da quanto volte è stato suddiviso lo spazio, ciò indica che un numero maggiore di regioni porta a una maggior precisione.

### Esempio

Per fare un esempio, l’atmosfera terrestre si può descrivere con differenziali parziali, ognuno di questi ha come variabili la velocità del vento, la temperatura dell’aria, eccetera.

Come si discretizza?

Ebbene, si divide l’atmosfera in X meridiani, Y paralleli e Z livelli, si ha quindi un numeri G di celle pari a:

Supponendo di effettuare T iterazioni, bisogna calcolare tutti i valori di tutte le V variabili, ognuna avente A operazioni in virgola mobile, quindi il numero totale di operazioni è il seguente:

Supponendo che si hanno 180 meridiani, 360 paralleli, 12 livelli di atmosfera, 500 iterazioni e 400 operazioni in virgola mobile, si ha che:

E’ tanto o poco? essenzialmente di cil dipende dalla velocità del processore, supponendo che ogni operazione richieda 100 ns, in totale ci vogliono secondi, ovvero circa 21.7 ore.

E’ tanto o poco? DIpende da quando servono i risultati, infatti se si ha poco tempo a disposizione, bisogna cercare una macchina più veloce.

Ciò indica che non è possibile predire il tempo a intervallo inferiori di 21.7 ore. Inoltre bisogna tenere conto che queste previsioni valgono per aree geografiche piuttosto ampie, utilizzare aree più piccole implica un maggior numero di variabili e livelli, facendo salire la complessità enormemente.

## Visualizzazione e animazione grafica

Anche nel rendering è possibile effettuare un ragionamento simile a quello precedente: supponendo che una scena è divisa in G aree, ognuna composta da R pixel e che per ottenere un’immagine fluida si ha bisogno di 60 iterazioni al secondo, si ha che:

Dal momento che G e R hanno sono tipicamente tendenti a e rispettivamente, il tempo totale è di secondi, inoltre se ogni pixel richiede N operazioni, questo valore viene moltiplicato per un fattore N.

# Come si può incrementare la velocità dei computer?

A questa domanda esistono due risposte:

* Velocizzare il processore aumentando la frequenza di clock, eseguendo le istruzioni più in fretta;
* Dotare il processore di più unità funzionali, parallelizzando le istruzioni.

Come si può vedere nel grafico a sinistra,

Il numero di transistor nei processori è aumentato e aumenterà esponenzialmente, questo trend ce l’aveva anche la frequenza fino al 2005.

A cosa è dovuto questo appiattimento?

Essenzialmente è dovuto a limiti fisici: la propagazione di un segnale nel vuoto è di 30 cm/ns, questo numero diminuisce a 20 cm/ns quando si passa dal vuoto ai mezzi fisici.

Considerando un processo con 10GHz di frequenza, il segnale può al massimo viaggiare per 2 cm, con 100GHz a 2 mm e così via.

Anche se fosse possibile realizzare transistor così piccoli, ci sarebbero da considerare i problemi riguardanti la dissipazione del calore: il calore prodotto da un transistor è direttamente proporzionale alla frequenza mentre la capacità di dissipazione è inversamente proporzionale alle sue dimensioni.

Ciò è un problema dato che, per creare un processore dieci volte più veloce, bisognerebbe dissipare il calore di un fattore cento.

Proprio per questi motivi la frequenza dei processori si è assestata a 5 GHz, questo valore aumenta con l’overclocking a 9 GHz ma ciò stressa enormemente il processore, facendolo durare meno.

### Seconda ipotesi: più core in un processore

La seconda ipotesi è quindi inserire nel processore più unità funzionali, creando i processori many-core. La densità dei circuiti integrati aumenta di continuo, tuttavia con scale molto piccolo non vale più l’effetto Dennand.

L’effetto Dennand è un fattore che lega la potenza necessaria all’unità di volume, facendo crescere la prima in modo costante.

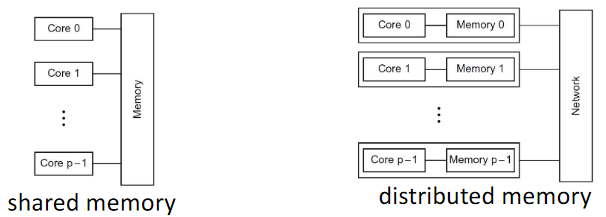
Senza questo effetto, la potenza necessaria per l’unità di volume cresce all’aumentare del numero di transistor e ciò comporta un consumo energetico eccessivo.

Nel grafico sopra si può vedere come la potenza necessaria cresce all’aumentare della densità, il leakage power infatti indica tutta la corrente che viene dispersa per alimentare il processore.

A questo punto l’unica soluzione è quella di inserire più processori in un calcolatore e farli lavorare assieme.

### Paradigmi di High performance

In cinquant’anni di computing ad alte performance, sono stati inventati differenti paradigmi:

* memoria condivisa: Tanti core indipendenti sono connessi alla stessa memoria, ciò permette la collaborazione scrivendo e leggendo in determinate locazioni;
* memoria distribuita: ogni core ha la sua memoria privata e, qualora servisse un valore agli altri, glielo invia attraverso una rete di intercomunicazione.

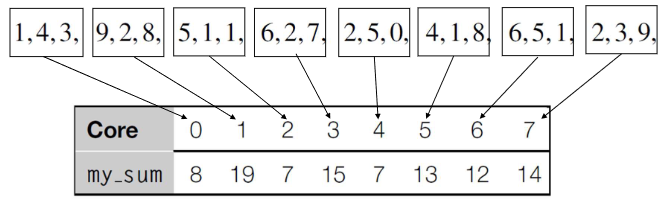
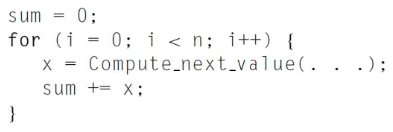
Nonostante i primi siano più semplici da programmare, questi non sono scalabili, cosa che invece permette la seconda famiglia di modelli. A oggi i sistemi più recenti utilizzano approcci ibridi.

# Come si programma un sistema parallelo?

Al momento non è stato ancora trovato un modo che permetta la conversione automatica da una programma sequenziale alla sua controparte parallela, o meglio non è ancora stato trovato un modo che sfrutti adeguatamente un calcolatore parallelo.

Infatti in un programma sequenziale è possibile riconoscere i costrutti e di conseguenza tradurli come si deve in modo efficiente, tuttavia la combinazione di questi potrebbe essere inefficiente.

### Esempio

Per fare un esempio, dato un numero p di core, si suppone di dover sommare n valori utilizzando un ciclo. Supponendo che p sia molto minore di n, quel che si può fare è dare ogni core n/p valori, quest’ultimo calcola il risultato e invia tutto al master, il quale fa la somma globale.

Il programma termina quando il core master termina dato che, oltre a sommare la sua porzione, deve anche fare la somma globale.

Quindi, supponendo che ogni somma ha costo 1, si hanno 8 core e 24 valori, in totale il master deve eseguire 7 addizioni mentre gli altri solamente una.

Si può fare di meglio?

## Riduzione ad albero

Per far fronte a questo problema si utilizza la riduzione ad albero, una tecnica in cui ogni core calcola la propria somma e divide il lavoro dei master tra i core, formando un albero.

Utilizzando gli stessi valori di prima, questa tecnica permette di calcolare il risultato utilizzando 4 operazioni anziché 7.

Cosa si può concludere da tutto questo?

Il primo metodo è O(p) mentre il secondo è , quindi con p molto grandi la differenza è notevole.

Infatti con p=1024, il primo metodo effettua 1024 operazione mentre il secondo solamente 10.

# Cos’è un calcolatore parallelo?

Un calcolatore parallelo è un’insieme di unità di processing (PE) che cooperano tra loro per risolvere un problema di computazione molto rapidamente.

Esistono diversi tipi di calcolatori paralleli, ognuno con le proprie caratteristiche, ciò viene riassunto nella foto a destra.

Nella foto a destra è presente un insieme di PE, ognuno con la sua memoria (M) e una propria interfaccia di comunicazione (CI), connessa a sua volta a una rete di comunicazione.

Queste ultime unità consentono la comunicazione dei PE attraverso dei messaggi, ciò permette loro di cooperare per la risoluzione del problema.

Ultimo ma non meno importante è il sistema di I/O, necessario quando sono richiesti input oppure bisogna inviare degli output.

Come già detto prima, questo schema è generale, questi calcolatori infatti differiscono per:

* tipologia di PE, può essere un ALU, un microprocessore completo di cache e memoria privata oppure un calcolatore completo;
* tipologia di memoria disponibile ai PE, qui ci sono tre sottogruppi:
  + cache e RAM private e accessibili a uno e un solo PE;
  + cache e RAM private, quest’ultima però è accessibile anche dagli altri PE;
  + cache privata e RAM comune a tutti i PE.
* Tipologia di rete di comunicazione tra i PE e/o la memoria, in mezzo infatti può esserci un bus oppure una rete di interconnessione fissa/programmabile.

## Come si classificano i calcolatori paralleli?

La classificazione dei calcolatori paralleli avviene in diversi criteri:

* La tassonomia di Flynn, cioè la classificazione in base al flusso di dati e istruzioni nel sistema;
* Il grado di interazione tra i vari PE;
* I meccanismi di accesso alla memoria.

## Tassonomia di Flynn

La tassonomia di Flynn vede il calcolatore come un sistema che esegue flussi di istruzioni (Instruction Stream - IS) su flussi di dati (Data Stream - DS), la classificazione avviene in base a quanti di questi vengono elaborati in simultanea.

Questi sistemi possono essere:

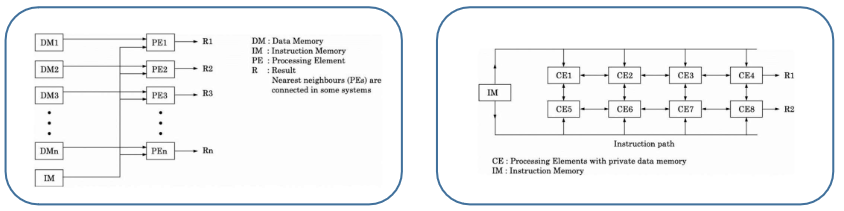
* Single oppure Multiple IS (SI o MI): eseguono uno o più flussi di istruzioni per volta;
* Single oppure Multiple DS (SD o MD): eseguono uno o più flussi di dati alla volta;

Combinando tutte le possibilità, escono fuori 4 possibili categorie:

* Per i calcolatori sequenziali ci sono i SISD;
* Per quelli paralleli esistono i SIMD, i MISD e i MIMD.

## SIMD

Nel caso SIMD, ogni PE esegue lo stesso programma ma riceve una parte del flusso dati in input, l’esecuzione di questi è sincrona.

Nel riquadro a sinistra, ogni PE ha la sua memoria (DM) con i propri dati all’interno e una instruction memory (IM) comune, a ogni ciclo di clock l’IM prende l’istruzione e la fa eseguire.

I risultati devono essere poi copiati nelle DM giuste.

Il riquadro a destra invece è un’evoluzione del precedente: qui la memoria è privata a ogni PE e quindi bisogna effettuare comunicazioni per inviare i risultati.

La differenza principale tra questi due schemi riguarda appunto l’interconnessione delle CE, infatti il secondo schema permette la comunicazione dei risultati intermedi.

La comunicazione nel primo schema avviene scrivendo in una delle memorie, l’estrazione dei dati dovrà quindi essere esplicita.

## MISD

I calcolatori MISD riguardano essenzialmente il pipelining, qui ogni PE ha la sua IM e ha una DM comune. Un caso interessante è l’elaborazione sistolica, detta anche pipelining bidimensionale.

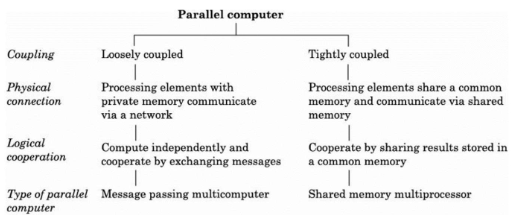
In questo tipo di pipelining, ogni PE propaga il suo risultato a quelli vicini secondo uno schema di interconnessione, precisamente ci sono delle fasi di elaborazione (contrazioni delle sistoli) e fasi di comunicazione (dilatazione).

## MIMD

Il calcolatore MIMD è il modello più generico della tassonomia di Flynn, ogni PE ha un proprio flusso di istruzioni e un proprio flusso dati, la comunicazione avviene tramite una rete di comunicazione.

## Classificazione in base al grado di accoppiamento

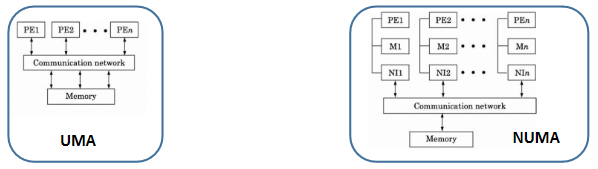
In questo contesto la classificazione si basa sulle velocità di comunicazione tra i PE, da ciò si possono distinguere:

* i sistemi lascamente accoppiati (Loosely coupled), aventi memoria privata e una rete per la comunicazione. La cooperazione avviene tramite lo scambio di messaggi. Proprio per questi motivi la comunicazione è relativamente lenta e vengono generalmente detti message-passed
* i sistemi strettamente accoppiati (tightly coupled), qui i PE condividono la memoria, utilizzata anche nella comunicazione scrivendo su apposite locazioni. A differenza dei primi, qui la comunicazione è relativamente veloce. Questi sistemi vengono denominati anche shared-memory.

## Classificazione in base ai meccanismi di accesso alla memoria

Le classificazioni di questo tipo sono differenti in base all’architettura, nei sistemi shared memory:

* la memoria può essere fisicamente condivisa, quindi ogni PE accede a ogni locazione allo stesso tempo (Uniform Memory Access - UMA), qui però possono esserci problemi di scalabilità;
* la memoria è privata a ogni PE ma logicamente condivisa a tutti, di conseguenza i tempi di accesso non sono uniformi (Non-Uniform Memory Access - NUMA).



## I supercomputer vettoriali

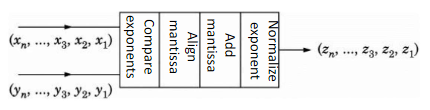
i supercomputer vettoriali sono calcolatori MISD che utilizzano il parallelismo temporale (pipelining) per essere sfruttati al meglio.

Per fare un esempio: dati due valori x e y in virgola mobile (formati da mantissa e esponente), cosa bisogna fare per fare la somma?

Essenzialmente bisogna effettuare 4 step:

* calcolare m come la differenza tra esponenti;
* Si confronta m con 0:
  + Se m>0 si shifta y di m posizioni a destra e si setta l’esponente di z come l’esponente di x;
  + Se m<0 si shifta x di m posizioni a destra e si setta l’esponente di z come l’esponente di y;
  + Se m==0, l’esponente di z è uguale all’esponente di x.
* Si trova la mantissa di z sommando le mantisse di x e y;
* Si fa un altro confronto con la mantissa di z e 1:
  + se mant(z)>1, si shifta di un bit a destra;
  + se uno o più bit significativi per z è uguale a 0, si fanno tanti shift a sinistra fino a quando z non è 0;
  + Ponendo p come numero di shift a sinistra fatti, si sottrae p dall’esponente di z.

## Come si parallelizza tutto ciò?

Tutto questo si estende per effettuare le somme vettoriali in modo efficiente. Con vettori lunghi n:

* a tempo 1, si calcolano gli esponenti della prima coppia;
* a tempo 2 si calcolano gli esponenti della seconda e, allo stesso tempo e si allinea la mantissa nella prima coppia e così via;
* eccetera;

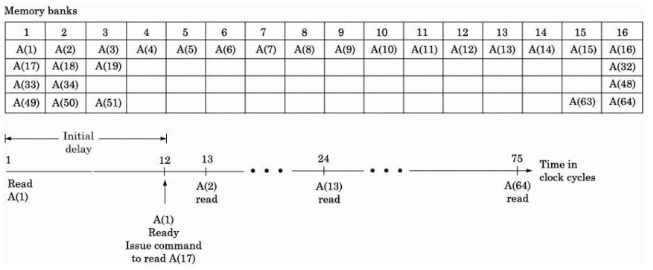
Ciò significa che si può creare un processore vettoriale sfruttando la pipeline per fare le operazioni tra i vettori, in alcuni casi è possibile avere anche più pipeline, a volte connesse tra loro (Vector Chaining).

Nei supercomputer vettoriali completi non è presente una sola pipeline che fa tutto, bensì ce ne sono tante specializzate come ad esempio quelle per la somma e per la moltiplicazione.

Oltre a ciò, ci sono anche dei vector register di dimensione prefissata, essi sono alimentati dalla memoria centrale e permettono di contenere vettori di dati.

## E in situazioni di stallo?

Lo schema a destra è stato popolare in passato, tuttavia ci possono essere stalli nel caso in cui non si riesca a dare continuamente input alla pipeline.

Per risolvere questo problema si utilizzano banchi di memoria indipendenti in cui piazzare elementi distinti dei vettori da elaborare, parallelizzandone il caricamento.

Per fare un esempio: dati 16 banchi di memoria, ognuno con un tempo di accesso di 12 cicli di clock e vettori di 64 elementi, come si possono utilizzare tutti insieme?

Ebbene, si dispongono tutti gli elementi in ogni banco sfruttando l’operazione di modulo, ciò permette quindi di effettuare 16 letture in parallelo una dopo l’altra. Come mostrato nella foto a destra, i primi 12 cicli servono a leggere il primo banco mentre il secondo è già pronto al ciclo 13.

Senza questa tecnica, il numero di cicli sarebbe 12\*64 in totale.

## Efficienza di questi sistemi

Con k stati e un periodo di clock la somma di due vettori lunghi n con un tempo di startup s è:

Ciò è dovuto al fatto che le fasi della pipeline sono in parallelo, infatti il primo risultato si ottiene a tempo mentre a tempo si ottiene tutto il risultato.

Il caso ideale si ottiene quando n tende a infinito, quindi:

Da queste due formule si ricava il rallentamento nel seguente modo:

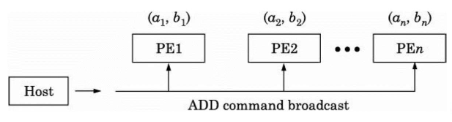
Ponendo s=8 e k=4:

Allo stesso modo, se si vuole ottenere un rallentamento del 10%, bisogna fare in modo che R(n)=1.1 trovando un valore di n adatto (in questo caso n=110).

## Array processor

Gli array processor sfruttano il cosiddetto parallelismo spaziale, cioè che vengono eseguite le stesse istruzioni su dati differenti.

Gli array processor sono calcolatori SIMD per le operazioni vettoriali, precisamente:

* ogni PE è connesso a un host ed esegue operazioni su un solo elemento di un vettore;
* i dati da elaborare sono stati caricati in precedenza nella memoria locale dei PE;
* L’host invia il comando da eseguire, quindi tutto avviene in modo sincrono.

## Multiprocessori

I multiprocessori sono calcolatori MIMD aventi memoria condivisa, ogni PE ha i suoi dati e il suo programma e utilizza la memoria per la sincronizzazione e l’esecuzione.

### Shared bus

Il calcolatore shared bus presenta tutti i processori e i banchi di memoria connessi allo stesso bus, ogni PE ha una cache utile per evitare il collo di bottiglia causato dagli accessi alla memoria.

Sono davvero utili le cache? Sì, infatti permettono di ridurre la probabilità di un ritardo nell’accesso ai dati, quindi che un PE trovi quel blocco di memoria completamente occupato.

Come si può dimostrare ciò?

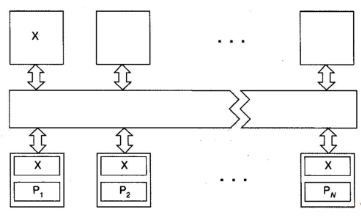
Il numero massimo dei processori supportato da un calcolatore dipende dai parametri strutturali, ad esempio:

| Utilizzando PE a 32 bit con processori a 2GHz che eseguono una istruzione per ciclo (di queste, il 15% sono letture e il 10% scritture), una cache write-through con hit rate pari al 95% e un bus che va a 20 GB/s, si ha che:   * il numero di transazioni di memoria al secondo è: * il numero medio di byte trasferiti da una memoria a un PE è: * il numero medio di processori supportati è: | Considerando gli stessi dati ma senza cache, i risultati sono i seguenti:   * il numero di transazioni di memoria al secondo è: * il numero medio di byte trasferiti da una memoria a un PE è: * il numero medio di processori supportati è: |
| --- | --- |

Da questi risultati si conclude che l’utilizzo delle cache permette l’inserimento di processori in più.

## Cache coherence

Il lato oscuro dell’inserimento della cache nei calcolatori è la coerenza: ogni cache di ogni PE può infatti contenere diversi valori per una locazione di memoria, rendendola in questo modo incoerente.

A fronte di questo problema, l’idea per risolverlo è quella di invalidare le linee di cache quando una data locazione di memoria viene sovrascritta, per fare ciò si utilizzano protocolli appositi.

Per i sistemi multiprocessore shared bus si può sfruttare il fatto che ogni PE osserva gli indirizzi di memoria e le operazioni su di essi nel bus, da questo principio nascono i protocolli di snoopy cache.

## Come bisogna comportarsi per avere coerenza in cache?

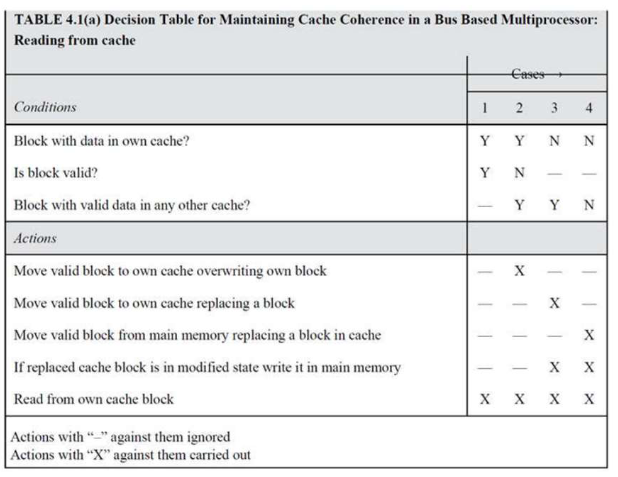
Considerando uno scenario come illustrato a sinistra, essenzialmente si utilizza una policy di cache coherence, implementata attraverso un protocollo, due di queste sono:

* write invalidate: Ogni blocco ha uno stato (valido o invalido). Un blocco valido indica che contiene il dato aggiornato ed è presente anche nelle altre cache, ciò ne permette la lettura. La scrittura di un blocco in cache rende invalide tutte le altre copie, ciò permette di richiedere l’unico valore valido alla rispettiva PE. Questa tecnica è adatta a una cache write-through;
* write-update: In questa policy non c’è un concetto di blocco valido o invalido, bensì di presenza o meno in cache. Di conseguenza, la lettura avviene quando il blocco è presente in cache, la scrittura invece, oltre a modificare il proprio blocco, modifica anche quelli presenti nelle altre cache. Con questa policy si ha quindi la certezza di ottenere un dato sempre aggiornato a ogni lettura, ciò è possibile con una cache write-back.

In generale, questi ultimi protocolli sono più onerosi da implementare.

### Esempio

Si considera ora un problema di cache coherence con policy write-invalidate, per farlo si utilizzano le tabelle decisionali, esse contengono lo stato dei blocchi e le rispettive azioni da eseguire.

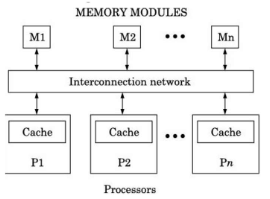
Nel caso della read, è possibile:

* leggere un blocco valido e sovrascriverlo nella propria cache, ciò si può fare quando il blocco è in cache, è invalido ed esiste un blocco valido in un’altra cache;
* Con le stesse condizioni è anche possibile leggere un blocco dalla propria cache;
* La stessa azione è possibile quando è presente un blocco valido in cache;
* Quando c’è un blocco in cache e si sa che esiste un blocco valido in un’altra, si può:
  + rimpiazzare il blocco con uno valido;
  + scrivere in memoria il blocco ma solo se è in modifica;
  + leggere un blocco dalla propria cache;
* Se non si ha un blocco in cache e non ne esiste uno valido nelle altre cache, si può:
  + muovere un blocco valido dalla memoria alla cache;
  + scrivere in memoria il blocco ma solo se è in modifica;
  + leggere un blocco dalla propria cache.

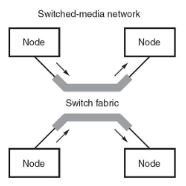
Per quanto riguarda la write invece, l’unica condizione è sapere se il blocco è in cache o no:

* in caso di risposta positiva, si può scrivere i dati in un blocco oppure invalidarlo nelle altre cache per bloccarne la condivisione;
* in caso di risposta negativa, oltre alle due azioni citate in precedenza, bisogna anche muovere un blocco dalla memoria alla cache per rimpiazzarlo e, se il blocco rimpiazzato viene modificato, scriverlo in memoria.

# Multiprocessori con rete di interconnessione

Nonostante i vantaggi della cache, i sistemi a bus condiviso non permettono l’inserimento di più di 10 processori perchè il bus fa da collo di bottiglia e la cache-coherence causa dell’overhead.

Nonostante i protocolli di cache coherence, non esiste una vera e propria soluzione effettiva, questi infatti possono impiegare tempo.

Cosa si fa allora? si sostituisce il bus con una rete di interconnessione, ottenendo di conseguenza più cammini tra PE e memoria anziché uno solo.

Come si può vedere dalla foto a destra, ogni PE ha la sua cache e comunica con le memorie attraverso la rete. Questo schema è detto UMA dato che il tempo di accesso alle memorie è uguale per tutte le PE.

Per ottenere prestazione migliori sull’accesso alla memoria, si passa da una rete condivisa a una commutata, per farlo si partiziona la capacità disponibile in porzioni indipendenti, queste vengono assegnate a ogni PE quando intende comunicare.

## Crossbar

La crossbar è il caso ideale di una rete commutata in termini di latenza e prestazioni, qui PE e memorie hanno in proprio bus, questi si incrociano formando una griglia, come si vede nella foto.

Nelle intersezioni sono presenti dei circuit switch (CS) che si aprono/chiudono in base all’input e quindi direzionare il traffico su una delle uscite.

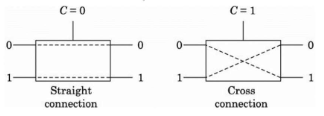
Il vantaggio è che gli N accessi possono andare in parallelo se ogni PE va in memorie differenti, si fa invece arbitraggio sui CS quando ci sono più richieste per una stessa memoria.

L’unico svantaggio riguarda il numero di CS, utilizzati, esso infatti è quadratico rispetto al numero di PE e memorie e quindi, con una complessità di , la complessità e il costo della rete salgono.

## Cosa si può fare per diminuire il numero di CS?

Per diminuire il numero di CS mantenendo una bassa probabilità di blocco, si riorganizzano in modo più adeguato quelle disponibili, precisamente si fa tutto questo in stadi consecutivi, ogni PE deve attraversarli tutti per andare a destinazione.

Le reti fatte in questo modo vengono dette Multistage Interconnection Network (MIN).

Ogni CS è partizionato in switching elements (SE) collegati attraverso una topologia, tipicamente hanno 2 ingressi e due uscite più un pin per il controllo che determina la connessione tra l’input e l’output:

* si parla di straight connection quando i pin di input sono connessi ai corrispondenti pin di output;
* si parla invece di cross connection se i collegamenti tra i pin si incrociano, come si vede in figura.

L'insieme degli stati di ogni SE corrisponde allo stato della MIN, in ognuno di essi gli SE non sono connessi tra loro.

Il collegamento tra SE di stadi contigui è determinato da una connection function mentre un sequenza di collegamenti e SE è detta path.

Un path particolare è il connection assignment e collega un input e un output specifici, ciò corrisponde a uno stato complessivo della MIN.

## Reti commutate

Queste reti commutate possono essere:

* bloccanti, cioè non è sempre possibile connettere un input e un output liberi;
* re-arrangeable non blocking: sono reti non bloccanti che connettono input e output liberi ricostruendo il cammino se necessario;
* non bloccanti: è sempre possibile connettere input e output liberi senza ricostruzioni;
* unicast: la corrispondenza tra input e output è 1 a 1, quindi viene collegato un pin di input a uno di output;
* multicast: connettono un pin di input a un insieme di pin di output, la corrispondenza è quindi 1 a N;
* singlepath: esiste un solo cammino per ogni coppia input-output;
* multipath: esistono più cammini per ogni coppia input-output.

## Proprietà delle MIN unicast

La complessità in spazio delle MIN unicast è data dal numero di SE minimo per realizzare ogni cammino possibile, nel crossbar ad esempio la complessità è O(n^2).

In tempo, la complessità equivale al numero di SE da attraversare per connettere un input a un output, ciò equivale al numero di stadi da attraversare. Nel caso della crossbar, la complessità in tempo è O(1).

Esiste un lowerbound sul numero di SE?

Dal momento che ogni coppia input-output è un connection assignment e indicano uno stato della MIN (per un totale di n!), ci sono due possibili stati per ogni SE, il numero totale è quindi il seguente:

Quanti stati servono per fare tutte queste connessioni insieme?

Ebbene, deve essere maggiore del numero delle connection assignement, quindi:

Da questi calcoli si conclude che il numero di SE minimo di una MIN è per connettere ogni input a ogni output.

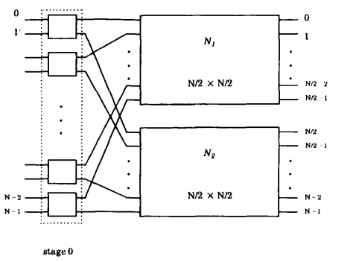
## Generalized Binary Network (GBN)

Le GBN sono MIN bloccanti, unicast e singlepath avente un numero NxN (con N=2^n) di SE, essa è definita nel seguente modo:

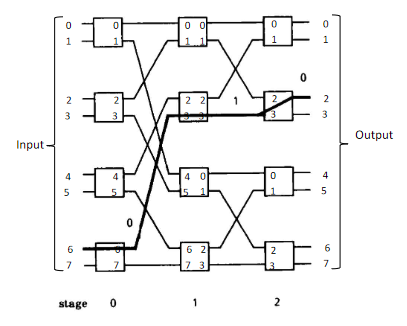
* tutte le SE sono 2x2;
* ha un numero stadi, numerati da 0 a n-1;
* ha SE per stadio;
* vale la buddy property: ogni coppia di SE dello stato i è connessa a una sola coppia di SE dello stato i+1;

Esistono diverse MIN appartenenti alle GBN, tutte queste sono equivalenti.

## Baseline

Nelle reti baseline, input e output sono rappresentati da numeri interi, di conseguenza, da una sequenza di bit. La connection function è la reverse shuffle e sposta il bit meno significativo il più possibile a sinistra, facendolo diventare il più significativo.

Queste reti vengono costruite ricorsivamente definendo uno stadio a ogni passaggio:

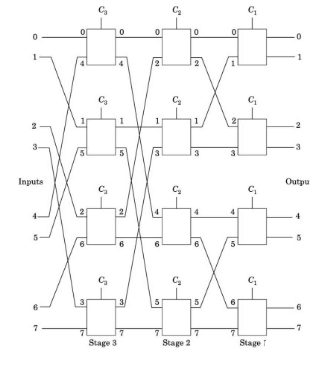
* lo stadio 0 è definito come un blocco NxN;
* per lo stadio 1, si divide il blocco iniziale in due sottoreti grandi la metà e si collegano tra loro mediante il reshuffle e così via.

A ogni passo si scompone la rete in due sottoreti e si divide sempre togliendo il bit più significativo.

Sempre a ogni passo le porte di output vengono rinumerate togliendo il bit più significativo, quelle di input sono invece uguali alle porte di output rispetto al precedente stadio. Tutte queste vengono poi connesse attraverso la funzione di reshuffle.

Per completare tutto c’è bisogno di passi e, dato che a ogni stadio il numero di porte si dimezza, la rete sarà composta da SE.

Con N=8, si hanno 3 stadi e 4 SE per un totale di 12. Per scegliere il connection assingnment si effettua lo XOR tra input e output, i bit del risultato ottenuto indicano i segnali di controllo partendo dal bit meno significativo:

Se si considerano l’input 6 e l’output 2:

6 XOR 2 = 110b XOR 010b = 100b

## Banyan

La rete di Banyan mantiene gli stadi numerati come interi, ciò però avviene specularmente rispetto alla baseline, numerando gli stadi partendo dall’output. Inoltre, tutte queste etichette differiscono al k-esimo bit, dove k è lo stadio.

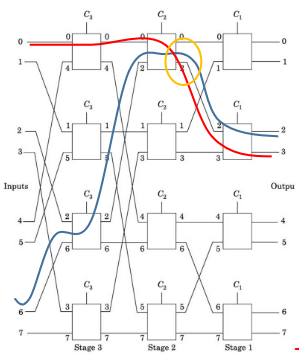
Tutte le porte tra due stadi contigui aventi etichette uguali vengono connesse, il routing avviene come nella baseline solo che la sequenza di bit ottenuta non viene invertita.

## Omega

Le reti omega presentano anche qui stadi numerati (a partire dall’input) ma utilizzano una funzione di shuffle differente detta perfect shuffle, uguale al reverse shuffle nel funzionamento con la differenza che è il bit più significativo a diventare il meno significativo.

I vari SE sono numerati secondo una notazione riga-colonna, essi indicano rispettivamente la riga e lo switch.

Tutte le porte tra stati contigui aventi la stessa etichetta vengono numerate.

Per quanto riguarda il routing, questo avviene allo stesso modo della baseline.

## Dove si verificano i blocchi?

Ebbene si verificano quando per ogni coppia indipendente ci sono SE comuni nell’instradamento.

Considerando una situazione come quella della figura a destra, i path 6→ 2 e 0→ 3 devono per forza passare dall’output 2, di conseguenza c’è bisogno di arbitraggio.

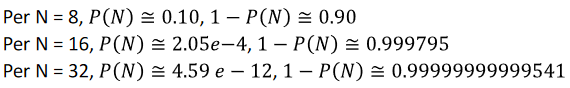
## Qual è la probabilità di blocco?

Dal momento che una GBN ha SE, ci possono essere assegnazioni differenti.

Assumendo che tutti gli output abbiano la stessa probabilità di essere richiesti, la probabilità di permutazione P{N} è realizzata nel seguente modo:

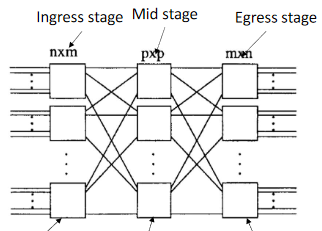
La probabilità di blocco sarà quindi la formula complementare alla precedente.

Da ciò si conclude che N! supera per N>=4, quindi non si possono realizzare tutti i CA.



Proprio per questo motivo, queste reti non vengono utilizzate per la produzione.

# MIN non bloccanti

Le GBN sono limitate a causa della probabilità di blocco, quindi come si può procedere? Si utilizzano reti non bloccanti, tutte figlie delle reti di Clos.

Le reti di Clos sono particolari reti aventi tre stadi e tre parametri, esse sono rappresentabili con un predicato C(n,m,p), cioè:

* il primo stadio ha p SE e collegano n input a m output;
* il secondo stadio ha m SE e collegano p input a p output;
* il terzo e ultimo stadio ha p SE che collegano m input a n output.

Le reti di Clos permettono un aumento del numero di SE e delle interconnessioni, grazie a ciò è possibile garantire l’assenza di blocchi.

Ogni SE dello stadio 2 è connesso a una porta dello stadio 1: il primo SE dello stadio 2 è quindi connesso alla porta 1 del primo SE e così via.

Lo stesso discorso vale anche per lo stadio 3 e il suo precedente.

Questa topologia è molto più malleabile e connessa rispetto alle GBN, si può infatti dimostrare che:

* con m>=2n+1 non sono bloccanti grazie al pigeon principle;
* con m>=n sono riarrangiabili, cioè che bisogna cambiare il percorso nel caso in cui un SE risulti occupato.

## Pigeon principle

Supponendo di collegare un’input i a un output j, nel caso peggiore tutte le porte sono occupate, quante sono? sono 2n+2 in totale (n+1 per parte), di conseguenza serve uno switch in più.

L’idea è quella di aggiungere più risorse in modo da evitare i blocchi, tutte queste aggiunte non vengono sprecate.

## Sistemi con memoria distribuita

I sistemi a memoria condivisa sono impraticabili quando ci sono più di 100 CPU a causa dei costi.

Per aumentare questo numero, bisogna rinunciare all’idea della memoria condivisa e che questa sia accessibile a tutti i PE alla medesima latenza.

Da ciò sono state create due soluzione:

* multiprocessori NUMA: sistemi con memoria fisicamente distribuita ma logicamente condivisa, ogni PE può accedere quindi alle memorie degli altri (con tempi di accesso maggiori);
* multicomputer: sistemi con memoria ad accesso privato, ogni PE non può accedere alle memorie degli altri. L’accesso ai dati viene quindi effettuato attraverso uno scambio di messaggi tra PE.

Per permettere ciò, sono quindi necessarie reti di interconnessione scalabili, cioè la capacità di erogare prestazioni a pieno carico. Per ottenere scalabilità, è necessario che la rete abbia:

* bassa latenza per consentire sincronizzazioni veloci;
* alto throughput per trasferire i dati tra PE in poco tempo;
* alta scalabilità, ovvero offrire prestazione dipendenti dalla quantità di risorse, quindi un carico maggiore al crescere dei PE.

Per essere scalabile, una rete di interconnessione deve avere le seguenti caratteristiche:

* Topologia: sapere quali sono i possibili cammini tra i pacchetti;
* Routing: tra tutti i cammini, quali sono quelli disponibili?;
* Arbitraggio: Come si gestiscono le contese di risorse tra pacchetti?
* Switching: come si allocano le risorse ai pacchetti?

## Multiprocessori NUMA

L’idea dei multiprocessori NUMA è quella di avere dei nodi indipendenti, ogni nodo è composto da una compute unit (CE) e un’interfaccia di rete (NIU).

La CE è composta da una PE, una memoria, una cache e una directory, quest’ultima è utile per la cache coherence.

Tutti questi componenti vengono messi in comunicazione attraverso un bus.

La NIU permette alla CU l’accesso ad altre memoria attraverso operazioni di LOAD e STORE, i tempi di accesso sono tuttavia più lunghi rispetto alla memoria locale.

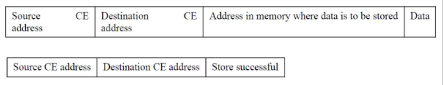
Esistono due varianti di questa architettura:

* NC-NUMA: NUMA senza cache coherence, presenta tempi di accesso più alti;
* CC-NUMA: NUMA con cache coherence, permette più velocità.

In questi sistemi, tutti i moduli di memoria formano uno spazio di indirizzamento condiviso.

### Accesso alla memoria remota

Per accedere alla memoria remota, si utilizzano degli indirizzi, precisamente:

* l’indirizzo viene scomposto in due parti: l’indirizzo del PE e l’offset del modulo di memoria;
* l’operazione di LOAD converte l’indirizzo e controlla se è locale o no. Se l’indirizzo è locale, allora carica in memoria tutti i dati che servono, altrimenti manda un messaggio tramite la rete;
* Il destinatario del messaggio raccoglie i dati e li invia al mittente.
* L’operazione di STORE è analoga al LOAD con la differenza che è anche presente il dato da scrivere.

### Quanto costa tutto ciò?

Il costo dipende dalle caratteristiche di sistema, considerando T come il tempo di invocazione di un comando sulla rete, B come il bandwidth e m come il numero di byte del pacchetto:

* il tempo di propagazione di un pacchetto di n byte è n/B.
* la lettura di una parola di memoria in remoto prende tempo q, stessa cosa vale per il destinatario.
* Il tempo totale è quindi il seguente:

Considerando un NUMA con:

* 256 PE, ognuno con 16 MB di memoria e un programma in cui il 10% sono LOAD e il 15% sono STORE;
* l’accesso alla memoria prende 5 cicli di clock più un overhead di 20 per iniziare la trasmissione;
* 100 MB/s di bandwidth;
* parole di 32 bit;
* 400k istruzioni eseguite.

| Qual è il tempo totale considerando solo accessi locali? | E invece se il 25% degli accessi è remoto? |
| --- | --- |

### Cache coherence nei sistemi CC-NUMA

Dai calcoli effettuati prima, è necessario ridurre il numero di accessi remoti. Una soluzione potrebbe essere gestire opportunamente la memoria, ciò però è troppo specifico per la macchina.

La soluzione quindi è dotare ai PE di una cache ed effettuare la cache coherence, l’assenza del bus però non permette di fare snoopy.

Cosa si fa in tal caso?

Si passa alla cache coherence di tipo directory-based: dal momento che i PE non possono osservare tutti gli accessi alla memoria, si permette tutto ciò in maniera indiretta utilizzando una sorta di database che memorizza in quale PE si trova una linea di cache e il suo stato.

Ogni volta che si fa riferimento alla cache, si interroga il database, tutto questo deve quindi essere memorizzato in una memoria veloce.

La directory è distribuita tra i vari PE, ognuno memorizza le entry corrispondenti alla sua memoria.

### Esempio

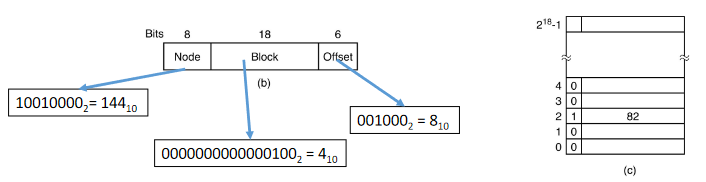
Per fare un esempio, considerando le stesse specifiche dell’esempio precedente, si aggiunge che:

* ogni cache line è lunga 64 B;
* ogni PE ha 2^24 B di RAM, corrispondenti a 2^18 cache line corrispondenti.

Supponendo che il PE 20 faccia una LOAD, la MMU interpreta l’indirizzo come segue:

* i primi n=log2(#PE) indica a quale PE appartiene;
* gli m=log2(#cacheline) successivi indicano quale blocco prendere;
* i bit restanti sono l’offset e indicano quale linea di cache considerare.

Considerando l’indirizzo 0x24000108, il PE 20 manda una richiesta al PE 144, quest’ultimo controlla la propria directory e scopre che la cache line 4 non è presente, quindi:

* legge il dato dalla propria memoria;
* lo invia al PE 20 che lo mette nella propria cache;
* il PE 144 aggiorna la propria cache indicando che la linea inviata è nella cache del PE 20.

Si suppone ora che il PE 20 invia una richiesta al PE 144 per la cache line 2, quest’ultimo controlla e nota che la linea è nella cache del PE 82 che è dirty.

Il PE 144 invia quindi un messaggio al PE 82 per dirgli di inviare la linea al PE 20, invalidando la copia di quest’ultima.

In caso di linea clean, la locazione viene inviata, altrimenti si notifica al PE corrispondente di inviare una copia. La linea rimarrà dirty finché tutto non verrà aggiornato, come una specie di commit.

Questo tipo di cache coherence è quindi più inefficiente dello snoopy, quanto spazio occupa?

Per ogni linea ci sono 9 bit, quindi in tutto esistono 2^18 linee per PE, quindi:

# Multicomputer

I multicomputer sono NUMA con memoria prima, anche logicamente. L’architettura non cambia molto rispetto a prima, l’unica cosa che manca infatti è la directory dato che non ce n’è bisogno.

La comunicazione anche qui utilizza reti scalabili, la differenza rispetto a prima è che queste reti sono dirette.

Le reti di interconnessione dirette sono simile a quelle utilizzate in precedenza, qui però i PE sono integrati con le risorse di comunicazione. Ogni PE costituisce un nodo della rete ed è connesso a tutti gli altri in modo diretto attraverso uno switch.I nodi tra loro connessi non hanno bisogno di switch esterni per la comunicazione, per quelli non connessi c’è bisogno del routing fatto da nodi intermedi.

La latenza della rete diretta dipende dal diametro, cioè la distanza nella rete tra i nodi più lontani, il cammino più lungo.

Diametro e latenza sono tra loro direttamente proporzionali, per diminuirla serve quindi un grafo più connesso.

Le prestazioni e il costo della rete dipendono dal costo dei nodi e quindi il loro grado. Un grado alto permette maggiori prestazioni ma allo stesso tempo un costo maggiore rispetto a uno basso.

Occorre quindi trovare un compromesso.

## Topologie

### Topologia a stella

Nella topologia a stella, un nodo centrale fa da commutatore (switch) tra sorgente e destinazione, ogni nodo ha quindi grado 1 (il minimo possibile) e c’è bassa latenza dal momento che il diametro è lungo 2.

Questa topologia è tuttavia inaffidabile dato che un guasto del commutatore fa venire meno la comunicazione, inoltre non è scalabile perché lo switch stesso fa da collo di bottiglia.

### Topologia ad anello

Nella topologia ad anello ogni nodo è connesso ad altri due (quindi tutti hanno grado 2), formando appunto un anello. Anche qui c’è poca scalabilità dato che, con N nodi, il diametro è lungo N/2, inoltre è presente scarsa affidabilità perchè il guasto dei nodi partiziona la reti in due insiemi non comunicanti, precisamente:

* nel routing unidirezionale, ciò avviene col guasto di un nodo;
* nel routing bidirezionale, il partizionamento si presenta col guasto di due nodi.

### Topologia a mesh

La topologia a mesh dispone i nodi su una griglia lunga Mx\*My, ogni nodo è connesso a tutti i vicini, quindi dato che alcuni di questi stanno sui bordi, il grado minimo è 2 (vertici) e quello massimo è 4 (centrali).

Il diametro è Mx+My-2 indipendentemente dal cammino fatto, ciò porta quindi a problemi di scalabilità, inoltre è presente un problema ingegneristico causato dai nodi di grado differente.

### Topologia a toroide

La topologia toroidale risolve alcuni problemi di quella a mesh rendendo tutti i nodi di grado 4, per farlo si collegano tra loro i nodi ai bordi o ai vertici.

Questi collegamenti in più diminuiscono il diametro (adesso è lungo Mx/2 + My/2) e permette maggior affidabilità in quanto sono presenti più cammini, tuttavia il costo è maggiore a causa dell’alto numero di connessioni tra nodi.

### Topologia a griglia 3D

Queste topologia è una generalizzazione di quella a mesh, infatti ne condivide alcune proprietà:

* il diametro si calcola in modo simile, solo riadattato alle 3 dimensioni:

Mx + My + Mz + 3

* anche qui il grado dei nodi dipende dalla posizione, il massimo è 6 e il minimo è 3.

Anche in questo caso è possibile inserire degli archi toroidali.

### Topologia a ipercubo

La topologia a ipercubo è una generalizzazione di quella a griglia 3D. Un ipercubo collega N nodi sulle sue n=log2(N) dimensioni, quindi ogni nodo ha grado n. A ogni nodo è associato un indirizzo intero (da 0 a N-1), l’assegnamento è fatto in modo tale che due nodi vicini differiscano di un solo bit (distanza di Hamming uguale a 1).

La distanza tra nodi equivale alla loro distanza di Hamming, quindi il diametro è al massimo n.

Proprio per questo motivo questa topologia è scalabile, infatti il diametro cresce in modo logaritmico in base alla crescita lineare del numero di nodi.

Tutto ciò ha uno svantaggio, cioè quello di ottenere un costo maggiore dato che, al crescere dei nodi, cresce anche il grado e quindi il numero di link.

### Topologia a ipercubo k-ario

Questa topologia connette tra logo N=k^n nodi, dove n è il numero di dimensioni e k la radice, il numero di nodi in ogni dimensione. Il diametro é n\*k/2 e tutti i nodi presentano lo stesso grado.

Tutte le topologie fino a ora citate funzionano bene quando il traffico è ben distribuito su tutti i nodi, cosa succede però in caso di congestioni? In questo caso è necessario spartire il traffico, ciò però causa dei colli di bottiglia.

Proprio per questo motivo, sono state create delle topologie adatte alla distribuzione del traffico.

### Topologia fat tree

L’idea è quella di utilizzare una topologia ad albero binario, qui i nodi intermedi sono switch mentre le foglie sono PE.

Il numero di link verso i figli è sempre uguale al numero verso il genitore quindi più uno switch è vicino alla radice, più traffico dovrà gestire, perciò si dota quel nodo di più link.

### Fat tree generalizzato

L’idea rimane quella precedente con la differenza che cambia la regola di connessione: qui non si ha una sola radice e ogni switch a un livello si connette a ognuno di quello successivo. In ogni switch, metà delle porte è connessa ai padri mentre l’altra metà verso i figli, se serve è possibile utilizzare anche il channel bonding.

E’ possibile aggiungere quanti switch si vogliono in ogni livello, un numero maggiore porta ad aumentare il numero di cammini tra due nodi di differenti cluster, aumentando però la latenza.

## Tecniche di switching

Le tecniche di switching indicano tutti i meccanismi che favoriscono il trasferimento di un messaggio nella rete.

### Circuit switching

Il circuit switching è una tecnica in cui prima vengono allocate le risorse (link e porte) e poi si trasmettono i messaggi come flussi continui di bit.

Ciò garantisce la ricezione del messaggio da parte del destinatario, inoltre il tempo di trasmissione diventa indipendente dalla lunghezza del cammino dato che si fa pipelining dei messaggi.

### Packet Switching

Il packet switching è la tecnica duale alla precedente, qui non vengono allocate risorse dato che la connessione è assente. I messaggi vengono divisi in pacchetti e inviati utilizzando lo store & forward, come funzione?

Ebbene, si inviano altri pacchetti solamente quando il pacchetto precedente viene ricevuto, ciò causa una dipendenza del tempo di trasmissione rispetto al cammino.

### Wormhole switching

Il wormhole switching è una tecnica che cerca di trovare un equilibrio tra le prime due. L’idea è quella di scomporre il messaggio in pacchetti e questi ultimi in flit (flow control unit). Il primo e l’ultimo flit indicano rispettivamente l’inizio e la fine del pacchetto mentre quelli intermedi indicano i dati veri e propri. Il primo flit contiene inoltre informazioni riguardanti il routing.

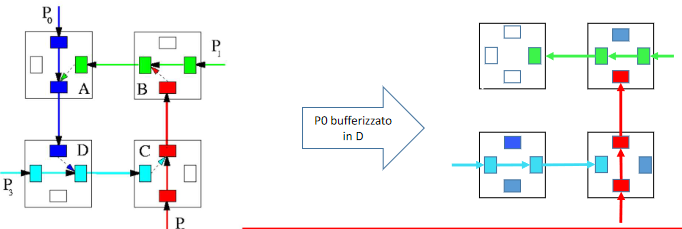
Quando il primo flit arriva in un nodo intermedio, viene esaminato e inoltrato nel link corretto, tutto ciò senza aspettare l’arrivo degli altri flit. Questa tecnica permette quindi il pipelining dato che un pacchetto diviso in flit può occupare più link in simultanea.

### E in caso di una linea occupata?

Nel caso in cui la linea dovesse essere occupata, viene eseguita la backpressure, ovvero si ferma l’invio dei flit finchè non torna disponibile.

La backpressure può però provocare deadlock a causa dell’attesa circolare, cosa si fa in questo casi?

### Virtual cut-through

Una prima soluzione è il virtual cut-through, cioè che, se un pacchetto è bloccato in un nodo, si ricevono e memorizzano tutti i flit del pacchetto su quel nodo, diventando di conseguenza uno store & forward.

Ciò significa che ogni nodo deve saper memorizzare un intero pacchetto e che, a causa di un blocco, si paga tutto in latenza per via dello store & forward.

### Virtual channel

Il virtual channel consiste nell’assegnare più buffer di input/output a ogni link (evitando le contese), permettendo a ogni buffer di avere pacchetti differenti allocati.

Dal momento che il link rimane unico, è possibile allocare più risorse e quindi fare multiplexing, ciò permette di evitare i blocchi e possibili attese circolari bloccanti.

## Routing

Il routing permette di scegliere il cammino utile per recapitare il messaggio, esso può essere:

* deterministico: il cammino dipende solo dalla posizione del mittente e del destinatario, sono molto semplici da progettare;
* adattativo: il cammino tiene conto anche di altri fattori come il traffico nei link, di conseguenza sono più onerosi da fare.

### Routing deterministico

Le famiglie più comuni del routing deterministico sono le dimension-ordered e consistono nell’organizzare i link in dimensioni consecutive, in questo caso il routing viene fatto per dimensioni crescenti.

Questi algoritmi evitano le attese attive dal momento che un link di una dimensione non essere bloccato per liberarne un altro di una dimensione inferiore.

Nel caso delle reti a mesh bidimensionali, il routing è XY: prima si fa routing sulla dimensione X e poi sulla Y.

Ad esempio: per inviare un messaggio da 2 a 7, prima si inoltre orizzontalmente da 2 a 1 e poi verticalmente da 1 a 4 e da quest’ultima a 7.

In presenza di canali congestionati, sono presenti delle attese e ciò è un problema.

### Routing adattativo

Questi algoritmi sono caratterizzati da tre criteri indipendenti:

* progressive: le scelte dei cammini non variano quando vengono prese, il contrario è il backtracking;
* profitable path: ogni link scelto permette di avvicinarsi alla destinazione, il contrario è misrouting path;
* completely adaptive: ogni possibile cammino può essere scelto, il contrario è partially adaptive.

L’algoritmo idle è progressive, profitable e completamente adattativo, esso si basa su un algoritmo deterministico che:

* sceglie un cammino iniziale e inizia a inviare il messaggio;
* se la porta di output scelta è occupata, allora ne sceglie un’altra libera che permette di avvicinarsi a destinazione;
* se invece la porta non esiste, il messaggio viene bloccato e attende che una porta si liberi.

Idle è minimale dato che sceglie sempre il cammino più breve (anche se i blocchi possono impedirlo), tuttavia non è deadlock free, eventuali deadlock devono essere rilevati e gestiti.

L’exhaustive profitable backtracking effettua una visita DFS della rete considerando solo i link profitable.

Il k-family routing controlla se la distanza sorgente-destinazione è minore di un valore k e, se è così, effettua una visita DFS, altrimenti effettua una ricerca euristica facendo pruning dei cammini.

# Cluster computing

I calcolatori paralleli ha subito un forte declino negli anni ‘90 a causa di:

* alti costi di produzione dovuti a componenti troppo specifiche;
* alti costi di sviluppo dato che gli ambienti utilizzati non sono standard;
* le aziende hanno smesso di produrli e quelle specializzate hanno chiuso.

Un’altra causa di questo declino è l’avvento dei cluster di calcolatori, cioè dei sistemi paralleli a basso costo i cui nodi sono normali calcolatori general purpose, tutti questi sono tra loro connessi da una rete general purpose.

L’approccio utilizzato è quello Commercial Off The Shelf (COTS), ciò permette di aumentare la potenza e le prestazioni del cluster semplicemente comprando un nuovo calcolatore.

Per nascondere la natura distribuita, i cluster utilizzano un middleware chiamato Single System Image.

Dal momento che i nodi di un cluster sono general purpose, l’ambiente di programmazione è uno standard e ciò permette di rendere portabili le applicazioni (semplicemente ricompilandole) e di sfruttare il parallelismo del sistema.

Rispetto ai normali calcolatori paralleli, i cluster sono a basso costo e ciò ha favorito il loro avvento.

## Architettura di un cluster

Un cluster non è tanto differente da un sistema parallelo, la differenza sta nella presenza del middleware, uno strato software che astrae i livelli inferiori offrendo servizi a quelli superiori, permettendo la Single System Image.

Non è necessario che ogni calcolatore abbia gli stessi componenti, l’omogeneità di questi è comunque una qualità dato che permette ottimizzazioni, tuttavia non è una condizione necessaria. Questo è dovuto al fatto che non ci si interfaccia direttamente coi calcolatori ma col middleware, quel che al massimo si può fare è indicare le varie specifiche.

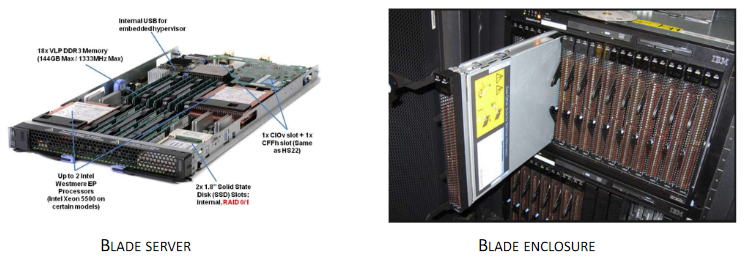
La rete di connessione non è solitamente proprietaria e dipende dalla connessione in sè.

Questo paradigma permette flessibilità supportando sia le applicazioni parallele (attraverso il Parallel Programming Environment), sia quelle sequenziali, ciò permette di supportare:

* l’high performance computing accelerando una singola applicazione parallela;
* l’high throughput computing massimizzando il numero di applicazioni completate nell’unità di tempo;
* l’high availability computing massimizzando la disponibilità di un servizio o dei dati nel tempo.

## A chi è venuta l’idea?

L’idea dei cluster è venuta a Donald Becker e Thomas Sterling al Caltech sviluppando un prototipo finanziato da NSF nel 1993. Questo prototipo prese il nome di Progetto Beowulf e aveva 16 PC con processore DX4 connessi tra loro da una rete Ethernet a 10 Mbps. Tutti questi nodi avevano inoltre Linux come sistema operativo, un driver per il channel bonding sviluppato ad hoc e una serie di librerie per lo sviluppo di applicazioni parallele.

L’approccio ebbe successo grazie ai bassi costi di realizzazione e ciò li portò a essere prodotti in fabbrica, il loro problema riguardava lo spazio occupato. Questo problema porto a un’evoluzione dei cluster grazie all’invenzione del blade: un calcolatore compatto da inserire in un apposito alloggiamento per connettersi con gli altri.

Le reti di comunicazione utilizzate dai blade erano inizialmente Ethernet, il tempo comunque portò alla creazione di soluzione specifiche come Infiniband e Myrinet.

# Sistemi many-core

I sistemi cluster sono diventati il ricettacolo dei many-core, essi vengono utilizzati come acceleratori su dati partizionabili, facendo poche cose molto velocemente. Questi sistemi sono per questi motivi accoppiati ai nodi tradizionali, rendendo questi sistemi eterogenei.

La variante di sistema many-core più diffusa è la GP-GPU (General Purpose Graphical Processing Unit), un processore creato da NVIDIA come interfaccia grafica programmabile per il rendering e poi evolutosi per fare anche altro.

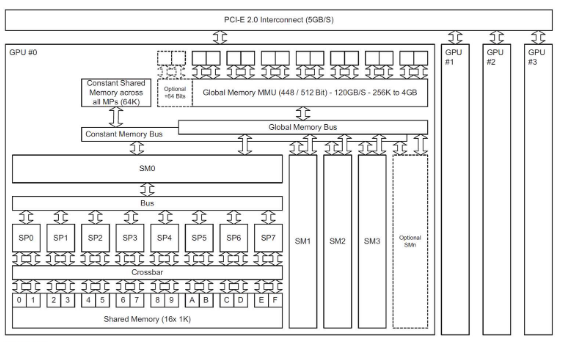
## Sistemi GP-GPU

I sistemi GP-GPU nascono basandosi sul fatto che in certi contesti occorre effettuare tante elaborazioni semplici in poco tempo, massimizzando il thoughput.

Le CPU infatti hanno bisogno di un ALU, cache, RAM e controllo, quindi con un instruction set semplice si possono limitare cache e controllo aumentare considerevolmente il numero di ALU, parallelizzando e aumentando il throughput.

Il collegamento tra CPU e GPU avviene tramite il bus PCI Express, utile per caricare i dati da una memoria all’altra.

## Architettura di una GP-GPU

L’architettura di una GP-GPU è fatta come visibile dalla foto. In alto c’è il bus PCI-Express per la comunicazione con “l’esterno”, questo bus è connesso a ogni GPU. Ogni GPU è formata nel seguente modo:

* una memoria;
* una Global memory MMU;
* una serie di memorie connesse tra loro da un bus detto global memory bus, queste vengono dette streaming multiprocessor (SM).

Ogni SM è connesso a un bus con coi comunica con degli streaming processor (SP), questi sono equivalenti ai core nelle CPU. Tutti gli SP comunicano attraverso una rete tipo crossbar con una memoria condivisa di capacità limitata.

## Come funziona?

* Ogni SP di una SM comunica con una memoria interna ubicata nel chip;
* Il register file è una memoria a latenza 0 che memorizza il contenuti dei registri dei vari thread in una SM, risulta quindi utile nei context switch;
* La Special Purpose Unit è un’unità speciale che esegue istruzioni particolari non general purpose, come ad esempio particolari istruzioni hardware;
* Texture e constant cache sono cache particolari utilizzate per il rendering.

## Come si programma?

Il paradigma di programmazione di una GP-GPU è Single Instruction Multiple Thread (SIMT), una variante di SIMD in cui:

* ogni SM supporta l’esecuzione di thread paralleli detti warp;
* ogni thread in un warp esegue la stessa istruzione su diverse partizioni dei dati;
* le istruzioni possono anche contenere salti condizionali, è quindi normale avere thread in un warp che fanno cose differenti.

# Programmazione parallela

Dal punto di vista astratto, l’obiettivo è sviluppare un programma parallelo che sfrutti adeguatamente le risorse in modo efficiente, come si fa? Si implementa l’algoritmo parallelo con un linguaggio di programmazione sul computer in questione.

Come si misurano le prestazioni di questo algoritmo e ci sono fattori che le limitano?

## Speedup

Per misurare le prestazione si utilizza lo speedup, l’incremento di velocità rispetto al miglior tempo sequenziale:

In cui:

* p è il numero di processori;
* T(1) è il tempo di esecuzione sequenziale;
* T(p) è il tempo di esecuzione con p processori;

Il caso ideale è quando S(p)=p, cioè che lo speedup è lineare.

## Efficienza e costo

L’efficienza è un’altra misura di performance che mette in relazione lo speedup col numero di processori utilizzati per ottenerlo:

Con uno speedup lineare, l’efficienza tende a 1, quindi si ottiene il caso migliore quando ogni processore viene utilizzato al meglio.

Il costo è la relazione tra tempo di esecuzione e il numero di processori utilizzati:

Il costo indica le risorse utilizzate da un algoritmo parallelo.

## Scalabilità

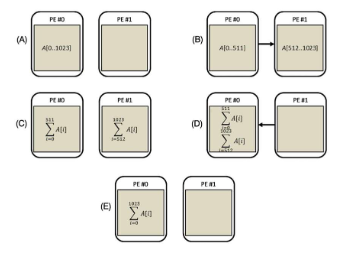
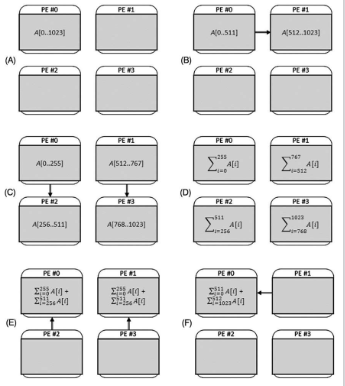
La Scalabilità non è altro che l'efficienza al variare del numero dei processori, essa può essere:

* strong quando il carico rimane sempre lo stesso con un numero di processori sempre crescente;
* weak quando, al variare del numero di processori, varia anche il carico in modo proporzionale.

### Esempio

Per fare un esempio: si considera la somma di un vettore di interi lungo n, come si fa a calcolarla con p processori?

Si considerano le seguenti ipotesi:

* n è una potenza di 2 equivalente a 2^k con un k fissato;
* ogni PE può sommare due numeri in memoria in un’unità di tempo U;
* la comunicazione tra PE richiede 3U, a prescindere dalla quantità di dati trasmessa;
* il vettore è inizialmente memorizzato nel PE 0, a sua volta il risultato deve essere memorizzato lì;
* I PE sono sincroni: in un certo istante tutti devono essere nello stesso stato.

Per prima cosa si calcola il tempo sequenziale, avendo un solo processore, questo effettuerà tutte le somme una dopo l’altra, quindi:

T(1)=n-1

### Con due processori?

Considerando due processori, si ha che:

* Il PE 0 manda metà vettore al PE 1;
* entrambi calcolano le somme parziali, mettendoci n/2 -1 U di tempo a testa;
* Il PE 1 manda il suo risultato a PE 0;
* PE 0 effettua la somma finale;

Quindi con un vettore lungo n si ha che:

T(2)=(3+n/2-1+3+1)\*U = 6U +nU/2

### Con 4 processori come si procede?

* All’inizio si distribuisce l’array tra tutti e 4 i processori facendo una sorta di riduzione ad albero, mettendo in tutto 6U, 3 da parte del PE 0 e 3 dal PE 1;
* Tutti i PE calcolano le somme parziali e ci mettono (n/4 -1)U;
* I PE 2 e 3 inviano i risultato ai propri mittenti (rispettivamente 0 e 1), mettendoci 3U;
* PE 0 e 1 calcolano le somme parziali mettendoci una U;
* PE 1 invia i dati a 0 in 3U;
* PE 0 calcola la somma finale.

Ricapitolando:

T(4)=(6+n/4 -1 +3 +1+3+1)U = (13+n/4)U

Con 8 processori si effettua lo stesso ragionamento fatto in precedenza mettendoci (20+n/8)U, cosa si può concludere?

Si conclude che, per uno stesso carico di lavoro, aggiungere processori non diventa conveniente dato che si perderebbe tempo in comunicazione.

Infatti, considerando un numero p=2^q di processori e n=2^k, si ha che:

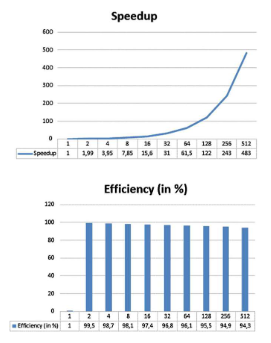
* la fase di distribuzione richiede 3U\*log2(p)=3qU;
* la fase di calcolo richiede 2^(k-q)-1 U;
* la fase di raccolta richiede sempre 3qU;
* la somma dei risultati parziali richiede qU.

Si conclude che:

T(p)=2^(k-q)-1+7q

### Osservazioni

Si osserva che con p << n, viene effettuato più calcolo rispetto alla computazione, di conseguenza il tempo di calcolo domina e ciò permette costi bassi e speedup ed efficienza alti.

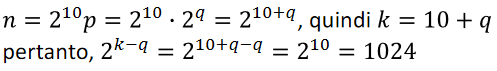
Al contrario, con p tendente a n, è il tempo di comunicazione a dominare dato che i vari PE deve inviare/ricevere i dati giusti, ciò causa un costo alto e speedup ed efficienza bassi.

Si conclude che l’algoritmo non è strongly scalable.

Si considera ora lo stesso esempio con la differenza che il numero di elementi n da sommare cresce insieme al numero p di processori, cosa si può osservare?

Essenzialmente il tempo di calcolo rimane costante al crescere di p, infatti:

Ponendo n=p\*1024, il tempo di calcolo rimane sempre uguale a 1024!



Al contrario, il tempo di comunicazione cresce in modo logaritmico rispetto a q.

Si conclude che l’algoritmo è weakly scalable.

## E se l’unità di tempo U non è sempre fissa?

Dal momento che U dipende dai tempi di comunicazione e di calcolo, si generalizza separando i due concetti:

* α è il tempo di calcolo;
* β è il tempo di comunicazione.

Da ciò si può dire che la distribuzione dei dati richiede un tempo β e il calcolo , inoltre ci vuole un tempo β\*q per raccoglierli e α\*q per sommarli.

Si conclude quindi che:

Da ciò si ricava la nuova formula dello speedup:

Ponendo come il rapporto tra calcolo e comunicazione, si ha che:

* con > 1 è il calcolo a prevalere;
* con < 1 è la comunicazione a farlo.

Si può riscrivere quindi la formula dello speedup:

## Come varia lo speedup al crescere della comunicazione?

Per effettuare questo controllo, si utilizza il limite:

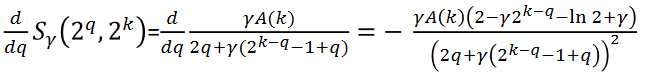
Ciò significa che al crescere della comunicazione, lo speedup va a 0, quindi si perde tempo a fare comunicazione anzichè fare calcolo.

Per capire l’andamento e se ci sono massimi e minimi, si utilizzano i differenziali:



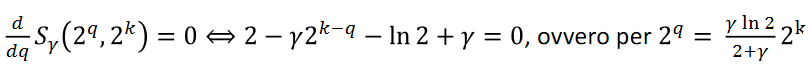
Ciò fa intendere che la funzione è crescente, quindi dati ᵧ1 < ᵧ2, si ha che

Fissando ᵧ e k, si può aumentare il numero di processori? Per capirlo si deriva rispetto a q:



Qual è il segno? Dipende dalle variabili, ha più magnitudine rispetto alle altre.

Ponendo tutto a 0, si ottiene che:



Quindi ed è un massimo per un dato valore di q.

Detto questo, si può concludere che per un dato valori di , esiste un numero ottimale p di processori e, al crescere di , decresce p, non avrebbe quindi senso aggiungere altri processori dato che non verrebbero sfruttati.

Inoltre il valore dello speedup dipende da p e , questo valore cresce al crescere di p fino a quando non raggiunge una soglia oltre la quale il valore decresce, questa soglia indica il numero ottimale di processori.

Di conseguenza, lo speedup ottimale dipende dal valore di dato che quest’ultimo influenza il numero ottimale di processori.

Per un valore n fissato, lo speedup diventa una funzione monotona.

## Legge di Amdahl

Tutto ciò che è stato detto prima dipende dalla legge di Amdahl, essa fissa un limite superiore e calcola il tempo di esecuzione in due parti:

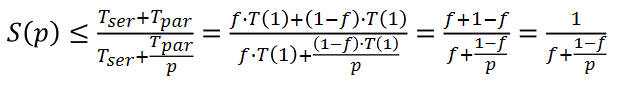
* Tser è il tempo di esecuzione non parallelo;
* Tpar è il tempo di esecuzione parallelo.

Con un processore, si ha che T(1)=Tser+Tpar.

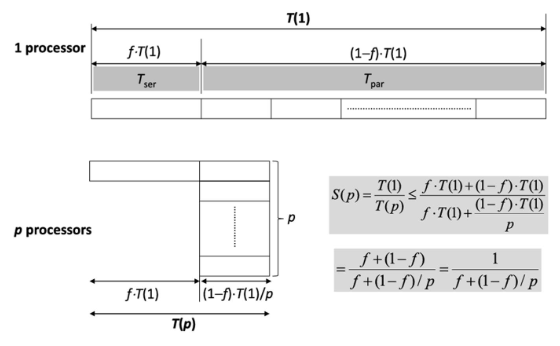
Nel caso parallelo ideale, T(p)= Tser+Tpar/p, in quello reale però T(p) è maggiore o uguale.

Dato che nel caso ideale lo speedup è lineare, si può quindi riscrivere nel seguente modo:

Si denota ora f = Tser/T(1), quindi si ha che Tser=f\*T(1) e Tpar=(1-f)\*T(1), inoltre si ridefinisce S(p) come segue:





In parole semplici, Tser e Tpar sono percentuali del tempo complessivo, rispettivamente equivalgono a f\*T(1) e (1-f)\*T(1).

Inoltre da questa formula si può intuire che Tser è la parte che determina il valore massimo di speedup ottenibile.

Un f piccolo infatti implica che c’è tanta parte parallelizzabile e quindi lo speedup è elevato, al contrario invece il programma risulta poco parallelizzabile e quindi lo speedup sarà basso.

Con 6 processori e un f pari al 5%, qual è il massimo speedup possibile?

Quant’è il massimo speedup ottenibile con una f al 10%?

Si calcola lo speedup per un numero di processori tendente a infinito:

Questo risultato si basa sulla strong scalability, la quantità di lavoro infatti è fissata.

Ciò non è particolarmente simpatico, quindi che si fa? si può cercare di ottimizzare il più possibile ma ciò ha dei limiti.

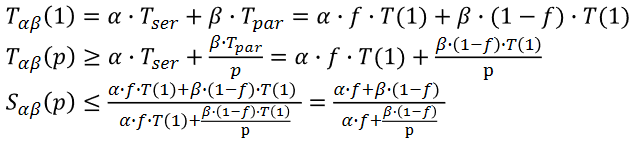
## Legge dell'upper bound scalato

La legge dell’upper bound scalato supera il limite della legge di Amdahl (cioè la strong scalability) aumentando la quantità di lavoro al crescere dei processori.

Questa legge funziona definendo le seguenti quantità:

* α, una funzione indicante come decresce il tempo sequenziale al crescere del numero di processori;
* β, una funzione in cui si indica come, al crescere dei processori, decresce il tempo parallelo.

Da queste quantità, si procede nel seguente modo:



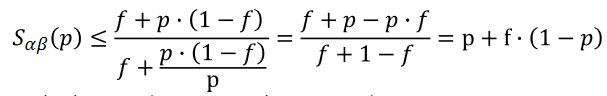
Quindi, denotando , si può riscrivere lo speedup come segue:

La legge di Amdahl è quindi un caso particolare in cui α e β non variano, quindi ˠ è uguale a 1.

## Legge di Gustafson

La legge di Gustafson è un importante caso particolare dell’upper bound scalato in cui α e β sono rispettivamente uguale a 1 e p, di conseguenza ˠ è anch’esso uguale a p.

Questa legge indica che la parte sequenziale non scala rispetto a p, cosa che invece fa in modo lineare quella parallelizzabile, quindi:



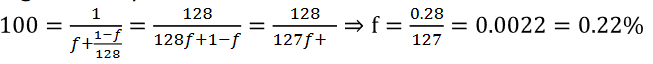
Lo speedup cresce al crescere del numero di processori p se cresce anche la quantità di lavoro, ciò permette di rispondere anche ad altre domande, ad esempio:

supponendo di avere una f al 15% e un numero di processori p pari a 50, quant’è lo speedup?

Quant’è il massimo speedup ottenibile scalando l’input di 100 volte?

Con uno speedup di 100 su 128 processori, quant’è la massima parte sequenziale in strong scalability?

Per trovare ciò, si inverte la formula.



E in weak scalability invece?



# Modelli di programmazione parallela

I modelli di programmazione parallela specificano i task che devono essere eseguiti in parallelo, assegnando a ciascuno di essi i dati su cui operare.

La necessità di un modello deriva da due fattori:

* il primo riguarda la natura del problema, infatti se non è possibile aggiungere lavoro, occorre arrangiarsi. In generale però ciò è sempre possibile dato che esistono applicazioni di questo tipo;
* Se è possibile sviluppare applicazioni che sfruttano il parallelismo, come si fa? si utilizza un modello di computazione parallelo, consentendo la specifica dei task e l’assegnamento dei dati da elaborare a questi ultimi.

I modelli paralleli possono essere:

* impliciti: il parallelismo non viene esplicitato dato che implementato nella struttura del programma;
* semi-impliciti: si identificano le sezioni parallele del programma, specificando i dati ai task ma non quanti bisogna allocarne.
* espliciti: tutto è esplicito, dalla creazione dei task, all’assegnamento dei dati, alla comunicazione.

## Modelli impliciti

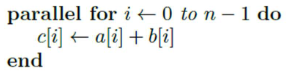
I modelli impliciti sono tipici dei linguaggi funzionali, in essi il programma è equiparabile a un grafo in cui i nodi sono le funzioni. Questa rappresentazione permette di individuare i vari percorsi tra loro indipendenti ed eseguirli in parallelo.

Altri tipi di modelli impliciti riguardano gli algoritmi a scheletro parallelo: tutto il parallelismo è nascosto al programmatore, quel che si fa è implementare il codice dei singoli task.

Un esempio di algoritmo a scheletro parallelo è il MapReduce.

Modelli semi

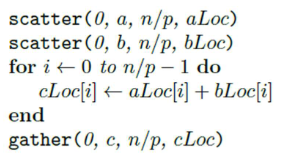
## Modelli semi-impliciti

I modelli semi-impliciti arricchiscono il linguaggio con delle primitive capaci di individuare le parti parallele del programma, qui però bisogna gestire in modo opportuno le varie dipendenze tra i dati.

In questi modelli non si specifica però quante risorse sono necessarie e a quali task assegnarle

Un esempio di primitiva parallela è il parallel for: un ciclo for che esegue ogni iterazione in parallelo.

## Modelli espliciti

I modelli espliciti sono quelli più a basso livello e sono supportati da librerie, qui è possibile gestire l’allocazione dei dati, la definizione dei task e a quali processori allocarli. Alcune librerie che supportano modelli espliciti sono:

* Message Passing Interface (MPI);
* OpenMP;
* Partitioned Global Address Space (PGAS).

## Come si sviluppa un programma parallelo?

Lo sviluppo di un programma parallelo è supportato da quattro fasi da effettuare in sequenza: Partitioning, Communication, Agglomeration e Mapping.

### Partitioning

Il partitioning identifica le possibili parti parallele dividendo il lavoro in task piccoli, l’obiettivo è massimizzare il numero di task, ciò viene detto parallelismo a grana fine.

La decomposizione avviene in due modi:

* domain decomposition: prima si partizionano i dati in unità da processare in parallelo e dopo specificando la loro elaborazione (definendo i task). Questa tecnica permette solitamente di trovare il maggior numero di task.
* functional decomposition: prima si decompone il lavoro in task indipendenti in base alle funzioni eseguibili in parallelo e poi si associano i dati a ciascuno.

### Communication, Agglomeration e Mapping

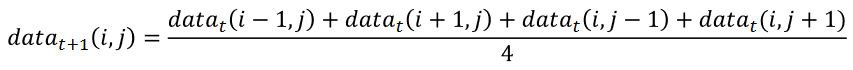
La communication permette di identificare le dipendenze tra i dati e quindi far comunicare adeguatamente i task.

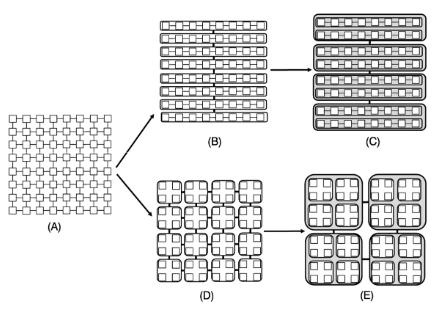
L’agglomeration mette insieme di task di granularità fine per diminuire la comunicazione, l’obiettivo infatti è quello di ottenere un rapporto tra comunicazione e computazione adeguato al sistema scelto e quindi uno speedup ottimale. I task agglomerati vengono posti nella stessa memoria logicamente condivisa al fine di velocizzare la comunicazione rispetto ai task separati.

Il mapping permette di allocare ogni task ai vari processori cercando di minimizzare la comunicazione, massimizzare il parallelismo e bilanciare il carico di lavoro. Tutti questi obiettivi sono in contrasto e inoltre il miglior bilanciamento si ottiene con un algoritmo NP-Completo, di conseguenza vengono utilizzate euristiche con risultati subottimali.

## Esempio di PCAM: Stencil code

Si vuole parallelizzare l’applicazione di uno stencil code su una matrice,a ogni iterazione si aggiornano tutti gli elementi (i,j) facendo una media tra i vicini:



* Partitioning: ogni elemento della matrice è un task, bisogna inoltre considerare che non tutti hanno quattro vicini dato che potrebbero essere sui bordi;
* Communication: riguarda i vicini stessi, questi devono essere comunicati per permettere l’operazione;
* Agglomeration: si uniscono diversi task, tutti gli elementi in un task comunicano attraverso la shared memory mentre utilizzano la rete per comunicare con task esterni. Sono possibili più combinazioni di agglomeration, ognuna con le proprie qualità.

Quale agglomerazione scegliere? Quella più efficiente! C’è un modo per stimarla? Sì!

Si suppone che T(n)=s+r\*n sia il tempo di inviare n byte con uno startup di s e un reciproco del throughput di r.

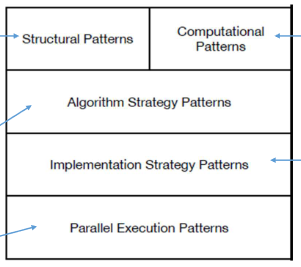
Considerando l’agglomerazione C nella foto, ogni processore scambia n byte con ogni vicino (facendo quindi due comunicazioni), quindi il tempo di comunicazione è Tc=2\*(s+r\*n).

Considerando l’agglomerazione E, ogni processori scambia n/2 byte con ogni vicini, quindi il tempo di comunicazione è .

Da ciò si conclude che la seconda soluzione è preferibile alla prima dato che, per un numero p di processori alto, il tempo di comunicazione decresce mentre nel primo caso rimane costante.

# Parallel Design Patterns

I design pattern paralleli forniscono un supporto allo sviluppo di programmi paralleli data la loro complessità, infatti è difficile:

* individuare il parallelismo sfruttabile nella soluzione di un dato problema;
* individuare la parti esprimibili in modo parallelo;
* implementare un algoritmo parallelo efficiente per una data architettura.

I design pattern paralleli possono essere:

* strutturali: definiscono le componenti che definiscono la struttura di un programma ad alto livello;
* computazionali: indicano gli elementi base per la costruzioni di componenti di un programma o di un pattern strutturale;
* algorithm strategy: definiscono algoritmi paralleli per pattern strutturali e computazionali;
* implementation strategy: indicano i pattern per implementare algoritmi paralleli in modo efficiente sui vari calcolatori, essi definiscono anche strutture dati che supportano gli accessi concorrenti in modo efficiente;
* parallel execution: sono quelli più a basso livello e permettono l’allocazione dei task e dei dati sui vari tipi di calcolatori.

## Esempi di pattern strutturali

### Model View Controller

Il model view controller è formato da tre componenti:

* il model cattura il comportamento dellapplicazione in termini di dominio, risolvendo e gestendo direttamente i dati, la logica e le regole dell’applicazione;
* il view rappresenta l’output delle informazioni prodotte dall’applicazione;
* il controller accetta l’input e lo converte in comandi per il model o per la view.

Questo design pattern è utilizzato particolarmente nelle interfacce grafiche.

### Pipe and Filter

Il pipe and filter è un pattern che funziona utilizzando due componenti:

* i filter, i quali eseguono trasformazioni sui dati;
* i pipe che connettono i vari filter tra loro.

### Agent and repository

Il pattern agent and repository funziona attraverso tre componenti:

* gli agent, cioè le entità computazionali che operano su dati condivisi;
* la repository, una struttura dati condivisa da tutti gli agenti;
* il manager, il componente che mantiene la consistenza tra i dati.

### Map Reduce

Il map reduce funziona attraverso due componenti:

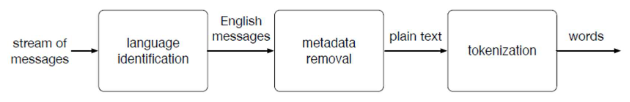
* la componente map effettua la stessa operazione su diversi oggetti di una stessa collezione;
* la componente reduce applica un’operazione ai risultati delle map.

### Esempio di applicazione

Si vuole analizzare un flusso di documenti per contare le occorrenze delle varie parole. Dal codice a destra possono essere estratte le seguenti fasi:

* estrazione delle parole da ogni documento;
* contare ogni parole utilizzando un dizionario.

Per parallelizzare la prima fase, si utilizza un pipe and filter che ottiene le parole prima identificando il linguaggio, poi rimuovendo i metadati e facendo la tokenizzazione.



Per parallelizzare la seconda fase si utilizza il map reduce, precisamente:

* si fa un’operazione su ogni occorrenza attraverso il map, ottenendo dalla stringa in input un dizionario di parole con il loro conteggio;
* tutti i dizionari vengono uniti attraverso una reduce, ottenendo un dizionario che contiene il conteggio di tutte le parole.

## Pattern computazionali

I pattern computazionali sono utili per esprimere le componenti dei pattern strutturali, essi vengono utilizzati solitamente per l’elaborazione.

## Pattern algorithm strategy

Questi pattern definiscono delle strategie ad alto livello per sfruttare la concorrenza, per farlo esistono diversi approcci:

* parallelismo dei dati: si fa la stessa elaborazione su dati differenti;
* parallelismo dei task: ogni task corrisponde a una specifica elaborazione effettuata su dati diversi o condivisi;
* slit ricorsivo: tutti i task vengono generati ricorsivamente;
* eventi discreti: l’elaborazione viene suddivisa in task semi-indipendenti i quali interagiscono in base al flusso dati;
* eccetera.

## Pattern Implementation strategy

Questi pattern definiscono le strategie per implementare algoritmi paralleli in modo efficiente, gli approcci utilizzati sono i seguenti:

* Single Program Multiple Data (SPMD), in cui lo stesso programma viene eseguito su dati differenti;
* Master-Worker: un insieme di task identici eseguono il lavoro dato da un task master;
* Loop parallelism: tutte le iterazioni indipendenti di un ciclo vengono eseguite in parallelo.

Gli implementation strategy pattern definiscono inoltre delle strutture dati che supportano la concorrenza in modo efficiente (in particolare gli accessi in lettura/scrittura), ad esempio:

* Shared Queue: una coda condivisa tra i task;
* Shared Hash Table: una hash table condivisa tra task;
* Array distribuito: distribuzione di slice distinte a task concorrenti.

## Pattern parallel execution

i pattern parallel execution indicano come un programma parallelo viene eseguito su un calcolatore parallelo, per farlo esistono varie strategie e meccanismi riguardanti l’avanzamento del program counter e il coordinamento dei task.

Ci sono tantissimi pattern di questo tipo, la sfida è capire qual è il migliore in base alla situazione, cosa si fa a decidere?

## Modello di esecuzione parallela

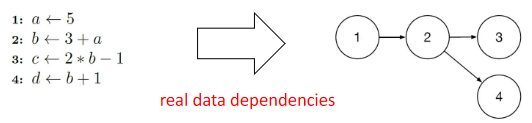
Il modello di esecuzione parallela è la risposta alla precedente domanda, esso è necessario per definire il parallelismo in modo da:

* rappresentare i task e le varie dipendenze astraendole al calcolatore;
* supportare il partitioning e l’agglomeration;
* consentire l’analisi teorica delle prestazione dell’algoritmo.

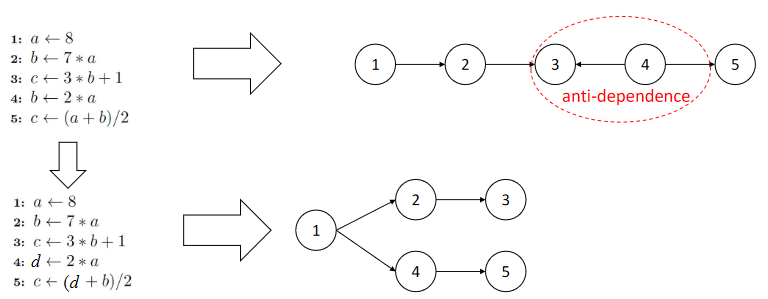
### Modello a task graph

Il modello di esecuzione parallela più efficace è quello a task graph,un grafo diretto e aciclico in cui:

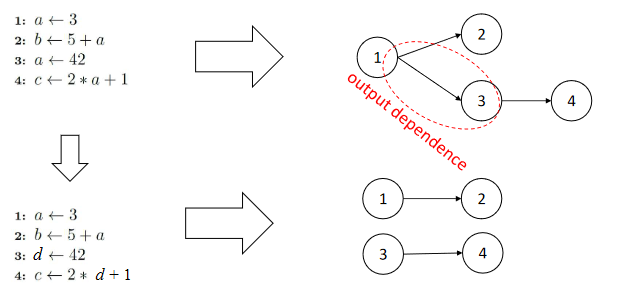
* l’unità elementare è il task, il quale rappresenta anche i nodi;
* gli archi del grafo indicano le varie dipendenze tra task.

L’idea è quella di eseguire prima i task senza dipendenze e poi tutti gli altri, quali sono però le dipendenze reali?

Considerando la foto a destra, ogni istruzione è un task, le dipendenze invece riguardano l’utilizzo delle variabili, infatti il task 2 dipende dal primo dato che viene utilizzata la variabile a.

Le istruzioni di controllo invece non possono essere parallelizzate, quindi vengono eseguite in un unico task.

### Dipendenze apparenti

Esistono anche forme di dipendenza che sono solo apparenti, nella foto a destra ad esempio c’è un anti-dependence dovuta al riassegnamento della variabile b, questo problema è però risolvibile utilizzando un’altra variabile.

Un’altra dipendenza apparente è l’output dependence, nella foto a destra si può notare sulla terza istruzione ed è risolvibile come nel caso precedente.

Detto ciò, come si organizzano i vari task? Ebbene, si esegue un mapping sul task graph per assegnare i vari task ai processori, indicando anche come agglomerarli.

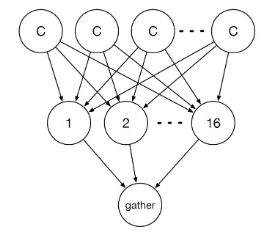
Il task graph fornisce informazioni utili per ottenere una granularità corrispondente al calcolatore scelto.

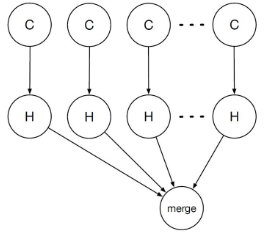
# Pattern algoritmici di decomposizione

Si vuole estrarre la distribuzione di complessità da un insieme di documenti, come si può estrarre tutto il parallelismo da ciò?

Un’ipotesi è quella di procedere calcolando un istogramma con 16 intervalli in cui:

1. si prende ogni documento e si calcola la sua complessità. Ogni documento rappresenta quindi un task;
2. a seconda dei valori ottenuti si incrementa il contatore degli intervalli corrispondenti.

Quest’ultimo passo può avere due composizioni:

* nella prima si associa un task a ogni documento per calcolarne la complessità. Ci sono poi 16 task (uno per ogni livello dell’istogramma) i quali raccolgono tutti i risultati che ricadono nel proprio intervallo e incrementando un contatore. Infine un task gather prende tutti gli intervalli e crea l’istogramma.
* la seconda è uguale alla prima nella fase di calcolo della complessità, qui però si associa un task a ogni risultato per calcolare degli istogrammi parziali, quindi in questo caso ogni task gestisce tutti i livelli. Tutti questi istogrammi vengono poi uniti in un task unico attraverso un task merge.

Quale soluzione è la migliore?

Nella prima soluzione i task del secondo passo rimangano costanti ma il loro carico cresce, nella seconda invece è duale e, proprio per questo motivo, si preferisce quest’ultima.

La decomposizione non deve basarsi su assunzioni relative al sistema parallelo, quel che si deve fare è creare il maggior numero di task possibile e che questo numero cresca al crescere del carico.

## Embarassingly parallel

In questo pattern non c’è bisogno di interazione tra i task se non per l’aggregazione, essi sono utili per i problemi di Monte Carlo, cioè quelli basati sul random sampling e/o utilizzando parametri casuali.

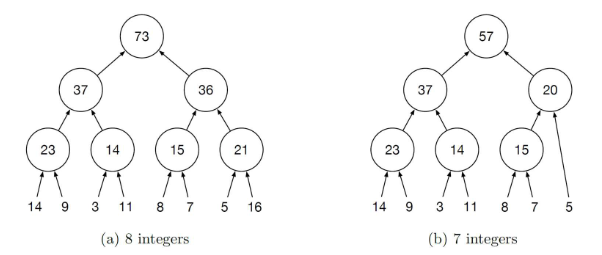
Per fare un esempio, si stima il valore di π con un metodo di Monte Carlo: dato un cerco di raggio 1 inscritto in un quadrato:

* si calcolano un certo numero di punti in modo casuale;
* si stima l’area del cerchio moltiplicando la percentuale di punti caduti dentro il cerchio per l’area del quadrato;
* si calcola π con la formula inversa.

## Riduzione

La riduzione è un altro pattern algoritmico che applica un’operazione binaria associativa su un insieme di dati. L’associatività permette il calcolo parallelo di valori parziali fino alla soluzione.

Per fare un esempio, supponendo di dover sommare n valori, si procede nel seguente modo:

* si dividono tutto l’insieme in coppie disgiunte di valori, in tutto sono n/2;
* si calcola la somma di ogni coppia, tutto ciò viene fatto in parallelo utilizzando n/2 task;
* si ritorna al primo punto tenendo in considerazione i risultati fino a che non si ottiene un unico valore.

In generale, al passo i vengono utilizzatitask per sommare lo stesso numero di coppie di risultati parziali.

Un altro esempio è il k-means parallelo, in questo caso, dato un insieme di punti e un numero k di cluster che si vogliono ottenere:

* si estraggono k punti random e si considerano come centri dei cluster;
* Per ogni punto si trova il centro più vicino calcolando la distanza, ciò viene fatto in parallelo.
* si ricalcolano i centri facendo la media e si ritorna al secondo punto.

## Scan

L’idea dello scan è quella di utilizzare l’operazione binaria associativa su un vettore ordinato in cui l’elemento i-esimo contiene il risultato dei precedenti. Lo scan può essere:

* inclusivo: l’i-esimo elemento è incluso nello scan;
* esclusivo: l’i-esimo elemento è escluso dallo scan.

Lo scan prende il nome di prefix sum quando l’operazione binaria è una somma.

Un’applicazione dello scan è il run-length encoding: la compressione di sequenze di lettere uguali in coppie (V,S) dove V è la lunghezza della sequenza di valori S contigui.

WWWWBBBWWBBBBB -> 4W3B2W5B

La decodifica può essere parallelizzata creando un task per ogni coppia, tuttavia i task non sanno dove posizionare la sottosequenza, come fare?

Si utilizza la prefix sum esclusiva delle coppie precedenti: 4W calcola 0, 3B calcola 4, 2W calcola 7 e 5B ottiene 9.

Rispetto alla riduzione, lo scan è molto più complesso da parallelizzare per via delle dipendenze tra task, come si fa a parallelizzare?

## Algoritmo di Blelloch

L’algoritmo di Blelloch genera un albero binario contenente tutte le informazioni necessarie per lo scan, ottenute utilizzando la somma per riduzione. Blelloch sfrutta il fatto che alcuni valori calcolabili sono già presenti nell’albero, come si calcolano quelli mancanti? Ebbene, si propaga il risultato nella parte destra dell’albero.

Ogni task associa il valore del nodo al figlio sinistro, quello destro invece ottiene sommando il valore del nodo con la riduzione del fratello a sinistra.



Il risultato corrisponde alla somma dei valori i-esimi dello scan esclusivo e inclusivo.

## Algoritmo di Hillis-Steele

L’idea dell’algoritmo di Hillis-Steele è quella di costruire un albero sul vettore sommando a ogni passo, al passo i-esimo si somma il contenuto dell’elemento j< con l’elemento , memorizzandolo in quest’ultima cella.

L’output da restituire è indicato dall’ultimo valore dell’array alla fine di ogni iterazione.

## Data decomposition

La data decomposition è l’approccio duale alla decomposizione in task, qui si decompongono prima le strutture dati in porzioni e dopo si assegnano queste ultime ai task.

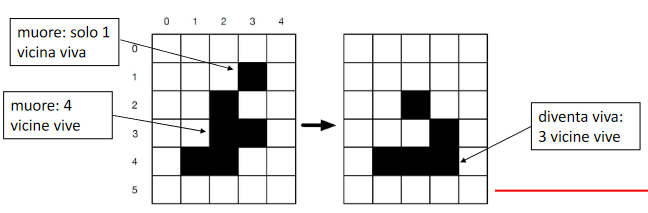
Ogni task esegue la stessa computazione su porzioni di dati differenti, proprio per questo motivo quest’approccio è detto data parallel.

La data decomposition è molto utilizzata nelle applicazioni a griglia, per esse si scandisce ogni punto della griglia ricalcolando il valore in base ai punti vicini.

### Esempio: Game of life

Un esempio di applicazione a griglia è il game of life: la matrice è un automa cellulare su cui sono disposte le cellule, ognuna di esse può essere viva o morta e dipende dallo stato di quelle vicine.

A ogni passo si aggiorna lo stato di ogni cellula nel seguente modo:

* una cellula al passo K rimarrà viva in quello successivo se ha 2 o 3 cellule vicine vive;
* una cellula morta al passo K ritornerà viva in quello successivo se ha 3 cellule vicine vive.

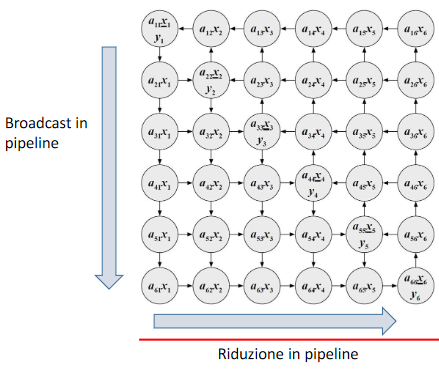
La data decomposition nel game of life si applica assegnando un task a ogni cellula, ogni task calcolerà lo stato della cellula e ha bisogno dei dati provenienti dalle 8 vicine.

### Esempio: prodotto matrice-vettore

Considerando ora come esempio la moltiplicazione di una matrice A(n,n) per il vettore x(n), come si può parallelizzare?

Le unità di lavoro elementari sono gli elementi di A e di x, il partizionamento organizza i task in una matrice T in cui:

* T(i,j) memorizza a(i,j);
* T(i,i) memorizza x(i) e memorizzerà il risultato y(i);
* T(i,j) calcolerà la moltiplicazione a(i,j)\*x(j).

Come si inviano tutti i dati che servono ai task?

Dato che T(i,J) ha bisogno di x(j) (che si trova in T(j,j)) per fare il calcolo, si fa un broadcast di tutti gli x(j) sulle colonne utilizzando una pipeline, in totale vengono trasmessi n-1 messaggi.

Dopo aver calcolato il risultato y(i), bisogna trasmetterlo a T(i,i), come fare? si effettua la riduzione per riga, anche questa possibile mediante pipeline.

Il task graph ottenuto è visibile nella foto a destra, si possono notare bene il broadcast per ogni colonna degli x(j) e la riduzione per riga di ogni y(i).

## Pattern implementativi

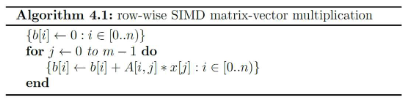
I pattern algoritmici non si prestano bene all’implementazione dato che:

* i task sono potenzialmente tanti e possono causare un elevato overhead per gestirli;
* troppe comunicazioni possono portare a problemi di efficienza;
* l’asincronicità dei task non sempre è supportata dalle architetture parallele.

Cosa si può fare? Si utilizzano i pattern implementativi per agglomerare più task.

### Esempio: prodotto matrice-vettore

Considerando come esempio il prodotto matrice-vettore, una prima idea è quella di dividere la matrice in righe e moltiplicarle per il vettore, si sarà quindi un task per ogni riga.

A ogni iterazione i-esima, si moltiplica l’i-esima colonna della matrice con l’i-esimo elemento del vettore, dopodichè si sommano i risultati di ogni riga.

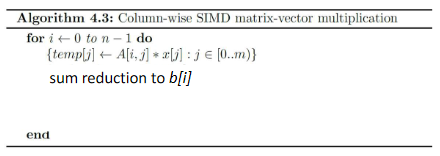
Nel codice a destra è un’implementazione SIMD del prodotto matrice-vettore, tra le graffe sono presenti le istruzioni parallelizzabili con SIMD, è un buon procedimento? Per rispondere a questa domanda bisogna considerare come viene memorizzata una matrice:

* in column-major, gli elementi di ogni colonna sono contigui, permettendo così il caricamento dei registri in una sola transazione di memoria;
* row-major ha lo stesso significato ma per le righe, quindi servono diverse transazioni per caricare gli elementi delle colonne nei registri.

Se nel primo caso non ci sono problemi, nel secondo invece occorre cambiare la disposizione dei task.

### Versione column-major

Dividendo per colonne la matrice, ogni task è associato una colonna e a un elemento del vettore, quindi calcola tutti i risultati in parallelo.

La matrice di risultati ottenuta viene infine ridotta per riga sfruttando l’algoritmo di Blelloch ma applicato su un input. 

Dato che SIMD non prevede la partecipazione di tutti i PE per via della condizione, si utilizza una struttura control-divergent per l’elaborazione: si gestisce l’operazione se e solo se vale una data condizione.

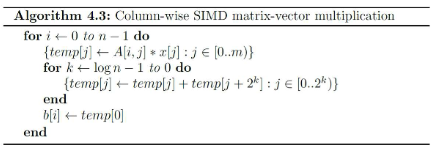
Questa proprietà non è comunque desiderabile dato che:

* le istruzioni condizionali sono più onerose rispetto a quelle aritmetiche;
* il parallelismo viene ridotto in mancanza di questa proprietà.

### Eliminazione della control divergence

Per eliminare la control divergence in questo esempio, si può osservare che al passo k c’è la seguente osservazione:



Gli elementi da sommare però sono in spazi di memoria non contigui, come fare? Si divide il vettore in due e si sommano le metà, procedendo ricorsivamente, ciò vale per via dell'associatività della somma per riduzione.

Una cosa che bisogna tenere a mente è quella di:

* evitare dipendenze tra iterazioni;
* evitare control divergence;
* adattare l’algoritmo alla disposizione dei dati in memoria.

# Implementazioni MIMD con Shared memory

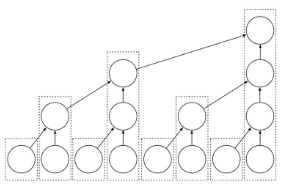
Nel parallelismo MIMD, la computazione è un insieme di thread, ognuno condivide il programma l’heap e la memoria globale, inoltre hanno un loro program counter e una propria memoria privata.

Il sistema operativo schedula i thread sui vari processori in vari modi:

* la migrazione, cioè che dopo una sospensione, il thread viene eseguito da un PE diverso dal precedente. Questa soluzione permette di bilanciare il carico ma fa diventare la cache cold, ripartendo quindi da una cache vuota (portando a rallentamenti);
* il pinning è la soluzione duale alla precedente: il task viene eseguito sempre sullo stesso PE anche dopo una sospensione.
* l’affinity scheduling invece sceglie i PE per i thread in base a dati criteri.

## Fork-join

Nell’implementazione fork-join, il lavoro viene suddiviso in task dinamicamente, ogni task ne crea altrettanti attraverso un’operazione detta fork. Per ottenere i risultati dai task figli, il task genitore effettua l’operazione di join, attendendo la fine dei figli.

Questa tecnica si adatta particolarmente per algoritmi divide and conquer, un esempio è la stima del π con metodo Monte Carlo: si decompone l’algoritmo in task formando un task graph ad albero binario. Ogni task intermedio crea i figli e accumula i risultati mentre i punti vengono generati dai task foglia. In questo modo si risolve anche il problema della soluzione embarassingly parallel, il quale non si pone il problema di come comunicare i dati.

Con un numero infinito di PE, si potrebbe assegnare un thread a ogni task, ogni thread creerebbe i propri figli aventi metà del carico assegnato e così via per un totale di 2\*n-1 task di cui n sono foglie, si può fare di meglio?

Sì, basta far generare un solo figlio anzichè due e far svolgere il carico rimanente al task genitore. Ciò permette di evitare che il task genitore rimanga in attesa dei figli e riduce a O(n) il numero di task.

Nonostante il parallelismo venga sfruttato bene, il numero di PE nella pratica è però limitato e minore di n, il numero di thread in questa soluzione rimane quindi troppo alto.

A fronte di ciò, esistono due possibili alternative:

* si creano comunque n thread e si fa fare tutto allo scheduler, ciò però sforza il sistema operativo e l’overhead sarebbe eccessivo;
* si mette un limite ai thread da generare non effettuando più il fork al suo raggiungimento, il thread in questo caso deve svolgere il lavoro in modo sequenziale.

## Come si assegnano i task ai thread?

L’assegnamento dei task ai thread avviene in due modi:

* nell’assegnamento esplicito, è il programmatore che deve assegnare ogni thread a ogni task, tuttavia è necessario conoscere le caratteristiche del sistema ed è inoltre limitante dato che:
  + nel parallelismo a grana fine c’è troppo overhead;
  + nel parallelismo a grana grossa si limita tutto il parallelismo raggiungibile;
  + ci possono essere problemi nel caso in cui i task richiedano quantità di lavoro differenti.
* Un approccio migliore è specificare quali task sono eseguibili e quali devono attendere i figli. Tutto ciò viene gestito da un thread pool in cui ogni thread ha una propria coda di task in cui inserire i figli. Il sistema operativo in alcuni casi può intervenire spostando i task tra le code. Il codice per permettere questo funzionamento è dato dalle seguenti istruzioni:
  + spawn f(): crea un thread che esegue la funzione f();
  + sync: attende il completamento dei figli.

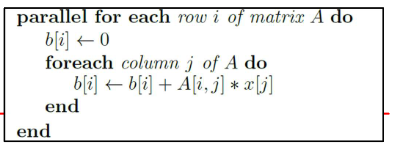
In generale per le implementazioni fork-join valgono le seguenti considerazioni:

1. non bisogna creare più thread del necessario dato che scheduling e prelazioni possono causare overhead;
2. utilizzare un cutoff per limitare il numero di thread e/o la ricorsione;
3. fare attenzione agli accessi concorrenti delle variabili condivise.

## Parallel loop

Il parallel loop è uno dei modi più semplici per fare parallelismo MIDM con shared memory, qui infatti ogni iterazione è parallela e ognuna non dipende dalle altre.

Le caratteristiche di questa implementazione sono le seguenti:

* ogni interazione ha un task associato;
* alla fine c’è una barriera di sincronizzazione che consente di andare avanti solo se tutte le iterazioni sono terminate;

Questa implementazione si presta molto bene ad algoritmi embarassingly parallel o data parallel, per questo motivo in alcuni linguaggi ci sono costrutti che consentono di fare tutto ciò in modo semplice.

L’assegnamento può avvenire in due modi:

* nel modo statico, è il compilatore a decidere quante e quali iterazioni vengono eseguite da ogni thread;
* ne metodo dinamico invece, il supporto a runtime assegna ogni iterazione a un thread non appena quest’ultimo si libera della precedente iterazione.

## Assegnamento statico

L’assegnamento static può avvenire in vari modi: due di questi sono:

* divide and conquer: si sfrutta il fork-join per schedulare i task dividendo il loop ricorsivamente in due parti fino a che non si raggiunge il cutoff;
* chunk contigui: l’idea è che, date n interazioni e nt thread, ogni thread si prende n/nt task contigui e li esegue.

La presenza della barriera di sincronizzazione non indica comunque l’assenza delle corse critiche dato che possono esserci accessi a variabili condivise, come si evitano queste dipendenze?

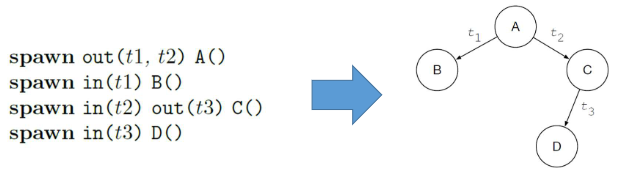
Si impongono forme di sincronizzazione prefissate, prima si sincronizzano i padri coi figli e dopo si sincronizzano tutti i thread alla fine di ogni iterazione.

Alcune classi di algoritmi però richiedono la sincronizzazione dei soli task aventi relazioni di dipendenza arbitrarie oppure che non richiedono la barriera di sincronizzazione.

### Come si risolvono le dipendenze tra task?

Un possibile approccio viene effettuato dal programmatore specificando le varie dipendenze e lasciando lo scheduling al supporto runtime, questo può essere:

* indiretto: si annotano input e output di ogni task per specificare le dipendenze, è più flessibile dato che non dipende dall’architettura del sistema, quindi si adatta sia alla shared memory, sia alla distributed memory;
* le dipendenze vengono specificate attraverso il task graph;

La comunicazione tra task viene implementata in modo automatico in entrambi gli approcci dal momento che i valori da comunicare sono già esplicitati.

Un modo per specificare queste dipendenze può essere ad esempio l’aggiunta di clausole in() e out() da applicare insieme all’istruzione spawn, come si può vedere nella foto a destra.

## Schema SPMD

Lo schema Single Program on Multiple Data si adatta bene alla data decomposition dato che:

* ogni task esegue lo stesso codice ma su porzioni differenti di dati;
* ogni thread è identificato da un id numerico univoco e lo utilizza per capire dove prendere i dati;
* Non c’è nessun meccanismo di sincronizzazione.

A fronte di tutti questi vantaggi, l’unica pecca è che l’assegnazione dei task ai thread viene effettuata dal programmatore.

Lo SPMD ha avuto successo con le GPU dato che l’approccio sta alla base di queste ultime.

## Come avviene la conversione da SPMD a SIMT?

* il programmatore specifica l’assegnamento dei task ai thread;
* I thread vengono divisi in blocchi della stessa dimensione;
* Si sceglie la grandezza dei blocchi, di solito coincide col numero di SP;
* ogni thread di ogni blocco viene eseguito in SIMD, la granularità è il programma eseguito e non la singola istruzione;
* ogni thread di un blocco vengono eseguito dalla stessa SM e di conseguenza condivido la risorse che ha quest’ultima.
* Si alloca la memoria in GPU utile per contenere i dati;
* Si trasferiscono i dati nella memoria appena allocata;
* Si converte il codice SPMD.

### Come si converte il codice SPMD?

* Si inseriscono istruzioni per far capire ai thread il proprio gruppo e il proprio id;
* si fanno calcolare a ogni thread le porzioni di dati che devono considerare;
* si aggiungono istruzioni che consentano l’aggregazione degli output di ogni blocco di thread.

## Esempio di riduzione SIMT

Supponendo una GPU con 32 SM da 64 SP (per un totale di 2048 SP), una possibile strutturazione è quella di utilizzare 32 blocchi da 64 thread. Supponendo un , ogni thread ha numeri.

Nel codice a destra:

* a e c sono array globali indicanti rispettivamente input e output;
* si definisce b come variabile condivisa in un blocco, essa prende l’id del thread e del gruppo e da essa calcola l’inizio e la fine delle parti di array da considerare per il calcolo.

Supponendo che l’id del thread e del gruppo siano rispettivamente 4 e 21 e ci sono 64 thread per blocco, si ha che:

Considerando ora il codice a destra:

* la variabile psum contiene la somma parziale del carico assegnato, il suo valore viene memorizzato in b[id];
* synGroup indica la barriera di sincronizzazione, tutti i thread del gruppo devono sincronizzarsi;
* Per garantire la terminazione di un’iterazione prima di iniziare la successiva, si utilizza un algoritmo SIMD con una barriera di sincronizzazione. Al termine b[0] conterrà la riduzione effettuata in modo globale dal gruppo di thread;
* l’assegnazione di c[groupId] è l’aggregazione del risultato.

## Master Worker

Lo schema master-worker è un approccio più generale che consiste nell’assegnamento dinamico dei task ai thread, per farlo:

* si utilizza una coda condivisa contenente tutti i task;
* I task vengono generati da un thread master;
* tutti gli altri thread sono worker e prendono i task dalla coda;
* Il programma termina quando la coda è vuota.

Con l’implementazione shared-memory, occorre evitare le corse critiche sulla coda, ogni thread (master compreso) si gestisce la propria sincronizzazione.

# Implementazione MIMD a memoria distribuita

L’implementazione MIMD a memoria distribuita, l’unità di esecuzione non è il thread bensì il processo, ognuno col proprio spazio privato di memoria.

La sincronizzazione e lo scambio di dati tra processi avviene attraverso uno scambio di messaggi esplicito.

Il controllo dei processi avviene con delle primitive che ne permettono il controllo (creazione, terminazione, eccetera) e la comunicazione, queste vengono fornite da apposite librerie le quali ne astraggono il meccanismo.

Questo paradigma permette di implementare ogni pattern visto nella MIMD a memoria condivisa, uno molto utilizzato è SPMD con la differenza che bisogna gestire lo scambio di messaggi.

Come si fa a gestirlo?

* Si utilizza un protocollo two-side, cioè che sia il mittente, sia il destinatario hanno bisogno di invocare una data funzione per permettere lo scambio;
* Questo messaggio è munito di un header (contenente le informazioni sul mittente e sul destinatario più eventuali tag) e un body, cioè i dati da trasmettere;
* si gestisce la comunicazione, questa è detta locale se avviene tra coppie di processi, è invece detta globale quando invece ne coinvolgono di più.

## Come avvengono le comunicazioni locali?

* Il mittente chiama una funzione send() mentre il destinatario invoca una receive();
* Quest’ultima funzione è bloccante, l’esecuzione ricomincerà solo quando il messaggio è stato ricevuto. Esiste anche una variante di receive() non bloccante;
* La funzione send() può invece essere:
  + sincrona, cioè completata quando il messaggio viene ricevuto;
  + asincrona bloccante, in questo caso l’invio viene gestito da terzi attraverso un buffer del sistema operativo. L’operazione viene completata quando il buffer può essere sovrascritto;
  + asincrona non bloccante: libera immediatamente il mittente quando viene chiamata, qui bisogna stare attenti a non sovrascrivere il buffer.

Gli scambi sono bidirezionali, di conseguenza ogni processo può essere sia mittente, sia destinatario, quindi l’utilizzo di chiamate bloccanti porterebbe a deadlock, come fare?

Si utilizzano operazioni asincrone, ad esempio: considerando il game of life, come si effettua tutto ciò?

Si dividono le celle in porzioni e ognuna viene assegnata a un processo, lo scambio di messaggi avviene per le celle vicine (quelle trasparenti nella foto a destra).

Ogni processo riceve n/(p+2) celle da elaborare, inoltre deve capire qual è il pezzo che deve elaborare e tutti i suoi vicini, alla fine tutti i risultati verranno inviati agli altri processi.

Prima di effettuare l’aggiornamento, ogni processo deve ricevere le informazioni relative ai bordi, come fare?

Si utilizza un processo master che legge i dati e li manda a ogni processi, per farlo ci vogliono p-1 messaggi in tutto.

Dato che l’elaborazione comincia quando tutti i processo hanno tutti i dati che servono, tutto ciò non è un bene dato che con tanti processi si rischia di aspettare troppo tempo.

## Comunicazioni globali

Come far fronte al problema dell’elaborazione? Si utilizza la comunicazione globale!

La comunicazione globale avviene attraverso le seguenti primitive:

* scatter divide il buffer in pezzettini e li invia a differenti processi;
* broadcast invia l’intero buffer a tutti i processi;
* gather è duale alla scatter e dispone un buffer di ricezione avente una data dimensione;
* reduce raccoglie i dati in un buffer e su di essi applica un’operazione specificata.

Le comunicazioni globali sono state implementate tramite comunicazioni locali, ciò permette di evitare di reinventare l’acqua calda ogni volta.

La chiamata scatter è formata da 4 parametri:

* l’id del processo mittente;
* il buffer di invio, questo parametro non è utilizzato per tutti i task diversi da quello mittente;
* dimensione dei dati in byte;
* il buffer di ricezione, non utilizzato dal mittente.

La chiamata reduce ha anch’essa quattro parametri:

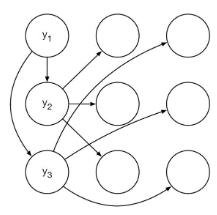
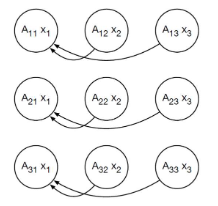
* l’id del processo che raccoglie;
* il buffer di ricezione;
* l’id del mittente;
* la dimensione dei dati in byte.

Broadcast utilizza gli stessi parametri della scatter, la differenza è che viene invocata solamente da un processo. La chiamata gather invece è complementare alla broadcast.

Tornando al precedente esempio,l’invio dei dati avviene con scatter mentre con gather si raccolgono i dati, precisamente:

* il processo 0 legge i dati;
* tutti i processi invocano la scatter con gli stessi parametri, qui 0 è il mittente e distribuisce la griglia a tutti;
* tutti gli altri processi diversi da 0 chiamano la gather per raccogliere i dati.

## Esempio: prodotto matrice-vettore

Con questo esempio, si utilizza una decomposizione bidimensionale, infatti data una matrice A n\*n e un vettore x lungo n, si hanno q^2 task in tutto, ognuno con una porzione della matrice grande n/q \* n/q e una porzione del vettore grande n/q.

Il risultato finale si ottiene sommando per riduzione su valori parziali sui processi della stessa riga, prima però devono inviarsi tutti i dati.

Il risultato finale viene calcolato attraverso una somma per riduzione e verrà infine distribuito in broadcast.

Ogni processo deve cercare di capire la sua porzione di matrice ed array calcolando q come radice di p e il numero della colonna che deve gestire, ottenuto facendo n/q.

## In generale

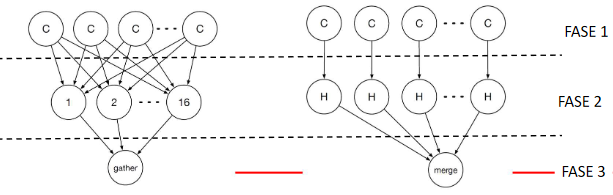
La memoria distribuita è più complicata da programmare, a fronte di ciò si ha un’efficienza più alta dal momento che scala meglio, infatti l’aggiornamento di una struttura condivisa richiede arbitraggio.

La raccolta e la distribuzione devono essere fatte minimizzando il più possibile, proprio per questo motivo si utilizza solamente con gruppi di processi.

Queste due fasi devono essere esplicite dal momento che non sono fatte in automatico.

# Valutazione della decomposizione

Per indicare quanto è buona una decomposizione, si effettua l’analisi work-depth, cioè si analizzano il lavoro e la profondità del task graph utilizzato.

Considerando il problema del calcolo dell’istogramma, quale delle due decomposizione considerate è migliore?

Considerando la prima fase, nel caso sequenziale ogni documento estrae le informazioni, il costo sarà quindi il seguente:



Nell formula, Li è la lunghezza del documento i mentre c1 e c2 sono due costanti.

Se m è la lunghezza del messaggio più lungo, la complessità sarà quindi O(m\*n).

Nel caso parallelo, tutto il lavoro viene fatto in contemporanea su ogni documento, quindi la velocità dell’algoritmo di ilpende dalla lunghezza del documento più lungo:



Confrontando i due risultati, si conclude che il caso parallelo è n volte più veloce di quello sequenziale.

Ora si considera la prima decomposizione, qui il tempo sequenziale è O(n\*m+16\*n+16) dato che ogni task nella seconda fase elabora tutti gli input della fase 1 e il gather fa lo stesso ma coi 16 input della precedente fase.

Il tempo parallelo invece è O(m+n), la fase 1 infatti è O(m) e ci vuole O(n) per elaborare la fase 2, la terza fase è invece indipendente da n.

Il rapporto totale è quindi O(n\*m/(n+m)).

Nella seconda decomposizione, il tempo sequenziale è O(n\*m+n+n) per i seguenti motivi:

* ogni task della seconda fase elabora un solo input di quella precedente;
* il task merge tiene conto di tutti gli output della fase 2.

L’esecuzione parallela ha complessità O(n+m) se si effettua una riduzione sequenziale nel merge, infatti fase 1 e 2 vengono fatte in parallelo e terminano quando l'ultima coppia di task elabora lo stesso messaggio (ciò spiega l’O(m) in più).

Il tempo complessivo della seconda decomposizione è quindi uguale al precedente, cioè O(n\*m/(n+m)).

## Concetti base

* La depth indica la misura del tempo di esecuzione parallelo tenendo conto di ogni cammino e dei tempi di esecuzione. In poche parole, le depth indica il tempo di esecuzione del cammino più lungo;
* Il work indica la somma dei tempi di esecuzione sequenziali, si dice infatti che un algoritmo parallelo è work-efficient se ha la stessa complessità della variante sequenziale;
* Il parallelismo è il rapporto work/depth.

## Cosa succede se si implementa la fase di merge in parallelo?

Applicando una riduzione parallela alla fase di merge, si ha una depth di O(log2(n))) e un work di O(n) con un albero di 2n-1 nodi, qui l’algoritmo è work efficient dal momento che la complessità sequenziale è O(n).

Il parallelismo è quindi O(n / log2(n)).

Applicando tutto ciò alla seconda decomposizione, si ha che:

* la depth è O(m+log2(n));
* il parallelismo p O(n\*m/(m+log2(n))).

Ciò significa che questa decomposizione è preferibile alla prima dato che quest’ultima ha un parallelismo inferiore.

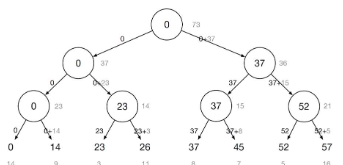
Inoltre con m << n si ha che la prima decomposizione ha complessità O(1) mentre la seconda è O(n/log2(n)).

Tutto questo vale in generale? No!

## Esempio: confronto tra l'algoritmo di Blelloch e quello di Hillis-Steele per il parallel scan

Il parallel scan fatto con l’algoritmo di Hillis-Steele ha come task graph la foto a destra, quanto è profondo e quanto task ci sono a ogni livello?

Ebbene, la sua profondità è O(log2(n)) mentre il numero di nodi in ogni livello è variabile: al livello k ci sono n-2^k nodi, quindi è O(n).

Il work è il numero di livelli moltiplicato per il numero di nodi, quindi è O(n\*log2(n)), ciò significa che l’algoritmo non è work-efficient dato che il miglior algoritmo sequenziale ha complessità O(n).

Il grado di parallelismo di questo algoritmo è O(n\*log2(n)/log2(n)) = O(n).

Per quanto riguarda l’algoritmo di Blelloch, il task graph è ad albero e viene visitato 2 volte, la complessità è quindi O(n) e ciò lo rende work-efficient. La sua depth però è O(log2(n)), il che rende il grado di parallelismo pari a O(n/log2(n)).

Quale dei due algoritmo è migliore? Non c’è una vera e propria risposta esatta dato che anche il contesto e il problema contano.

Per considerare però una scelta migliore in generale, l’analisi work-depth non basta, quindi si tiene conto delle caratteristiche del sistema parallelo.

## Analisi delle performance

Considerando la paralleli reduction, il suo work è O(n), quindi allocando p=n/2 processori, al primo passo verranno utilizzati mentre dal secondo in poi diminuiranno ogni volta di metà.

Complessivamente il costo sarà O(n\*log2(n)), di conseguenza non è work-efficient dato che il miglior algoritmo sequenziale ha complessità O(n).

Da ciò si può dire che il suo speedup è di O(n(log2(n)).

Lo speedup come definito in precedenza è quello assoluto, cioè equivale al miglior tempo sequenziale, ciò però non conviene è quindi si utilizza lo speedup relativo: il tempo di un programma parallelo che gira su un processore.

Da un algoritmo work-efficient è possibile ottenere uno speedup più alto? Supponendo ciò, si modifica l’algoritmo nel seguente modo:

* al primo passo si dividono gli n input tra i p processori, ognuno avrà n/p valori che verranno sommati tra loro;
* dopo si utilizza l’algoritmo di riduzione.

Ciò permette di ottenere le seguenti statistiche:

Come tempo di esecuzione parallelo



Con n >> p, il costo diventa lineare dato che si uniforma la complessità, l’algoritmo è work-efficient. Inoltre il tempo parallelo diventa O(n/p), rendendo in questo modo lo startup lineare.

Da ciò si può dire che un algoritmo è work-efficient se la sua efficienza è pari a 1, indipendentemente dal numero di processori.

Quando non si può sapere l’efficienza di un algoritmo, si utilizza il throughput come metrica, ciò è utile nei casi in cui non esiste una versione sequenziale dell’algoritmo, come nel caso delle GPU. La misura per throughput utilizza i FLOPS come unità di misura, ciò il numero di operazioni in virgola mobile effettuate in un secondo.

Riconsiderano il precedente confronto:

| per Hillis-Steele si utilizza un sistema SIMD con registri grandi n, quindi il tempo parallelo è log2(n) e il suo startup parallelo è O(n/log2(n)).  Questo algoritmo è work-inefficient!  Considerando un numero p di processori minore di n, ogni passo esegue n/p addizione, quindi il tempo parallelo per p è n\*log2(n)/p mentre lo startup è n\*log2(n).  L’algoritmo rimane work-inefficient. | per Blelloch si utilizza un numero p di processori pari a n/2, quindi T(n) è 2\*log2(n) e di conseguenza S(n) è O(n/log2(n)).  Anche questo algoritmo è work-inefficient!  Con questo algoritmo verranno eseguite al massimo p addizioni in parallelo, quindi con n>p bisognerà serializzarle come segue:   * al primo passo si serializzano n/2/p addizioni; * al secondo passo si serializzano n/4/p addizioni e così via fino a quando non si raggiungono p task, dopo si procederà in parallelo.   Da ciò si conclude che T(p) equivale a O((n/p)\*log2(n)), facendo rimanere work-inefficient l'algoritmo. |
| --- | --- |

Dato che entrambi gli algoritmi danno gli stessi risultati, bisogna cambiare logica.

## Esempio: scan inclusivo su un vettore lungo n

Dato un vettore lungo n:

* ogni processore si prende n/p valore e su di essi effettua lo scan;
* si effettua la riduzione utilizzando Hillis-Steele oppure Blelloch.

Il tempo parallelo è O((n/p)+log2(p)) dato che si sfrutta uno dei due algoritmi citati prima, il costo è quindi O(n + p\*log2(p)).

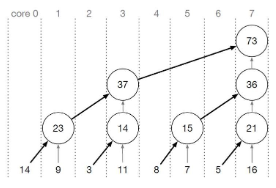
Con n>>p si ha un costo lineare, l’algoritmo è quindi work-efficient.

Inoltre per lo stesso motivo si ha che il tempo parallelo è O(n/p) mentre lo startup è lineare.

## Come limitare la comunicazione

La comunicazione è un fattore che limita lo speedup, come si può far fronte a ciò?

Modellando il tempo di invio di un messaggio come dove λ è la latenza in secondi causata dall’overhead, m è la lunghezza del messaggio e β il throughput del canale, si può caratterizzare il costo della comunicazione di un algoritmo parallelo.

Precisamente:

* al passo i, vengono effettuate comunicazioni, denotate nella foto a destra dalle frecce scure;
* Tutte le comunicazioni in ogni passo vengono effettuate in parallelo, lo smistamento richiede tempo log2P più il tempo per gestirli;
* la comunicazione si calcola quindi con la seguente formula:



* Quindi, il tempo complessivo è:



## Esempio: comunicazione nel game of life

Utilizzando l’agglomerazione per riga, ogni processore invia due messaggi ai suoi vicini, per farlo ci mette tempo 2\*tcom. In tutto il tempo di comunicazione per questa agglomerazione è il seguente:



Utilizzando l’agglomerazione bidimensionale, a ogni passo ogni processore comunica un valore a 4 processori e ai restanti 4, quindi si considera quest’ultimo per il calcolo del tempo complessivo dato che il primo è trascurabile:



Al crescere di n, il calcolo cresce linearmente mentre la comunicazione lo fa come radice, si conclude che questa agglomerazione è migliore della prima.