PERFORMANCE

L’ottimizzazione del programma è un fattore importante soprattutto nel caso in cui lo si voglia parallelizzare. Una buona parallelizzazione di un programma si basa sull’andare a concentrarsi su quelle aree in cui il lavoro svolto dai threads è considerevole. Tali aree vengono chiamate “hotspots” (aree calde).

# DATA DEPENDENCY

Il **Data Dependency** è uno dei maggiori responsabili di un’errata parallelizzazione e riguarda quelle aree di codice dove conta l’ordine di esecuzione delle istruzioni in quanto questo influirebbe sul risultato finale.

# Task-Dependency Graph

Il **Dependency Graph** è usato per indicare quali task hanno bisogno del risultato della computazione di altri task (dipendenza). E’ formato da:

* Nodi
  + Rappresentano task
* Archi
  + Rappresentano le dipendenze tra task

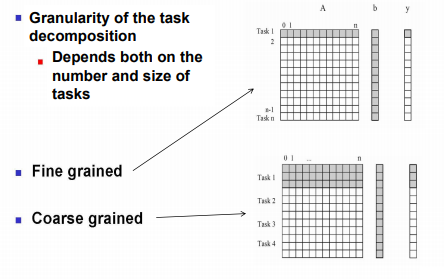
# Task Interaction Graph

Descrive l’interazione tra attività (ovvero quando un task condivide dati con altri task); anche tale fattore, così come la dipendenza è limitante per un sistema parallelo in quanto provoca un decremento prestazionale.

**CI POTREBBE ESSERE INTERAZIONE TRA TASKS ANCHE SE NON VI SONO DIPENDENZE.**

# Granularità

Dai grafici di interazione e dipendenza si può decidere il livello di granularità. La **Granularità** è un indice che va ad indicare il rapporto tra lavoro e comunicazione.

Può essere:

* **Fine**
  + Più task
  + Meno Lavoro
  + Più comunicazione
* **Grossa**
  + Meno task
  + Più lavoro
  + Meno comunicazione

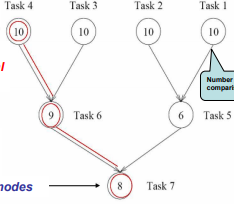
Esiste un limite su quanto possa essere fine la granularità, poiché, dopo un certo limite, i task che hanno bisogno di comunicare sarebbero troppi e questo porterebbe ad un overhead significativo.

# Concorrenza

E’ collegata alla granularità e al Graph Dependency: più è fine e più il grado di concorrenza tra attività aumenta. Indica quante istruzioni si possono eseguire simultaneamente. Un buon programma parallelo dovrebbe garantire un buon numero di istruzioni eseguibili contemporaneamente. In caso contrario si andrebbe contro al concetto di “parallelismo”.

* **Massimo grado di concorrenza**
  + Massimo numero di task che possono essere eseguite simultaneamente.
* **Grado medio di concorrenza**
  + Numero medio di task che possono essere eseguite simultaneamente.
* **Grado minimo di concorrenza**
  + Numero minimo di task che possono essere eseguite simultaneamente.

# Percorso Critico

Il **Percorso Critico** è il percorso più lungo tra ciascuna coppia di nodi finali e iniziali nel **Dependency Graph.** Serve a calcolare il grado medio di concorrenza data un valore di granularità. Supponiamo che in un grafo ogni nodo possieda la quantità di lavoro del task associato.

***In poche parole, esso fornisce il grado massimo di dipendenza fra i task di un dato algoritmo***.

**Per migliorare l’efficienza di un programma parallelo è necessario trovare il giusto compromesso fra decomposizione (= scelta numero task) ed overhead dovuto alle relative comunicazioni.**

# PROCESSORS AND MAPPING

La **Mappatura** serve ad associare un task ai corrispettivi processi fisici. Il **Task Interaction** e il **Task Dependency Graph** sono usati per effettuare una buona mappatura. Anche se non è compito del programmatore mappare i task, sarebbe molto lecito capire ciò che avviene dietro le quinte.

* Statico
  + Conosco a priori il grafo delle interazioni e delle dipendenze
  + La mappatura avviene prima dell’inizio dell’esecuzione del programma
* Dinamico
  + Si assegna un task ad un core quando ha finito il proprio lavoro
  + La mappatura avviene dopo dell’inizio dell’esecuzione del programma, **a un-time.**

Una buona mappatura determina il:

* **LOAD BALANCING**
  + Bilanciamento del lavoro fra i vari task per diminuire lo *starvation* (= tempo di attesa);
* **LOAD UNBALANCING**
  + Sbilanciamento del sistema
  + Una delle maggiori cause di overhead.

Tante volte è possibile che diminuendo il numero di processori si ottengano prestazioni migliori.

L’uso di meno processori rispetto a quanti se ne ha a disposizione viene chiamato ***scaling-down*** (o *ridimensionamento*):

* Scaling by emulation: costruisce un programma per un certo numero di processori e poi effettivamente usarne di meno per simularne un numero maggiore. Il guadagno effettivo c’è se già il sistema è cost-optimal, altrimenti no in quanto i processi virtuali vengono mappati sullo stesso processore: proprio per questo non ci sarebbe bisogno di comunicazione essendo sullo stesso processore bensì questo tipo di mappatura simula anche le comunicazioni non necessarie.
* Scaling intelligente: non simula le comunicazioni fra processi virtuali mappati sullo stesso processore fisico, bensì simula solo le comunicazioni necessarie.

# SpeedUp super lineare

Si parla di SpeedUp super-lineare quando un singolo processore eseguirà il programma più velocemente della versione seriale (ma non si verifica quasi mai in quanto per il calcolo dello SpeedUp è opportuno considerare il miglior algoritmo seriale possibile).

# SCALABILITA’

* Capacità di un sistema parallelo di *incrementare le proprie prestazioni in relazione allo Speed-up e al numero di risorse aggiunte.* ***(È un fattore che caratterizza SOLO alcuni sistemi paralleli: non è un valore!)****.*
* Con questo concetto si riesce, a volte, a superare la legge di Amdahl in quanto, quest’ultimo non considera l’aumento delle dimensioni del problema, bensì le considera costanti.
* In un sistema scalabile:
  + Efficienza costante
    - Costante processori
    - Costante size
  + Efficienza diminuisce
    - = size
    - + processori
  + Efficienza aumenta
    - + size
    - = processori

C:\Users\Utente\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Immagine.png

# Costo

* Il costo di risoluzione di un problema in un sistema parallelo è definito come il prodotto del tempo di esecuzione e il numero di processori impiegati.

# **Efficienza** (p=processori)

* Grado di miglioramento del sistema parallelo. Da informazioni su quanto lavora un processore durante l’esecuzione di un algoritmo parallelo. Minore è il tempo di inattività dei processori (starvation) e maggiore è l’efficienza.
* Il rapporto tra lo SpeedUp ottenuto e il numero di processori impiegati.

*In alcuni sistemi noi possiamo aumentare simultaneamente le dimensioni del problema e il numero di processori tenendo costante l’efficienza; tale sistema viene chiamato* ***scalable parallel systems.***

**In questi casi “l’aumento delle dimensioni del problema” sarebbe l’aumento del tempo sequenziale.**

# Isoefficienza

Abbiamo visto come con il concetto di scalabilità si può aggirare la legge di Amdahl. Sappiamo inoltre che il tempo sequenziale di un programma rappresenta la dimensione del problema stesso.

La funzione che descrive l’isoefficienza calcola la dimensione del problema (ovvero il tempo seriale) in funzione del numero dei processi **in modo da mantenere costante l’efficienza** e quindi, anche la scalabilità.

*Praticamente ci dice di quanto deve essere incrementata la dimensione del problema quando viene incrementato anche il numero di processi.*

***È opportuno ricordare che non per tutti i sistemi scalabili si riesce a trovare un equazione che soddisfi la condizione di isoefficienza***.

Un sistema è cost-optimal quando l’efficienza è uguale a 1.