Programmazione Parallela

Corso tenuto dal Professor Nicola Bombieri

Università di Verona

Alessio Gjergji

Indice

Capitolo 1

Introduzione

1.1 Cos'è la programmazione parallela?

Tradizionalmente, il software è stato scritto per delle computazioni sequenziali. Questo significa che le istruzioni sono eseguite una dopo l'altra, in un ordine ben definito. Le applicazioni, quindi, venivano eseguite su un singolo computer con una singola unità centrale di elaborazione (`CPU`).

Nel senso più elementare, il calcolo parallelo consiste nell'utilizzo simultaneo di diverse risorse di elaborazione per affrontare un problema computazionale. Questo processo prevede:

- **L'impiego di più unità di elaborazione** (`CPU`): per distribuire l'esecuzione del compito su diversi processori, accelerando così il tempo di elaborazione.
- **La divisione del problema in parti discrete**: ogni problema viene scomposto in segmenti più piccoli che possono essere processati in parallelo, ovvero contemporaneamente, su diverse `CPU`.
- **La suddivisione di ogni parte in una serie di istruzioni**: ciascuna frazione del problema viene poi ulteriormente frammentata in istruzioni specifiche, che definiscono esattamente cosa deve essere fatto.
- **L'esecuzione simultanea delle istruzioni su differenti** `CPU`: le istruzioni appartenenti a segmenti differenti del problema vengono eseguite nello stesso momento ma su processori distinti, permettendo così una soluzione più rapida del problema complessivo.

Questo approccio sfrutta al massimo le capacità delle moderne architetture informatiche, permettendo di risolvere problemi complessi in tempi significativamente ridotti rispetto al calcolo sequenziale, dove le istruzioni vengono eseguite una dopo l'altra su un'unica `CPU`.

1.2 Perché la programmazione parallela?

La **computazione parallela** sfrutta l'uso simultaneo di molteplici risorse di calcolo per risolvere problemi computazionali. Questo approccio offre diversi vantaggi significativi, tra cui:

- **Risparmio di Tempo e Denaro:** La distribuzione di un compito su più CPU può ridurre il tempo di completamento e i costi.
- **Risolvere Problemi Più Grandi:** Alcuni problemi sono troppo grandi o complessi per essere gestiti da un singolo computer.
- **Concorrenza:** Diverse risorse di calcolo permettono di eseguire molteplici operazioni in parallelo.
- **Uso di Risorse Non Locali:** L'accesso a risorse di calcolo su reti geografiche estese o su Internet consente di superare le limitazioni delle risorse locali.

1.2.1 Limiti del Calcolo Seriale

Il calcolo seriale presenta limiti fisici e pratici, tra cui:

- **Velocità di Trasmissione:** I limiti alla velocità di trasmissione dei dati impongono un tetto alle prestazioni dei computer seriali.
- **Limiti alla Miniaturizzazione:** Esiste un limite fisico a quanto possano essere piccoli i componenti di un processore.
- **Limitazioni Economiche:** Aumentare la velocità di un singolo processore è progressivamente più costoso.
- **Consumo Energetico:** I core paralleli tendono a consumare meno energia rispetto a un equivalente core sequenziale.

1.2.2 Tendenze e Futuro del Calcolo Parallelo

L'evoluzione delle architetture informatiche evidenzia un crescente affidamento sul parallelismo hardware, attraverso:

- Unità di esecuzione multiple
- Istruzioni in pipeline
- Processori Multi-core e Many-core

Queste tendenze confermano che il futuro del calcolo è orientato verso il parallelismo.

1.3 Concetti di base e terminologia

1.3.1 Tipologie di Computer e Sistemi

Ci sono diversi tipi di computer e sistemi che possono essere utilizzati per eseguire applicazioni parallele, tra cui:

Computer Desktop

I computer desktop sono sistemi personali comunemente usati in ambienti domestici e uffici per svariate applicazioni, da quelle produttive a quelle di intrattenimento.

Computer Embedded

I computer embedded sono sistemi specializzati progettati per eseguire compiti specifici all'interno di dispositivi più grandi, come automobili, elettrodomestici e sistemi di controllo industriale.

Internet of Things (`IoT`)

- **Computer Embedded Collegati a Internet:** Dispositivi embedded che sono connessi a Internet per fornire funzionalità avanzate, come il monitoraggio remoto e il controllo.
- **Sistemi Smart:** L'`IoT` abilita la creazione di sistemi intelligenti che combinano sensori, attuatori e connettività per interagire con il mondo fisico in modi avanzati.

Dispositivi Mobili Personali (`PMDs`)

Include smartphone, tablet e altri dispositivi portatili che forniscono una vasta gamma di funzionalità, dalla comunicazione all'accesso a Internet e applicazioni specializzate.

Server

Potenti computer progettati per gestire richieste di dati e servizi da parte di altri computer e dispositivi all'interno di reti aziendali e su Internet.

Cluster e Computer su Scala di Magazzino

- **Cluster:** Insiemi di computer connessi che lavorano insieme come un'unica entità per fornire elevate prestazioni di calcolo e disponibilità.
- **Computer su Scala di Magazzino:** Grandi infrastrutture informatiche che supportano applicazioni di cloud computing e servizi Internet su larga scala.
- **Supercomputer vs. Cluster:** Mentre i supercomputer sono sistemi altamente specializzati per compiti di calcolo intensivo, i cluster rappresentano un approccio più scalabile e flessibile al calcolo ad alte prestazioni.

1.3.2 La Struttura von Neumann

La struttura von Neumann, che prende il nome dal matematico ungherese John von Neumann, rappresenta il modello di base seguito dalla maggior parte dei computer moderni. Questo modello fu descritto per la prima volta nei documenti del 1945, evidenziando i requisiti generali per un computer elettronico. A differenza dei primi computer, programmati attraverso un cablaggio fisso, la struttura von Neumann introduce un design flessibile e potente.

La struttura è composta da quattro componenti principali:

1. **Memoria:** Serve per memorizzare le istruzioni del programma e i dati. Le istruzioni sono codificate per dire al computer cosa fare, mentre i dati sono le informazioni elaborate dal programma.
2. **Unità di Controllo:** Preleva le istruzioni e i dati dalla memoria, decodifica le istruzioni e coordina le operazioni per eseguire il compito programmato.
3. **Unità Logica Aritmetica (`ALU`):** Esegue le operazioni aritmetiche e logiche di base.
4. **Input/Output:** Funge da interfaccia tra il computer e l'utente, permettendo l'ingresso e l'uscita dei dati.

La memoria a accesso casuale (`RAM`), che permette sia la lettura che la scrittura, è fondamentale in questa architettura per la memorizzazione sia delle istruzioni che dei dati necessari per l'esecuzione del programma.

1.3.3 Tassonomia di Flynn

La tassonomia di Flynn è un sistema di classificazione per le architetture dei computer multiprocessore, basato sul numero di flussi di istruzioni e dati che possono gestire in parallelo. Utilizza due dimensioni: Istruzione e Dati, ognuna delle quali può essere Singola o Multipla. Questo porta a quattro possibili classificazioni nella tassonomia di Flynn, che forniscono un quadro di riferimento per comprendere le diverse modalità di calcolo parallelo.

1.4 Tassonomia di Flynn

La tassonomia di Flynn classifica le architetture di calcolo parallelo basandosi su due dimensioni: il numero di flussi di istruzioni e il numero di flussi di dati che il sistema può gestire. Ogni dimensione può essere Singola o Multipla, portando a quattro categorie principali:

- `SISD` **(Single Instruction, Single Data)**: un processore esegue un flusso di istruzioni su un flusso di dati.
- `SIMD` **(Single Instruction, Multiple Data)**: un'istruzione controlla simultaneamente più operazioni su diversi flussi di dati.
- `MISD` **(Multiple Instruction, Single Data)**: più istruzioni operano su un singolo flusso di dati, utilizzato raramente.

- `MIMD` **(Multiple Instruction, Multiple Data)**: più processori eseguono istruzioni diverse su flussi di dati diversi, comunemente usato per applicazioni parallele general-purpose.

In un sistema di elaborazione, l'esecuzione di programmi e la gestione dei dati sono basate su cinque componenti chiave, che insieme formano il cuore funzionale di qualsiasi computer moderno:

- `IS` (Instruction Stream): il flusso di istruzioni, ovvero le operazioni che il sistema deve eseguire, organizzate in sequenza.
- `DS` (Data Stream): il flusso di dati comprende gli operandi sui quali operano le istruzioni e i risultati di tali operazioni.
- `CU` (Control Unit): l'unità di controllo, che si occupa di prelevare le istruzioni dalla memoria, decodificarle e coordinare l'esecuzione.
- `PU` (Processing Unit): l'unità di elaborazione, costituita dall'`ALU` (*Arithmetic Logic Unit*) e dai registri, esegue le istruzioni operative.
- `MM` (Main Memory): la memoria principale, dove vengono allocati i dati e le istruzioni necessari per l'esecuzione di un programma.

Questi componenti interagiscono tra loro per processare efficacemente i dati e le istruzioni, permettendo al sistema di eseguire una vasta gamma di compiti.

1.4.1 Single instruction, single data - `SISD`

Nella struttura di Von Neumann, l'Unità di Controllo (`CU`) ha il compito di prelevare le istruzioni dalla Memoria Principale (`MM`), mentre l'Unità di Elaborazione (`PU`) esegue tali istruzioni interagendo con la `MM` per modificare i dati. Questo schema rappresenta il funzionamento base della struttura di Von Neumann, in cui un singolo programma è in esecuzione e si basa su un unico flusso di dati.

La `CU` coordina il processo di esecuzione leggendo sequenzialmente le istruzioni dal programma memorizzato nella `MM`, decodificandole e trasferendole alla `PU` per la loro esecuzione. La `PU`, a sua volta, esegue le operazioni aritmetiche e logiche specificate dalle istruzioni, utilizzando i dati memorizzati nella `MM`. Questo processo iterativo tra `PU`, `PU` e `MM` permette l'elaborazione dei programmi secondo il modello di flusso di dati e di controllo definito dalla struttura di Von Neumann.

Un computer seriale (*non parallelo*) si caratterizza per il singolo flusso di istruzioni e dati. Durante ogni ciclo di clock, la `CPU` elabora:

- **Singola istruzione:** Viene processato solo un flusso di istruzioni.
- **Singolo dato:** Viene utilizzato come input un solo flusso di dati.

Questo comporta un'esecuzione deterministica, in cui il risultato del calcolo è direttamente determinato dall'algoritmo e dai dati in ingresso. Esempi comuni di computer seriali includono mainframe di vecchia generazione, minicomputer e workstation.

1.4.2 Single instruction, multiple data - SIMD

Un tipo di computer parallelo conosciuto come SIMD (*Single Instruction, Multiple Data*) possiede le seguenti caratteristiche:

- **Singola istruzione:** Tutte le unità di elaborazione eseguono la stessa istruzione in qualsiasi ciclo di clock.
- **Dati multipli:** Ogni unità di elaborazione può operare su un elemento di dati diverso.

Questa architettura è particolarmente adatta per problemi specializzati caratterizzati da un alto grado di regolarità, come l'elaborazione di grafica e immagini. L'esecuzione è sincrona (*lockstep*) e deterministica.

La maggior parte dei computer moderni, in particolare quelli dotati di unità di elaborazione grafiche (GPU), impiega istruzioni SIMD e unità di esecuzione.

1.4.3 Multiple instruction, single data - MISD

Un computer parallelo MISD (*Multiple Instruction, Single Data*) è caratterizzato da:

- Un singolo flusso di dati che viene elaborato da più unità di elaborazione.
- Ogni unità di elaborazione processa i dati in modo indipendente attraverso propri flussi di istruzioni.

Questa architettura è raramente utilizzata, poiché è difficile da implementare e non offre vantaggi significativi rispetto ad altre architetture parallele.

1.4.4 Multiple instruction, multiple data - MIMD

Il computer parallelo di tipo MIMD (*Multiple Instruction, Multiple Data*) è attualmente la forma più comune di calcolo parallelo e la maggior parte dei computer moderni rientra in questa categoria. Le caratteristiche distintive sono:

- **Istruzioni Multiple:** ogni processore può eseguire un flusso di istruzioni diverso.
- **Dati Multipli:** ogni processore può lavorare con un proprio flusso di dati.
- L'esecuzione può essere sincrona o asincrona, deterministica o non deterministica.

Esempi di questa architettura includono la maggior parte dei supercomputer attuali, i cluster di computer paralleli connessi in rete e le "griglie" di calcolo, i computer SMP (*Symmetric Multi-Processing*) con più processori e i PC multi-core.

Nota 1.4.1 *Molte architetture MIMD includono anche sottocomponenti di esecuzione SIMD.*

1.5 Concetti di Esecuzione

1.5.1 Task

Un task è un'unità di lavoro computazionale che corrisponde a un programma o a una sequenza di istruzioni eseguite da un processore. Ogni task è progettato per completare una parte specifica del lavoro generale richiesto dal programma completo.

1.5.2 Esecuzione Seriale

L'esecuzione seriale implica il processamento di istruzioni una alla volta, in una sequenza ordinata. Questo metodo di esecuzione è tipico dei computer con un singolo processore e si basa su un modello computazionale che non prevede l'esecuzione contemporanea di più istruzioni o task.

Implicazioni dell'Esecuzione Seriale Questo tipo di esecuzione è caratterizzato da un flusso di lavoro prevedibile e da una facile individuazione e risoluzione degli errori. È particolarmente efficace in applicazioni dove le operazioni devono essere svolte in una sequenza specifica e dove le istruzioni successive dipendono dai risultati di quelle precedenti.

1.5.3 Esecuzione Parallela

Contrariamente all'esecuzione seriale, l'esecuzione parallela permette a più task di essere eseguiti simultaneamente. Questo approccio sfrutta l'architettura dei computer multi-processore per ridurre il tempo totale di elaborazione.

Benefici dell'Esecuzione Parallela L'abilità di eseguire più task contemporaneamente porta a una riduzione significativa del tempo di esecuzione per i problemi che possono essere suddivisi in parti indipendenti, ottimizzando l'uso delle risorse di elaborazione disponibili.

1.5.4 Pipelining

Il pipelining è un'efficace strategia di esecuzione parallela in cui un task è diviso in diverse fasi. Ogni fase è elaborata da una diversa unità di processamento, permettendo un flusso continuo di esecuzione simile a quello di una catena di montaggio industriale.

Efficienza del Pipelining Attraverso il pipelining, diverse fasi di un processo possono essere eseguite simultaneamente, migliorando l'efficienza e la velocità complessive del sistema di elaborazione.

1.6 Memoria nei Sistemi Paralleli

1.6.1 Memoria Condivisa

In architetture con memoria condivisa, tutti i processori accedono a una memoria fisica comune, consentendo una comunicazione e sincronizzazione efficienti tra i task. Tuttavia, que-

sto modello può comportare dei collo di bottiglia dovuti alla competizione per l'accesso alla memoria.

1.6.2 Memoria Distribuita

Nei sistemi con memoria distribuita, ciascun processore accede a una propria memoria locale. Questo approccio migliora la scalabilità del sistema ma richiede meccanismi di comunicazione complessi per coordinare i task distribuiti su diversi processori.

1.7 Comunicazione e Sincronizzazione

1.7.1 Comunicazione

La comunicazione tra i task è fondamentale in un ambiente di calcolo parallelo. I meccanismi di comunicazione variano a seconda dell'architettura e possono includere l'uso di bus di memoria condivisa o reti di comunicazione.

1.7.2 Sincronizzazione

Per mantenere la coerenza e l'ordine nell'esecuzione parallela, i task devono sincronizzarsi periodicamente. Ciò è spesso realizzato attraverso punti di sincronizzazione nel programma, dove ogni task deve attendere gli altri prima di procedere.

1.8 Prestazioni e Scalabilità

1.8.1 Granularità

La granularità nel calcolo parallelo descrive il livello di suddivisione del lavoro computazionale e ha un impatto diretto sull'equilibrio tra calcolo e comunicazione. La granularità fine potrebbe richiedere una comunicazione più frequente, mentre quella grossolana meno frequente.

1.8.2 Speedup e Overhead Parallelo

Lo speedup misura l'efficacia dell'esecuzione parallela rispetto a quella seriale. L'overhead parallelo, che include il tempo di avvio dei task, le sincronizzazioni e le comunicazioni, può influenzare negativamente questo indicatore di prestazione.

$$Speedup = \frac{Tempo\ di\ esecuzione\ seriale}{Tempo\ di\ esecuzione\ parallelo}$$

1.9 Architetture e Computazione Parallela

1.9.1 Processori Multi-core

I processori multi-core contengono più core di elaborazione in un unico chip, permettendo l'esecuzione parallela di task su un singolo dispositivo fisico.

1.9.2 Cluster Computing

Il cluster computing utilizza un insieme di unità di calcolo, spesso commerciali, configurate per lavorare insieme come un unico sistema parallelo.

1.9.3 Supercomputing

Il supercomputing si basa sull'uso di computer ad alte prestazioni per affrontare problemi computazionali di grande scala, dove la velocità e la capacità di elaborazione sono essenziali.

1.9.4 Edge Computing

L'edge computing mira a portare la potenza di calcolo e la memorizzazione dei dati più vicino al punto di necessità, riducendo i tempi di risposta e il consumo di banda.

1.10 Memoria Condivisa

I computer paralleli a memoria condivisa presentano diverse caratteristiche, ma in generale condividono la capacità per tutti i processori di accedere a tutta la memoria come uno spazio di indirizzamento globale.

- I processori multipli possono operare in modo indipendente, ma condividono le stesse risorse di memoria.
- Le modifiche in una posizione di memoria effettuate da un processore sono visibili a tutti gli altri processori.
- Le macchine a memoria condivisa possono essere suddivise in due classi principali in base ai tempi di accesso alla memoria: `UMA` (*Uniform Memory Access*) e `NUMA` (*Non-Uniform Memory Access*).

1.10.1 `UMA` (*Uniform Memory Access*)

Le architetture `UMA` sono comunemente rappresentate oggi dalle macchine Symmetric Multiprocessor (`SMP`), caratterizzate da processori identici e accesso uniforme alla memoria con tempi di accesso uguali. Questo modello è noto anche come `CC-UMA` (*Cache Coherent `UMA`*), dove la coerenza della cache indica che se un processore aggiorna una posizione nella memoria condivisa, tutti gli altri processori vengono informati dell'aggiornamento. La coerenza della cache è ottenuta a livello hardware.

Vantaggi e Svantaggi

- **Vantaggi:** Lo spazio di indirizzamento globale offre una prospettiva di programmazione user-friendly per la memoria. La condivisione dei dati tra i task è sia rapida che uniforme grazie alla prossimità della memoria ai CPU.
- **Svantaggi:** Il principale svantaggio è la mancanza di scalabilità tra memoria e CPU. Aggiungere più CPU può aumentare geometricamente il traffico sul percorso memoria-CPU condiviso e, per i sistemi con coerenza della cache, aumentare geometricamente il traffico associato alla gestione della cache/memoria. È responsabilità del programmatore utilizzare costrutti di sincronizzazione che assicurino un accesso "corretto" alla memoria globale. Inoltre, diventa sempre più difficile e costoso progettare e produrre macchine a memoria condivisa con un numero crescente di processori.

1.10.2 NUMA (*Non-Uniform Memory Access*)

Le architetture NUMA sono spesso realizzate collegando fisicamente due o più SMP. Un SMP può accedere direttamente alla memoria di un altro SMP, ma non tutti i processori hanno tempi di accesso uguali a tutte le memorie. L'accesso alla memoria attraverso il collegamento è più lento. Se la coerenza della cache è mantenuta, queste architetture possono anche essere chiamate CC-NUMA (*Cache Coherent NUMA*).

Vantaggi e Svantaggi

- **Vantaggi:** Similmente a UMA, NUMA offre uno spazio di indirizzamento globale che facilita la programmazione e la condivisione dei dati tra i task. La struttura di NUMA permette una migliore scalabilità rispetto a UMA quando si aggiungono processori, grazie alla distribuzione della memoria tra i vari SMP.
- **Svantaggi:** L'accesso non uniforme alla memoria può portare a prestazioni inconsistenti, specialmente in carichi di lavoro che richiedono un accesso frequente alla memoria attraverso i collegamenti SMP. La gestione della coerenza della cache, sebbene fornisca una visione coerente della memoria, può introdurre overhead significativo, specialmente in sistemi di grande dimensione.

1.11 Memoria Distribuita

Come i sistemi a memoria condivisa, anche quelli a memoria distribuita variano notevolmente ma condividono una caratteristica comune: richiedono una rete di comunicazione per connettere la memoria tra i vari processori. In questi sistemi, ogni processore dispone di una propria memoria locale e gli indirizzi di memoria in un processore non sono mappati su un altro processore, eliminando così il concetto di spazio di indirizzamento globale.

1.11.1 Funzionamento Indipendente e Coerenza della Cache

Poiché ogni processore ha la propria memoria locale, opera indipendentemente. Le modifiche che effettua nella sua memoria locale non influenzano la memoria degli altri processori,

rendendo inapplicabile il concetto di coerenza della cache. Quando un processore necessita di accedere ai dati in un altro processore, spesso è compito del programmatore definire esplicitamente come e quando i dati vengono comunicati. Anche la sincronizzazione tra i task è responsabilità del programmatore.

1.11.2 Tessuto di Rete

Il "tessuto" di rete utilizzato per il trasferimento dei dati varia ampiamente, sebbene possa essere semplice come Ethernet.

Vantaggi e svantaggi

- **Vantaggi**
  - La memoria è scalabile con il numero di processori. Aumentando il numero di processori, la dimensione della memoria aumenta proporzionalmente.
  - Ogni processore può accedere rapidamente alla propria memoria senza interferenze e senza l'overhead necessario per mantenere la coerenza della cache.
  - Costo-efficacia: è possibile utilizzare processori e reti commerciali.
- **Svantaggi**
  - Il programmatore è responsabile di molti dettagli associati alla comunicazione dei dati tra i processori.
  - Può essere difficile mappare le strutture dati esistenti, basate sulla memoria globale, a questa organizzazione della memoria.
  - Tempi di accesso alla memoria non uniformi (`NUMA`).

1.12 Memorie ibride, distribuite e condivise

I supercomputer più grandi e veloci al mondo oggi impiegano architetture ibride che combinano elementi di memoria condivisa e memoria distribuita.

1.12.1 Componente di Memoria Condivisa

La componente di memoria condivisa è solitamente costituita da una macchina `SMP` (*Symmetric Multiprocessing*) con coerenza della cache. I processori all'interno di un dato `SMP` possono indirizzare la memoria della macchina come se fosse globale, permettendo un accesso rapido e efficiente ai dati condivisi.

1.12.2 Componente di Memoria Distribuita

La componente di memoria distribuita si realizza tramite il collegamento in rete di più macchine `SMP`. Ogni `SMP` è a conoscenza soltanto della propria memoria e non di quella presente

su un altro SMP. Di conseguenza, sono necessarie comunicazioni di rete per spostare i dati da un SMP all'altro.

Tendenze Attuali Le tendenze attuali sembrano indicare che questo tipo di architettura di memoria continuerà a prevalere e ad espandersi nell'alta fascia del calcolo per il futuro prevedibile. L'approccio ibrido offre il meglio di entrambi i mondi: l'efficienza e la facilità di programmazione della memoria condivisa e la scalabilità e flessibilità della memoria distribuita.

Vantaggi e svantaggi

- **Vantaggi**
  - **Scalabilità:** L'architettura ibrida permette ai supercomputer di scalare efficacemente aggiungendo più SMP, aumentando la potenza di calcolo e la memoria disponibile.
  - **Flessibilità:** Gli sviluppatori possono ottimizzare le prestazioni sfruttando la memoria locale nei nodi SMP per l'accesso ad alta velocità e utilizzare la memoria distribuita per il lavoro collaborativo tra SMP.
  - **Efficienza:** La combinazione di memoria condivisa e distribuita può migliorare l'efficienza complessiva del sistema, bilanciando carico di lavoro e comunicazioni di rete.
- **Svantaggi**
  - **Complessità:** La programmazione e la gestione di architetture ibride sono più complesse a causa della necessità di bilanciare l'uso di memoria condivisa e distribuita.
  - **Costo:** La costruzione e manutenzione di supercomputer con architetture ibride possono essere costose, data la complessità del hardware e del software.
  - **Coerenza dei Dati:** Mantenere la coerenza dei dati tra la memoria condivisa e quella distribuita può richiedere meccanismi di sincronizzazione avanzati, aggiungendo un ulteriore livello di complessità.

Capitolo 2

Modelli di programmazione parallela

2.1 Introduzione

Ci sono diversi modelli di programmazione parallela, ognuno con i propri vantaggi e svantaggi.

- **Modelli di memoria condivisa**: i processi condividono un unico spazio di indirizzamento.
- **Modelli di memoria distribuita**: i processi hanno spazi di indirizzamento separati.

Sebbene possa sembrare, i modelli non sono legati ad una specifica architettura hardware, ma possono essere implementati (*teoricamente*) su qualsiasi architettura.

È importante notare che non c'è un modello migliore rispetto ad un altro, ma dipende dal problema che si vuole risolvere e dalle caratteristiche dell'architettura hardware a disposizione.

2.2 Modelli di memoria condivisa

Nel **modello di programmazione a memoria condivisa**, i task condividono uno spazio di indirizzi comune, che leggono e scrivono in modo asincrono. Esistono vari meccanismi, come i lock o i semafori, utilizzati per controllare l'accesso alla memoria condivisa. Da un punto di vista del programmatore, un vantaggio di questo modello è che manca la nozione di "proprietà" dei dati. Ciò implica che:

- Non è necessario specificare esplicitamente la comunicazione dei dati tra i task.
- Lo sviluppo del programma può spesso essere semplificato.

Tuttavia, un importante svantaggio, in termini di prestazioni, è che diventa più difficile comprendere e gestire la **località dei dati**. Mantenere i dati locali al processore che ci lavora su conserva gli accessi alla memoria, i refresh della cache e il traffico sul bus che si verifica

quando più processori utilizzano gli stessi dati. Sfortunatamente, controllare la località dei dati è difficile da capire ed è al di fuori del controllo dell'utente medio.

2.3 Modello a Thread

Nel modello di programmazione parallela basato sui **thread**, un singolo processo può avere più percorsi di esecuzione concorrenti. Questa modalità permette di eseguire diverse parti di un programma in parallelo, aumentando l'efficienza e riducendo il tempo di esecuzione.

2.3.1 Analogia e Funzionamento

Un'analogia semplice per descrivere i thread è il concetto di un singolo programma che include un numero di subroutine:

- Il programma principale `a.out` viene schedulato per l'esecuzione dal sistema operativo nativo. `a.out` carica e acquisisce tutte le risorse di sistema e utente necessarie per l'esecuzione.
- `a.out` esegue del lavoro seriale e poi crea un numero di task (*thread*) che possono essere schedulati ed eseguiti contemporaneamente dal sistema operativo.
- Ogni thread ha dati locali, ma condivide anche tutte le risorse di `a.out`, risparmiando così l'overhead associato alla replicazione delle risorse del programma per ogni thread. Ogni thread beneficia anche di una visione globale della memoria perché condivide lo spazio di memoria di `a.out`.
- Il lavoro di un thread può essere descritto come una subroutine all'interno del programma principale. Qualsiasi thread può eseguire qualsiasi subroutine allo stesso tempo degli altri thread.
- I thread comunicano tra loro tramite la memoria globale (*aggiornando le posizioni degli indirizzi*). Ciò richiede costrutti di sincronizzazione per assicurare che più di un thread non stia aggiornando lo stesso indirizzo globale contemporaneamente.
- I thread possono essere creati e terminati, ma `a.out` rimane presente per fornire le risorse condivise necessarie fino al completamento dell'applicazione.

2.3.2 Implementazioni e Standardizzazione

I thread sono comunemente associati con architetture di memoria condivisa e sistemi operativi. Dal punto di vista della programmazione, le implementazioni dei thread comprendono comunemente:

- Una libreria di subroutine che vengono chiamate all'interno del codice sorgente parallelo.
- Un insieme di direttive del compilatore integrate nel codice sorgente, sia seriale che parallelo.

In entrambi i casi, il programmatore è responsabile della determinazione di tutto il parallelismo.

Le implementazioni basate su thread non sono una novità nel campo dell'informatica. Storicamente, i fornitori di hardware hanno implementato le loro versioni proprietarie di thread, le quali differivano sostanzialmente l'una dall'altra, rendendo difficile per i programmatori sviluppare applicazioni threaded portabili. Sforzi di standardizzazione non correlati hanno risultato in due implementazioni molto diverse di thread:

- `POSIX Threads`
- `OpenMP`

2.4 Modello Message Passing

Il modello di passaggio di messaggi dimostra le seguenti caratteristiche principali:

- Un insieme di task che utilizzano la propria memoria locale durante il calcolo. Più task possono risiedere sulla stessa macchina fisica così come su un numero arbitrario di macchine.
- I task scambiano dati attraverso la comunicazione inviando e ricevendo messaggi.
- Il trasferimento di dati richiede di solito operazioni cooperative da eseguire da ciascun processo. Ad esempio, un'operazione di invio deve avere un'operazione di ricezione corrispondente.

Dal punto di vista della programmazione, le implementazioni del passaggio di messaggi comprendono comunemente una libreria di subroutine che sono incorporate nel codice sorgente.

Il programmatore è responsabile della determinazione di tutto il parallelismo.

2.5 Modello di Parallelismo dei Dati

Il modello di parallelismo dei dati dimostra le seguenti caratteristiche principali:

- La maggior parte del lavoro parallelo si concentra sull'esecuzione di operazioni su un insieme di dati. L'insieme di dati è tipicamente organizzato in una struttura comune, come un array o un cubo.
- Un insieme di task lavora collettivamente sulla stessa struttura di dati, tuttavia, ogni task lavora su una partizione diversa della stessa struttura di dati.
- I task eseguono la stessa operazione sulla loro partizione di lavoro, per esempio, "aggiungere 4 a ogni elemento dell'array".

Sulle architetture a memoria condivisa, tutti i task possono avere accesso alla struttura di dati tramite la memoria globale. Sulle architetture a memoria distribuita, la struttura di dati è suddivisa e risiede come "chunk" nella memoria locale di ogni task.

2.5.1 Programmazione con il Modello di Parallelismo dei Dati

La programmazione con il modello di parallelismo dei dati si realizza solitamente scrivendo un programma con costrutti di parallelismo dei dati. I costrutti possono essere chiamate a una libreria di subroutine di parallelismo dei dati o direttive del compilatore riconosciute da un compilatore di parallelismo dei dati.

Direttive del Compilatore

Permettono al programmatore di specificare la distribuzione e l'allineamento dei dati. Le implementazioni Fortran sono disponibili per le piattaforme parallele più comuni.

Implementazioni a Memoria Distribuita

Le implementazioni di questo modello su memoria distribuita di solito hanno il compilatore che converte il programma in codice standard con chiamate a una libreria di passaggio di messaggi (*solitamente* `MPI`) per distribuire i dati a tutti i processi. Tutto il passaggio di messaggi è invisibile al programmatore.

2.6 Altri modelli di programmazione parallela

Oltre ai modelli di programmazione parallela precedentemente menzionati, esistono certamente altri modelli, che continueranno a evolversi insieme al mondo sempre in cambiamento dell'hardware e del software per computer. Qui ne vengono menzionati solamente tre tra i più comuni: ibrido, `SPMD` (*Single Program Multiple Data*) e `MPMD` (*Multiple Program Multiple Data*).

2.6.1 Modello Ibrido

In questo modello, vengono combinati due o più modelli di programmazione parallela. Esempi comuni di modello ibrido includono:

- La combinazione del modello di passaggio di messaggi (`MPI`) con il modello dei thread (*`POSIX` threads*) o il modello di memoria condivisa (`OpenMP`). Questo modello ibrido si presta bene all'ambiente hardware sempre più comune di macchine SMP in rete.
- La combinazione del parallelismo dei dati con il passaggio di messaggi. Come menzionato nella sezione relativa al modello di parallelismo dei dati, le implementazioni di parallelismo dei dati su architetture a memoria distribuita utilizzano effettivamente il passaggio di messaggi per trasmettere dati tra i task, in modo trasparente per il programmatore.

2.6.2 Single Program Multiple Data (`SPMD`)

- `SPMD` è un modello di programmazione "di alto livello" che può essere costruito su qualsiasi combinazione dei modelli di programmazione parallela precedentemente menzionati.
- Un singolo programma viene eseguito simultaneamente da tutti i task.

- In un dato momento, i task possono eseguire le stesse o diverse istruzioni all'interno dello stesso programma.
- A differenza di `SIMD`, in `SPMD`, processori autonomi eseguono simultaneamente lo stesso programma in punti indipendenti, anziché in modo sincronizzato come impone `SIMD` su dati diversi.
- I programmi `SPMD` di solito hanno la logica necessaria programmata per permettere ai diversi task di eseguire condizionalmente solo quelle parti del programma che sono progettati per eseguire.

2.6.3 Multiple Program Multiple Data (`MPMD`)

- Come `SPMD`, anche `MPMD` è un modello di programmazione "di alto livello" che può essere basato su qualsiasi combinazione dei modelli di programmazione parallela menzionati.
- Le applicazioni `MPMD` tipicamente hanno più file oggetto eseguibili (*programmi*). Mentre l'applicazione viene eseguita in parallelo, ciascun task può eseguire lo stesso programma o programmi diversi rispetto agli altri task.

Capitolo 3

Misurazione delle Performance

3.1 Introduzione

Ci sono due punti di vista nella misurazione delle performance:

- **Utente**: tempo di risposta, throughput, tempo di completamento.
- **Sviluppatore**: tempo di esecuzione, tempo di `CPU`, tempo di `I/O`, tempo di comunicazione.

Il **tempo di risposta** è il tempo che intercorre tra la partenza del codice e la fine dell'esecuzione. Il **throughput** è la quantità di lavoro fatto dal sistema in un determinato periodo di tempo (*si considera l'ammontare di lavoro*). La **latenza** di un'istruzione, nel caso della pipeline, è il tempo che intercorre tra l'arrivo di un'istruzione e la sua completa esecuzione.

L'affermazione "X è più veloce di Y" nel contesto della misurazione delle prestazioni significa che il tempo di risposta o il tempo di esecuzione per un determinato compito è inferiore in X rispetto a Y. Questo si riferisce direttamente all'efficienza con cui X completa un compito rispetto a Y.

Quando diciamo che "X è n volte più veloce di Y", stiamo esprimendo un rapporto di velocità tra X e Y. Formalmente, questo è descritto dalla formula:

$$n = \frac{\texttt{Tempo di esecuzione di } Y}{\texttt{Tempo di esecuzione di } X}$$

Questo rapporto, noto come *speedup*, indica quante volte X è più veloce di Y. In termini di prestazioni, il tempo di esecuzione è inversamente proporzionale alla performance. Pertanto, possiamo riformulare il *speedup* utilizzando le prestazioni relative di X e Y:

$$n = \frac{\texttt{Tempo di esecuzione di } Y}{\texttt{Tempo di esecuzione di } X} = \frac{\frac{1}{\texttt{Prestazioni di } Y}}{\frac{1}{\texttt{Prestazioni di } X}} = \frac{\texttt{Prestazioni di } X}{\texttt{Prestazioni di } Y}$$

Questa equazione ci mostra che il *speedup* è il rapporto tra le prestazioni di X e quelle di Y, offrendo un modo quantitativo per valutare e confrontare l'efficienza di due sistemi o componenti software.

3.2 Benchmark

Quando misuriamo le performance di un sistema, dobbiamo capire come misurarle in termini di architettura. Un **benchmark** è un programma che misura le performance di un sistema. I benchmark possono essere:

- **Sintetici**: programmi che eseguono operazioni tipiche di un'applicazione e stimolano quindi certi comportamenti dell'architettura.
- **Kernel**: frammenti di codice che rappresentano operazioni fondamentali e critiche per la performance di sistemi computazionali.
- **Toy**: programmi molto semplici che eseguono operazioni elementari.
- **Suits**: insiemi di benchmark che permettono una valutazione complessiva delle performance di un sistema.

Sul piatto della bilancia ci sono le **performance** e il **costo**. Generalmente, si esegue un benchmark più volte e si calcola la media dei risultati ottenuti per ottenere una misura affidabile delle performance.

3.3 Principi Quantitativi

La prima cosa da fare per migliorare le performance di un sistema è capire come parallelizzare un codice sequenziale. Innanzitutto, si inizia analizzando e ottimizzando la parte di codice che viene eseguita più frequentemente, noto come *hot spot* del codice.

3.3.1 Legge di Amdahl

La legge di Amdahl ci dice che la velocità di un sistemap arallelo è limitata dalla frazione sequenziale del codice. Tale legge ci aiuta a comprendere quanto possiamo migliorare le performance di un sistema. Non ci interessano le righe di codice, ma le funzioni che vengono eseguite più frequentemente. Per identificarle, generalmente vengono utilizzati dei *profiler*.

La formula della legge di Amdahl per il calcolo dello speedup $S(n)$ è:

$$S(n) = \frac{\texttt{ExecutionTime}_{\texttt{old}}}{\texttt{ExecutionTime}_{\texttt{new}}} = \frac{1}{(1 - \texttt{Fraction}_{\texttt{improved}}) + \frac{\texttt{Fraction}_{\texttt{improved}}}{\texttt{Speedup}_{\texttt{improved}}}}$$

Di natura, la legge di Amdahl ci indica che non possiamo migliorare le performance di un sistema in maniera illimitata. Dovremo quindi cercare di parallelizzare solamente le parti che presentano un potenziale parallelismo.

Ne segue che lo speedup globale può essere calcolato come:

$$S(n) = \frac{1}{(1 - \texttt{Fraction}_{\texttt{improved}}) + \frac{\texttt{Fraction}_{\texttt{improved}}}{n}} \leq \frac{1}{(1 - \texttt{Fraction}_{\texttt{improved}})}$$

Ovvero, lo speedup è limitato dalla frazione sequenziale del codice. Con il 50% di codice parallelo, lo speedup massimo è 2.

Esempio

Supponiamo che una parte di un programma possa essere migliorata di 10 volte e che questa parte sia eseguita nel 40% del tempo totale. Lo speedup massimo sarà:

$$S(n) = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = \frac{1}{0.6 + 0.04} = \frac{1}{0.64} = 1.56$$

3.4 Il tempo è un'unità di misura

Il tempo di esecuzione di un programma è una misura delle performance. Nel tempo di `CPU` non si considera il tempo di I/O, mentre nel tempo di risposta si considera anche il tempo di latenza di accesso al disco.

Un altro parametro da considerare è il numero di istruzioni del programma e il numero di cicli di clock per queste istruzioni, ovvero il `CPI`.

$$\texttt{CPI} = \frac{\texttt{ClockCycles}}{\texttt{Instructions}}$$

$$\texttt{CPU time} = \texttt{Instructions} \cdot \texttt{CPI} \cdot \texttt{clock cycle time} = \frac{\texttt{Instructions} \cdot \texttt{CPI}}{\texttt{clock frequency}}$$

$$\texttt{CPU time} = \texttt{CI} \cdot \texttt{CPI} \cdot \texttt{T}_{\texttt{clock}}$$

Il tempo di `CPU` dipende da tre fattori:

- **Cicli di clock (*o frequenza*)**: dipende dall'architettura del processore.
- `CPI`: dipende dall'organizzazione dell'architettura e dall'insieme di istruzioni.
- **Numero di istruzioni**: dipende dall'insieme di istruzioni e dalla tecnologia di compilazione.

3.5 MIPS o GIPS

Il `MIPS` (*Million Instructions Per Second*) è una misura delle performance di un processore, legata al *throughput*. Tante volte quando si usano queste unità di misura, in alcuni casi, sono controituitive, non viene fatta una distinzione tra istruzioni complesse e istruzioni semplici.

Nasce quindi il `GFLOPS` (*Giga Floating Point Operations Per Second*), che misura il numero di operazioni in virgola mobile eseguite in un secondo.

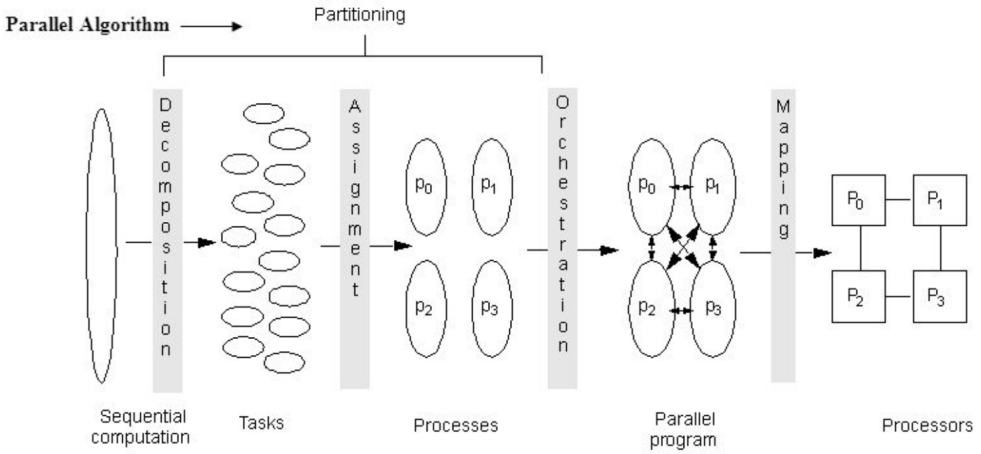
$$\texttt{GFLOPS} = \frac{\texttt{Numero di operazioni floating point nel programma}}{10^9}$$

Il problema di queste unità di misura è che non tengono conto della complessità delle operazioni. Usiamo quindi il `GFLOPS` normalizzato, che tiene conto della complessità delle operazioni.

Capitolo 4

Prospettiva sulla programmazione parallela

4.1 4 passi nella creazione di un programma parallelo



1. **Decomposizione**: si divide il problema in sotto-problemi più piccoli.
2. **Assegnazione**: si assegna ogni sotto-problema ad un processore.
3. **Orchestrazione**: si sincronizzano i processori.
4. **Mapping**: si ricombinano i risultati.

4.2 Capire il problema e il programma

Indubbiamente, il primo passo nello sviluppo di software parallelo è comprendere a fondo il problema che si desidera risolvere in parallelo. Se si parte da un programma seriale esistente,

questo necessita anche di una profonda comprensione del codice attuale. Questo aspetto è cruciale poiché la natura del problema influisce sulla fattibilità di un approccio parallelo e su come esso potrebbe essere strutturato.

Prima di investire tempo nello sviluppo di una soluzione parallela, è essenziale determinare se il problema in questione è adatto alla parallelizzazione. Non tutti i problemi possono beneficiare dell'esecuzione parallela, e riconoscere questo aspetto in anticipo può risparmiare tempo e risorse significative.

Un chiaro esempio di problema che può essere parallelizzato è il calcolo dell'energia potenziale per ciascuna di diverse migliaia di conformazioni indipendenti di una molecola. Una volta completati tutti i calcoli, rimane il compito di trovare la conformazione con l'energia minima.

Un esempio tipico di problema non parallelizzabile è il calcolo della serie di Fibonacci utilizzando la formula:

$$F(k+2) = F(k+1) + F(k)$$

Questo rappresenta un problema non parallelizzabile perché il calcolo della sequenza di Fibonacci, come mostrato, implica calcoli dipendenti piuttosto che indipendenti.

- Il calcolo del valore $F(k+2)$ utilizza i valori di $F(k+1)$ e $F(k)$. Questi tre termini non possono essere calcolati indipendentemente e, quindi, non possono essere eseguiti in parallelo.

La dipendenza diretta tra i termini consecutivi della serie impedisce qualsiasi decomposizione che permetta un'elaborazione parallela efficace, dimostrando così le limitazioni di alcuni tipi di problemi rispetto alla parallelizzazione.

- Questo problema si presta bene al processo parallelo perché ogni conformazione molecolare può essere determinata indipendentemente dalle altre.
- Il compito successivo di identificazione della conformazione di energia minima è anch'esso parallelizzabile, in quanto coinvolge il confronto di risultati che possono essere elaborati in parallelo.

Questo esempio illustra come i compiti possono essere decomposti ed eseguiti in parallelo, accelerando significativamente il processo di calcolo complessivo nelle applicazioni adatte.

4.2.1 Identificazione dei Punti Critici in un Programma

Identificare i Hotspots del Programma: Identificare i *hotspots*, ovvero le zone dove il programma compie la maggior parte del lavoro, è essenziale. Molti programmi scientifici e tecnici concentrano gran parte delle loro operazioni in poche aree critiche. L'uso di strumenti di profilazione e analisi delle prestazioni è quindi cruciale per individuare queste aree. Concentrarsi sulla parallelizzazione di questi hotspots può aumentare notevolmente l'efficienza del programma, mentre le sezioni che utilizzano meno `CPU` possono essere trascurate in questa fase iniziale.

Identificare i Colli di Bottiglia nel Programma: È importante riconoscere le aree del programma che sono sproporzionatamente lente o che causano interruzioni o ritardi nel lavoro

che potrebbe essere parallelizzato. Spesso, operazioni come l'`I/O` sono responsabili di questi rallentamenti. Modificare la struttura del programma o adottare un diverso algoritmo può aiutare a ridurre o eliminare queste inefficienze, migliorando così le prestazioni complessive.

Identificare gli Inibitori al Parallelismo: Un altro aspetto fondamentale è identificare gli inibitori al parallelismo. La dipendenza dai dati è un esempio comune di questi ostacoli, come dimostrato dalla sequenza di Fibonacci. Queste dipendenze creano una situazione in cui i calcoli devono essere eseguiti in un ordine specifico, il che limita le opportunità di eseguire il processo in parallelo.

Esplorare Altri Algoritmi: Infine, l'esplorazione di altri algoritmi può rappresentare la considerazione più importante nella progettazione di un'applicazione parallela. Trovare un algoritmo più adatto al parallelismo può spesso offrire soluzioni più efficienti e performanti.

4.3 Decomposizione

Identificare la concorrenza in un programma e decidere il livello al quale sfruttarla è fondamentale per ottimizzare l'esecuzione parallela. Questo processo inizia con la suddivisione del calcolo in compiti che possono essere distribuiti tra diversi processi. È importante notare che i compiti possono diventare disponibili dinamicamente e che il numero di compiti disponibili può variare nel tempo.

L'obiettivo principale è avere abbastanza compiti per mantenere i processi occupati, ma non troppi; infatti, il numero di compiti disponibili in un dato momento rappresenta il limite superiore della velocità di esecuzione che può essere raggiunta. Troppi compiti potrebbero sovraccaricare i processi e diminuire l'efficienza complessiva, mentre troppo pochi compiti potrebbero lasciare alcune risorse inutilizzate, riducendo la performance.

Per quanto riguarda la suddivisione del lavoro computazionale tra i task paralleli, esistono due metodi fondamentali: la decomposizione per dominio e la decomposizione funzionale. La **decomposizione per dominio** implica dividere i dati su cui opera il programma in parti che possono essere processate in parallelo, mentre la **decomposizione funzionale** comporta la divisione delle funzioni del programma in sotto-funzioni che possono essere eseguite contemporaneamente.

4.3.1 Decomposizione per Dominio

La decomposizione per dominio è una tecnica comune per suddividere il lavoro in un programma parallelo. Questo approccio prevede la divisione dei dati in parti che possono essere processate indipendentemente. Questo metodo è particolarmente utile quando i dati sono indipendenti tra loro e possono essere elaborati senza interazioni tra i processi.

Un esempio di decomposizione per dominio è l'analisi di un'immagine in cui ogni pixel può essere elaborato indipendentemente dagli altri. In questo caso, l'immagine può essere divisa in sezioni che possono essere processate in parallelo, accelerando notevolmente il processo di analisi.

La decomposizione per dominio può essere svolta in diversi modi, nel caso mono-dimensionale si può dividere il lavoro in blocchi di dati o eseguire il lavoro in modo ciclico. Nel caso bidimensionale, si può dividere il lavoro in righe o colonne, o in blocchi di dimensioni maggiori. Anche nel caso bidimensionale è possibile eseguire il lavoro in modo ciclico.

4.3.2 Decomposizione Funzionale

La decomposizione funzionale è un'altra tecnica importante per la progettazione di programmi paralleli. Questo approccio prevede la divisione delle funzioni del programma in sotto-funzioni che possono essere eseguite contemporaneamente. Questo metodo è particolarmente utile quando le funzioni del programma possono essere eseguite indipendentemente l'una dall'altra.

Un esempio di decomposizione funzionale è l'elaborazione di un documento di testo in cui ogni paragrafo può essere analizzato indipendentemente dagli altri. In questo caso, il documento può essere suddiviso in paragrafi che possono essere processati in parallelo, accelerando notevolmente il processo di analisi.

4.4 Assegnazione

Nel contesto della programmazione parallela, è cruciale considerare i compiti come "cose da fare" e i thread come "lavoratori". Questa analogia aiuta a visualizzare la distribuzione del lavoro tra i processi. Ad esempio, potremmo decidere quale processo calcola le forze su quali stelle, o quali raggi vengono calcolati da quale processo. L'obiettivo principale è bilanciare il carico di lavoro, riducendo i costi di comunicazione e gestione, noto anche come *load balancing*.

4.4.1 Approcci Strutturati alla Divisione dei Compiti

Gli approcci strutturati tendono a funzionare bene in questo contesto:

- L'ispezione del codice (*come i loop paralleli*) o la comprensione dell'applicazione possono guidare la divisione efficace dei compiti.
- L'uso di euristiche ben note può facilitare questo processo.
- Si considerano assegnazioni statiche versus dinamiche dei compiti, a seconda della natura e delle esigenze dell'applicazione.

Come programmatori, tendiamo a preoccuparci prima della partizione, che di solito è indipendente dall'architettura o dal modello di programmazione. Tuttavia, il costo e la complessità nell'uso delle primitive possono influenzare le decisioni.

4.4.2 Bilanciamento del Carico

Il bilanciamento del carico si riferisce alla pratica di distribuire i compiti tra i processi in modo che tutti i processi siano costantemente occupati, minimizzando il tempo di inattività dei processi. Questo aspetto è fondamentale per le prestazioni dei programmi paralleli. Ad esempio, se tutti i processi sono soggetti a un punto di sincronizzazione a barriera, il compito più lento determinerà le prestazioni complessive.

Come Raggiungere il Bilanciamento del Carico

- L'assegnazione dinamica del lavoro può essere utilizzata per gestire i compiti in modo flessibile.
- Alcune classi di problemi risultano in squilibri di carico anche se i dati sono distribuiti uniformemente tra i processi:
  - Array sparsi - alcuni processi avranno dati effettivi su cui lavorare mentre altri hanno prevalentemente "zeri".
  - Metodi di griglia adattivi - alcuni processi potrebbero dover raffinare la loro mesh mentre altri no.
  - Simulazioni N-body - dove alcune particelle possono migrare verso o lontano da un processo.
- Quando il lavoro che ogni processo eseguirà è intenzionalmente variabile o non prevedibile, può essere utile utilizzare un approccio di scheduler - task pool. Man mano che ogni processo completa il suo lavoro, si mette in coda per ottenere un nuovo pezzo di lavoro.
- Potrebbe diventare necessario progettare un algoritmo che rilevi e gestisca gli squilibri di carico man mano che si verificano dinamicamente all'interno del codice.

4.4.3 Granularità nel Calcolo Parallelo

Rapporto Computazione/Comunicazione

Nel calcolo parallelo, la granularità è una misura qualitativa del rapporto tra computazione e comunicazione. I periodi di computazione sono tipicamente separati dai periodi di comunicazione attraverso eventi di sincronizzazione. Questa distinzione è fondamentale per comprendere e ottimizzare le prestazioni dei sistemi paralleli.

Granularità del Parallelismo

Il parallelismo può essere classificato in base alla granularità delle operazioni che vengono eseguite:

Fine grain parallelism dove piccole quantità di lavoro computazionale sono seguite da eventi di comunicazione, con un basso rapporto tra computazione e comunicazione. Questo tipo di parallelismo può aiutare a ridurre gli overhead dovuti agli squilibri di carico, ma potrebbe incrementare gli overhead di comunicazione e sincronizzazione.

Coarse grain parallelism caratterizzato da quantità relativamente grandi di lavoro computazionale che intervallano gli eventi di comunicazione/sincronizzazione. Questo comporta un alto rapporto computazione/comunicazione, suggerendo maggiori opportunità di incremento delle prestazioni, anche se può essere più difficile da bilanciare efficacemente il carico di lavoro.

Parallelismo Fine o Grossolano?

La granularità più efficiente dipende dall'algoritmo specifico e dall'ambiente hardware in cui opera. In molti casi, l'overhead associato alle comunicazioni e alla sincronizzazione è elevato rispetto alla velocità di esecuzione, rendendo vantaggiosa una granularità grossolana. Tuttavia, il parallelismo fine può essere utile per ridurre gli overhead dovuti agli squilibri di carico. La scelta tra granularità fine o grossolana deve quindi essere ponderata in base alle specifiche esigenze dell'applicazione e alle caratteristiche del sistema di calcolo utilizzato.

4.5 Orchestrazione

L'orchestrazione in calcolo parallelo si concentra sulla strutturazione della comunicazione, sulla sincronizzazione e sull'organizzazione delle strutture di dati e sulla programmazione temporale dei compiti. Gli obiettivi principali di questa fase includono:

- Ridurre i costi di comunicazione e sincronizzazione.
- Preservare la località del riferimento ai dati.
- Programmare i compiti in modo da soddisfare le dipendenze il più presto possibile.
- Ridurre l'overhead della gestione del parallelismo.

Le scelte in questa fase dipendono dal modello di programmazione adottato, dall'astrazione della comunicazione e dall'efficienza delle primitive offerte dai progettisti di sistemi.

4.5.1 Comunicazioni nel Calcolo Parallelo

Chi necessita delle comunicazioni?

La necessità di comunicazioni tra compiti dipende dalla natura del problema:

Non necessita comunicazioni: Alcuni tipi di problemi possono essere decomposti ed eseguiti in parallelo senza quasi alcuna necessità di condivisione di dati tra i compiti. Ad esempio, un'operazione di elaborazione di immagini dove ogni pixel in un'immagine in bianco e nero deve avere il suo colore invertito. I dati dell'immagine possono essere facilmente distribuiti a più compiti che agiscono indipendentemente l'uno dall'altro per svolgere la loro parte di lavoro. Questi tipi di problemi sono spesso chiamati parallelamente imbarazzanti perché sono così diretti.

Necessita comunicazioni: La maggior parte delle applicazioni parallele non è così semplice e richiede che i compiti condividano dati tra di loro. Ad esempio, un problema di diffusione del calore in `3D` richiede che un compito conosca le temperature calcolate dai compiti che hanno dati adiacenti. Le modifiche ai dati adiacenti hanno un effetto diretto sui dati del compito.

Fattori importanti nella progettazione delle comunicazioni inter-task

- **Costo delle comunicazioni:** La comunicazione inter-task implica quasi sempre un overhead. I cicli di macchina e le risorse che potrebbero essere utilizzati per la computazione vengono invece utilizzati per impacchettare e trasmettere dati.
- **Latency vs Bandwidth:**
  - La *latency* è il tempo necessario per inviare un messaggio minimo ($0byte$) da un punto A a un punto B.
  - La *bandwidth* è la quantità di dati che può essere comunicata per unità di tempo, comunemente espressa in megabyte/sec o gigabyte/sec.
- **Visibilità delle comunicazioni:** Con il modello di passaggio di messaggi, le comunicazioni sono esplicite e generalmente ben visibili e sotto il controllo del programmatore. Con il modello di parallelismo sui dati, le comunicazioni spesso avvengono trasparentemente per il programmatore, particolarmente su architetture a memoria distribuita.
- **Comunicazioni sincrone vs asincrone:** Le comunicazioni possono essere sincrone, bloccanti o non bloccanti, a seconda delle necessità dell'applicazione.

Tipi e Complessità delle Comunicazioni

Nella progettazione di codici paralleli, è essenziale determinare quali compiti devono comunicare tra loro. Queste comunicazioni possono essere implementate sia in modo sincrono che asincrono e si dividono in due categorie principali:

Comunicazione Punto-a-Punto: Coinvolge due compiti specifici dove uno agisce come mittente/produttore di dati e l'altro come ricevitore/consumatore. Questa tipologia è ideale per il trasferimento diretto di dati tra due entità.

Comunicazione Collettiva: Implica la partecipazione di più di due compiti, generalmente definiti come membri di un gruppo collettivo. Le varianti comuni includono:

- **Broadcast:** Dati inviati da un mittente a tutti i partecipanti.
- **Scatter:** Dati divisi e inviati a diversi ricevitori dallo stesso mittente.
- **Gather:** Dati raccolti da diversi mittenti in un singolo ricevitore.
- **Allgather, Reduce, e Allreduce:** Operazioni che combinano le funzionalità di raccolta e distribuzione dati, con tutti i partecipanti che inviano o ricevono dati.

Sovraccarico e Complessità: Le comunicazioni introducono un sovraccarico derivante dalla necessità di sincronizzare i compiti, impacchettare e trasmettere dati, e gestire il traffico di rete che può saturare la banda disponibile. La scelta del tipo di comunicazione (*sincrona o asincrona*) può influenzare significativamente l'efficienza del sistema parallelo. Le comunicazioni asincrone possono ridurre i tempi di attesa e migliorare la fluidità dell'esecuzione, mentre quelle sincrone possono semplificare il design ma a costo di potenziali ritardi.

Queste considerazioni sono fondamentali per ottimizzare le prestazioni e l'efficienza di un sistema di calcolo parallelo, bilanciando le esigenze di velocità e coerenza dei dati all'interno dell'applicazione.

4.5.2 Sincronizzazione nel Calcolo Parallelo

La sincronizzazione è un aspetto cruciale del calcolo parallelo, necessario per coordinare l'operato dei processi paralleli. Qui di seguito vengono descritti i tipi principali di sincronizzazione utilizzati nei programmi paralleli.

Tipi di Sincronizzazione

Barriera: Una barriera è un punto di sincronizzazione dove tutti i processi devono arrivare prima di poter procedere. È comunemente utilizzata per separare le fasi di computazione e garantire che tutti i processi raggiungano un certo stato prima di avanzare.

Lock e Semaphore: Questi meccanismi controllano l'accesso a risorse condivise per prevenire conflitti e inconsistenze. Un lock è un meccanismo di mutua esclusione, mentre un semaforo può permettere un accesso limitato a più processi.

Comunicazioni Sincrone: Le operazioni di comunicazione sincrone richiedono che entrambi i processi, mittente e ricevente, siano pronti per inviare o ricevere dati, fungendo da forma di sincronizzazione.

Sincronizzazione di Eventi Globali

Una sincronizzazione globale avviene tramite l'uso di una `BARRIER(nprocs)`, che richiede che tutti i processi coinvolti raggiungano questo punto prima di continuare. Questo tipo di sincronizzazione è spesso utilizzato per assicurare che tutte le operazioni precedenti, come l'accumulo di somme globali, siano completate prima di procedere alla fase successiva del calcolo.

Sincronizzazione di Eventi Punto-a-Punto

In alcuni scenari, un processo potrebbe dover notificare un altro evento per poter procedere, tipico del modello produttore-consumatore. In programmi paralleli con spazio di indirizzi condiviso, ciò può essere gestito con semafori o variabili ordinari usate come bandiere. Questa pratica, nota come *busy-waiting* o *spinning*, comporta che un processo rimanga in attesa attiva fino alla ricezione del segnale per procedere.

Sincronizzazione di Eventi di Gruppo

Questa forma di sincronizzazione coinvolge solo un sottoinsieme di processi, che possono usare barriere o bandiere per coordinare l'azione tra di loro. Gli scenari tipici includono:

- Produttore singolo, consumatori multipli.
- Produttori multipli, consumatore singolo.

- Produttori e consumatori multipli.

Questi metodi permettono di sincronizzare efficacemente sottoinsiemi di processi in base alle necessità specifiche del problema e del design dell'applicazione.

4.6 Mapping

La mappatura di processi su una topologia di rete specifica pone interrogativi importanti:

- Saranno eseguiti processi multipli sullo stesso processore?
- Come si possono sfruttare al meglio le connessioni fisiche e logiche nella rete per ottimizzare la comunicazione?

Condivisione dello Spazio

La condivisione dello spazio implica la divisione della macchina in sottoinsiemi, ognuno dei quali può ospitare un'applicazione per volta. Questo può essere gestito in due modi:

- I processi possono essere vincolati a processori specifici.
- I processi possono essere lasciati alla gestione del sistema operativo, che decide dinamicamente l'allocazione.

Allocazione del Sistema

Nel mondo reale, l'utente specifica alcuni desideri riguardo all'allocazione dei processi e il sistema gestisce alcuni aspetti automaticamente. La visione comune è quella di associare un processo a un processore, ma questa può variare in base alle esigenze e alla configurazione del sistema.

Prospettiva dell'Architettura

L'architettura del sistema deve decidere:

- Cosa può essere migliorato con un design hardware migliore?
- Quali sono le questioni fondamentalmente legate alla programmazione?

4.7 Obiettivi di Alto Livello e Processo di Parallelizzazione

Segue una tabella che riassume gli step nel processo di parallelizzazione e i loro obiettivi principali:

Step	Dipendente dall'Architettura?	Obiettivi di Performance
Decomposizione	No	Esporre sufficiente concorrenza ma non troppo
Assegnazione	No	Bilanciare il carico di lavoro
Orchestrazione	Sì	Ridurre i volumi di comunicazione
Mappatura	Sì	Sfruttare la località nella topologia di rete

Questi obiettivi mirano a ottimizzare le prestazioni, come il miglioramento della velocità rispetto ai programmi sequenziali, riducendo al contempo l'utilizzo delle risorse e lo sforzo di sviluppo. Queste considerazioni sono cruciali sia per i progettisti di algoritmi che per gli architetti di sistema.

4.8 Come appare un programma parallelo

Consideriamo una versione semplificata che calcola le simulazioni oceaniche. Utilizziamo il metodo di Gauss-Seidel per aggiornare i punti interni di una griglia. Ogni punto della griglia è aggiornato in base alla media ponderata del suo valore corrente e dei valori dei suoi vicini immediati.

La formula per il calcolo è la seguente:

$$A_{i,j} = 0.2 \cdot (A_{i,j} + A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1})$$

Capitolo 5

General Purpose Graphics Processing Unit

5.1 Introduzione

Le GPU sono state introdotte per fare rendering di immagini e video, ma sono state utilizzate anche per fare calcoli scientifici. Principalmente si utilizzano per fare calcoli di tipo matriciale.

L'instruction set delle GPU è molto più semplice rispetto a quello delle CPU, in quanto è stato progettato per fare calcoli matriciali.

Le general purpose GPU a differenza delle normali GPU hanno un instruction set molto più ricco, in quanto sono state progettate per fare calcoli scientifici.

Nelle GPU sono fatte per il l'indipendenza dei dati, quindi sono molto adatte per fare calcoli paralleli. Il problema delle CPU sono dei colli di bottiglia,

Le GPU hanno tante ALU. Molto meglio calare la frequenza, in modo che si consumi meno energia e si possa fare più calcoli. Poiché vi è un andamento esponenziale tra la frequenza e il consumo di energia.

5.2 CUDA

La CPU verrà chiamata `host`, mentre la GPU si chiamerà `device`. Il `CUDA kernel` è costituito da un insieme di thread, che sono raggruppati in blocchi (*grid*). Tutte queste thread eseguono lo stesso codice scritto nel `CUDA kernel`. Per distinguere i thread si utilizza l'`ID` del thread, in questo modo si può fare un calcolo diverso per ogni thread. Quindi ogni grid è composta da blocchi, e ogni blocco è composto da thread. Le thread che appartengono allo stesso blocco possono comunicare tra di loro in maniera molto efficiente, poiché l'hardware delle GPU è stato progettato per fare ciò avendo una memoria condivisa tra le thread dello stesso blocco. Le thread che appartengono a blocchi diversi non possono comunicare in modo diretto, ma possono comunicare tramite la memoria globale, che è molto più lenta. Ogni kernel ha

associato un `grid` e nella `grid` ci sono dei blocchi, e in ogni blocco ci sono delle thread, perciò l'identificativo delle thread è perciò espresso fino a tre dimensioni.

Ogni thread ha un suo insieme di registri e l'accesso è molto veloce, in quanto i registri sono molto veloci e indipendenti tra di loro.

È importante differenziale i puntatori tra la memoria dell'host e la memoria del device, in quanto sono due memorie diverse.

Capitolo 6

GPU

6.1 Interrograzione del Device

Per interrograre il device si può utilizzare la funzione `cudaGetDeviceProperties` che restituisce una struttura `cudaDeviceProp` con le informazioni sul device.

```
cudaDeviceProp dev_prop;
for (i = =; i < deviceCount; i++) {
    cudaGetDeviceProperties(&dev_prop, i);
}
```

6.2 Analisi quantitativa sugli accessi in memoria

Una `GPU`. Quando abbiamo un codice e guardiamo il `CPU` bound e il `GPU` bound (*relazione tra il tempo di calcolo e il tempo di accesso alla memoria*).

$$\frac{\#\texttt{computations}}{\#\texttt{communications}}$$

Nel caso della moltiplicazione tra matrici, se consideriamo una matrice larga `width`, prende un elemento di m, un elemento di n e li moltiplica.

Se ho un `flops` ho 4 `b/s` ...

6.3 Memory Coalescing

La coalescenza della memoria è un'ottimizzazione che permette di.

Consideriamo il sistema di mantenimento dell'informazione in memoria. Il tempo per far scaricare il condensatore nel corso del tempo non è aumentato, ma il tempo per leggere un byte è aumentato, cercando di miniaturizzare i transistor e aumentare la densità di memoria.

Quando una GPU accede alla memoria in elementi contigui, la GPU può fare una sola richiesta per tutti gli elementi contigui.

Nel caso di una matrice, se la matrice è memorizzata per righe, la GPU può fare una sola richiesta per tutti gli elementi di una riga. Se la matrice è memorizzata per colonne, la GPU deve fare una richiesta per ogni elemento.

Una soluzione potrebbe essere quella di trasporre la matrice, ma questo comporta un costo computazionale.

La coalescenza della memoria ha solo il problema dell'accesso in global memory.

COnsidero thread dello stesso warp, che accedono a memoria contigua. Se i thread accedono a memoria contigua, la GPU può fare una sola richiesta per tutti i thread.