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.

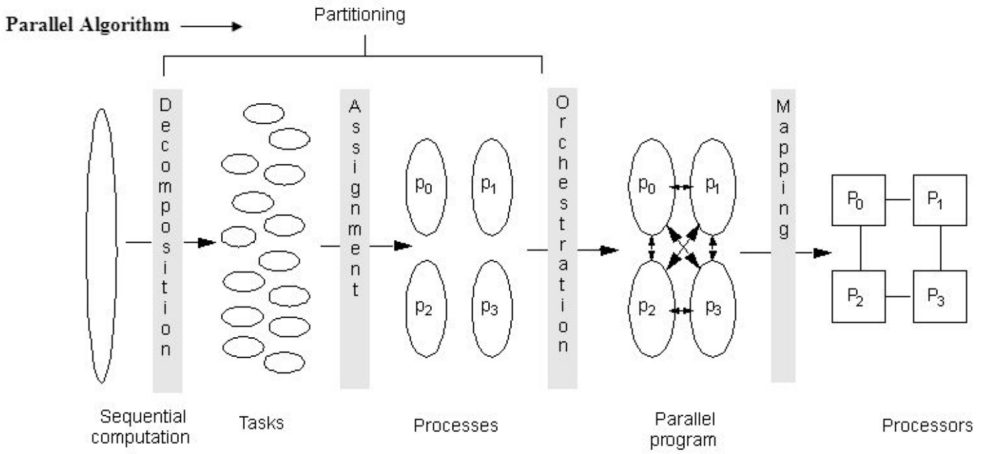
$$\texttt{GFLOPS} = \frac{\texttt{Numero di operazioni floating point nel programma}}{10^9}$$

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

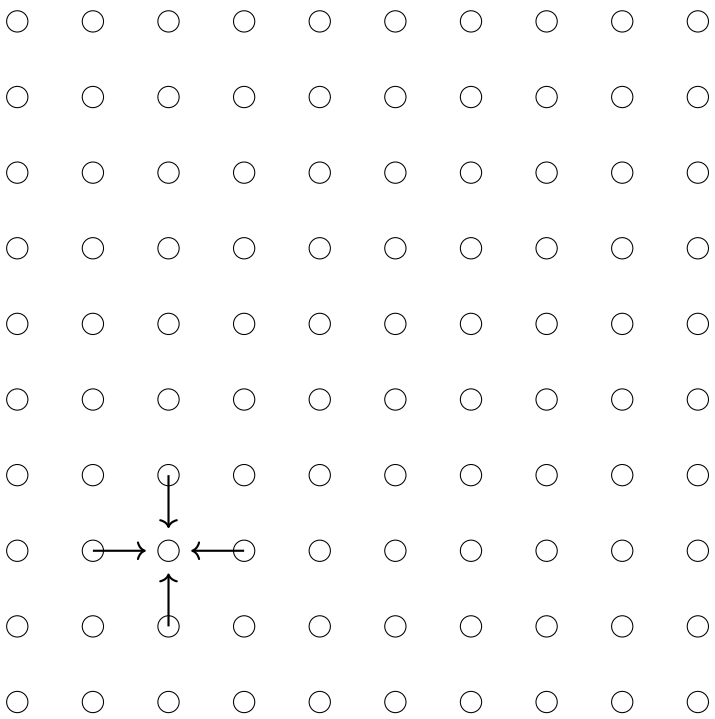
Questi obiettivi mirano a ottimizzare le prestazioni, come il miglioramento della velocità rispetto ai programmi sequenziali, riducendo al contempo l'utilizzo delle risorse e lo sforzo di sviluppo. Queste considerazioni sono cruciali sia per i progettisti di algoritmi che per gli architetti di sistema.

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:

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



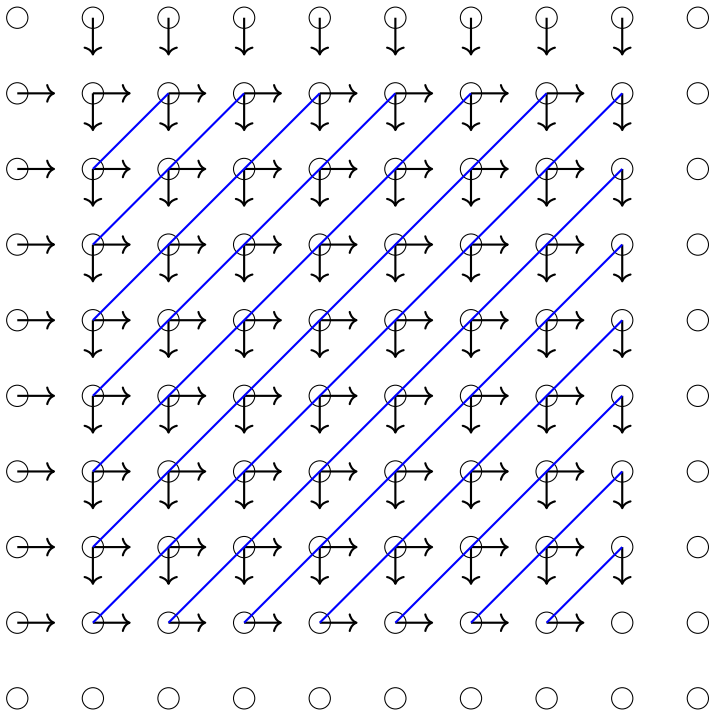
Una versione sequenziale di questo programma potrebbe essere scritta come segue:

Input: Dimensione della griglia n
Output: Griglia aggiornata A dopo la convergenza

```
n ← read_input()
A ← allocate_grid(n + 2)
initialize_grid(A)
Solve(A)
Function Solve(A):
    done ← 0
    while not done do
        diff ← 0
        for i ← 1 to n do
            for j ← 1 to n do
                tmp ← A[i][j]
                A[i][j] ← 0.2 × (A[i][j] + A[i − 1][j] + A[i + 1][j] + A[i][j − 1] + A[i][j + 1])
                diff ← diff + abs(A[i][j] − tmp)
            end
        end
        if diff/(n × n) < threshold then
            done ← 1
        end
    end
```

Algorithm 1: `Simulazioni Oceaniche - Versione Sequenziale`

Il primo passo per parallelizzare questo programma è quello di identificare le dipendenze tra i calcoli. In questo caso, ogni punto della griglia dipende dai valori dei suoi vicini immediati. Questo implica che i calcoli per ogni punto della griglia devono essere eseguiti in un ordine specifico per garantire che i valori dei vicini siano disponibili prima di calcolare il valore del punto stesso.



Quello che possiamo notare è che i calcoli per ogni punto della griglia possono essere eseguiti in parallelo, a patto che i valori dei vicini siano disponibili. Questo significa che possiamo dividere la griglia in sezioni più piccole e assegnare ciascuna sezione a un thread parallelo. Ogni thread può quindi calcolare i valori dei punti nella sua sezione in parallelo, riducendo notevolmente il tempo di calcolo complessivo.

L'idea per migliorare le performance del'algoritmo è quella di cambiare l'ordine in cui le celle vengono aggiornate. Il nuovo algoritmo itera le stesse soluzioni, ma converge diversamente.

Uno dei metodi utili nella risoluzione di sistemi di equazioni su griglie di calcolo, soprattutto in contesti paralleli, è l'adozione di un ordinamento specifico per la traversata della griglia. L'ordinamento "red-black", o scacchiera, è un esempio di questo approccio che consente di migliorare la convergenza degli aggiornamenti iterativi.

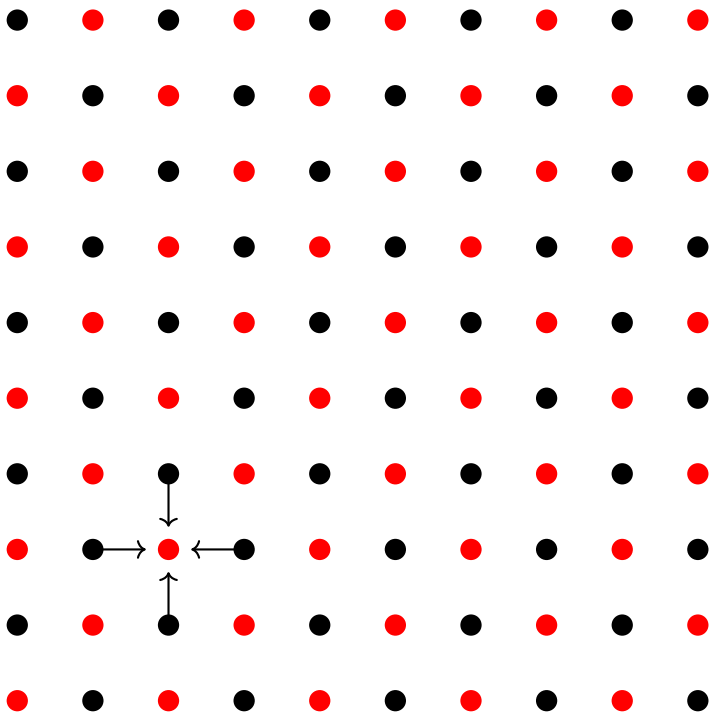
4.8.1 Ordinamento rosso-nero

L'ordinamento "rosso-nero" suddivide la griglia di calcolo in due insiemi disgiunti, comunemente denominati come *red* e *black*. Questi due insiemi sono aggiornati sequenzialmente:

- **Red Sweep:** Durante la fase red, solo i nodi rossi sono aggiornati.
- **Black Sweep:** Successivamente, durante la fase black, solo i nodi neri sono aggiornati.

La caratteristica principale di questo metodo è che ciascun "sweep" (*passata*) può essere eseguito in modo completamente parallelo, poiché non ci sono dipendenze dirette tra i nodi dello stesso colore durante un singolo sweep. Questo riduce significativamente il tempo di attesa per le sincronizzazioni globali tra i thread o i processi.

Nonostante la parallelizzazione, è necessaria una sincronizzazione globale tra le due fasi per garantire che tutti i nodi rossi siano stati completamente aggiornati prima di iniziare gli aggiornamenti dei nodi neri e viceversa. Questa sincronizzazione, sebbene conservativa, è conveniente per mantenere la coerenza dei dati tra le fasi di aggiornamento.



L'assegnamento dei processi a ciascuna fase può seguire un modello di assegnazione statica o dinamica, a seconda delle esigenze dell'applicazione e delle caratteristiche dell'architettura sottostante. L'obiettivo principale è garantire che i processi siano costantemente occupati e che i tempi di attesa siano ridotti al minimo.

Assegnamento Statico

L'assegnamento statico decompone la griglia in righe o colonne e assegna ciascuna sezione a un processo specifico. Questo approccio è semplice e prevedibile, ma può portare a squilibri di carico se le sezioni non sono bilanciate in termini di complessità computazionale.

Assegnamento Dinamico

L'assegnamento dinamico si basa su un modello di coda di lavoro in cui i processi richiedono e ricevono nuovi compiti man mano che terminano quelli precedenti. Questo metodo è particolarmente vantaggioso in contesti in cui il carico di lavoro è variabile, poiché permette una migliore adattabilità e minimizza i tempi di inattività dei processi.

4.8.2 Considerazione delle Dipendenze nel Flusso di Dati

Il processo di calcolo parallelo richiede una gestione accurata delle dipendenze per massimizzare l'efficienza. La seguente sezione descrive il flusso di lavoro diviso in fasi di calcolo e comunicazione:

1. Esecuzione parallela dell'aggiornamento delle celle rosse.
2. Attesa fino al completamento dell'aggiornamento da parte di tutti i processori.
3. Comunicazione delle celle rosse aggiornate agli altri processori.
4. Esecuzione parallela dell'aggiornamento delle celle nere.
5. Attesa fino al completamento dell'aggiornamento da parte di tutti i processori.
6. Comunicazione delle celle nere aggiornate agli altri processori.
7. Ripetizione del processo.

Questo metodo alterna fasi di computazione e comunicazione, essenziale per mantenere un flusso di lavoro efficiente e coordinato tra i diversi processori.

4.8.3 Come pensare alla parallelizzazione di un programma

Ci sono diversi aspetti da considerare quando si pensa alla parallelizzazione di un programma. Alcuni di questi includono:

- **Pensare alla parallelizzazione dei dati:** Considerare come i dati possono essere suddivisi tra i processi e come le operazioni possono essere eseguite in parallelo.
- `SPMD`**/spazi di indirizzi condivisi:** Pensare a come i processi possono condividere dati e comunicare tra di loro in un ambiente di spazio di indirizzi condiviso.
- **Passaggio di messaggi:** Considerare come i processi possono comunicare tra di loro utilizzando il passaggio di messaggi e come possono coordinare le loro attività.

Sincronizzazione in Ambiente di Memoria Condivisa

La programmazione in un ambiente di memoria condivisa impone agli sviluppatori la responsabilità della sincronizzazione tra i thread. Le primitive comuni di sincronizzazione includono:

- **Locks**: Forniscono l'esclusione mutua, permettendo a un solo thread per volta di entrare in una regione critica.
- **Barriers**: I thread devono attendere che tutti raggiungano questo punto di barriera prima di procedere.

Modello di Programmazione `SPMD`

Il modello di programmazione Single Program, Multiple Data (`SPMD`) non segue il lockstep, ovvero non implica necessariamente l'esecuzione delle stesse istruzioni contemporaneamente da parte di tutti i processi:

- L'assegnazione è controllata dai valori delle variabili utilizzate come limiti dei loop.
- Le operazioni speciali più interessanti in questo modello sono quelle di sincronizzazione, necessarie per assicurare l'accesso esclusivo a risorse condivise e per le operazioni di riduzione globale.
- La necessità di barriere deriva dalla necessità di assicurare che tutte le operazioni precedenti siano state completate prima di procedere.

4.8.4 Primitive di Sincronizzazione con Barriere

Le barriere sono un metodo di sincronizzazione essenziale nei sistemi di calcolo parallelo. Servono per garantire che tutti i thread o i processi raggiungano un certo punto nel loro esecuzione prima di procedere. Questo permette di gestire le dipendenze tra diverse fasi di calcolo.

Una barriera, tipicamente invocata come `barrier(num_threads)`, assicura che tutti i thread abbiano completato le loro operazioni prima di superare questo punto di sincronizzazione. Le caratteristiche principali includono:

- Le barriere dividono il calcolo in fasi ben distinte.
- Tutti i calcoli eseguiti da tutti i thread prima della barriera devono essere completati prima che qualsiasi thread inizi i calcoli previsti dopo la barriera.
- In sostanza, si presume che tutti i calcoli che seguono la barriera dipendano da tutti quelli che la precedono.

Le barriere sono quindi un modo conservativo per esprimere le dipendenze all'interno di un'applicazione parallela, assicurando che non ci siano race condition o inconsistenze nei dati condivisi.

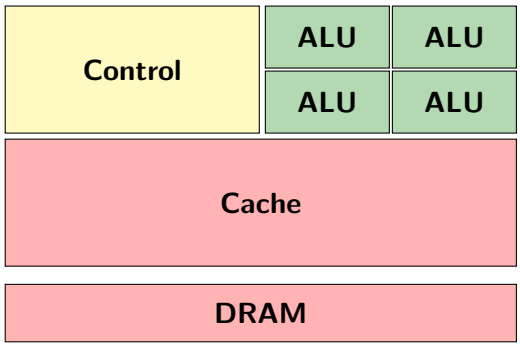
4.9 Message Passing Grid Solver

TODO

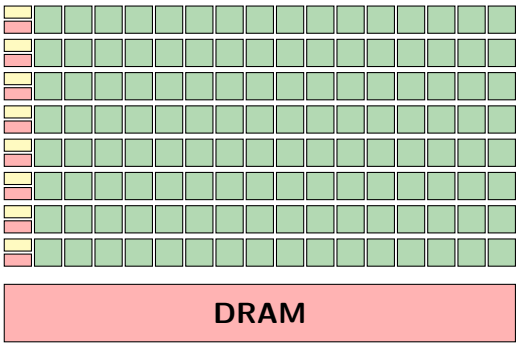
Capitolo 5

General Purpose GPU - GP-GPU

Le GPU (Graphics Processing Units) e le CPU (Central Processing Units) presentano differenze significative nella loro architettura e nel design, riflettendo le loro finalità di utilizzo orientate rispettivamente al throughput e alla latenza.



Central Processing Unit (CPU)



Graphical Processing Unit (GPU)

5.1 Architettura delle CPU

Le CPU sono progettate con un'orientamento alla riduzione della latenza:

- **Grandi cache**: Per convertire gli accessi alla memoria a lunga latenza in accessi alla cache a breve latenza.
- **Controllo sofisticato**: Includono la previsione dei branch per ridurre la latenza dei branch e il forwarding dei dati per ridurre la latenza dei dati.
- **ALU potenti**: Unità di elaborazione aritmetica progettate per ridurre la latenza delle operazioni.

5.2 Architettura delle GPU

Al contrario, le GPU sono progettate per massimizzare il throughput:

- **Piccole cache**: Ottimizzate per aumentare il throughput della memoria.
- **Controllo semplice**: Non dispongono di previsione dei branch né di forwarding dei dati.
- **ALU efficienti dal punto di vista energetico**: Numerose ALU, caratterizzate da lunghe latenze ma fortemente pipeline per un alto throughput.
- **Necessità di un numero massivo di thread**: Per tollerare le latenze grazie all'alto numero di thread in esecuzione.

5.3 Differenze nella Gestione della Memoria

Entrambe le architetture utilizzano la DRAM come memoria principale, ma si differenziano significativamente nelle loro strategie di gestione della cache e nel controllo delle unità di elaborazione, riflettendo i loro obiettivi di progettazione di bassa latenza per le CPU e alto throughput per le GPU.

5.3.1 Comparazione delle Prestazioni tra CPU e GPU

CPU per Codice Sequenziale Le CPU (Central Processing Units) sono ottimizzate per l'esecuzione di codice sequenziale dove la latenza è un fattore critico. Grazie alla loro architettura complessa, che include una gestione avanzata della cache e unità di controllo sofisticate, le CPU possono eseguire codice sequenziale molto più rapidamente rispetto alle GPU. Questo le rende particolarmente adatte per applicazioni che richiedono un elevato grado di interattività o tempi di risposta rapidi, poiché possono essere fino a 10 volte o più veloci delle GPU nell'esecuzione di codice sequenziale.

GPU per Codice Parallelo Al contrario, le GPU (Graphics Processing Units) sono progettate per massimizzare il throughput, specialmente utile in applicazioni che beneficiano dell'elaborazione parallela. Con migliaia di core più piccoli, le GPU gestiscono efficacemente compiti paralleli, distribuendo il lavoro su molti core per elaborare grandi quantità di dati simultaneamente. Questo le rende ideali per computazione scientifica, elaborazione di immagini e applicazioni di machine learning, dove possono superare le CPU di un fattore di 10 volte o più in termini di velocità.

Scelta dell'Hardware Adatto La scelta tra CPU e GPU deve essere fatta in base alla natura del carico di lavoro. Mentre le CPU sono preferibili per operazioni che richiedono decisioni rapide e poca parallelizzazione, le GPU sono la scelta migliore per lavori che possono essere facilmente suddivisi in compiti più piccoli e gestiti in parallelo. La decisione dovrebbe quindi basarsi sull'analisi del carico di lavoro specifico e delle esigenze di prestazione richieste.

5.4 Introduzione a CUDA C

CUDA C è un'estensione del linguaggio di programmazione C che consente di scrivere codice per le GPU di NVIDIA. Questo linguaggio permette di sfruttare la potenza di calcolo delle GPU per eseguire operazioni parallele su grandi quantità di dati. CUDA C è progettato per essere simile a C, ma include alcune estensioni e funzionalità specifiche per la programmazione parallela su GPU.

5.4.1 Interazione tra CPU e GPU

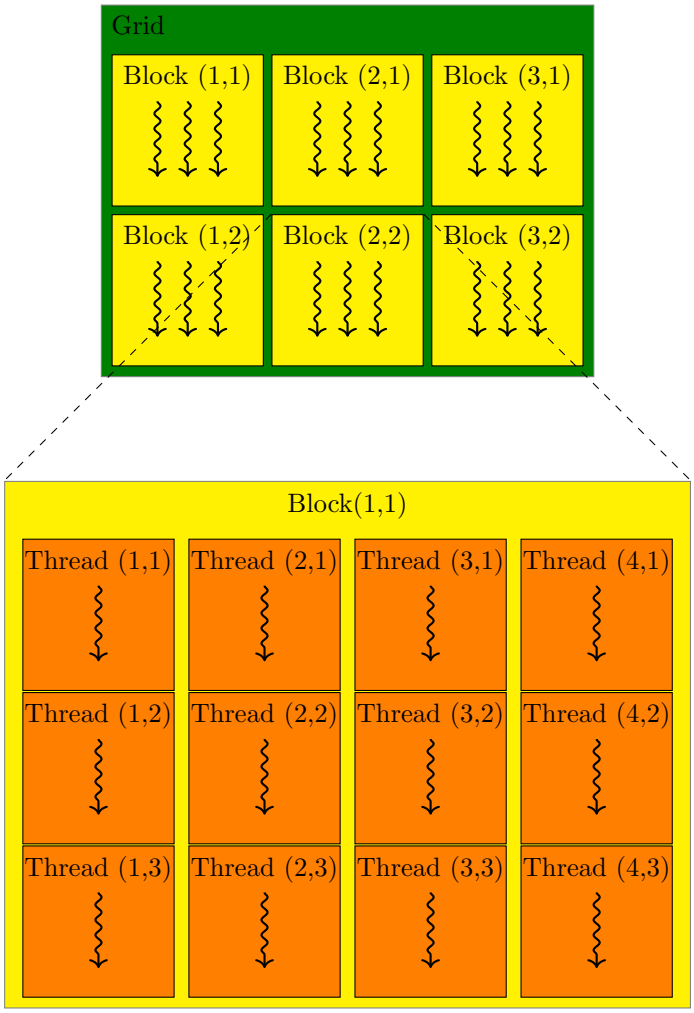
In un'applicazione CUDA, la CPU funge da **host** e la GPU come **device**. Il programma inizia la sua esecuzione sulla CPU con la funzione `main` e, al richiamo di una funzione **kernel**, trasferisce l'esecuzione sulla GPU. Questo permette di sfruttare l'elaborazione parallela per poi ritornare alla CPU una volta completate le operazioni sulla GPU.

5.4.2 Esecuzione del Kernel CUDA

Un **Cuda kernel** è eseguito da una griglia (*array*) di thread. Tutti i thread in questa griglia eseguono lo stesso codice scritto nel kernel. È fondamentale identificare ogni thread mediante un indice unico per garantire che ciascuno esegua il codice in modo appropriatamente "diverso".

5.4.3 Struttura della Grid e Blocchi di Thread

Le thread non sono aggregate in un unico gruppo all'interno della griglia, ma sono organizzate in **blocchi**. Quindi, abbiamo una griglia composta da blocchi di thread. Questa strutturazione è essenziale per facilitare la comunicazione e la sincronizzazione delle thread all'interno dello stesso blocco, grazie alla presenza di una memoria condivisa. Le thread in blocchi diversi, tuttavia, hanno capacità limitate di cooperazione diretta, rendendo la memoria condivisa all'interno dei blocchi uno strumento cruciale per l'efficienza del parallelismo.



5.5 Identificare le Thread in un Kernel CUDA

La programmazione in CUDA permette di identificare le thread in uno spazio che può avere fino a tre dimensioni, rappresentate come (x, y, z). Questo consente agli sviluppatori di adattare l'assegnazione delle thread alla struttura dei dati che devono essere elaborati.

5.5.1 Array Monodimensionale

Consideriamo il caso di un array di 128 elementi. In una struttura monodimensionale, è naturale assegnare a ogni blocco 128 thread, utilizzando solo la dimensione x per identificarle. In questo modo, ogni thread può essere associato direttamente a un elemento dell'array, e l'indice `x` del thread corrisponde all'indice dell'elemento nell'array:

```
    index = threadIdx.x;
    array[index] = ...; // Operazioni su ogni elemento dell'array
```

5.5.2 Matrice Bidimensionale

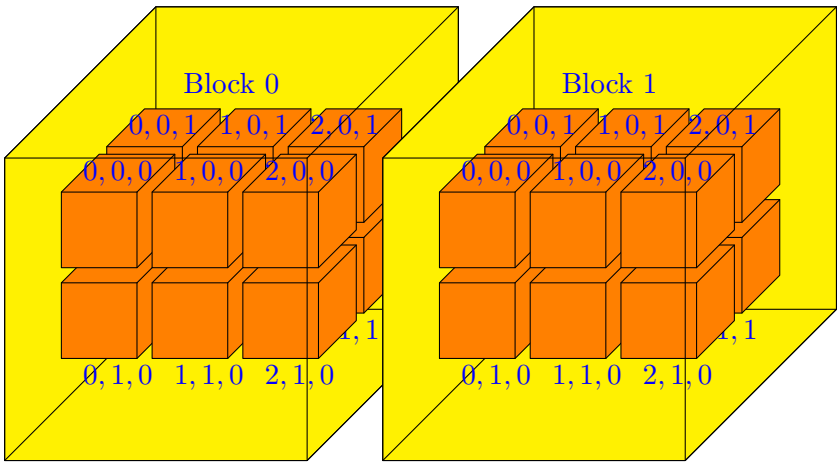
Per una matrice di dimensioni 20×20 400 *elementi*, possiamo utilizzare una configurazione bidimensionale per i blocchi e le thread. Assegnando 400 thread per blocco, ogni thread può essere identificato tramite le coordinate (x, y), permettendo di mappare direttamente le coordinate della matrice:

```
row = threadIdx.y;
col = threadIdx.x;
matrix[row][col] = ...; // Operazioni sulla matrice
```

5.5.3 Strutture Tridimensionali

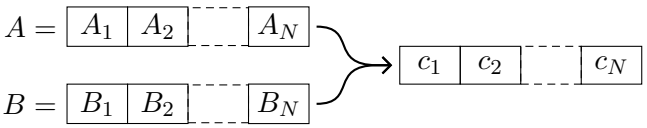
Nel caso di strutture tridimensionali, come un cubo, possiamo estendere ulteriormente questo concetto. I thread e i blocchi possono essere configurati in tre dimensioni (x, y, z), ognuna corrispondente a una dimensione del cubo. Questo permette di indirizzare un elemento specifico dentro un array tridimensionale in modo diretto:

```
row = threadIdx.y;
col = threadIdx.x;
depth = threadIdx.z;
cube[row][col][depth] = ...; // Operazioni sul cubo
```



5.6 Somma tra Vettori in CUDA

Per esemplificare il concetto di programmazione parallela in CUDA, consideriamo il problema della somma tra due vettori. Questo problema è particolarmente adatto alla programmazione parallela, poiché ogni elemento del vettore può essere sommato indipendentemente dagli altri.



5.6.1 Versione Sequenziale

La versione sequenziale di questo algoritmo è molto semplice: basta scorrere i due vettori e sommare gli elementi corrispondenti.

```
// Somma tra due vettori
void vecAdd(float *A, float *B, float *C, int n) {
  for (int i = 0; i < n; i++) {
    C[i] = A[i] + B[i];
  }
}

int main()
{
  // Allocazione e inizializzazione dei vettori
  float *A, *B, *C;
  int n = 1024;
  A = (float *)malloc(n * sizeof(float));
  B = (float *)malloc(n * sizeof(float));
  C = (float *)malloc(n * sizeof(float));
  for (int i = 0; i < n; i++) {
    A[i] = i;
    B[i] = i;
  }
  // Somma tra i vettori
  vecAdd(A, B, C, n);
  // Deallocazione della memoria
  free(A);
  free(B);
  free(C);
  return 0;
}
```

5.6.2 Processo di Parallelizzazione in CUDA

La programmazione in CUDA per l'elaborazione parallela su dispositivi GPU si articola generalmente in tre fasi principali. Queste fasi sono fondamentali per il corretto trasferimento e elaborazione dei dati tra la CPU (host) e la GPU (device).

Allocazione della Memoria su Device

Il primo passo in un'applicazione CUDA consiste nell'allocazione della memoria sul device per le variabili necessarie. Per esempio, se abbiamo due array A e B che devono essere sommati per ottenere un array C, è necessario allocare la memoria per A, B, e C sul device:

```
  cudaMalloc(&A, size);
  cudaMalloc(&B, size);
  cudaMalloc(&C, size);
```

Esecuzione del Kernel

Una volta che la memoria è stata allocata e i dati iniziali sono stati trasferiti al device, il prossimo passo è il lancio del kernel. Il kernel è il codice eseguito parallelamente dai thread della GPU:

```
dim3 threadsPerBlock(256);
dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x);
addKernel<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Questo esempio mostra un kernel chiamato `addKernel` che somma due array.

Copia dei Risultati alla Host Memory

Dopo l'esecuzione del kernel, l'ultimo passo è copiare il risultato dall'array C dalla memoria del device alla memoria dell'host. Questo permette all'host di utilizzare i risultati calcolati dalla GPU:

```
cudaMemcpy(hostC, C, size, cudaMemcpyDeviceToHost);
```

5.6.3 Versione Parallela in CUDA

La versione parallela di questo algoritmo utilizza la programmazione parallela per sfruttare la potenza di calcolo della GPU. Il kernel `vecAdd` viene eseguito da un insieme di thread, ognuno dei quali somma un elemento dei vettori A e B per produrre l'elemento corrispondente del vettore C.

```
__global__ void vecAdd(float *A, float *B, float *C, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {
    C[i] = A[i] + B[i];
  }
}

int main()
{
  // Allocazione e inizializzazione dei vettori
  float *A, *B, *C;
  float *d_A, *d_B, *d_C;
  int n = 1024;
  A = (float *)malloc(n * sizeof(float));
  B = (float *)malloc(n * sizeof(float));
  C = (float *)malloc(n * sizeof(float));
  for (int i = 0; i < n; i++) {
    A[i] = i;
    B[i] = i;
  }
  // Allocazione della memoria sul device
  cudaMalloc(&d_A, n * sizeof(float));
  cudaMalloc(&d_B, n * sizeof(float));
  cudaMalloc(&d_C, n * sizeof(float));
  // Copia dei dati dalla host alla device
  cudaMemcpy(d_A, A, n * sizeof(float), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_B, B, n * sizeof(float), cudaMemcpyHostToDevice);
    // Esecuzione del kernel
    vecAdd<<<(n + 255) / 256, 256>>>(d_A, d_B, d_C, n);
    // Copia dei risultati dalla device alla host
    cudaMemcpy(C, d_C, n * sizeof(float), cudaMemcpyDeviceToHost);
    // Deallocazione della memoria
    free(A);
    free(B);
    free(C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    return 0;
}
```

5.6.4 Dimensionamento dei Blocchi e delle Thread

Quando si esegue un kernel CUDA, è necessario specificare il numero di blocchi e il numero di thread per blocco. Questi valori dipendono dalla dimensione del problema e dalla capacità della GPU. In generale, è consigliabile utilizzare un numero di thread per blocco che sia un multiplo di 32, poiché le GPU di NVIDIA organizzano i thread in gruppi di 32 chiamati *warps*.

```
dim3 DimGrid((N / 256), 1, 1);
if (N % 256 != 0) DimGrid.x++;
dim3 DimBlock(256, 1, 1);

vecAdd<<<DimGrid, DimBlock>>>(d_A, d_B, d_C, N);
```

5.6.5 Definizione di Funzioni in CUDA

Le funzioni in CUDA sono classificate secondo il contesto in cui possono essere eseguite e da cui possono essere chiamate:

- `__device__ type fun()` - Definisce una funzione che può essere chiamata e eseguita solo sul dispositivo (device). Queste funzioni sono utilizzate per eseguire calcoli direttamente sulla GPU.
- `__global__ type fun()` - Specifica una funzione che, pur essendo definita per il device, viene lanciata dall'host. Questo tipo di funzione è comunemente noto come *kernel* e rappresenta il cuore dell'esecuzione parallela in CUDA.
- `__host__ type fun()` - Una funzione destinata a essere chiamata ed eseguita sull'host, ossia la CPU.

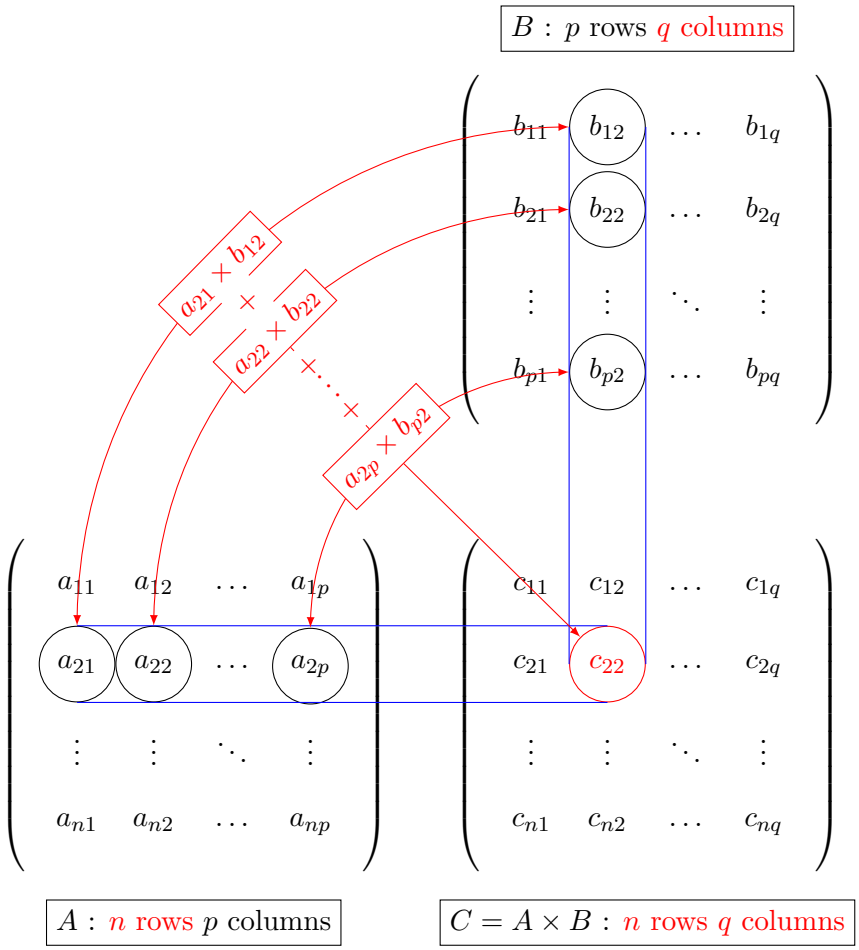
In CUDA, è anche possibile definire funzioni che possono essere eseguite sia sull'host che sul device utilizzando le direttive `__host__ __device__`.

5.7 Moltiplicazione di Matrici in CUDA

La moltiplicazione delle matrici può essere espressa come $C = A \times B$, dove A, B, e C sono matrici. Ogni elemento della matrice C è il prodotto scalare di una riga di A e una colonna di

B. Questo problema è particolarmente adatto alla programmazione parallela, poiché i calcoli per ogni elemento di C possono essere eseguiti indipendentemente dagli altri.

Ogni thread calcola un elemento della matrice C utilizzando la formula $C_{ij} = \sum_{k=0}^{N-1} A_{ik} \times B_{kj}$, dove i e j sono gli indici di riga e colonna di C, rispettivamente, e k è l'indice della somma.



5.7.1 Versione Sequenziale

La versione sequenziale di questo algoritmo è molto semplice: basta scorrere le righe di A e le colonne di B per calcolare ogni elemento di C.

```
// Moltiplicazione tra due matrici
void matMul(float *A, float *B, float *C, int n, int p, int q) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < q; j++) {
      C[i * q + j] = 0;
      for (int k = 0; k < p; k++) {
        C[i * q + j] += A[i * p + k] * B[k * q + j];
      }
    }
  }
```

```
}

int main()
{
  // Allocazione e inizializzazione delle matrici
  float *A, *B, *C;
  int n = 1024, p = 512, q = 256;
  A = (float *)malloc(n * p * sizeof(float));
  B = (float *)malloc(p * q * sizeof(float));
  C = (float *)malloc(n * q * sizeof(float));
  for (int i = 0; i < n * p; i++) {
    A[i] = i;
  }
  for (int i = 0; i < p * q; i++) {
    B[i] = i;
  }
  // Moltiplicazione tra le matrici
  matMul(A, B, C, n, p, q);
  // Deallocazione della memoria
  free(A);
  free(B);
  free(C);
  return 0;
}
```

5.7.2 Versione Parallela in CUDA

La programmazione parallela su GPU sfrutta la potenza di calcolo elevata delle unità di elaborazione grafica per eseguire operazioni matematiche complesse, come la moltiplicazione di matrici, in modo significativamente più veloce rispetto ai processori tradizionali.

Per sfruttare al meglio le capacità delle GPU, è cruciale adottare una rappresentazione efficiente delle matrici. Le matrici A, B e C vengono comunemente rappresentate come array monodimensionali per facilitare l'accesso e migliorare le prestazioni. Questa rappresentazione, nota come *row-major order*, organizza le righe della matrice in modo contiguo nella memoria.

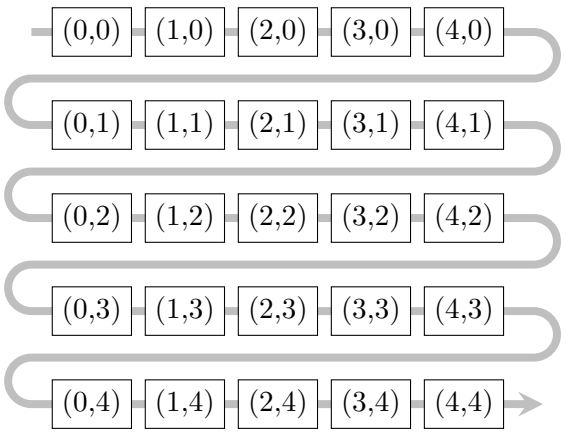


Figura 5.7.1: Rappresentazione di una matrice in *row-major order*

Nell'ambito della programmazione CUDA, l'accesso agli elementi di una matrice organizzata secondo il *row-major order* è effettuato attraverso la seguente formula:

$$\texttt{index} = \texttt{row} \times \texttt{width} + \texttt{col}$$

Questa formula calcola l'indice lineare dell'elemento nella memoria unidimensionale, dove `row` rappresenta l'indice della riga, `width` la larghezza della matrice, e `col` l'indice della colonna.

L'utilizzo del *row-major order* permette un accesso alla memoria più rapido e prevedibile, essenziale per il parallelismo su GPU. Gli accessi alla memoria che seguono il pattern naturale della memoria cache e dei banchi di memoria della GPU riducono i colli di bottiglia e migliorano il throughput delle operazioni.

5.7.3 Gestione dei grid e dei blocchi

Nella moltiplicazione di matrici tramite CUDA, il dimensionamento dei blocchi spesso si basa su una tecnica chiamata *tiling*. Questo approccio permette di ottimizzare l'uso della memoria e migliorare le prestazioni globali del sistema. Ogni blocco è responsabile del calcolo di un sottoinsieme della matrice risultante C, e ogni thread all'interno di un blocco calcola un singolo elemento di C.

Il dimensionamento dei grid e dei blocchi si effettua in modo da dividere la matrice in "piastrelle" che possono essere elaborate efficacemente dai blocchi di thread:

```
// Dimensionamento dei blocchi e dei grid
dim3 dimGrid(width/TILE_WIDTH, width/TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
matMul<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, width);
```

Una volta configurati i grid e i blocchi, il kernel CUDA viene eseguito su ciascun blocco per calcolare i valori corrispondenti nella matrice C. Il kernel dettagliato è mostrato di seguito:

```
// Definizione del kernel CUDA
__global__ void matMul(float *A, float *B, float *C, int width) {
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  float p_val = 0;
  if (row < width && col < width) {
      for (int k = 0; k < width; ++k) {
          p_val += A[row * width + k] * B[k * width + col];
      }
      C[row * width + col] = p_val;
  }
}
```

Questo kernel esegue un ciclo attraverso ogni elemento corrispondente nelle righe di A e nelle colonne di B per accumulare il prodotto nel valore p_val, che viene poi assegnato all'elemento corrispondente in C.

5.8 Thread CUDA

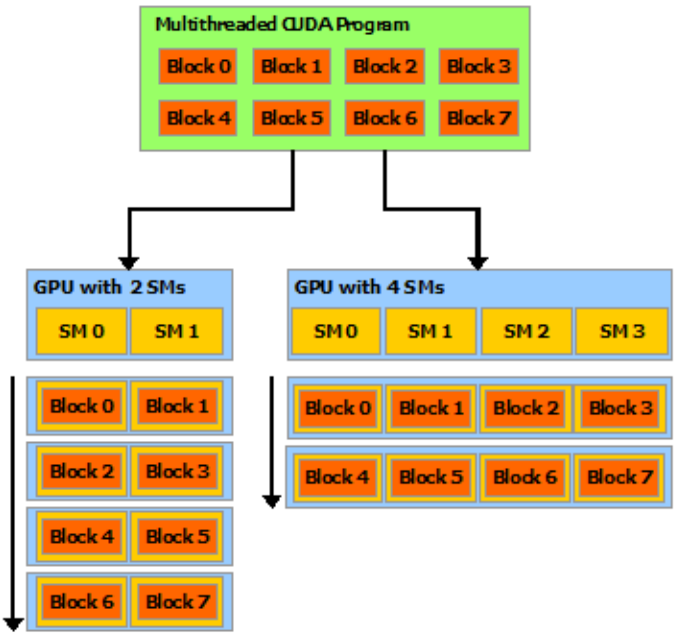
Le thread in CUDA sono organizzate in blocchi e griglie, che forniscono una struttura gerarchica per l'esecuzione parallela. Ogni thread è identificata da un indice unico all'interno del blocco e della griglia, che può essere utilizzato per accedere ai dati e coordinare le operazioni tra le thread.

5.9 Modello di Programmazione CUDA

Il modello di programmazione CUDA consente una scalabilità trasparente attraverso un'assegnazione dinamica dei blocchi di thread e una gestione flessibile delle esecuzioni, adattandosi automaticamente al numero di processori paralleli disponibili.

5.9.1 Scalabilità Trasparente nel Modello CUDA

Nel modello CUDA, l'hardware ha la completa libertà di assegnare i blocchi di thread a qualsiasi processore in qualsiasi momento, consentendo al kernel di scalare su un numero arbitrario di processori paralleli. Questa flessibilità è fondamentale per massimizzare l'efficienza di esecuzione e le prestazioni del sistema.



5.9.2 Dettagli Tecnici di Scheduling e Gestione dei SM

I blocchi di thread sono organizzati in griglie di blocchi, e ogni blocco può essere eseguito in un ordine indipendente rispetto agli altri. La granularità dell'assegnazione dei thread ai multiprocessori streaming (SM) varia in base alla generazione del processore:

- Gli SM Fermi possono gestire fino a 1536 thread, con configurazioni come 256 thread per blocco per 6 blocchi o 512 thread per blocco per 3 blocchi.
- Gli SM Kepler possono gestire fino a 2048 thread, con i thread che vengono eseguiti contemporaneamente.

Gli SM mantengono e gestiscono gli identificativi di thread e blocco, pianificando l'esecuzione dei thread in modo efficiente.

5.9.3 Warps come Unità di Scheduling

All'interno degli SM, ogni blocco viene eseguito come warps di 32 thread, che sono le unità di scheduling. Ad esempio, se a un SM sono assegnati 3 blocchi e ogni blocco ha 256 thread, il numero totale di warps per SM sarà di 24, calcolato come $256/32 = 8$ warps per blocco moltiplicato per 3 blocchi.

5.10 Partizione dei Blocchi di Thread e Controllo del Flusso in CUDA

Comprendere come i blocchi di thread sono partizionati e gestiti all'interno del modello di esecuzione di CUDA è fondamentale per ottimizzare le prestazioni e garantire un comportamento corretto dell'applicazione.

5.10.1 Partizione dei Blocchi di Thread

I blocchi di thread in CUDA sono partizionati in warp, con ID di thread consecutivi e crescenti all'interno di un warp. Questa partizione è consistente tra le esecuzioni, il che permette ai programmatori di anticipare e pianificare il comportamento dei thread all'interno dei warp:

- Ogni warp inizia con l'ID di thread 0.
- I warp sono strutturati in modo che gli ID di thread siano sempre consecutivi, migliorando la prevedibilità nel controllo del flusso.

Tuttavia, i programmatori non dovrebbero fare affidamento su un ordine specifico tra i warp a causa delle potenziali dipendenze tra i thread, che necessitano di una sincronizzazione esplicita utilizzando `__syncthreads()` per ottenere risultati di esecuzione corretti.

5.10.2 Controllo del Flusso in CUDA

Il controllo del flusso all'interno di CUDA è soggetto a problemi di divergenza, dove i thread all'interno di un singolo warp possono seguire percorsi di esecuzione differenti:

- La divergenza si verifica quando i thread in un warp seguono percorsi di controllo differenti, che vengono poi serializzati, potenzialmente degradando le prestazioni.
- Un esempio di divergenza: se `(threadIdx.x > 2)`, questa condizione crea due percorsi all'interno del primo warp.

- Per minimizzare la divergenza, è consigliabile allineare le condizioni di ramificazione con i confini dei warp, ad esempio, `if (threadIdx.x / WARP_SIZE > 2)`.

5.10.3 Schedulazione e Esecuzione dei Warp

La schedulazione dei warp è gestita dai Multiprocessori di Streaming (SM) con zero sovraccarico:

- I warp vengono eseguiti in base alla prontezza degli operandi e alle politiche di schedulazione prioritarie.
- Questo assicura che tutti i thread in un warp eseguano la stessa istruzione simultaneamente quando selezionati, ottimizzando il throughput.

5.10.4 Considerazioni sulla Granularità dei Blocchi per una Performance Ottimale

Scegliere la dimensione giusta del blocco è cruciale per la performance, specialmente in applicazioni come la moltiplicazione di matrici:

- Per una configurazione di blocco 8×8 (64 *thread per blocco*), un SM può supportare fino a 24 blocchi, ma a causa dei vincoli hardware, spesso solo 8 blocchi possono essere attivi contemporaneamente.
- Una configurazione di blocco 16×16 (256 *thread per blocco*) permette a un SM di operare vicino alla piena capacità con fino a 6 blocchi.
- Blocchi più grandi, come 32×32 (1024 *thread per blocco*), possono superare la capacità per SM di alcune versioni di CUDA e hardware, riducendo l'efficienza.

Ciascuna di queste considerazioni gioca un ruolo significativo nel determinare quanto efficacemente un programma CUDA può operare, e comprenderle è fondamentale per sfruttare appieno le capacità di CUDA.

Capitolo 6

GPU e memoria

6.1 Panoramica della Memoria in CUDA e Dichiarazione delle Variabili

6.1.1 Accesso alla Memoria nei Thread CUDA

In CUDA, ogni thread può accedere a diverse tipologie di memoria con tempi di accesso variabili, che influenzano le prestazioni dell'applicazione:

- **Registri per thread:** Ogni thread può leggere e scrivere nei registri specifici per thread con un tempo di accesso molto basso, circa 1 ciclo.
- **Memoria condivisa per blocco:** La memoria condivisa accessibile da tutti i thread di un blocco ha un tempo di accesso di circa 5 cicli.
- **Memoria globale per griglia:** Tutti i thread possono accedere alla memoria globale con un tempo di accesso significativamente più elevato, circa 500 cicli.
- **Memoria costante per griglia:** Accessibile in sola lettura, la memoria costante ha un tempo di accesso di circa 5 cicli se c'è caching.

6.1.2 Qualificatori delle Variabili in CUDA

Le variabili in CUDA possono essere qualificate in modo specifico per definire la loro visibilità e il luogo di memorizzazione. Di seguito è presentata una tabella che riassume i principali qualificatori delle variabili, la loro visibilità e i luoghi di memorizzazione.

Dichiarazione	Memoria	Visibilità	tempo di vita
`int LocalVar;`	Registri	Thread	Thread
`__device__ int GlobalVar;`	Globale	Grid	Applicazione
`__device__ __shared__ int SharedVar;`	Condivisa	Blocco	Blocco
`__device__ __constant__ int ConstVar;`	Costante	Grid	Applicazione

Tabella 6.1: Qualificatori delle variabili in CUDA e loro proprietà

Dove Dichiarare le Variabili?

- Variabili `global` e `constant` vanno dichiarate all'esterno di qualsiasi funzione per essere accessibili dall'host.
- Variabili `register` (automatiche) e `shared` sono dichiarate all'interno dei kernel.

Esempio

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
    {
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
```

6.2 Strategia di Programmazione Comune in CUDA: Tiling e Blocking

6.2.1 Utilizzo della Memoria Globale e Condivisa

La memoria globale, situata nella memoria del dispositivo (DRAM), presenta tempi di accesso lenti. Un metodo efficace per eseguire calcoli sul dispositivo è quindi quello di usare la tecnica di tiling, che sfrutta la memoria condivisa più veloce:

- **Partizionamento dei dati:** I dati vengono suddivisi in sottoinsiemi che si adattano alla memoria condivisa.
- **Caricamento dei dati:** Ogni blocco di thread carica un sottoinsieme dalla memoria globale alla memoria condivisa, utilizzando più thread per sfruttare il parallelismo a livello di memoria.
- **Computazione sui dati:** I thread eseguono calcoli sui dati nella memoria condivisa. Ogni thread può passare efficientemente più volte su qualsiasi elemento dei dati.
- **Copia dei risultati:** I risultati vengono copiati dalla memoria condivisa alla memoria globale.

6.2.2 Vantaggi del Tiling e del Blocking

- **Efficienza:** L'utilizzo di memoria condivisa riduce la latenza di accesso ai dati e aumenta la velocità di esecuzione.

- **Parallelismo a livello di memoria:** Più thread possono caricare e scrivere dati in modo simultaneo, migliorando ulteriormente le prestazioni.

6.2.3 Considerazioni sul Timing di Accesso dei Thread

- **Accesso simile:** Il tiling è particolarmente vantaggioso quando i thread hanno tempi di accesso simili, poiché questo permette di massimizzare l'efficienza e minimizzare i conflitti di accesso alla memoria.
- **Accesso diverso:** Se i tempi di accesso dei thread sono molto diversi, il tiling potrebbe non essere efficace e potrebbe portare a una serializzazione indesiderata delle operazioni, riducendo le prestazioni.

Queste strategie di programmazione sono cruciali per ottimizzare l'utilizzo delle risorse del dispositivo e migliorare le prestazioni generali delle applicazioni CUDA.

6.3 Moltiplicazione di Matrici in CUDA: Tiling e Blocking

6.3.1 Descrizione dell'Algoritmo

La moltiplicazione di matrici in CUDA può essere notevolmente ottimizzata attraverso l'uso del tiling e del blocking. Queste tecniche sfruttano la memoria condivisa del dispositivo per ridurre il numero di accessi alla memoria globale, che è più lenta. L'idea di base è suddividere le matrici in blocchi o "tiles" più piccoli che possono essere caricati nella memoria condivisa, permettendo così ai thread di lavorare su porzioni di dati più gestibili contemporaneamente.

6.3.2 Implementazione in CUDA

Il kernel CUDA per la moltiplicazione di matrici utilizzando il tiling segue questi passaggi principali:

1. Partizionare le matrici di input in blocchi di dimensioni TILE_WIDTH x TILE_WIDTH.
2. Ogni blocco di thread carica un tile di una matrice in memoria condivisa, sincronizzandosi con gli altri thread per assicurare che tutti i dati siano caricati prima di procedere al calcolo.
3. Calcolare il prodotto del blocco moltiplicando i corrispondenti tile delle due matrici.
4. Ogni thread accumula il risultato in una variabile locale e, una volta completato il calcolo per tutti i tile, scrive il risultato nella memoria globale.

6.3.3 Codice CUDA per il Kernel di Moltiplicazione

Di seguito è riportato un esempio di implementazione di un kernel CUDA per la moltiplicazione di matrici utilizzando il tiling:

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
    {
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
```

```
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        ds_M[ty][tx] = d_M[Row * Width + (m * TILE_WIDTH + tx)];
        ds_N[ty][tx] = d_N[Col + (m * TILE_WIDTH + ty) * Width];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k) {
            Pvalue += ds_M[ty][k] * ds_N[k][tx];
        }
        __syncthreads();
    }
    d_P[Row * Width + Col] = Pvalue;
}
```

Questo kernel CUDA esegue la moltiplicazione di matrici utilizzando il tiling per ottimizzare l'accesso alla memoria e migliorare le prestazioni complessive.

6.3.4 Spiegazione del Codice

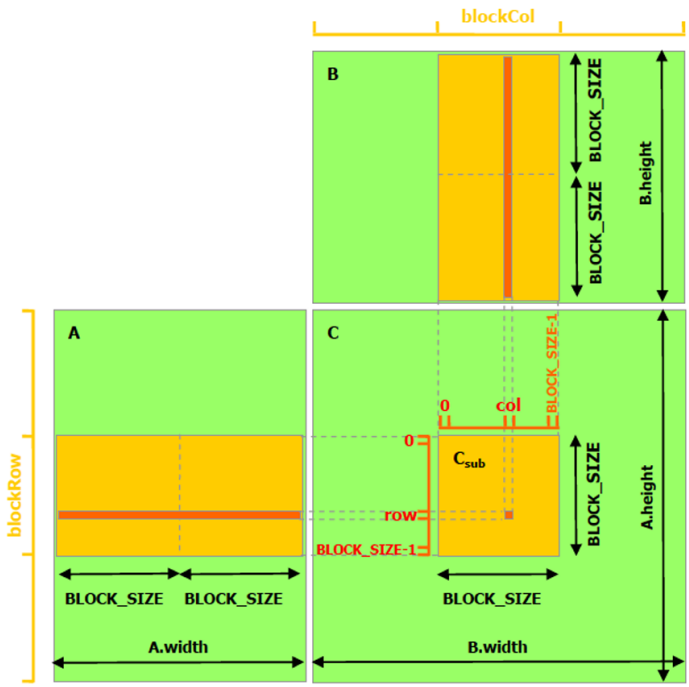


Figura 6.3.1: Moltiplicazione di matrici con tiling

- Il kernel carica i tile delle matrici M e N in memoria condivisa e calcola il prodotto del blocco.

- I thread sincronizzano l'accesso alla memoria condivisa per evitare conflitti e assicurare che tutti i dati siano disponibili per il calcolo.
- Il risultato viene accumulato in una variabile locale e scritto nella memoria globale.

Questo approccio sfrutta la memoria condivisa per ridurre i tempi di accesso alla memoria globale e migliorare le prestazioni della moltiplicazione di matrici su CUDA.

6.3.5 Dimensionamento dei Blocchi di Thread e Utilizzo della Memoria Condivisa

La scelta della dimensione dei blocchi di thread e della larghezza delle tessere (TILE_WIDTH) ha un impatto significativo sulla performance dei programmi CUDA.

- Un TILE_WIDTH di 16 porta a blocchi di 256 thread ($16 \cdot 16$), mentre un TILE_WIDTH di 32 comporta blocchi di 1024 thread (32x32).
- Per una TILE_WIDTH di 16, ogni blocco esegue 512 caricamenti di float dalla memoria globale e può eseguire $8,192$ operazioni di moltiplicazione/addizione.
- Con una TILE_WIDTH di 32, il numero di operazioni di moltiplicazione/addizione sale a 65536, per un totale di 2048 caricamenti di float.

Queste operazioni di caricamento influenzano direttamente l'utilizzo della memoria condivisa e la capacità di avere più blocchi attivi contemporaneamente su un singolo Multiprocessore di Streaming (SM):

- Con TILE_WIDTH $= 16$, ogni blocco di thread utilizza $2KB$ di memoria condivisa ($2*256*4B$), consentendo potenzialmente fino a 8 blocchi di thread attivi contemporaneamente.
- Con TILE_WIDTH $= 32$, l'utilizzo di memoria condivisa aumenta a $8KB$ per blocco, limitando il numero di blocchi attivi a 2 o 6 a seconda della configurazione della memoria condivisa e della dimensione totale disponibile sull'SM.

Utilizzando un tiling di 16x16, si riducono gli accessi alla memoria globale di un fattore 16, incrementando così efficacemente la larghezza di banda utilizzabile e il throughput di calcolo.

6.3.6 Interrogazione delle Proprietà del Dispositivo

Per sfruttare al meglio le risorse hardware, è essenziale interrogare le proprietà del dispositivo CUDA disponibile:

```
int dev_count;
cudaGetDeviceCount(&dev_count);
cudaDeviceProp dev_prop;
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&dev_prop, i);
    // Decisioni basate su dev_prop.maxThreadsPerBlock,dev_prop.
    sharedMemPerBlock, ...
}
```

Questo codice permette di determinare il numero di dispositivi e le loro specifiche, come il numero massimo di thread per blocco e la memoria condivisa per blocco, elementi cruciali per configurare correttamente i kernel.

6.3.7 Riepilogo - Struttura Tipica di un Programma CUDA

Un programma CUDA tipico segue questa struttura:

1. Definizione dei kernel e configurazione dei blocchi di thread.
2. Allocazione e inizializzazione delle strutture di dati.
3. Trasferimento dei dati dall'host al device.
4. Esecuzione dei kernel.
5. Copia dei risultati dal device all'host.
6. Liberazione delle risorse.

Ripetere questi passi secondo necessità per ottenere il comportamento desiderato e massimizzare le prestazioni.

Capitolo 7

GPU e considerazioni sulle performance

7.1 Coalescenza della Memoria in CUDA

La coalescenza della memoria è un concetto chiave per ottimizzare le prestazioni delle applicazioni CUDA. La coalescenza si riferisce all'accesso sequenziale e allineato alla memoria da parte dei thread, che consente alla GPU di combinare le richieste di accesso in un singolo ciclo di memoria, migliorando così l'efficienza e riducendo i tempi di accesso.

Consideriamo una matrice, rappresentata come un array monodimensionale, in condizioni normali, l'accesso ai dati da parte dei thread non è coalescente, poiché i thread accedono a posizioni di memoria non contigue. Tuttavia, organizzando la matrice in modo che i thread accedano a posizioni contigue, è possibile ottenere un accesso coalescente, che consente alla GPU di combinare le richieste di accesso in un singolo ciclo di memoria.

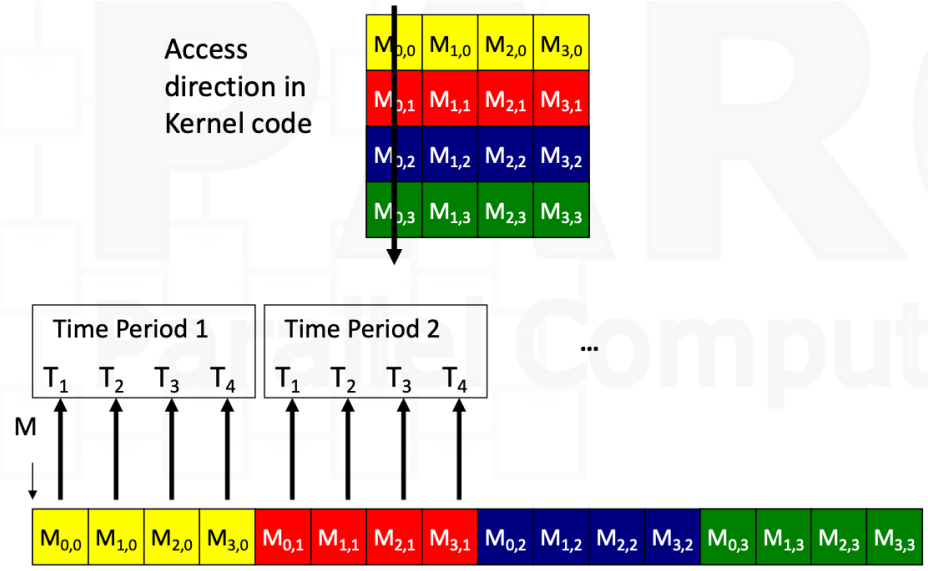


Figura 7.1.1: Accesso coalescente alla memoria

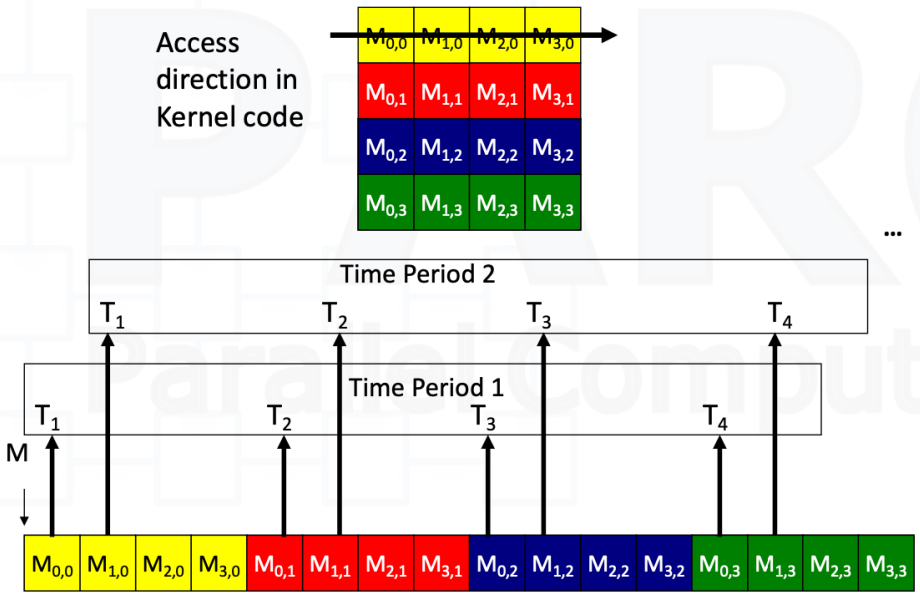


Figura 7.1.2: Accesso non coalescente alla memoria

7.1.1 Coalescenza della memoria

La coalescenza della memoria si occupa degli accessi alla memoria globale. A differenza della memoria condivisa, la memoria globale può subire penalità di prestazioni a causa di accessi non coalescenti. Gli accessi alla memoria coalescenti riguardano i thread dello stesso warp;

l'hardware verifica se gli accessi alla memoria globale sono eseguiti dai thread dello stesso mezzo warp e, in tal caso, coalesce gli accessi dei thread in un unico accesso consolidato.

7.2 Partizione dinamica delle risorse di esecuzione

7.2.1 Risorse di esecuzione in uno SM

Le risorse di esecuzione in uno Streaming Multiprocessor (SM) includono registri, memoria condivisa e slot per blocchi di thread. Analizziamo un dispositivo con un massimo di 1536 thread per SM:

- Con blocchi da 512 thread, è possibile ospitare 3 blocchi per SM, una configurazione ottimale.
- Con blocchi da 128 thread, si teorizzano 12 blocchi per SM, ma questo supera le capacità e si limita a 8 blocchi da 128 thread ciascuno, totalizzando 1024 thread, sotto-utilizzando così le capacità dello SM.

7.2.2 Gestione dei registri per SM

Considerando uno SM con 16384 registri disponibili:

- Se un kernel istanzia 10 variabili (32 bit) per thread:
  - Con blocchi da 256 thread, si necessitano 2560 registri per blocco.
  - Con un rapporto di 1536 thread per SM suddivisi in blocchi da 256 thread, si possono ospitare 6 blocchi, che utilizzano 15360 registri in totale, una configurazione accettabile.
  - Aggiungere anche solo un blocco ulteriore per SM comporterebbe il superamento del limite di registri disponibili, richiedendo una riduzione nel numero di blocchi.
- Aggiungendo due variabili per kernel (12 variabili in totale per thread):
  - Ogni blocco da 256 thread richiederebbe 3072 registri.
  - Sei blocchi da 3072 registri ciascuno comportano un totale di 18432 registri, superando il limite. La soluzione comporta una riduzione del numero di blocchi a 5, rientrando così nei 15360 registri disponibili, ma riducendo il numero di thread attivi per SM a 1280 anziché 1536, a causa dell'aumento del numero di variabili.

7.3 Parallelizzazione dei task per il trasferimento dati

Quando si schedula un kernel, è possibile definire un grafo di dipendenza tra le esecuzioni dei kernel e i trasferimenti di memoria. Per esempio, il kernel A non può eseguire fino a quando non sono completati i trasferimenti di dati A e B, ma può iniziare mentre il trasferimento C è ancora in corso.

CUDA offre la possibilità di eseguire operazioni di trasferimento dati e computazione in modo asincrono attraverso l'uso degli stream. Gli stream permettono di eseguire trasferimenti di dati e kernel in parallelo, migliorando l'utilizzo delle risorse e riducendo i tempi di attesa.

7.4 Device Overlap

Nel contesto della programmazione CUDA, il deviceOverlap rappresenta una funzionalità importante per aumentare l'efficienza e la velocità di esecuzione dei programmi. Questa funzione permette ai dispositivi CUDA di eseguire simultaneamente un kernel e una operazione di copia di memoria tra il dispositivo e l'host.

Verifica del Supporto a deviceOverlap

Per determinare se un dispositivo CUDA supporta il deviceOverlap, è possibile utilizzare il seguente codice:

```
int dev_count;
cudaDeviceProp prop;
cudaGetDeviceCount(&dev_count);
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&prop, i);
    if (prop.deviceOverlap) {
        // Il dispositivo supporta deviceOverlap
    }
}
```

Questo frammento di codice prima interroga il numero totale di dispositivi CUDA disponibili. Successivamente, per ogni dispositivo, acquisisce e analizza le sue proprietà attraverso cudaGetDeviceProperties per verificare la presenza della caratteristica deviceOverlap.

Utilizzo del Timing Sovrapposto (Pipelined)

Una tecnica efficace per sfruttare al meglio il deviceOverlap consiste nel dividere i vettori di grandi dimensioni in segmenti più piccoli e gestire il trasferimento e il calcolo di questi segmenti in maniera sovrapposta. Questo approccio, noto come *pipelined timing*, permette di ridurre i tempi morti in cui il dispositivo potrebbe altrimenti rimanere inattivo, ottimizzando così sia le operazioni di trasferimento che quelle di calcolo.

7.4.1 Stram di CUDA

Il parallelismo di task in CUDA è realizzato mediante divers streams. Le operazioni inserite in streams differenti possono essere eseguite in parallelo. Questo è particolarmente utile per sovrapporre la copia di memoria e l'esecuzione di kernel, migliorando l'efficienza complessiva del programma.

Le richieste di dispositivo effettuate dal codice host sono inserite in una coda, che viene letta ed elaborata in modo asincrono dal driver e dal dispositivo. Il driver assicura che i comandi

nella coda siano processati in sequenza, garantendo che le copie di memoria siano completate prima del lancio dei kernel, tra gli altri.

Per permettere la copia e l'esecuzione di kernel in modo concorrente, è necessario utilizzare più code, denominate `streams`. Gli "eventi" `CUDA` permettono al thread host di interrogare e sincronizzarsi con le code individuali.

Codice Host Multi-Stream Semplice

Di seguito è presentato un esempio di codice host che utilizza due `streams` per gestire le operazioni in modo parallelo:

```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

float *d_A0, *d_B0, *d_C0; // memoria del dispositivo per stream 0
float *d_A1, *d_B1, *d_C1; // memoria del dispositivo per stream 1

// Allocazione di cudaMalloc per d_A0, d_B0, d_C0, d_A1, d_B1, d_C1

for (int i = 0; i < n; i += SegSize * 2) {
    cudaMemCpyAsync(d_A0, h_A + i, SegSize * sizeof(float), ..., stream0);
    cudaMemCpyAsync(d_B0, h_B + i, SegSize * sizeof(float), ..., stream0);
    vecAdd<<<SegSize / 256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    cudaMemCpyAsync(d_C0, h_C + i, SegSize * sizeof(float), ..., stream0);

    cudaMemCpyAsync(d_A1, h_A + i + SegSize, SegSize * sizeof(float), ...,
    stream1);
    cudaMemCpyAsync(d_B1, h_B + i + SegSize, SegSize * sizeof(float), ...,
    stream1);
    vecAdd<<<SegSize / 256, 256, 0, stream1>>>(d_A1, d_B1, ...);
    cudaMemCpyAsync(d_C1, h_C + i + SegSize, SegSize * sizeof(float), ...,
    stream1);
}
```

Il problema di questo codice è che non avviene in trasferimento dei dati nel momento in cui il kernel esegue la somma vettoriale. Questo problema può essere risolto riorganizzando il codice in modo che i trasferimenti di dati avvengano in modo corretto:

```
for (int i = 0; i < n; i += SegSize * 2) {
    cudaMemCpyAsync(d_A0, h_A + i, SegSize * sizeof(float), ..., stream0);
    cudaMemCpyAsync(d_A1, h_A + i + SegSize, SegSize * sizeof(float), ...,
    stream1);
    cudaMemCpyAsync(d_B0, h_B + i, SegSize * sizeof(float), ..., stream0);
    cudaMemCpyAsync(d_B1, h_B + i + SegSize, SegSize * sizeof(float), ...,
    stream1);

    vecAdd<<<SegSize / 256, 256, 0, stream0>>>(d_A0, d_B0, ...);
    vecAdd<<<SegSize / 256, 256, 0, stream1>>>(d_A1, d_B1, ...);

    cudaMemCpyAsync(d_C0, h_C + i, SegSize * sizeof(float), ..., stream0);
```

```
    cudaMemCpyAsync(d_C1, h_C + i + SegSize, SegSize * sizeof(float), ...,
    stream1);
}
```

7.4.2 Concorrenza in Fermi e Precedenti

Le architetture GPU di NVIDIA precedenti a Kepler, come Fermi, supportavano una concorrenza a 16 vie. Questo significava che fino a 16 griglie potevano essere eseguite contemporaneamente. Tuttavia, le CUDA streams venivano multiplexate in una singola coda hardware per l'esecuzione, limitando la concorrenza a livello di stream.

7.4.3 Hyper Queue in Kepler

Con l'introduzione di Kepler, NVIDIA ha implementato Hyper Queue, che fornisce multiple code reali per ciascun motore, permettendo una maggiore concorrenza. Questo approccio consente ad alcune stream di fare progressi per un motore mentre altre possono essere bloccate, migliorando l'efficienza e la scalabilità.

7.4.4 Miglioramento della Concorrenza con Kepler

Kepler ha significativamente migliorato la concorrenza rispetto a Fermi, consentendo una concorrenza a 32 vie. Ogni stream ha una propria coda di lavoro dedicata, che elimina le dipendenze inter-stream e consente la concorrenza a livello di intero stream, migliorando così le prestazioni generali:

- **Concorrenza 32-vie:** Ogni GPU Kepler può gestire fino a 32 stream contemporaneamente.
- **Code di lavoro multiple:** Una coda di lavoro per ogni stream.
- **Nessuna dipendenza inter-stream:** Ogni stream opera indipendentemente dagli altri, permettendo una migliore scalabilità e efficienza.

7.5 Introduzione al Prefix Scan

L'operazione di **Prefix Scan**, prende in input:

- Un operatore binario associativo $\oplus$.
- Un array di N elementi $[x_0, x_1, \ldots, x_{N-1}]$.

Essa restituisce un array trasformato che può essere:

- **Inclusivo:** $[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \cdots \oplus x_{N-1})]$.
- **Esclusivo:** $[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \cdots \oplus x_{N-2})]$.

Esempio

Per $\oplus$ come addizione (somma prefissa):

- **Inclusivo:** Da $[3, 1, 7, 4, 6]$ a $[3, 4, 11, 15, 21]$.
- **Esclusivo:** Da $[3, 1, 7, 4, 6]$ a $[0, 3, 4, 11, 15]$.

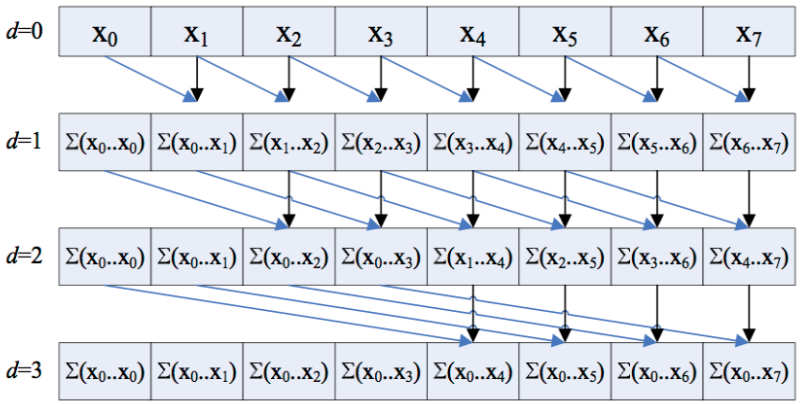


Figura 7.5.1: Esempio di Prefix Scan

Conversione tra Scansioni Inclusive ed Esclusive

- Da esclusiva a inclusiva:
  - Spostare l'array risultante di un elemento a sinistra.
  - Inserire alla fine la somma dell'ultimo elemento della scansione e l'ultimo elemento dell'array di input.
- Da inclusiva a esclusiva:
  - Spostare l'array risultante di un elemento a destra.
  - Inserire l'identità all'inizio.

7.5.1 Usi del Prefix Scan

Il Prefix Scan è una primitiva chiave in molti algoritmi paralleli per convertire calcoli seriali in paralleli, utilizzato in:

- Ordinamento (counting sort/radix sort)
- Istogramma
- Inserimento in coda di massa
- Compattazione di flusso/Partizione
- Moltiplicazione di matrici sparse

- Costruzione di strutture dati in parallelo
- e altro...

7.5.2 Implementazione Sequenziale

Inclusivo:

```
Out[0] = In[0];
for (int i = 1; i < N; ++i) {
  Out[i] = Out[i-1] + In[i];
}
```

Esclusivo:

```
Out[0] = 0;
for (int i = 1; i < N; ++i) {
  Out[i] = Out[i-1] + In[i-1];
}
```

Nota: Sono necessarie N addizioni per N elementi, con una complessità di $O(n)$.

7.5.3 Prefix Scan Parallelo

Di seguito è riportato l'algoritmo per un'implementazione parallela naive del Prefix Scan, usando un approccio iterativo che aumenta progressivamente la distanza tra gli elementi combinati ad ogni step.

for *int* `level` $= 0$; $level < \log_2(N)$; `++level` **do**
  for $\forall i \in N$ **do in parallel**
    `offset` $= 2^{\texttt{level}}$
  end
  if $i \geq$ `offset` **then**
    $X[i] = X[i - \texttt{offset}] + X[i]$
  end
end

Algorithm 2: Naive Parallel Prefix Scan

La complessità parallela, *span*, di questo algoritmo è $O(\log(N))$, mentre la complessità di lavoro è $O(n \cdot \log(n))$, dove n è il numero di elementi da scansionare, poiché ogni elemento deve essere combinato con tutti gli altri.

7.6 Prefix scan per l'efficienza di lavoro

L'algoritmo consiste in due fasi principali:

- *up-sweep*: calcola la somma prefissa per ogni potenza di 2.
- *down-sweep*: calcola la somma prefissa finale.

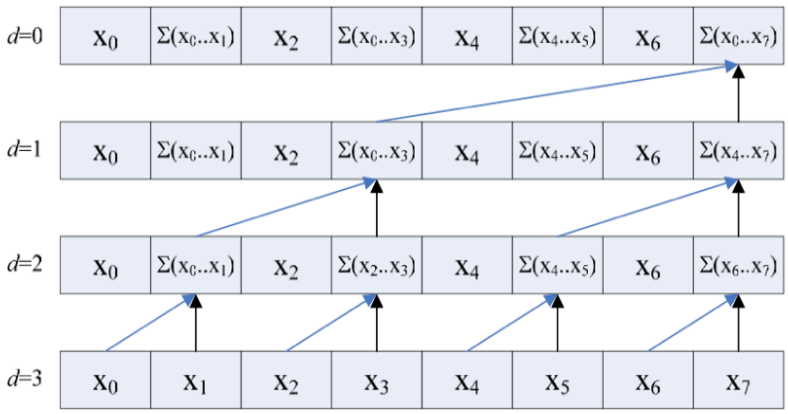


Figura 7.6.1: Fase 1: Up-Sweep

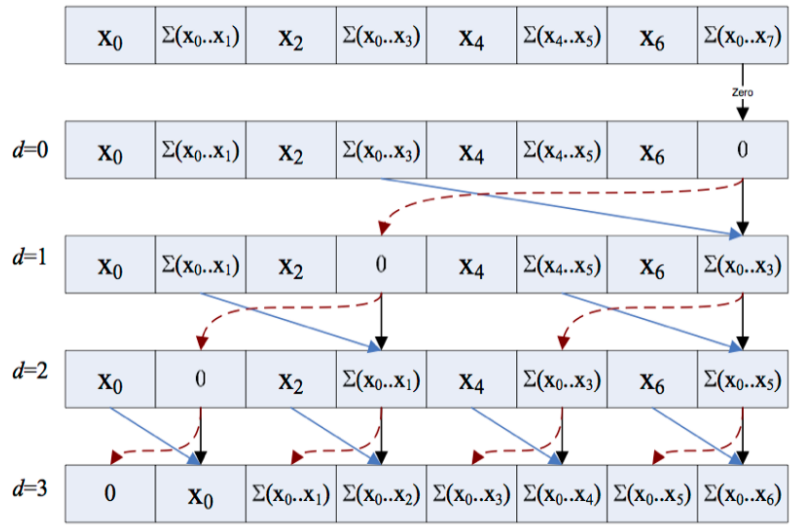


Figura 7.6.2: Fase 2: Down-Sweep

Di seguito è riportato il codice per l'implementazione parallela del Prefix Scan con efficienza di lavoro:

for $level = 0$; $level < \log_2(N)$; $++level$ **do**
  $step = 2^{level}$
  for $\forall i \mid i \mod (step \cdot 2) = 0$ **do in parallel**
    $valueRight = (i + 1) \cdot (step \cdot 2) - 1$
    $valueLeft = valueRight - step$
    $X[valueRight] = X[valueRight] + X[valueLeft]$
  end
end

Algorithm 3: Efficient Parallel Prefix Scan

In questo modo si riduce la complessità di lavoro a $O(n)$, mantenendo una complessità parallela di $O(\log(n))$.

Capitolo 8

Algoritmo BFS

La prima cosa è guardare la struttura dei dati, esploriamo alcune alternative e poi esaminiamo ciò che abbiamo allo stato dell'arte. È fondamentale comprendere bene la struttura dei dati e l'algoritmo. Affronteremo anche il problema del load balancing, che è particolarmente rilevante in questo contesto.

8.1 Rappresentazione del grafo in GPU

Tra le varie rappresentazioni abbiamo:

- Liste di adiacenza
- Matrice di adiacenza
- Edge list

8.1.1 Matrice di Adiacenza

La matrice di adiacenza è una rappresentazione comune dei grafi che memorizza le relazioni tra i nodi in una matrice booleana. In una matrice di adiacenza, ogni riga e colonna rappresenta un nodo del grafo, e il valore in posizione (i, j) indica se esiste un arco tra i nodi i e j.

Nel caso di grafi pesati, si utilizza una matrice di adiacenza con valori interi o in virgola mobile per rappresentare il peso degli archi:

$$M[i,j] = \begin{cases} 0 & \text{se } i = j \\ w(i,j) & \text{se } i \neq j \wedge (i,j) \in E \\ \infty & \text{altrimenti} \end{cases}$$

Il problema di tale approccio è che la matrice di adiacenza occupa spazio $O(V^2)$, che può essere inefficiente per grafi di grandi dimensioni. Inoltre, la matrice di adiacenza richiede $O(V^2)$ operazioni per inizializzare e accedere ai dati, rendendo l'algoritmo meno efficiente.

8.1.2 Liste di Adiacenza

Le liste di adiacenza sono un'altra rappresentazione comune dei grafi che memorizza la struttura del grafo in una serie di liste di adiacenza per ogni nodo. Ogni lista contiene i nodi adiacenti al nodo corrispondente. Abbiamo quindi due liste, una per i nodi e una per gli archi. Nell'array dei vertici vengono memorizzati gli offset, ovvero il numero dei figli di ogni nodo. Nell'array degli archi vengono memorizzati i nodi figli.

La dimensione di questa rappresentazione è $O(V + E)$, che è più efficiente della matrice di adiacenza per grafi sparsi. Si tratta di un problema lineare in memoria, che è molto più efficiente per grafi di grandi dimensioni. Il problema è che l'accesso ai dati è più lento rispetto alla matrice di adiacenza, poiché richiede $O(E)$ operazioni per accedere ai dati.

8.1.3 Edge List

L'edge list è una rappresentazione dei grafi che memorizza ogni arco del grafo come una tupla di nodi. Questa rappresentazione è compatta e richiede $O(2E)$ spazio, rendendola ideale per grafi di grandi dimensioni.

8.1.4 Scelta della Rappresentazione

Nella scelta della rappresentazione del grafo, è importante considerare le caratteristiche del grafo e le operazioni che devono essere eseguite. Nel caso del BFS, è necessario considerare:

- Il memory footprint
- L'accesso ai dati
- Tempo richiesto per trovare i figli dato un vertice

Consideriamo le più importanti ovvero la coalescenza e il load balancing. La tabella seguente confronta le varie rappresentazioni in termini di utilizzo dello spazio, efficienza dell'accesso, bilanciamento del carico, e coalescenza:

Tabella 8.1: Confronto delle rappresentazioni di grafi

Rappresentazione	Spazio	$(u, v) \in E$	$(u, v) \in \text{adj}(v)$	Load Bal.	Coalescenza
Matrice di Adiacenza	$O(V^2)$	$O(1)$	$O(V)$	Sì	Sì
Liste di Adiacenza	$O(V + E)$	$O(\deg(v))$	$O(\deg(v))$	Difficile	Difficile
Edge List	$O(2E)$	$O(E)$	$O(E)$	Sì	Sì

8.2 Algoritmo BFS sequenziale

L'algoritmo BFS è un algoritmo di ricerca in ampiezza che esplora tutti i nodi di un grafo a partire da un nodo sorgente. L'algoritmo visita tutti i nodi adiacenti al nodo sorgente, poi i nodi adiacenti a questi nodi, e così via, fino a quando non ha visitato tutti i nodi raggiungibili dal nodo sorgente.

L'algoritmo BFS può essere implementato in modo sequenziale utilizzando una coda per mantenere traccia dei nodi da visitare. L'algoritmo visita i nodi in ordine di distanza dal nodo sorgente, garantendo che i nodi più vicini vengano visitati prima dei nodi più lontani.

foreach *vertex* $u \in V(G)$ **do**
    u.dist $= \infty$
    $u.\pi = -1$
end
v_0.dist $= 0$
$v_0.\pi = -1$
$Q = \{v_0\}$
while $Q \neq \emptyset$ **do**
    $u = Q$.dequeue()
    foreach *vertex* $v \in adj(u)$ **do**
        if $v.dist = \infty$ **then**
            v.dist $= u$.dist $+ 1$
            $v.\pi = u$
            Q.enqueue(v)
        end
    end
end

Algorithm 4: BFS sequenziale

Nell'algoritmo BFS sequenziale, la coda Q viene utilizzata per mantenere traccia dei nodi da visitare. L'algoritmo inizia impostando la distanza di tutti i nodi dal nodo sorgente a ∞, tranne il nodo sorgente stesso, che ha distanza 0. Il nodo sorgente viene aggiunto alla coda Q, e l'algoritmo continua finché la coda non è vuota. Ad ogni iterazione, il nodo u viene rimosso dalla coda e i suoi nodi adiacenti vengono visitati. Se un nodo adiacente v ha distanza ∞, la sua distanza viene impostata a u.dist $+ 1$ e il nodo viene aggiunto alla coda.

8.3 Algoritmo BFS in CUDA

Nella versione parallela, la coda verra utilizzata come frontiera. L'idea è di utilizzare una frontiera per mantenere traccia dei nodi da visitare, e utilizzare un kernel per visitare i nodi adiacenti a ciascun nodo nella frontiera.

Il filtraggio serve a non considerare nodi la cui distanza non è infinito ed ad eliminare figli che hanno più padri.

8.3.1 Possibili soluzioni

Una possibile soluzione per rimuovere duplicati è l'utilizzo di una hash table, l'idea è quella che con delle chiamate a basso livello si riesca a prendere un nodo ed ad infilarlo nella hash table, se il nodo deve essere controllato lo si controlla dalla hash table.

8.3.2 Varie tecniche di implementazione

- Prefix sum esclusiva: è una tecnica che permette di calcolare la somma di tutti gli elementi precedenti ad un certo indice. Nella BFS serve quando ho una frontiera e devo allocare i nodi alla varie thread. L'idea è di assegnare i nodi con un certo bilanciamento, per capire se un nodo ha figli o meno. La prefix sum deve essere implementata in modo efficiente, in quanto è un'operazione critica per il bilanciamento del carico.
- Warp virtuali dinamiche:
- Parallelismo dinamico: permette che la ricorsione sia implementata nel kernel, in modo che ogni thread possa chiamare la funzione ricorsiva. In questo modo le thread e i blocchi di thread possono essere creati in modo dinamico e creati runtime. Il problema è che è pesantissimo.
- Ricerca degli archi.
- Single-Block vs Multi-Block: è possibile utilizzare un solo blocco di thread per eseguire l'intero algoritmo BFS, oppure utilizzare più blocchi di thread per eseguire l'algoritmo in parallelo. L'utilizzo di più blocchi di thread può migliorare le prestazioni dell'algoritmo, ma richiede una sincronizzazione tra i blocchi.
- Lettura/Scrittura coalescente.

8.4 Appunti 27/06

8.4.1 Blocco singolo vs Multi blocco

La calibrazione della threshold impatta nell'organizzazione della memoria condivisa di ogni stream multiprocessor. la memoria condivisa si organizza in modi diversi rispetto al numero di blocchi. Nella versione single block, la memoria condivisa è organizzata con una grossa parte dedicata alla hash table e una parte dedicata alla frontiera attuale e alla prossima. Nel caso multi block la frontiera è gestita nella memoria globale, ma la memoria condivisa ha un numero di variabili e ogni blocco di thread ha la propria hash table.

Single Block tempo

$$\texttt{F\_threashold} = \texttt{max\_threads\_per\_block} \cdot K_5$$

$$\texttt{HashT}_{\texttt{size}} = \texttt{neaest\_power\_of\_2}(|\texttt{SM}| - \texttt{Var}_{\texttt{size}})$$

Multi Block tempo

$$\texttt{HashT}_{\texttt{size}} = \texttt{nearest\_power\_of\_2}\left(\frac{(|\texttt{SM}| - \texttt{Var}_{\texttt{size}}) - K_6}{\texttt{max\_blocks\_per\_Multiprocessor}}\right)$$

8.4.2 Coalescenza di lettura/scrittura

Se lasciamo lavorare le thread indipendentemente, si ha un problema di totale non coalescenza.

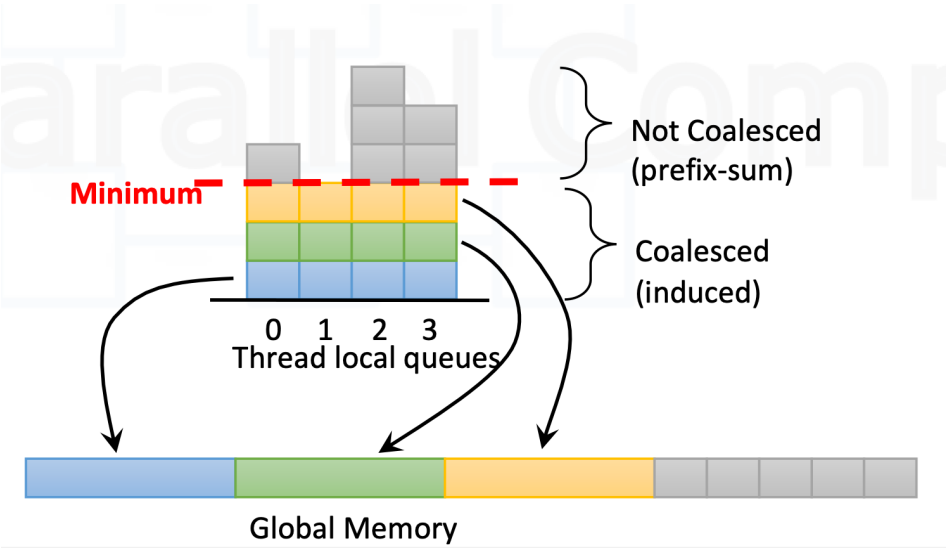


Figura 8.4.1: Coalescenza di lettura/scrittura

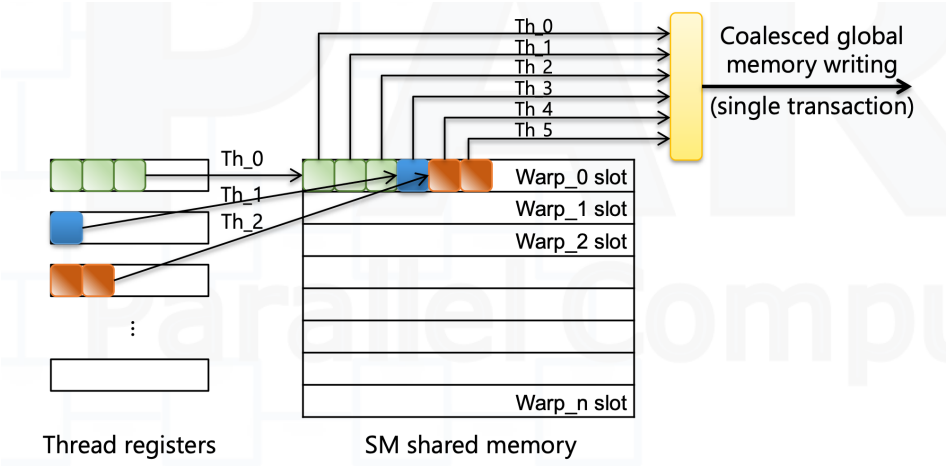


Figura 8.4.2: Coalescenza di lettura/scrittura

Capitolo 9

MPI - *Message Passing Interface*

9.1 MPI

MPI (*Message Passing Interface*) è una libreria di comunicazione per la programmazione parallela su sistemi distribuiti. MPI fornisce Si utilizza quando non è presente una memoria condivisa tra i processi, in modo che i processi possano comunicare tra loro tramite messaggi.

9.1.1 Introduzione

MPI è una libreria di comunicazione per la programmazione parallela su sistemi distribuiti. MPI fornisce un'interfaccia standard per la comunicazione tra processi, consentendo a programmi paralleli di scambiare messaggi e sincronizzarsi in modo efficiente.

La primitiva più semplice di comunicazione in MPI è la MPI_Send e la MPI_Recv, che consentono a due processi di scambiare messaggi in modo sincrono. Queste primitive possono essere utilizzate per implementare algoritmi di comunicazione più complessi, come la raccolta di dati, la distribuzione di lavoro e la sincronizzazione tra processi.

Gli obiettivi di MPI includono:

- **Portabilità:** MPI è progettato per funzionare su una vasta gamma di architetture di calcolo parallelo, inclusi cluster, supercomputer e sistemi di calcolo distribuito.
- **Scalabilità:** MPI è progettato per supportare applicazioni parallele di grandi dimensioni con migliaia o milioni di processi, consentendo una crescita flessibile delle risorse di calcolo.
- **Portabilità software:** MPI fornisce un'interfaccia standard per la comunicazione tra processi, consentendo ai programmatori di scrivere codice parallelo che può essere eseguito su diverse piattaforme senza modifiche. L'interfaccia è stata definita per C/C++, Fortran, Python e altri linguaggi di programmazione.
- **Portabilità hardware:** MPI è progettato per funzionare su una vasta gamma di hardware.

Struttura di un programma MPI

```
#include "mpi.h"

// Inizializzazione dell'ambiente MPI

// Corpo del programma, il programma che viene
// manualmente parallelizzato

// Terminazione dell'ambiente MPI
```

Per garantire la portabilità i tipi di dati in MPI sono definiti come MPI_Datatype.

9.1.2 Comunicazione e rango

MPI utilizza oggetti chiamati *communicators* per gestire la comunicazione tra processi. Un communicator è un gruppo di processi che possono scambiare messaggi tra loro. Ogni processo in un communicator è identificato da un numero intero univoco chiamato *rank*.

Se non viene specificato un communicator, MPI utilizza il communicator predefinito MPI::COMM_WORLD, che include tutti i processi in esecuzione.

I *rank* vengono assegnati in modo sequenziale, partendo da 0 per il primo processo, 1 per il secondo e così via.

9.1.3 Funzioni chiave di MPI

Inizializzazione e terminazione

```
MPI::Init(&argc, &argv);
```

Rank e dimensione del communicator

```
int MPI::COMM::Get_rank();
int MPI::COMM::Get_size();
```

Nome del processore

```
MPI::Get_processor_name(char* name, int& len);
```

Terminazione dell'ambiente MPI

```
MPI::Finalize();
```

```
MPI::COMM::Abort(int errorcode);
```

Comunicazione

```
bool MPI::Is_initialized();
```

Minutaggio e temporizzazione

```
double MPI::Wtime();
```

```
double MPI::Wtick();
```

9.1.4 Hello World in MPI

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

int main(int argc, char** argv) {
  // Inizializzazione dell'ambiente MPI
  MPI::Init(argc, argv);
  // Ottenere il numero di processi e il rank del processo
  int world_size = MPI::COMM_WORLD.Get_size();
  int world_rank = MPI::COMM_WORLD.Get_rank();
  // Ottenere il nome del processore
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI::Get_processor_name(processor_name, name_len);
  // Stampa del messaggio di saluto
  std::cout << "Hello from processor " << processor_name << ", rank " <<
  world_rank << " out of " << world_size << " processors" << std::endl;
  // Terminazione dell'ambiente MPI
  MPI::Finalize();
  return 0;
}
```

9.1.5 Applicazioni e buffer di comunicazione

Un *system buffer* è un'area di memoria utilizzata per memorizzare i dati trasmessi tra i processi. Questo buffer può essere utilizzato per inviare e ricevere messaggi tra i processi, consentendo la comunicazione e la sincronizzazione tra i processi.

Lo spazio di indirizzamento è gestito dall'utente.

Una `SEND` bloccante ritorna il risultato solo dopo che la modifica dell'application buffer è stata completata in modo corretto. Ciò significa che il processo mittente si sblocca quando il processo destinatario ha completato la ricezione del messaggio. Quando il processo invoca una `RECIVE` bloccante, il processo si blocca fino a quando il messaggio non è stato ricevuto correttamente. Se la chiamata di ricezione viene eseguita prima che il messaggio sia stato inviato, il processo si blocca fino a quando il messaggio non è stato inviato.

Quando un processo invia una sequenza di messaggi, il processo destinatario riceve i messaggi nello stesso ordine in cui sono stati inviati.

`MPI` non garantisce il principio di equità, il che significa che non garantisce che i messaggi vengano ricevuti nell'ordine in cui sono stati inviati.

9.1.6 Comunicazione bloccante

```
MPI::COMM::Send(void* buf, int count, MPI::Datatype datatype& datatype, int
  dest, int tag);
```

```
MPI::COMM::Recv(void* buf, int count, MPI::Datatype datatype& datatype, int
  source, int tag, MPI::Status& status);
```

```
MPI::COMM::Sendrecv(void* sendbuf, int sendcount,
          MPI::Datatype sendtype, int dest, int sendtag,v
          oid* recvbuf, int recvcount, MPI::Datatype recvtype,
          int source, int recvtag, MPI::Status& status);
```

```
MPI::COMM::Ssend(void* buf, int count, MPI::Datatype datatype, int dest, int
   tag);
```

```
MPI::COMM::Rsend(void* buf, int count, MPI::Datatype datatype, int dest, int
   tag);
```

9.1.7 Comunicazione non bloccante

```
MPI::COMM::Isend(void* buf, int count, MPI::Datatype datatype, int dest, int
   tag);
```

```
MPI::COMM::Irecv(void* buf, int count, MPI::Datatype datatype, int source,
  int tag);
```

```
MPI::COMM::Irsend(void* buf, int count, MPI::Datatype datatype, int dest,
  int tag);
```

```
MPI::COMM::Issend(void* buf, int count, MPI::Datatype datatype, int dest,
  int tag);
```