SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" Università di Roma
Ingegneria dell'Informazione,
Informatica e Statistica
Dipartimento di Informatica

Programmazione di Sistemi Embedded e Multicore

Appunti integrati con il libro "An Introduction to Parallel Programming", Peter Pacheco

Author
Simone Bianco

8 dicembre 2023

Indice

Informazioni e Contatti

Appunti e riassunti personali raccolti in ambito del corso di *Programmazione di Sistemi Embedded e Multicore* offerto dal corso di laurea in Informatica dell'Università degli Studi di Roma "La Sapienza".

Ulteriori informazioni ed appunti possono essere trovati al seguente link: **https://github.com/Exyss/university-notes**. Chiunque si senta libero di segnalare incorrettezze, migliorie o richieste tramite il sistema di Issues fornito da GitHub stesso o contattando in privato l'autore :

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

Gli appunti sono in continuo aggiornamento, pertanto, previa segnalazione, si prega di controllare se le modifiche siano già state apportate nella versione più recente.

Prerequisiti consigliati per lo studio:

Apprendimento del materiale relativo al corso *Sistemi Operativi II*.

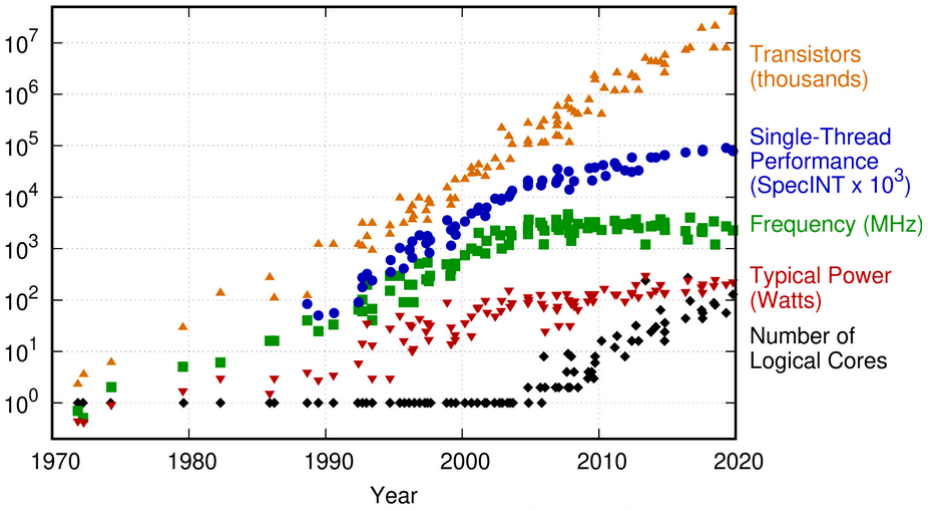
Licence:

1

Introduzione al Parallelismo

Dal 1986 al 2002, la velocità dei microprocessori è salita notevolmente, aumentando le prestazioni per una media del 50% annuo. Tuttavia, da tale periodo l'aumento è sceso a circa il 20% annuo.

Nonostante la famosa **legge di Moore**, la quale stabilisce che il numero di transistor nei circuiti integrati raddoppi circa ogni due anni, si riveli tuttora vera, le altre statistiche legate ai microprocessori non seguono tale trend:



Affinché il numero di transistor possa incrementare senza aumentare la superficie su cui posizionarli, nel corso degli anni sono state notevolmente ridotte le loro dimensioni. Procedendo in tal modo, tuttavia, risulta evidente il **limite fisico** di tale procedura: transistor più piccoli implicando processori più veloci, aumentando la quantità di energia consumata e di conseguenza anche il calore generato, portando i processori stessi ad essere inconsistenti.

Per tanto, non potendo più diminuire le dimensioni dei transistor e non volendo aumentare le dimensioni dei circuiti, l'unica soluzione risulta essere l'uso di **sistemi con più processori** in grado di comunicare tra di loro.

L'uso di architetture basate su più processori, tuttavia, richiedono la **conversione di programmi seriali in programmi paralleli**. Poiché alcuni costrutti di codice possono essere automaticamente riconosciuti e parallelizzati,inizialmente si provò ad effettuare la transizione tramite programmi di conversione automatica, i quali tuttavia generavano programmi inefficienti e poco ottimizzati. La soluzione migliore, dunque, risulta essere la **riscrittura** completa dei programmi basandosi su un **approccio parallelo**.

Supponiamo di voler calcolare n valori tramite una particolare funzione, per poi sommare tra di loro i valori ottenuti.

Un **approccio seriale** prevede la seguente soluzione banale:

```
sum = 0;
for(i = 0; i < n; i++){
    x = compute_next_value(...);
    sum += x;
}
```

richiedendo quindi l'uso di un singolo core per essere eseguito

Un **approccio parallelo**, invece, prevede l'uso di p core (dove $p < n$), assegnando ad ogni core un insieme di $\frac{n}{p}$ valori da calcolare. Per tanto, ogni core eseguirà il seguente frammento di codice:

```
my_sum = 0;
my_first_i = ...;
my_last_i = ...;
for(my_i = my_first_i; i < my_last_i; i++){
    my_x = compute_next_value(...);
    my_sum += my_x;
}
```

Osservazione 1: Variabili locali

Ogni variabile è **locale per ogni core**, ossia ogni core possiede una propria copia indipendente dalle altre su cui poter lavorare

Una volta che ogni core avrà terminato la computazione, le p somme parziali verranno sommate dal **master core**, ossia il core designato come principale:

```
if(I'm teh master core){
    sum = my_x;
    for each core other than myself{
        sum += value received from that core;
    }
}
else{
    send my_x to the master core;
}
```

In tal modo, ogni core secondario svolgerà $\frac{n}{p}$ somme, mentre il master core svolgera $\frac{n}{p} + p$ somme poiché deve sommare le p somme parziali al proprio risultato.

Notiamo quindi che tale soluzione presenta ancora alcune **inefficienze**, poiché solo il master core lavora durante la somma finale, mentre tutti gli altri core rimangono inutilizzati. La soluzione risulta ovvia: **distribuire** nuovamente il carico di lavoro.

Metodo 1: Computazione ad albero

Dato un problema di calcolo con p valori da eseguire su p core, la computazione parallela può essere sviluppata tramite il seguente algoritmo:

1. Ad ogni core viene assegnato un indice, dove 0 è il master core
2. Viene posto $k = 1$
3. Ogni processore con indice i multiplo di k somma il proprio valore calcolato x_i al valore x_j calcolato dal successivo core di indice j multiplo di k (dunque $x_i + x_j$)
4. Viene incrementato k, tornando al passo precedente finché la computazione non verrà svolta da un singolo core

In tal modo, ogni core svolgerà un massimo di $\frac{n}{p} + \log_2(p)$ somme parallele

Esempio:

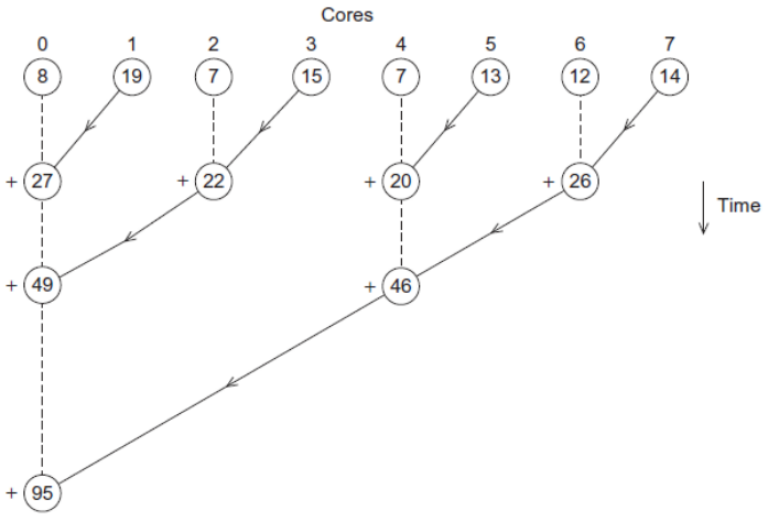
- Consideriamo il seguente insieme di $n = 24$ valori da sommare tra di loro:

$$1, 4, 3, 9, 2, 8, 5, 1, 1, 5, 2, 7, 2, 5, 0, 4, 1, 8, 6, 5, 1, 2, 3, 9$$

- Avendo $p = 8$ core, distribuiamo $\frac{n}{p} = \frac{24}{8} = 3$ valori a ciascun core:

$$1, 4, 3 \quad 9, 2, 8 \quad 5, 1, 1 \quad 5, 2, 7 \quad 2, 5, 0 \quad 4, 1, 8 \quad 6, 5, 1 \quad 2, 3, 9$$

- Una volta calcolate le somme parziali da parte di ogni core, la computazione verrà sviluppata ad albero:



Definizione 1: Task parallelism e Data parallelism

Dato un carico di lavoro, definiamo come:

- **Task parallelism** una distribuzione del carico basata sulla suddivisione dei compiti da svolgere
- **Data parallelism** una distribuzione del carico basata sulla suddivisione dei dati su cui lavorare

Esempio:

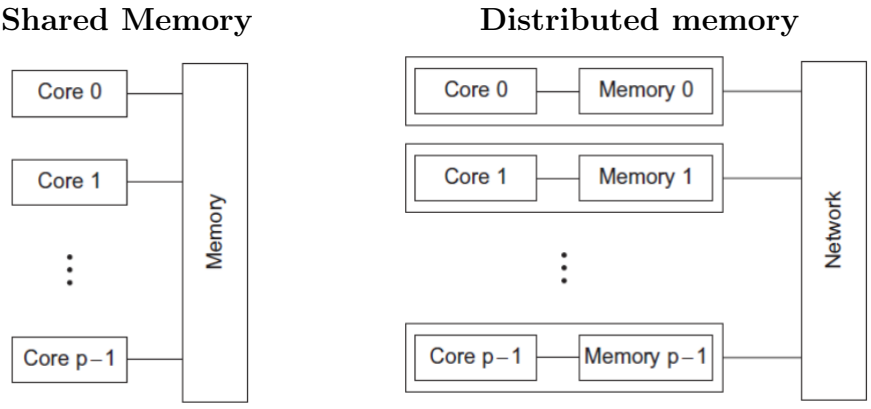
- Tre professori vogliono suddividersi la valutazione di 300 consegne di un esame composto da 15 domande.
- Una suddivisione basata sul task parallelism prevede che ogni professore corregga tutte le 300 consegne, ma che il primo professore corregga solo le domande dalla 1 alla 5, il secondo dalla 6 alla 10 e il terzo dalla 11 alla 15
- Una suddivisione basata sul task parallelism prevede che ogni professore corregga tutte le 15 domande, ma che ogni professore debba correggere solo 100 consegne

Definizione 2: Shared Memory System

Definiamo come **shared memory system** un sistema in cui i core possono condividere l'accesso alla memoria del computer, richiedendo coordinazione per svolgere operazioni di lettura e scrittura sul contenuto condiviso

Definizione 3: Distributed Memory System

Definiamo come **distributed memory system** un sistema in cui ogni core possiede la propria memoria, richiedendo uno scambio di messaggi tramite una rete di interconnessione per effettuare il passaggio di dati



1.1 Parallelismo a livello hardware

1.1.1 Cache, memoria virtuale e TLB

Le moderne architetture prevedono componenti hardware aggiuntivi rispetto all'architettura minimale basata sul **modello di Von Neumann**, il quale ricordiamo essere basato sulla sola presenza di una CPU costituita da una Control Unit (CU) ed una Arithmetic Logic Unit (ALU), una memoria principale e periferiche di I/O.

In particolare, le moderne CPU sono dotate di piccole **cache**, tipicamente situate sullo stesso chip, le quali sono accessibili più velocemente rispetto alla memoria principale.

Definizione 4: Caching

Definiamo come **caching** l'uso di una collezione di locazioni di memoria accessibili più velocemente rispetto ad altre locazioni, le quali possono essere all'interno della stessa o di una diversa memoria

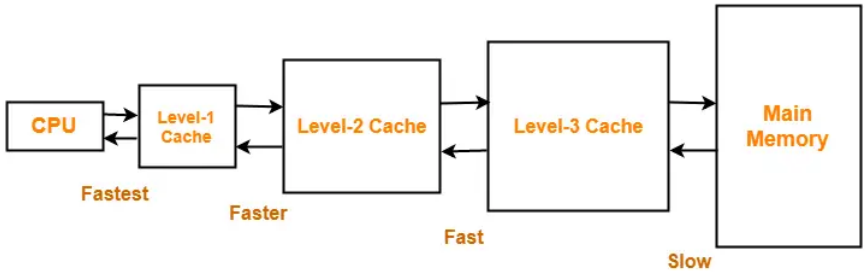
L'uso del caching genera un alto impatto sulle performance delle moderne architetture per via del **principio di località**.

Principio 1: Principio di località

Il **principio di località** afferma che durante l'esecuzione di un qualsiasi programma prevalgono le due seguenti proprietà:

- **Località spaziale**: gli accessi ad una locazione di memoria sono per lo più seguiti da accessi a locazioni ad essa adiacenti
- **Località temporale**: una locazione di memoria viene solitamente acceduta ripetute volte solo nel breve periodo di tempo, per poi non venir più acceduta durante l'esecuzione del programma

Per alleviare il **bottleneck** (*collo di bottiglia*) tra cache e memoria principale, viene utilizzata una **gerarchia di cache**, dove ogni sottolivello è costituito da una memoria più lenta ma anche più capiente rispetto alla precedente, fino a raggiungere la memoria principale stessa. Solitamente, l'ultima cache prima della memoria principale viene detta **Last Level Cache (LLC)**.



Essendo più piccole della memoria principale, ogni cache può gestire solo una **limitata quantità di dati**: quando un dato richiesto da parte di una CPU viene trovato all'interno della sua cache, esso viene immediatamente restituito (**cache hit**). In caso contrario, tale dato verrà cercato nella memoria di livello direttamente inferiore (**cache miss**), finché esso non verrà eventualmente trovato (nel caso peggiore verrà prelevato direttamente dalla memoria principale).

Per mantenere i propri dati, ogni cache è dotata di m **linee**, ognuna composta da n **set** all'interno dei quali vengono conservati byte di dati. Le mappature blocco-set vengono gestite tramite una delle seguenti modalità:

- **Direct-mapped cache**: la cache è dotata di m linee ciascuna con un singolo set
- n**-way set associative cache**: la cache è dotata di m linee ciascuna con m set
- **Fully-associative cache**: la cache è dotata di una sola linea contenente n set

In base alla mappatura utilizzata, ogni indice della memoria (corrispondete ad un byte) viene mappato ad una determinata linea. Tuttavia, essendo le linee limitate, più indici verranno **mappati sulla stessa linea**.

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

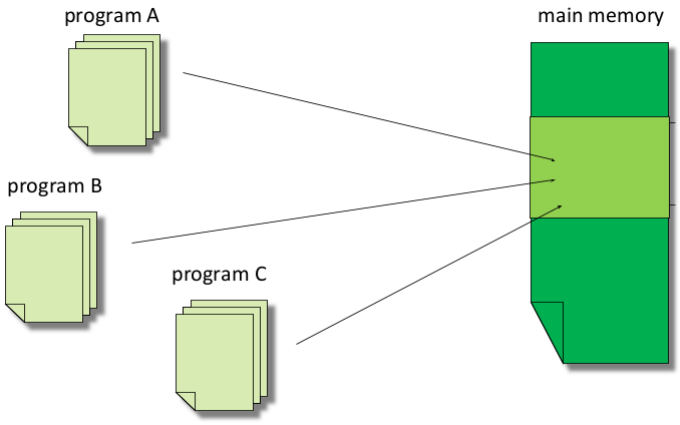
Per tanto, quando una linea non avrà più posti disponibili, sarà necessario **rimpiazzare** il dato contenuto in uno dei suoi set. Solitamente, viene rimpiazzato quello utilizzato meno recentemente (**Least Recently Used Policy - LRU**) o quello utilizzato di meno in generale (**Least Frequently Used Policy - LFU**).

Inoltre, quando vengono scritti dati su una cache, tali dati potrebbero essere **inconsistenti** con quelli della memoria principale. Per gestire tale dinamica, le due strategie più comuni sono:

- **Write-through**: il dato viene immediatamente scritto sulla memoria principale
- **Write-back**: il dato viene marcato come **dirty** tramite un bit speciale. Quando il set della cache contenente tale dato verrà rimpiazzato, il dato verrà prima scritto sulla memoria

Eseguendo un programma di grandi dimensioni o che accede a grandi insiemi di dati, tutte le istruzioni e i dati potrebbero non entrare all'interno della memoria principale.

Con **memoria virtuale** intendiamo una modalità di gestione della memoria basata sulla tecnica del **paging**, ossia la suddivisione della memoria principale in **frame** e della memoria di ogni programma in **pagine**, dove frame e pagine sono blocchi di byte della stessa dimensione. Le pagine di un programma possono essere **inattive**, ossia conservate sullo **swap space** della memoria secondaria, o **attive**, ossia attualmente caricate in memoria ed associate ad un frame. In particolare, ogni frame può conservare una sola pagina per volta.



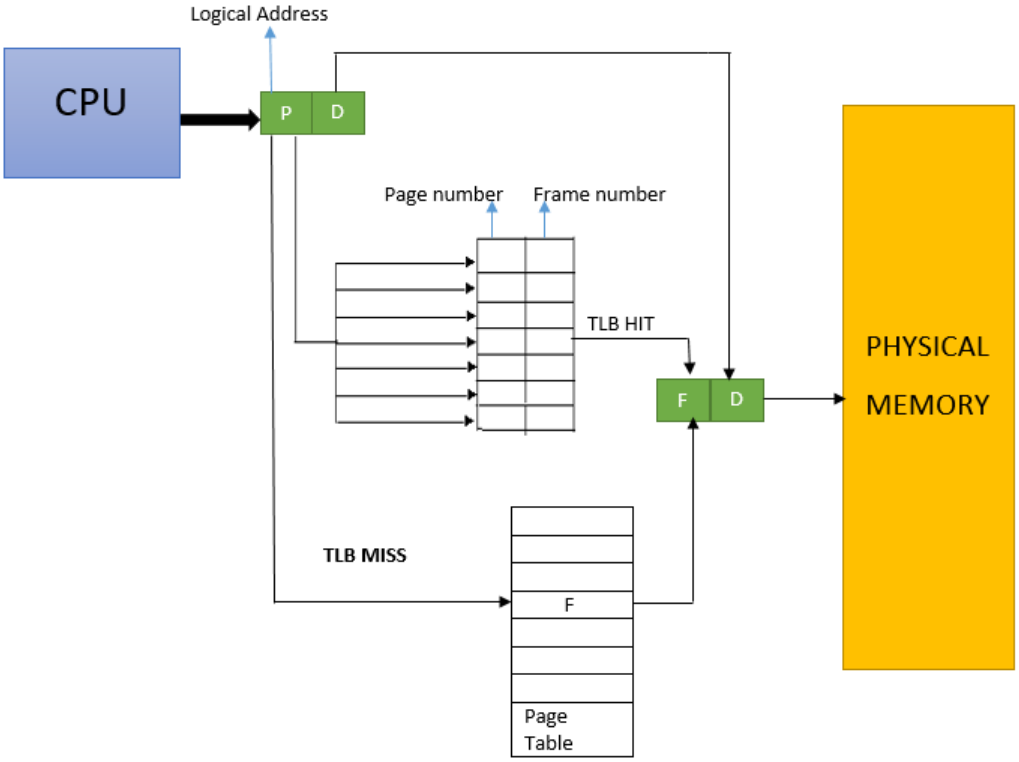
Quando un programma viene compilato, alle sue pagine vengono assegnati dei **virtual page number**, tramite cui vengono generati i **virtual address** di istruzioni e dati. Una volta che tale programma verrà eseguito verrà creata una sua **page table** all'interno della quale vengono conservate le **traduzioni** tra indirizzi virtuali e fisici. Nel caso i cui si tenti di accedere all'indirizzo fisico di una pagina attualmente non presente sulla memoria principale (**page fault**), essa verrà prelevata dal disco e associata ad un frame libero (rimpiazzandone uno se necessario).

Nonostante l'uso della memoria virtuale migliori notevolmente le prestazioni, le page table vengono comunque conservate all'interno della memoria principale, rendendo lento il processo di traduzione. Pertanto, viene utilizzata una **Translation Lookaside Buffer (TLB)**, ossia una cache speciale addetta solo al salvataggio di poche entrate delle varie page table.

Per gestire i **TLB miss**, vengono utilizzate due strategie principali:

- **Hardware managed**: a seguito del miss, la CPU scorre automaticamente la page table cercando la pagina richiesta
- **Software manager**: a seguito del miss, viene generata una TLB miss exception e il sistema operativo si occupa di scorrere la page table non appena possibile

Se la pagina richiesta non viene trovata neanche in page table, essa verrà prelevata dallo swap space per poi venire salvata in TLB, generando anche una page fault exception.



1.1.2 Parallelismo nelle istruzioni e multi-threading

Definizione 5: Instruction level parallelism (ILP)

L'**instruction level parallelism (ILP)** prevede l'utilizzo di CPU composte da più unità funzionali in grado di eseguire simultaneamente istruzioni diverse, permettendo di ottenere parallelismo all'interno delle CPU stesse.

La prima modalità di implementazione dell'ILP prevede l'uso del **pipelining** (*catena di montaggio*), ossia la suddivisione dell'esecuzione di una singola istruzione in più **fasi**, dove ogni fase è eseguita da un componente a se stante, permettendo quindi l'esecuzione simultanea di più istruzioni.

Esempio:

- Supponiamo di voler eseguire il seguente frammento di codice:

```
...
float x[1000], y[1000], z[1000];
...

for(i = 0; i < 1000; ++i){
    z[i] = x[i] + y[i];
}
```

- La somma tra due numeri float è suddivisibile nelle seguenti 7 fasi. Ad esempio, la somma dei numeri $9.87 \cdot 10^4$ e $6.54 \cdot 10^3$ viene svolta nel seguente modo:

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

- Assumendo che ognuna delle 7 fasi impieghi 1 ns, il tempo totale impiegato dal loop corrisponderebbe a 7000 ns
- Suddividendo il floating point adder (ossia l'adder della CPU addetto a tale compito) in 7 unità funzionali a cui vengono associate le 7 fasi della somma, tali somme verrebbero svolte simultaneamente:

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

  richiedendo quindi solo 1005 ns per eseguire l'intero loop

La seconda modalità di implementazione, invece, prevede l'uso del **multiple issue**, ossia l'esecuzione sincrona di istruzioni dello stesso tipo su copie diverse della stessa unità funzionale (es: con due floating point adder è possibile eseguire due somme float contemporaneamente). Per utilizzare tale modalità è richiesto uno **scheduling** delle unità funzionali da far utilizzare alle istruzioni. In particolare, lo **static multiple issue** prevede che le unità funzionali vengano schedulate al tempo di compilazione, mentre il **dynamic multiple issue** prevede che ciò avvenga in fase di esecuzione stessa.

Per poter schedulare correttamente le istruzioni, il compilatore (o la CPU nel caso del dynamic multiple issue) deve essere in grado di **speculare** su quali istruzioni possano essere avviate simultaneamente. In caso di speculazione errata, l'esecuzione viene terminata e rieseguita correttamente.

Per quanto riguarda il multithreading, invece, alcune volte risulta impossibile eseguire simultaneamente thread diversi. Per tale motivo, viene implementato del **multithreading a livello hardware** in grado di permettere ad un sistema di continuare a svolgere lavoro quando una task attualmente in esecuzione rimane in attesa (**stalled**):

- **Fine-grained hardware multithreading** (*a grana fine*): il processore alterna i thread ad ogni istruzione, saltando quelli in attesa. In tal modo viene generalmente ridotto il tempo sprecato a seguito degli stalli, ma un thread che deve eseguire molte istruzioni richiederà più tempo per terminare il lavoro
- **Coarse-grained hardware multithreading** (*a grana grossa*): il processore alterna solo i thread che sono in attesa, permettendo agli altri di continuare l'esecuzione. In tal modo viene generalmente ridotta la quantità di switch tra i thread, ma il processore potrebbe rimanere inattivo durante stalli brevi per via del continuo switch tra thread inattivi
- **Simultaneous hardware multithreading**: variante del fine-grained dove viene permesso ai thread di utilizzare molteplici unità funzionali

1.1.3 Tassonomia di Flynn

La **tassonomia di Flynn** è un sistema di classificazione delle architetture degli elaboratori a seconda della **molteplicità** del flusso di **istruzioni** e del flusso dei **dati** che possono gestire:

- **Single Instruction stream with Single Data stream (SISD)**: vi è un solo flusso di istruzioni che opera su un solo flusso di dati. Esempio tipico di tale architettura è il modello di Von Neumann.
- **Single Instruction stream with Multiple Data stream (SIMD)**: vi è un solo flusso di istruzioni che opera su più flussi di dati in parallelo. Tale architettura è molto utilizzata nei sistemi moderni richiedenti la manipolazione di grandi quantità di dati.
- **Multiple Instruction stream with Single Data stream (MISD)**: vi sono molteplici flussi di istruzioni che operano in parallelo sullo stesso flusso di dati. Tale architettura viene utilizzata molto di rado in quanto estremamente situazionale.
- **Multiple Instruction stream with Multiple Data stream (MIMD)**: vi sono molteplici flussi di istruzioni che operano in parallelo sullo molteplici flussi di dati. I moderni processori multicore rientrano in tale categoria.

All'interno delle architetture SIMD, il **parallelismo** viene ottenuto suddividendo i flussi di dati tra i vari processori, dove ognuno di essi applicherà la stessa istruzione (**data parallelism**). Visualmente, è possibile immaginare l'esecuzione di istruzioni sulle architetture SIMD come un macchinario industriale di una fabbrica: lo stesso compito viene eseguito su n oggetti contemporaneamente.

Esempio:

- Consideriamo il seguente codice

```
...
for(i = 0; i < n; ++i){
    x[i] += y[i];
}
...
```

- Supponendo di avere m core, abbiamo che:
  - Se $m \geq n$ allora ogni somma di tale loop viene assegnata ad un core, per poi venir eseguite tutte parallelamente
  - Se $m < n$ allora i dati su cui svolgere le somme vengono assegnate iterativamente, eseguendo quindi m somme parallele per poi ripetere la fase di assegnamento iterativo

Nonostante la loro semplicità, le architetture SIMD presentano alcuni **svantaggi**:

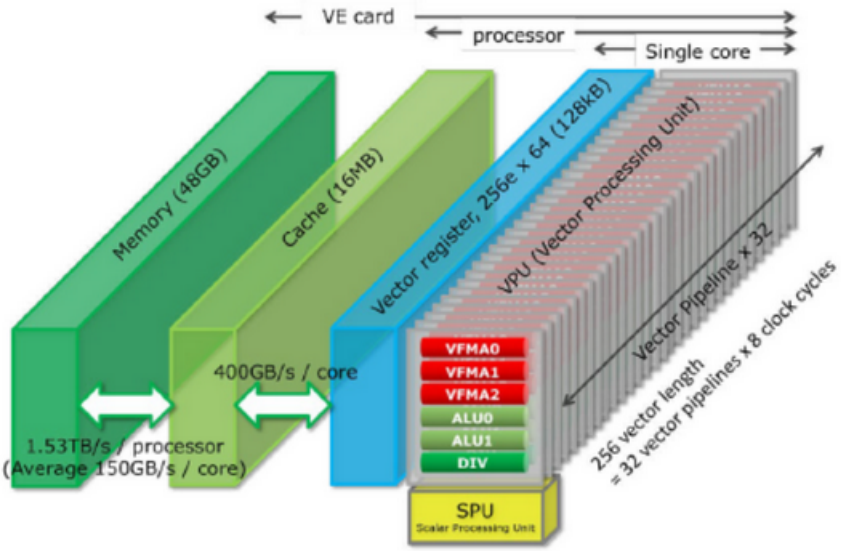
- Poiché i core devono eseguire la stessa istruzione degli altri core, ogni ALU deve scegliere se eseguire l'istruzione attuale o attendere la prossima
- Nei design classici è richiesto che le ALU siano sincronizzate tra loro
- Molto efficiente solo per programmi paralleli con molti dati ed istruzioni semplici

Esempi tipici di sistemi basati su architettura SIMD sono i sistemi dotati di **processori vettoriali**, i quali operano su **array** (detti *vettori*) **di dati** invece che su singoli elementi (detti *scalari*). Sono dotati di:

- **Registri vettoriali** capaci di contenere vettori di scalari ed operare simultaneamente su di essi
- **Unità funzionali vettoriali** capaci di applicare la stessa operazione ad ogni scalare dei vettori su cui si sta lavorando
- **Memoria interlacciata** composta da molteplici banchi di memoria, i quali possono essere acceduti più o meno indipendentemente. Gli elementi dei vettori vengono distribuiti tra i banchi, riducendo o persino eliminando i ritardi dovuti a load/store di elementi successivi
- **Accesso progressivo alla memoria (hardware scatter/gather)**, ossia la possibilità di accedere agli elementi di un vettore solo ad intervalli fissi

Grazie ai **compilatori vettoriali**, i quali ottimizzano il codice e forniscono informazioni sulle parti di codice non vettorializzabili, l'uso dei processori vettoriali risulta veloce e semplice. Inoltre, la loro natura stessa favorisce notevolmente l'uso delle **cache**, in quanto i vettori sono contigui.

Tuttavia, quest'ultimi non sono in grado di gestire **strutture dati irregolari**, ossia non contigue (es: liste, alberi, ...), e possiedono **scarsa scalabilità** per problemi di dimensioni più grosse.



Schematizzazione del processore vettoriale Aurora NEC

Basate sull'architettura SIMD sono anche molte **moderne GPU**. Difatti, rispetto quest'ultime sono improntate **Throughput Oriente Design**:

- Molteplici piccoli core e dotati di piccole cache
- Assenza di branch prediction e di data forwarding
- Una grande quantità di ALU ma energeticamente efficienti (dunque più deboli) e richiedenti molteplici e lunghe latenze, compensate tramite un uso estensivo del pipelining
- Richiedono un enorme numero di thread per gestire le latenze

Le **normali CPU** (dunque del modello SISD), invece, sono improntate ad un **Latency Oriented Design**:

- Una sola CPU dotata di cache di grandi dimensioni
- Presenza di branch prediction e data forwarding
- ALU potenti e richiedenti meno latenze

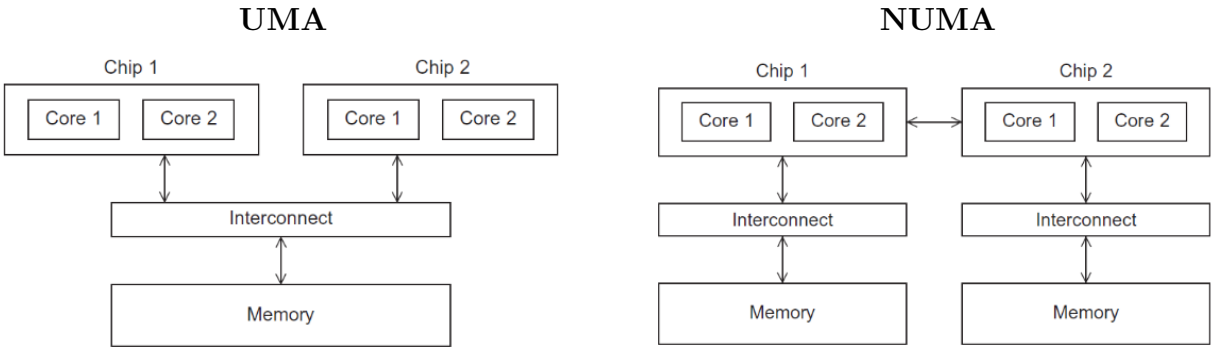
All'interno delle architetture MIMD, il **parallelismo** viene ottenuto suddividendo i flussi di dati tra vari processori i quali, a differenza dell'architettura SIMD, sono **completamente indipendenti** l'uno dall'altro, permettendo l'esecuzione contemporanea di flussi di istruzioni diverse.

In particolare, possiamo individuare due **sottocategorie** di architetture MIMD:

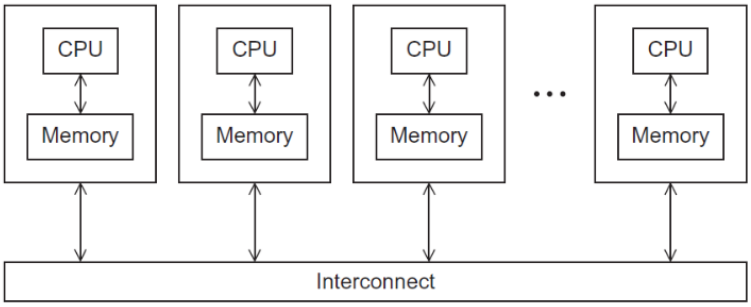
- **Sistemi a memoria condivisa**: sistemi composti da collezioni di CPU autonome connesse ad un sistema di memoria tramite una rete di interconnessione. Ogni core può accedere ad ogni locazione di memoria, comunicando con gli altri core tramite strutture dati condivise.

  Le architetture di tale categoria si suddividono a loro volta in:

  - **Uniform memory access (UMA)**, dove vi è un'unica memoria condivisa uniformemente da tutti i core
  - **Non-uniform memory access (NUMA)**, dove vi sono più memorie ciascuna accessibile solo da un determinato gruppo di core

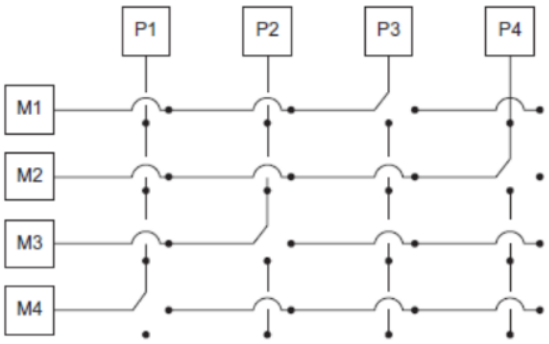


- **Sistemi a memoria distribuita**: sistemi composti da collezioni di CPU autonome dotate ognuna del proprio sistema di memoria. Ogni core può lavorare solo su dati locali e, se necessario, può scambiare dati con gli altri core tramite una rete di interconnessione. Tra i più comuni sistemi a memoria distribuita troviamo i **cluster**.



Nel caso dei **sistemi a memoria condivisa**, le reti di interconnessione possono essere realizzate tramite:

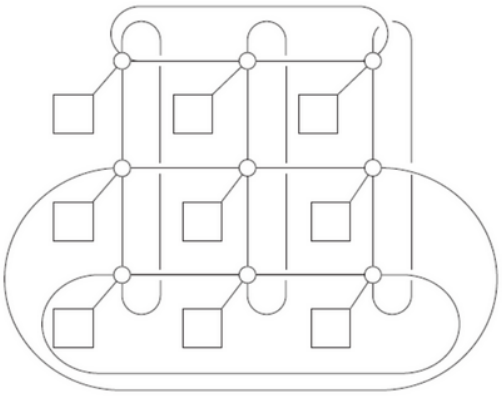
- **Bus di interconnessione**, ossia una collezione di fili di comunicazione parallela controllati da hardware e condivisi dai dispositivi a loro connessi. Il bus può essere acceduto solo da un dispositivo alla volta; dunque maggiore è il numero dei dispositivi connessi, maggiore è la contesa per il bus.
- **Switch di interconnessione**, ossia una collezione di interruttori che controllano l'instradamento dei dati tra dispositivi connessi (**crossbar**)



Esempio di crossbar

Per quanto riguarda i **sistemi a memoria distribuita**, invece, le reti di interconnessione possono essere realizzate tramite:

- **Interconnessioni dirette**, dove ogni interruttore è direttamente connesso ad una coppia core-memoria e ad ogni altro interruttore
- **Interconnessioni indirette**, dove gli interruttori potrebbero non essere direttamente connessi



Esempio di maglia toroidale (interconnessione diretta)

1.1.4 Coerenza tra le cache

Nel caso in cui vi siano più core sullo stesso sistema ed ognuno di essi sia dotato di cache propria, tali cache devono essere in grado di mantenere **coerenti** tra loro i valori dei **dati condivisi**, ossia evitare situazioni in cui il valore di un dato sia stato modificato in una cache ma non nelle altre, portando i core non aggiornati ad utilizzare il valore sbagliato.

Tra le tecniche più utilizzate per mantenere le cache coerenti tra loro troviamo:

- **Snooping cache coherence**, dove i core condividono un bus tramite quale possono segnalare un aggiornamento dei valori a tutti i core, i quali ascoltano passivamente il bus (**snooping**) ed aggiornano il valore una volta rilevato il segnale
- **Directory based cache coherence**, dove viene utilizzata una struttura dati condivisa detta **directory** all'interno della quale vengono conservati gli stati di ogni linea. Quando una variabile viene aggiornata, la directory viene consultata e i controller delle cache che rilevano tale variabile come presente in una delle proprie linee rendono tali linee invalide.

Definizione 6: False sharing

Definiamo come **false sharing** una situazione in cui un core accede periodicamente a dati che **non vengono modificati da un altro core**, ma che sono presenti all'interno dello **stesso blocco di cache** di altri dati che invece vengono modificati da altri core.

In tal caso, il protocollo di coerenza tra le cache può forzare il primo core a ricaricare costantemente l'intero blocco di cache nonostante la mancanza di necessità, portando ad un **notevole degrado delle prestazioni**.

Esempio:

- Consideriamo il seguente frammento di codice:

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count;
double y[m] = {...};
iter_count = m/core_count;

/* Core 0 does this */
for (i = 0; i < iter_count; i++){
    for (j = 0; j < n; j++){
        y[i] += f(i,j);
    }
}
```

```
/* Core 1 does this */
for (i = iter count+1; i < 2*iter count; i++){
    for (j = 0; j < n; j++){
        y[i] += f(i,j);
    }
}
```

- Sebbene i due core lavorino su elementi diversi dell'array y, nel momento in cui ognuno di tali elementi venga caricato assieme ad esso verrà caricata anche la sua porzione di memoria adiacente (ossia il blocco che lo contiene).
- Di conseguenza, alcuni elementi dell'array faranno parte dello stesso blocco di memoria, implicando che ogni volta che una delle due cache dei core marchi tale blocco come invalido, l'altra cache sarà costretta a ricaricare l'intero blocco al fine di mantenere coerenza tra i valori
- Tuttavia, tali ricaricamenti sono in realtà innecessari, poiché come già discusso i due core lavorano in realtà su elementi del tutto diversi
- Per ridurre il fenomeno del false sharing, in tale situazione è sufficiente utilizzare due variabili temporanee che possano accumulare la somma, in modo che le due cache vadano ad aggiornare i valori contigui il minor numero di volte possibile:

```
...

/* Private variable */
int sum;

/* Core 0 does this */
for (i = 0; i < iter_count; i++){
    sum = y[i];
    for (j = 0; j < n; j++){
        sum += f(i,j);
    }
    y[i] = sum;
}

/* Core 1 does this */
for (i = iter count+1; i < 2*iter count; i++){
    sum = y[i];
    for (j = 0; j < n; j++){
        sum += f(i,j);
    }
    y[i] = sum;
}
```

1.2 Parallelismo a livello software

1.2.1 Concetti generali

Come discusso, hardware e compilatori riescono a "tenere il passo" con i cambiamenti previsti dalle moderne architetture.

Per quanto riguarda il software, in generale la procedura di realizzazione di un programma parallelo nei **sistemi a memoria condivisa** consiste nell'avvio di un processo, il quale poi creerà dei thread ai quali suddividere le task da svolgere, mentre nei **sistemi a memoria distribuita** richiede l'avvio di processi multipli, i quali svolgono in prima persona le task, suddividendole.

In seguito, indicheremo come **thread 0** il primo thread creato all'avvio del programma stesso in un sistema a memoria condivisa, mentre indicheremo come **processo 0** il primo processo creato da un programma in un sistema a memoria distribuita.

Molti software paralleli sono basati sul modello **Single Program Multiple Data (SPMD)**, ossia la scrittura di un singolo eseguibile in gradi di comportarsi come se fosse più programmi tramite l'uso di branch condizionali. Idealmente, ogni branch è indipendente dagli altri e viene eseguito da un singolo thread o processo figlio.

```
void main(){
    pid_t pid = fork();

    if(pid = 0){    //child process
        ...
    }
    else{   //original process
        ...
    }
}
```

Nei **sistemi a memoria condivisa**, i thread possono essere gestiti tramite due modalità:

- **Thread dinamici**: un master thread (o thread 0) attende del lavoro da svolgere, creando nuovi thread quando richiesto e terminandoli una volta che il lavoro sia concluso.

  Tale modalità fa un uso efficiente delle risorse del sistema, ma la creazione e terminazione continua di thread risulta essere dispendioso a livello di tempo

- **Thread statici**: vengono create delle pool di thread, assegnando del lavoro da svolgere ad ognuna di esse, terminando ogni thread solo quando la sua pool viene cancellata.

  Tale modalità migliora le prestazioni, ma genera un potenziale spreco di risorse del sistema

Definizione 7: Non determinismo

Definiamo come **non determinismo** il fenomeno per cui lo stato futuro del programma possa dipendere dall'ordine con cui i thread svolgono gli accessi a **sezioni critiche** condivise (ad esempio eventuali race condition tra i thread).

Definizione 8: Busy waiting

Definiamo come **busy waiting (o spinning)** la tecnica secondo cui un thread rimane "bloccato" all'interno di un loop fino a quando un altro thread non lo sbloccherà a fine di prevenire il non determinismo

Esempio:

```
my_val = Compute_val ( my_rank ) ;
if(my_rank == 1){
    while(!ok_for_1);   //Thread 1 spins here
}

x += my_val ;   // Critical section

if( my_rank == 0){
    ok_for_1 = true;    //Let Thread 1 update x
}
```

Definizione 9: Message passing

Definiamo come **message passing** la modalità di scambio di informazioni tra due thread o processi basata sulle direttive `send()` e `receive()`

Esempio:

```
char message[100];
my_rank = Get_rank();

if(my_rank == 1){
    sprintf(message, "Greetings from process 1") ;
    send(message, MSG_CHAR, 100, 0);
}
else if(my_rank == 0){
    receive(message, MSG_CHAR, 100, 1) ;
    printf("Process 0 > Received: %s\n", message) ;
}
```

Per quanto riguarda la gestione di **input** ed **output** nei programmi paralleli, è consigliato l'uso delle seguenti **regole**:

- Solo il processo/thread 0 accede allo `stdin`
- Tutti i processi/thread possono accedere a `stdout` e `stderr`. Tuttavia, al fine di prevenire il non determinismo dell'output, nella maggior parte dei casi viene incaricato un unico processo/thread di accedere a tali canali.
- L'output di ogni processo/thread dovrebbe sempre includere il rank o ID del chiamante
- Ogni file deve essere acceduto da un solo processo/thread alla volta

1.2.2 Performance dei programmi paralleli

Per misurare la performance di un programma parallelo, vengono fondamentalmente considerati due parametri:

- La **velocizzazione**, ossia il rapporto tra il tempo di esecuzione della versione parallela del programma rispetto a quella seriale

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

- L'**efficienza**, ossia il rapporto tra la velocizzazione del programma e il numero di core utilizzati:

$$E = \frac{S}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$

In particolare, se si verifica che $T_{\text{parallel}} = \frac{T_{\text{serial}}}{p}$, la velocizzazione avrà un incremento **lineare** rispetto al numero di core:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{T_{\text{serial}} \cdot p}{T_{\text{serial}}} = p$$

mentre l'efficienza sarà **massima**:

$$E = \frac{S}{p} = \frac{p}{p} = 1$$

Teorema 1: Legge di Amdahl

La **legge di Amdahl** afferma che, a meno che esso non venga venga ipoteticamente completamente parallelizzato, la velocizzazione di un programma seriale è **limitata** indipendentemente dal numero di core utilizzati

Dimostrazione.

- Supponiamo che un processore impieghi t secondi per eseguire un programma seriale, dunque che $T_{\text{serial}} = t$
- Un software parallelizzato sarà formato da una percentuale $\alpha \in [0, 1]$ di codice parallelizzabile, mentre la restante percentuale $1 - \alpha$ di codice dovrà essere eseguita sequenzialmente
- Dato un numero p di core, Il tempo di esecuzione del programma parallelo corrisponderà quindi a:

$$T_{\text{parallel}} = \alpha \cdot \frac{T_{\text{serial}}}{p} + (1 - \alpha) \cdot T_{\text{serial}} = \frac{\alpha t}{p} + (1 - \alpha)t$$

  implicando quindi che la velocizzazione sia:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{t}{\frac{\alpha t}{p} + (1 - \alpha)t} = \frac{1}{\frac{\alpha}{p} + 1 - \alpha}$$

- A questo punto, all'aumentare del numero di core si ha che:

$$\lim_{p \to +\infty} S = \lim_{p \to +\infty} \frac{1}{\frac{\alpha}{p} + 1 - \alpha} = \frac{1}{1 - \alpha}$$

  per tanto concludiamo che, indipendentemente dal numero di core utilizzati, l'unico parametro che influenzi la velocizzazione sia la percentuale di codice non parallelizzato

□

Teorema 2: Legge di Gustafson

La **legge di Gustafson** afferma che sistemi con un elevato numero di core permettono la computazione di grandi quantità di dati in un tempo fisso.

In altre parole, oltre a velocizzare l'esecuzione, un sistema parallelo permette di lavorare con **istanze più grandi** di un problema.

Dimostrazione.

- Supponiamo che un processore impieghi t secondi per eseguire un programma parallelo, dunque che $T_{\text{parallel}} = t$
- Sia $\alpha \in [0, 1]$ la percentuale di codice parallelizzabile del programma, implicando che la restante percentuale $1 - \alpha$ di codice sarà quella eseguita sequenzialmente
- Dato un numero p di core, il tempo di esecuzione della versione seriale del programma corrisponderà quindi a:

  $$T_{\text{serial}} = (1 - \alpha) \cdot T_{\text{parallel}} + \alpha p \cdot T_{\text{parallel}} = (1 - \alpha)t + \alpha pt$$

  implicando quindi che la velocizzazione sia:

  $$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{(1 - \alpha)t + \alpha pt}{t} = (1 - \alpha) + \alpha p$$

  e quindi che l'efficienza sia:

  $$E = \frac{S}{p} = \frac{(1 - \alpha) + \alpha p}{p} = \frac{1 - \alpha}{p} + \alpha$$

- Di conseguenza, all'aumentare dei numero di core si ha che:

  $$\lim_{p \to +\infty} E = \lim_{p \to +\infty} \frac{1 - \alpha}{p} + \alpha = \alpha$$

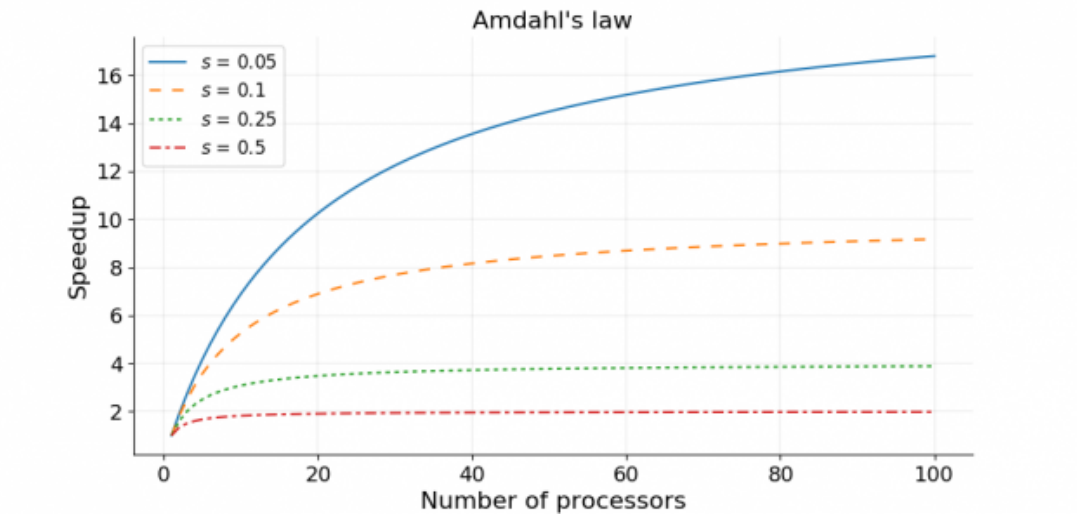
□

Definizione 10: Scalabilità

Definiamo un problema come **scalabile** se è possibile gestire sue istanze di dimensioni crescenti senza avere impatti sull'efficienza

Se l'**efficienza** di un programma rimane **fissa** con un numero crescente di processori utilizzati ma dimensioni del problema costanti, il problema risulta avere **scalabilità forte**, poiché ciò implica che le dimensioni del problema non vengano compensate dal numero di thread/core e quindi che esse non abbiano impatto sull'efficienza.

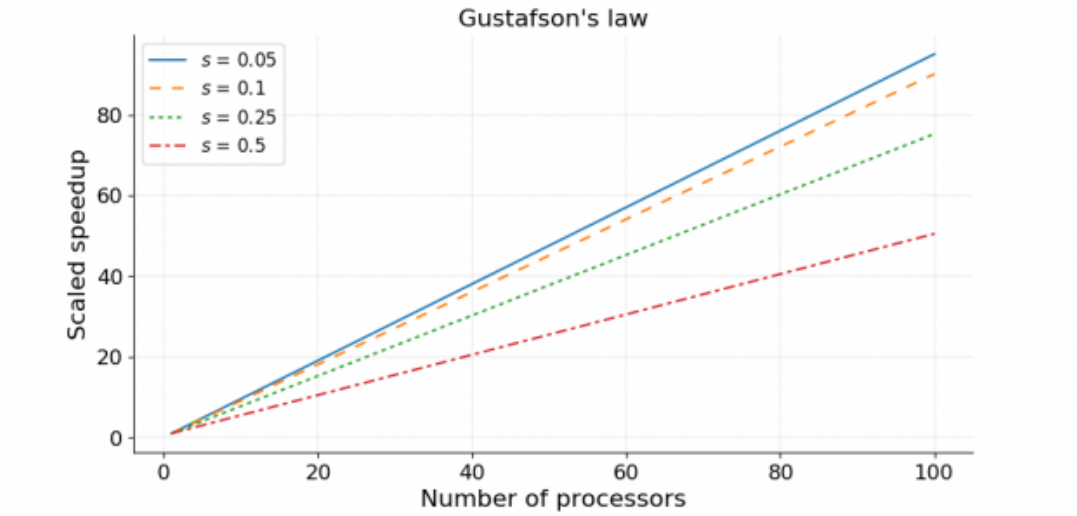
Risulta quindi evidente che la **legge di Amdahl** abbia una **forte scalabilità**:



dove s è la percentuale di lavoro sequenziale svolto.

Se invece l'**efficienza** di un programma rimane **fissa** con un numero crescente di processori utilizzati e dimensioni del problema, il problema risulta avere **scalabilità forte**, poiché ciò implica che le dimensioni del problema debbano essere compensate dal numero di thread/core e quindi che esse abbiano un grande impatto sull'efficienza.

Risulta quindi evidente che la **legge di Gustafson** abbia una **scalabilità debole**:



dove s è la percentuale di lavoro sequenziale svolto.

1.2.3 Design pattern di programmi paralleli

Per **parallelizzare** un programma sequenziale, viene solitamente utilizzata la **metodologia di Foster**, composta da quattro fasi:

1. **Partizionamento**: la computazione e i dati su cui operare vengono suddivisi in task di piccole dimensioni, cercando di priorizzare la loro esecuzione parallela
2. **Comunicazione**: vengono determinati quali comunicazioni debbano essere effettuate tra le varie task
3. **Agglomerazione**: alcune task vengono combinate con le task con cui dovrebbero comunicare

   (es: se la task B deve necessariamente attendere che la task A sia completata, la scelta migliore potrebbe essere quella di combinarle in un'unica task poiché esse sarebbero comunque non parallelizzabili tra loro)
4. **Mappature**: le task rimanenti vengono assegnate ai processi/thread, cercando di minimizzare le comunicazioni e di bilanciare il carico di lavoro

Per quanto riguarda la **strutturazione** della modalità di parallelismo di un programma, essi possono rientrare in due categorie:

- **Globally Parallel, Locally Sequential (GPLS)**: l'applicazione è in grado di svolgere task parallelamente, dove ogni task in se svolge del lavoro sequenziale.

  Rientrano in tale categoria i programmi basati sui modelli:

  - **Single Program Multiple Data (SPMD)**, dove viene inizializzato il programma procedendo con l'enumerazione tramite ID dei processi/thread utilizzati, i quali verranno eseguiti parallelamente tramite i vari branch, per poi terminare il programma una volta che essi hanno concluso
  - **Multiple Programs Multiple Data (MPMD)**, identici al quelli basati sul modello SPMD, ma utilizzando programmi multipli
  - **Master-Worker**: uno o più processi/thread master si occupano di assegnare lavoro ai processi/thread worker, collezionandone i risultati ed eseguendo operazioni di I/O per essi. Al fine di ridurre il bottleneck tra master e worker, è richiesta la strutturazione di una gerarchia basata su più livelli (dunque alcuni worker saranno anche i master di altri worker e così via)
  - **Map-Reduce**: variante del modello Master-Worker, dove i worker possono eseguire solo due tipi di task:
    - **Map**: viene utilizzata una funzione su dei dati, generando un risultato parziale
    - **Reduce**: viene derivato un risultato completo da risultati parziali

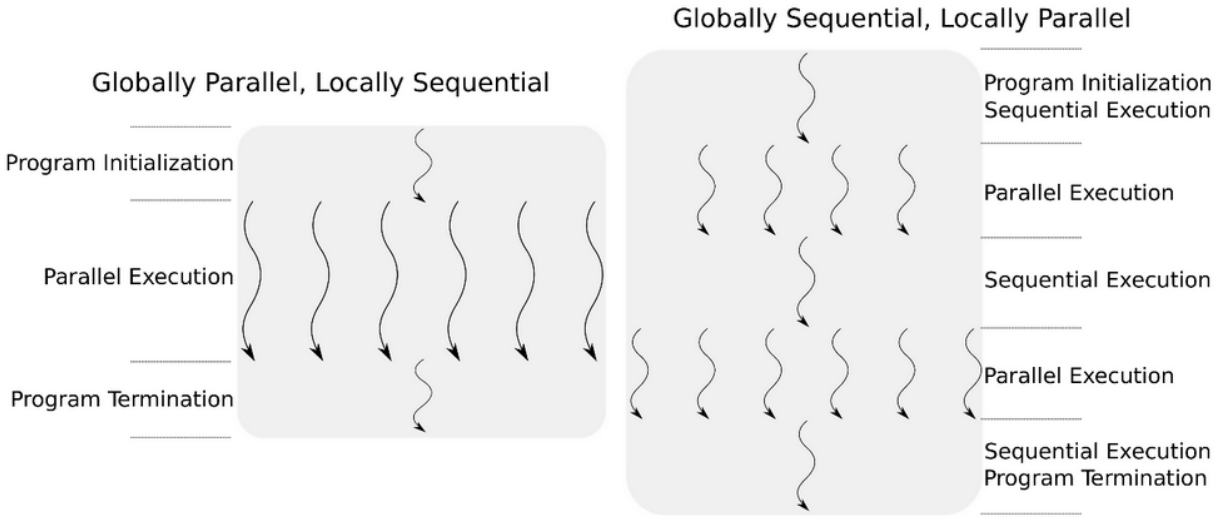
- **Globally Sequential, Locally Parallel (GSLP)**: l'applicazione esegue codice sequenziale, le cui parti individuali vengono svolte in parallelo quando richiesto.

  Rientrano in tale categoria i programmi basati sui modelli:

  - **Fork-Join**: vi è un singolo processo/thread genitore e i processi/thread figli vengono creati dinamicamente per eseguire task quando necessario. Il genitore può terminare l'esecuzione solo se tutti i figli hanno concluso le loro task
  - **Loop parallelism**: le iterazioni dei loop vengono "suddivise" manipolando le variabili di controllo.

    (es: un loop di 100 iterazioni viene sostituito con due loop da 50 iterazioni eseguibili in parallelo)

    Tale modello viene solitamente utilizzato per la migrazione di vecchi software sequenziali che si preferisce non riscrivere da capo, risultando tuttavia poco efficienti



2

Message Passing Interface (MPI)

2.1 Introduzione ad MPI

Il protocollo **Message Passing Interface (MPI)** è un protocollo di comunicazione per sistemi paralleli in grado di supportare sia la comunicazione point-to-point sia la comunicazione collettiva. In particolare, MPI supporta i programmi basati sui **modelli SPMD e MPMD**.

Su sistemi Linux è possibile installare la libreria `mpi` del linguaggio C tramite il seguente comando:

```
sudo apt install openmpi
```

Una volta installata, tramite il wrapper `mpicc` è possibile compilare programmi scritti con tale libreria:

```
mpicc -g -Wall <souce_file> -o <output_file>
```

I programmi compilati con `mpicc` possono essere comodamente avviati con un **numero variabile di processi** eseguenti il codice definito nel programma:

```
mpirun -n <processes_number> <executable>
```

Inoltre, è possibile avviare un programma MPI anche su **host distribuiti**, permettendo persino di scegliere come vengano suddivisi i processi avviati

(es: `mpirun -n 10 -host node1:4,node2:3,node3:3 ./mpi_hello`)

Ogni processo avviato viene identificato da un intero non negativo detto **rank**.

Le funzioni, variabili e costanti definite da MPI sono identificate dal **prefisso** `MPI_`, dove la prima lettera a seguito del trattino basso è in maiuscolo.

Per poter realizzare una porzione di codice con MPI, tale codice deve essere racchiuso tra le seguenti direttive:

- **MPI_Init**, la quale effettua il necessario setup per poter eseguire codice MPI

```
int MPI_Init(
    int*    argc_p,     // in/out
    char*** argv_p      // in/out
)
```

- **MPI_Finalize**, la quale effettua il necessario cleanup delle allocazioni svolte da MPI

```
int MPI_Init()
```

Un programma MPI, per tanto, comprende la seguente struttura base:

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

int main(int argc, char* argv[]){
    MPI_Init(&argc, &argv);

    ...
    //MPI calls are done ONLY here
    ...

    MPI_Finalize();
    ...

    return 0;
}
```

Definizione 11: Comunicatore

Definiamo come **comunicatore** un canale di comunicazione al quale hanno accesso alcuni (o tutti) processi, tramite quale essi possono scambiarsi messaggi

Osservazione 2

Quando viene chiamata la direttiva `MPI_Init()`, viene creato un **comunicatore di default** chiamato `MPI_COMM_WORLD` il quale è composto da tutti i processi creati all'avvio del programma

I comunicatori permettono inoltre di ottenere informazioni base sui processi:

- **MPI_Comm_size** restituisce il numero di processi del comunicatore dato in input

```
int MPI_Comm_size(
    MPI_Comm    comm,       // in
    int*        comm_size   // out
)
```

- **MPI_Comm_rank** restituisce il rank all'interno del comunicatore del processo chiamante (potrebbe cambiare in base al comunicatore)

```
int MPI_Comm_rank(
    MPI_Comm    comm,       // in
    int*        my_rank_o   // out
)
```

2.2 Comunicazione point-to-point

Per effettuare comunicazioni **point-to-point**, ossia tra solo due processi, la libreria MPI fornisce le seguenti direttive base:

- **MPI_Send** permette di inviare un messaggio ad un processo sul comunicatore designato.

```
int MPI_Send(
    void*           msg_buf_p,  // in
    int             msg_size,   // in
    MPI_Datatype    msg_type,   // in
    int             dest,       // in
    int             tag,        // in
    MPI_Comm        comm        // in
)
```

- **MPI_Recv** permette di ricevere un messaggio inviato da un processo sul comunicatore designato

```
int MPI_Recv(
    void*           msg_buf_p,  // out
    int             buf_size,   // in
    MPI_Datatype    buf_type,   // in
    int             source,     // in
    int             tag,        // in
    MPI_Comm        comm,       // in
    MPI_Status*     status_p    // in
)
```

Di seguito viene mostrato il programma `mpi_hello`, un esempio basilare di programma scritto tramite MPI:

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

#define MAX_STRING 100;

int main(){
    char greeting[MAX_STRING];  //string buffer
    int comm_sz;    //number of processes
    int my_rank;    //private process rank

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if(my_rank != 0){
        sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0,
            0, MPI_COMM_WORLD);
    }
    else{
        printf("Greetings from process %d of %d!", my_rank, comm_sz);
        for(int i = 1; i < comm_sz, i++){
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, i, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Eseguendo il comando `mpiexec -n 1 ./mpi_hello`, l'output sarà:

```
Greetings from process 0 of 1!
```

mentre tramite `mpiexec -n 4 ./mpi_hello`, l'output sarà:

```
Greetings from process 0 of 1!
Greetings from process 1 of 1!
Greetings from process 2 of 1!
Greetings from process 3 of 1!
```

Osservazione 3: Ordine di invio

La libreria MPI richiede che i messaggi siano **non sorpassanti**, ossia che nel caso in cui un processo invii più messaggi allo stesso processo, il primo messaggio dovrà arrivare prima del secondo.

Tuttavia, ciò non è richiesto nel caso in cui siano più processi ad inviare messaggi allo stesso processo

La libreria MPI fornisce anche dei **datatype** equivalenti ai principali datatype del linguaggio C. A differenza di quest'ultimi, i datatype di MPI vengono gestiti internamente da MPI stesso, venendo ottimizzati in base all'architettura.

Datatype MPI	Datatype C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_FLOAT	unsigned float
MPI_UNSIGNED_DOUBLE	unsigned double
MPI_UNSIGNED_LONG_DOUBLE	unsigned long double
MPI_UNSIGNED_BYTE	
MPI_UNSIGNED_PACKED	

Osservazione 4: Message matching

Un messaggio viene ricevuto **correttamente** se:

- Il datatype dell'invio è uguale al datatype del ricevimento
- La dimensione del messaggio inviato è minore del buffer del ricevimento

Un destinatario può ricevere un messaggio anche senza essere a conoscenza della dimensione del messaggio, del mittente del messaggio (tramite `MPI_ANY_SOURCE`) o il tag del messaggio (tramite `MPI_ANY_TAG`)

In particolare, la prima informazione è ottenibile tramite la direttiva `MPI_Get_count()`, mentre le altre due informazioni sono ottenibili tramite l'argomento di tipo `MPI_Status*` della direttiva `MPI_Recv()`.

Per loro natura stessa, le direttive di invio e ricevimento presentano alcune problematiche:

- `MPI_Send()` potrebbe comportarsi diversamente in base alla dimensione del buffer, a discontinuità e blocchi della comunicazione
- `MPI_Recv()` rimarrà in attesa finché non avverrà un message matching con tale chiamata
- Se una chiamata `MPI_Send()` viene bloccata e non vi è un `MPI_Recv()` corrispondente, il processo mittente può rimanere in attesa infinita
- Se una chiamata `MPI_Send()` viene bufferizzata e non vi è un `MPI_Recv()` corrispondente, il messaggio verrà perso
- Se il rank del processo destinatario è lo stesso del processo mittente, il processo rimarrà in attesa infinita o riceverà un messaggio casuale da parte di un altro processo

La maggior parte dei programmi MPI permettono solo al processo 0 di leggere da `stdin`, per poi condividere i dati letti tramite `MPI_CONN_WORLD`:

Una corretta funzione per la lettura di input viene pertanto strutturata come segue:

```
void Get_input(
    int     my_rank,    // in
    int     comm_sz,    // in
    double* a_p,        // out
    double* b_p,        // out
    int*    n_p         // out
){
    if(my_rank == 0){
        printf("Enter the values of a, b and n: ");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for(int dest = 1; dest < comm_sz; dest++){
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    }
    else{
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
}
```

Le direttive `MPI_Send()` e `MPI_Recv` gestiscono automaticamente un **buffer** interno al comunicatore utilizzato. Inoltre, la modalità di comunicazione prevista può essere di tipo **blocking** o **non-blocking**, ossia prevedere che i processi coinvolti rimangano o no in attesa che la comunicazione sia terminata. In particolare, distinguiamo due categorie di blocking:

- **Locally blocking**: solo un lato della comunicazione rimane in attesa
- **Globally blocking**: entrambi i lati della comunicazione rimangono in attesa

La libreria MPI fornisce le seguenti modalità di comunicazione:

- **Standard** (`MPI_Send()`): se il messaggio è abbastanza piccolo, la chiamata ritorna prima che il corrispettivo receive venga richiesto (locally blocking). In caso contrario, la chiamata ritorna solo quando la destinazione recupera completamente il messaggio inviato (globally blocking).
- **Buffered** (`MPI_Bsend()`): la chiamata ritorna non appena il messaggio viene copiato sul buffer (locally blocking). Inoltre, il buffer utilizzato deve essere fornito dall'utente.

  L'utilizzo tipico di tale modalità corrisponde a:

  ```
  MPI_Buffer_attach(...)
  ...
  MPI_Bsend(...)
  ...
  MPI_Buffer_detach(...)
  ```

- **Synchronous** (`MPI_Ssend()`): la chiamata ritornerà solo dopo che il destinatario abbia avviato la procedura di recupero del messaggio (globally blocking)
- **Ready** (`MPI_Rsend()`): l'invio va a buon fine solo se al momento della chiamata è già attiva la chiamata di ricevimento da parte del destinatario. In caso contrario, viene generato un errore
- **Immediate** (`MPI_Isend()`, `MPI_Ibsend()`, `MPI_Issend()`, `MPI_Irsend()`, `MPI_Irecv()`): ritorna immediatamente appena viene avviato il trasferimento (non-blocking), implicando che i due estremi della comunicazione non possano sapere se il trasferimento sia corretto o anche solo avvenuto.

Alcune direttive di comunicazione di MPI prevedono l'uso del datatype `MPI_Request`, il quale permette di ottenere un handle per controllare lo status di una comunicazione (simile ad `MPI_Status`). In particolare, una volta ottenuto l'handle di una comunicazione, è possibile controllarne lo stato tramite due direttive:

- `MPI_Wait()`: attende finché la comunicazione non è terminata (blocking), distruggendo l'handle
- `MPI_Test()`: restituisce immediatamente lo status attuale della comunicazione (non-blocking), senza distruggere l'handle

Osservazione 5

Se tutti i processi eseguono una chiamata di **invio bloccante**, nessuno dei processi sarà in grado di effettuare una chiamata di ricevimento, portando ad un **deadlock**

Per evitare i deadlock, dunque, è **buona prassi** che ogni chiamata di invio sia immediatamente seguita dalla corrispondente chiamata di ricevimento.

Esempio:

```
...
if(my_rank % 2 == 0){
    MPI_Send(msg, size, MPI_INT, (my_rank + 1) % comm_sz, 0, comm);
    MPI_Recv(new_msg, size, MPI_INT, (my_rank + comm_sz - 1) % comm_sz,
        0, comm, MPI_STATUS_IGNORE);
}
else{
    MPI_Recv(new_msg, size, MPI_INT, (my_rank + comm_sz - 1) % comm_sz,
        0, comm, MPI_STATUS_IGNORE);
    MPI_Send(msg, size, MPI_INT, (my_rank + 1) % comm_sz, 0, comm);
}
...
```

Nelle versioni più recenti di MPI viene introdotta la direttiva **MPI_Sendrecv**, la quale permette di svolgere contemporaneamente una chiamata di invio ed una chiamata di ricezione. Tale direttiva risulta particolarmente utile in quanto MPI schedula le comunicazioni in modo tale che il programma sia **a prova di blocchi perenni e crash**.

Osservazione 6

L'uso della direttiva `MPI_Sendrecv` non corrisponde all'uso di una direttiva `MPI_Send` seguita da una direttiva `MPI_Recv`, poiché nel secondo caso le due chiamate non sono svolte contemporaneamente

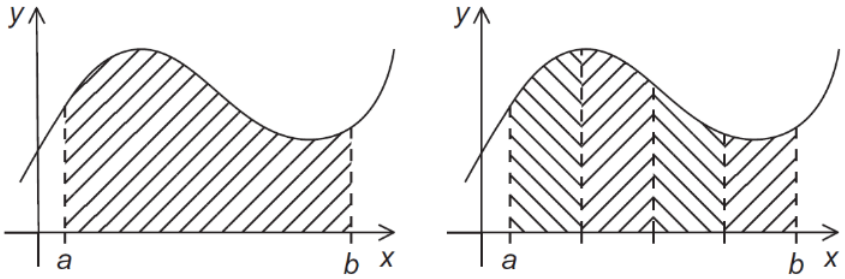
Esempio:

- Il codice dell'esempio precedente può essere riscritto direttamente come segue, ma le comunicazioni verranno automaticamente schedulate in modo da essere eseguibili contemporaneamente:

```
...
MPI_Sendrecv(
    msg, size, MPI_INT, (my_rank + 1) % comm_sz, 0,
    new_msg, size, MPI_INT, (my_rank + comm_sz - 1) % comm_sz, 0,
    comm, MPI_STATUS_IGNORE
);
...
```

2.2.1 Regola del trapezoide in MPI

Data una funzione f, vogliamo calcolare l'**integrale definito di** f tra due estremi a e b. Per calcolare tale integrale, utilizziamo la **regola del trapezoide**, ossia l'approssimazione dell'integrale tramite la somma di n trapezoidi che approssimino l'integrale.



In particolare, la regola del trapezoide è definita dal seguente algoritmo:

1. L'intervallo tra a e b viene suddiviso in n intervalli:

$$x_0 = a, \quad x_1 = a + h \quad x_2 = a + 2h \quad \dots \quad x_{n-1} = a + (n-1)h \quad x_n = b$$

   dove $h = \dfrac{b-a}{n}$

2. Per ogni $i \in [0, n-1]$ viene calcolata l'area del trapezoide corrisponde a:

$$A_i = \frac{h}{2}[f(x_i) + fx_{i+1}]$$

3. Vengono sommate le aree dei trapezoidi:

$$A_{tot} = \sum_{i=0}^{n} A_i = h\left[\frac{f(x_0)}{2} + f(x_1) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}\right]$$

L'implementazione seriale della regola del trapezoide, dunque, corrisponde a:

```
double approxIntegral(double a, double b, int n){
    double h, approx;
    h = (b-a)/n;
    approx = (f(a)+f(b))/2.0;

    for(int i = 1; i <= n-1; i++){
        x_i = a + i * h;
        approx += f(x_i);
    }

    return h * approx;
}
```

Vogliamo quindi parallelizzare tale codice applicando la metodologia di Foster:

1. **Partizionamento**: viene definita una task per ognuno degli n trapezoidi ed una task finale per la somma delle loro aree
2. **Comunicazione**: ognuna delle n task deve inviare il risultato dalla task finale
3. **Agglomerazione**: non necessaria poiché l'ultima task deve avvenire dopo ognuna delle n task
4. **Mappature**: vengono suddivise le n task tra i p processi avviati dal programma

```
double Trap(double left_endpt, double right_endpt,
                int trap_count, double base_len){
    double estimate, x;

    estimate = (f(left_endpt) + f(right_endpt))//2.0;

    for(int i=0; i <= trap_count-1; i++){
        x = left_endpt + i * base_len;
        estimate += f(x);
    }

    return estimate * base_len;
}

double approxIntegral(double a, double b, int n){
    int my_rank, comm_sz, local_n;
    double h, local_a, local_b, local_int, total_int;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    h = (b-a)/n;
    local_n = n/comm_sz;    //number of trapezoids per process

    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);

    if(my_rank != 0){
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
    else{
        total_int = local_int;
```

```
        for(int source = 1; source < comm_sz; source++){
            MPI_Recv(&local_int, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
            total_int += local_int;
        }
    }

    MPI_Finalize();
    return total_int;
}
```

2.3 Comunicazione collettiva

Oltre alla comunicazione point-to-point, la libreria MPI permette di effettuare **comunicazioni collettive**, ossia tra più di due processi. In particolare la direttiva `MPI_Reduce` permette a tutti i processi coinvolti nella comunicazione di inviare un dato, per poi applicare un **operatori collettivo** su tutti i dati della comunicazione.

```
int MPI_Reduce(
    void*           input_data_p    // in
    void*           output_data_p   // out
    int             count           // in
    MPI_Datatype    datatype        // in
    MPI_Op          operator        // in
    int             dest_process    // in
    MPI_Comm        comm            // in
)
```

Valore di `MPI_Op`	**Operazione**
`MPI_MAX`	Massimo
`MPI_MIN`	Minimo
`MPI_SUM`	Somma
`MPI_PROD`	Prodotto
`MPI_LAND`	AND logico
`MPI_LOR`	OR logico
`MPI_LXOR`	XOR logico
`MPI_BAND`	Bitwise AND
`MPI_BOR`	Bitwise OR
`MPI_BXOR`	Bitwise XOR
`MPI_MAXLOC`	Massimo e posizione del massimo
`MPI_MINLOC`	Minimo e posizione del minimo

Osservazione 7

All'interno di un comunicatore, tutti i processi coinvolti nella comunicazione devono utilizzare la **stessa direttiva**

Esempio:

- Se un processo esegue `MPI_Reduce` e un altro processo esegue `MPI_Recv`, il programma si bloccherà e/o andrà in errore

Osservazione 8

L'output di una chiamata `MPI_Reduce` viene inserito solo nella locazione puntata dal valore `output_data_p` passato dal **processo destinatario**.

Tramite la comunicazione collettiva, possiamo realizzare una versione estremamente ridotta del codice parallelo per la **regola del trapezoide**:

```
double Trap(...){
    ...
}

double approxIntegral(double a, double b, int n){
    int my_rank, comm_sz, local_n;
    double h, local_a, local_b, local_int, total_int;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    h = (b-a)/n;
    local_n = n/comm_sz;    //number of trapezoids per process

    local_a = a + my_rank * local_n * h;
    local_b = local_a + local_n * h;
    local_int = Trap(local_a, local_b, local_n, h);

    MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM,
        0, MPI_COMM_WORLD);

    MPI_Finalize();
    return total_int;
}
```

Per far sì che **tutti i processi** coinvolti nella comunicazione **ricevano l'output**, viene fornita la direttiva `MPI_Allreduce`:

```
int MPI_Allreduce(
    void*           input_data_p    // in
    void*           output_data_p   // out
    int             count           // in
    MPI_Datatype    datatype        // in
    MPI_Op          operator        // in
    MPI_Comm        comm            // in
)
```

Nel caso in cui invece si voglia **distribuire un dato** all'interno di un comunicatore senza svolgere alcuna operazione, viene fornita la direttiva `MPI_Bcast`:

```
int MPI_Bcast(
    void*           data_p          // in/out
    int             count           // in
    MPI_Datatype    datatype        // in
    int             source_proc     // in
    MPI_Comm        comm            // in
)
```

Ad esempio, possiamo definire una seconda versione di `Get_input` realizzata tramite `MPI_Bcast`:

```
void Get_input(
    int     my_rank,    // in
    int     comm_sz,    // in
    double* a_p,        // out
    double* b_p,        // out
    int*    n_p         // out
){
    if(my_rank == 0){
        printf("Enter the values of a, b and n: ");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

2.4 Distribuzione del lavoro su vettori

Supponiamo di voler calcolare una somma tra due vettori di n elementi:

$$(x_0, \ldots, x_{n-1}) + (y_0, \ldots, y_{n-1}) = (x_0 + y_0, \ldots, x_{n-1} + y_{n-1})$$

Il codice seriale in grado di svolgere tale somma risulta banale:

```
void vectorSum(double x[], double y[], double z[], int n){
    for(int i=0; i < n; i++){
        z[i] = x[i] + y[i];
    }
}
```

Sebbene tale codice possa essere parallelizzato anche tramite le direttive mostrate precedentemente, MPI permette di parallelizzare il lavoro su vettori tramite direttive più adatte.

In particolare, la direttiva **MPI_Scatter()** distribuisce in modo equo il contenuto di un vettore del processo 0 tra tutti i processi all'interno di un comunicatore (incluso il processo 0 stesso). Deve essere chiamata *sia* dal processo distributore sia dai processi riceventi.

```
int MPI_Scatter(
    void*           send_buf_p  // in
    int             send_count  // in
    MPI_Datatype    send_type   // in
    void*           recv_buf_p  // out
    int             recv_count  // in
    MPI_Datatype    recv_type   // in
    int             src_proc    // in
    MPI_Comm        comm        // in
)
```

Esempio:

- La seguente funzione legge in input un vettore di n elementi, per poi distribuire gli elementi letti:

```
void readAndScatterVector(
    double local_vec[]  // out
    int local_n         // in
    int n               // in
    char vec_name[]     // in
    int my_rank         // in
    MPI_Comm            // in
){
    double* vec = NULL;
```

```
        if(my_rank == 0){
            vec = malloc(n * sizeof(double));
            printf("Enter the vector %s\n", vec_name);
            for(int i=0; i < n; i++){
                scanf("%lf", &vec[i]);
            }

            // Process 0 sends local_n elements to every process
            MPI_Scatter(vec, local_n, MPI_DOUBLE, local_vec, local_n,
                MPI_DOUBLE, 0, comm);
            free(vec);
        }
        else{
            // Other processees receive the data
            MPI_Scatter(vec, local_n, MPI_DOUBLE, local_vec, local_n,
                MPI_DOUBLE, 0, comm);
        }
    }
```

Viceversa, la direttiva **MPI_Gather()**: raccoglie all'interno di un vettore del processo 0 il contenuto di altri vettori posseduti dai processi di un comunicatore. Deve essere chiamata sia dal processo raccoglitore sia dai processi invianti.

```
int MPI_Gather(
    void*           send_buf_p  // in
    int             send_count  // in
    MPI_Datatype    send_type   // in
    void*           recv_buf_p  // out
    int             recv_count  // in
    MPI_Datatype    recv_type   // in
    int             dest_proc   // in
    MPI_Comm        comm        // in
)
```

Esempio:

- La seguente funzione raccoglie all'interno di un vettore il contenuto dei vettori dei processi, per poi stampare il vettore finale

```
    void printAndGatherVector(
        double local_vec[]  // in
        int local_n         // in
        int n               // in
        char vec_name[]     // in
        int my_rank         // in
        MPI_Comm            // in
    ){
```

```
        double* vec = NULL;

        if(my_rank == 0){
            vec = malloc(n * sizeof(double));

            // Process 0 gathers previously distributed element
            MPI_Gather(vec, local_n, MPI_DOUBLE, local_vec, local_n,
                MPI_DOUBLE, 0, comm);

            printf("Vector %s elements:\n", vec_name);
            for(int i=0; i < n; i++){
                printf("%lf", &vec[i]);
            }

            free(vec);
        }
        else{
            // Other processees send the data
            MPI_Gather(vec, local_n, MPI_DOUBLE, local_vec, local_n,
                MPI_DOUBLE, 0, comm);
        }
    }
```

Nel caso in cui il vettore finale collezionato voglia essere reso disponibile a tutti i processi partecipanti al raccoglimento, viene fornita la direttiva `MPI_Allgather`.

Osservazione 9

Se un vettore distribuito tramite `MPI_Scatter` non è perfettamente distribuibile, l'ultimo processo riceverà meno elementi

Esempio:

- Supponiamo che V sia un vettore di $n = 7$ elementi e che venga eseguita una chiamata `MPI_Scatter` inviando $k = 3$ elementi ad ogni processo su un comunicatore di $p = 3$ processi:
  - Il primo processo riceverà gli elementi $V[0], V[1]$ e $V[2]$
  - Il secondo processo riceverà gli elementi $V[3], V[4]$ e $V[5]$
  - Il terzo processo riceverà gli elementi $V[6], D$ e D, dove D è un valore di default definito dal `MPI_Datatype` utilizzato nella chiamata

2.5 Datatype derivati

Nel linguaggio C viene fornita la possibilità di creare degli **struct**, ossia dei **datatype derivati** composti da altri tipi.

Ad esempio, possiamo definire il tipo `Point` come una collezione di tre tipi `double` corrispondenti alle coordinate di un punto in tre dimensioni:

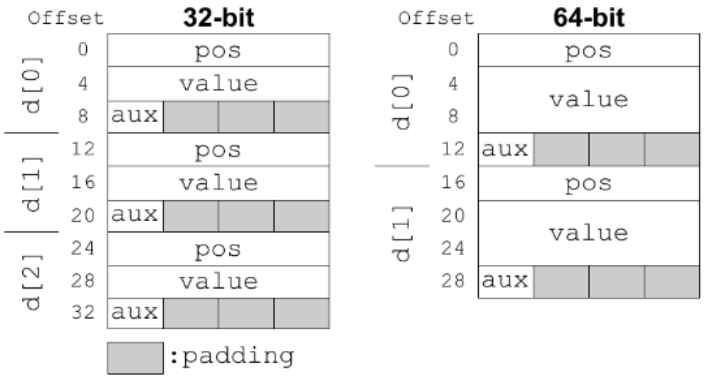
```
typedef struct Point{
    double x;
    double y;
    double z;
};
```

Tuttavia, l'uso dei normali struct all'interno di un codice MPI, potrebbe generare delle **problematiche**:

- Supponiamo di aver avviato un programma parallelo con due processi attivi, il primo risiedente su un sistema a 32 bit e il secondo su un sistema a 64 bit
- Supponiamo inoltre che tale programma utilizzi il seguente vettore `d` di tipo `T`:

```
struct T{
    int pos;
    long value;
    char aux;
} d[n]
```

- Nei sistemi a 32 bit, il tipo `long` coincide ad un intero di 4 byte (dunque analogo ad un tipo `int`), mentre nei sistemi a 64 bit esso coincide con un intero di 8 byte. Di conseguenza, nei due processi il vettore `d` viene rappresentato come:



- Dunque, utilizzare il tipo `MPI_BYTE` (il quale invia byte per byte) all'interno di una comunicazione porterà i due processi ad avere valori inconsistenti tra loro

Per risolvere tale problematica, viene fornita la direttiva **MPI_Type_create_struct**, la quale permette di creare dei datatype derivati tramite i tipi base di MPI stesso.

```
int MPI_Type_create_struct(
    int             count                   // in
    int             arr_of_blockleghts[]    // in
    MPI_Aint        arr_of_displacements[]  // in
    MPI_Datatype    arr_of_types[]          // in
    MPI_Datatype*   new_type_p              // out
)
```

Inoltre, vengono fornite le seguenti direttive per lavorare con i datatype derivati:

- `MPI_Get_address()`: restituisce l'indirizzo di memoria puntato da `location_p` utilizzando il formato adatto previsto dal sistema operativo

  ```
  int MPI_Get_address(
      void* location_p        // in
      MPI_Aint* address_p     // out
  )
  ```

- `MPI_Type_commit()`: permette ad MPI di ottimizzare la rappresentazione interna del datatype al fine di poterlo utilizzare nelle comunicazioni.

  Tale direttiva è obbligatoria e va utilizzata solo quando la definizione del tipo da creare è completata

  ```
  int MPI_Type_commit(
      MPI_Datatype* new_mpi_t_p   // in/out
  )
  ```

- `MPI_Type_free()`: permette di liberare la memoria una volta concluso di utilizzare il datatype derivato

  ```
  int MPI_Type_free(
      MPI_Datatype* old_mpi_t_p   // in/out
  )
  ```

Esempio:

```
void build_MPI_type(double* a_p, double* b_p, double* n_p,
        MPI_Datatype* input_mpi_t_p){

    MPI_Aint a_addr, b_addr, n_addr;

    int AoB[3] = {1, 1, 1};
    MPI_Datatype AoT[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
    MPI_Aint AoD[3] = {0};
```

```
    MPI_Get_address(a_p, &a_addr);
    MPI_Get_address(b_p, &b_addr);
    MPI_Get_address(n_p, &n_addr);

    AoD[1] = b_addr - a_ddr;
    AoD[2] = n_addr - a_ddr;
    MPI_Type_create_struct(3, AoB, AoD, AoT, input_mpi_t_p);
    MPI_Type_commit(input_mpi_t_p);
}

void getInput(int my_rank, int comm_sz, double* a_p,
        double* b_p, int* n_p){
    MPI_Datatype MPI_Input_t;
    build_MPI_type(a_p, b_p, n_p, &MPI_Input_t);

    if(my_rank == 0){
        printf("Enter a, b and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }

    MPI_Bcast(a_p, 1, MPI_Input_t, 0, MPI_COMM_WORLD);
    MPI_Type_free(&MPI_Input_t);
}
```

2.6 Performance e barriere

Per sapere il tempo impiegato da un programma seriale, il linguaggio C fornisce la funzione `GET_TIME()` presente all'interno della libreria `<timer.h>`:

```
#include <timer.h>

double start, finish;

void main(){
    GET_TIME(start);

    ...

    GET_TIME(finish);
    printf("Elapsed time = %e seconds", finish - start);
}
```

Analogamente, la libreria MPI fornisce la direttiva `MPI_Wtime`:

```
double start, finish;

void main(){
    ...

    start = MPI_Wtime();

    ...

    finish = MPI_Wtime();
    printf("Process %d > Elapsed time = %e seconds",
        my_rank, finish - start);
}
```

Tuttavia, in tal modo ogni processo restituirà il proprio tempo impiegato. Nel caso in cui si voglia ottenere il tempo totale impiegato per l'esecuzione del programma parallelo, può essere utilizzata la direttiva `MPI_Barrier`, la quale metterà **ogni processo chiamante in attesa** che tutti i processi del comunicatore abbiano "raggiunto la barriera" (ossia abbiano chiamato la direttiva).

```
double local_start, local_finish, local_elapsed, elapsed;

void main(){
    ...

    // wait for all processes in comm to reach the barrier
    MPI_Barrier(comm);

    local_start = MPI_Wtime();

    ...

    local_finish = MPI_Wtime();
    local_elapsed = local_finish - local_start;
    MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
        MPI_MAX, 0, comm);

    if(my_rank == 0){
        printf("Elapsed time = %e seconds", finish - start);
    }
}
```

2.7 Remote Memory Access

La libreria MPI-2 (e successive versioni) forniscono la possibilità di effettuare comunicazioni unilaterali, ossia effettuate completamente da un singolo processo.

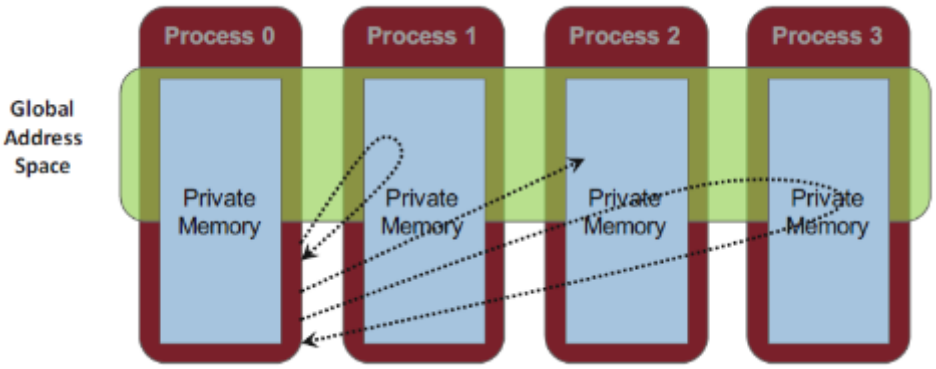
In particolare, tale tipo di comunicazione viene realizzata tramite il **Remote Memory Access (RMA)**, ossia la possibilità di un processo di poter accedere e modificare porzioni di memoria di un altro processo, inviando così un messaggio.

Il RMA risulta una valida alternativa alle comunicazioni bilaterali soprattutto in situazioni in cui le modalità di comunicazione cambiano durante l'esecuzione oppure richiedono operazioni globali e/o polling.

Le operazioni necessarie al fine di poter utilizzare il RMA sono le seguenti:

1. Creazione di una **finestra**, ossia una struttura dati tramite cui accedere alle zone di memoria da remoto
2. Svolgimento delle comunicazioni, sincronizzando i processi tra di loro
3. Distruzione della finestra

In una delle sue modalità operative, il meccanismo del RMA ricorda quello dei Mutex nell'ambito dei programmi multi-thread a memoria condivisa.



- `MPI_Alloc_mem`: alloca una porzione di memoria condivisibile
- `MPI_Win_create`: ogni processo espone una propria zona di memoria creando una finestra. NOTA: gli argomenti passati devono rispettare i valori di una precedente chiamata `MPI_Alloc_mem`
- `MPI_Win_alloc`: analogo all'uso di `MPI_Alloc_mem` seguito da `MPI_Create_win`
- `MPI_Info`: è un contenitore di coppie (chiave, valore) utilizzabili per ottimizzare le comunicazioni
- `MPI_Win_free`: distrugge una finestra precedentemente creata
- `MPI_Get`: copia dati dalla memoria del target alla memoria del chiamante. È una direttiva non bloccante e richiede sincronizzazione esterna tramite altre modalità.

- `MPI_Put`: copia dati dalla memoria del chiamante alla memoria del target. È una direttiva non bloccante e richiede sincronizzazione esterna tramite altre modalità.
- `MPI_Accummulate`: scrive dati locali sulla memoria del target. È una direttiva non bloccante e richiede sincronizzazione esterna tramite altre modalità.

Definizione 12: Active e Passive target

Per effettuare le operazioni di **sincronizzazione** tra i processi al fine di svolgere correttamente le comunicazioni, vengono utilizzati due approcci:

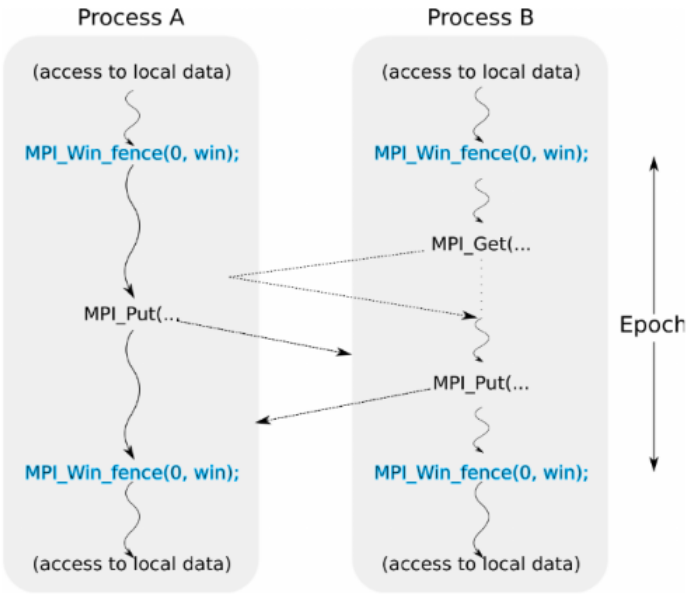
- **Active target**: solo il processo target controlla lo scambio dei dati. Il lasso di tempo in cui il target può ricevere comunicazioni unilaterali viene detto **era di esposizione** ed è controllato dal processo target stesso
- **Passive target**: solo i processi chiamanti controllano lo scambio dei dati. Il lasso di tempo in cui ogni chiamante può svolgere comunicazioni unilaterali viene detto **era di accesso** ed è controllato dal singolo processo chiamante.

Osservazione 10

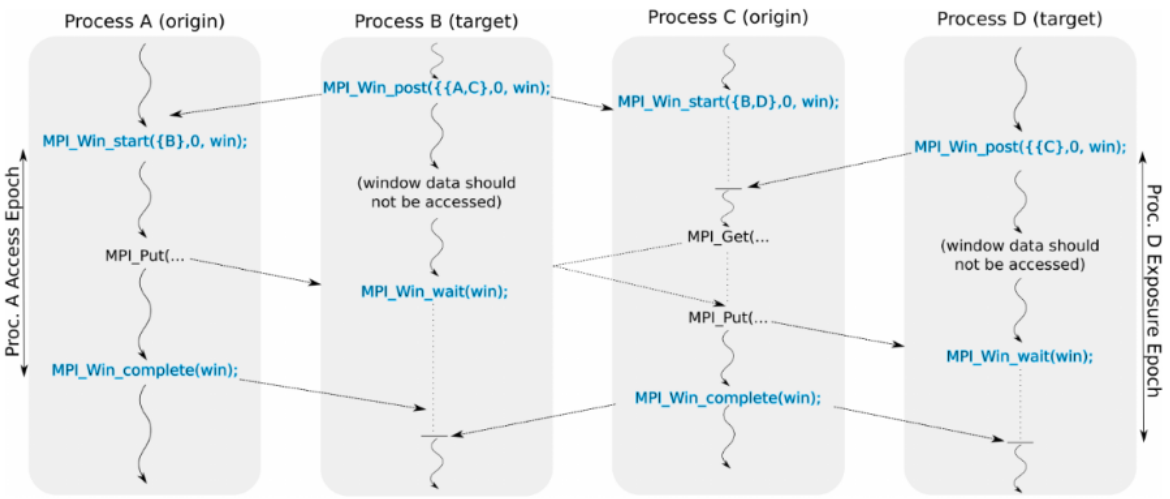
Qualsiasi tentativo di accesso alla memoria del target di una comunicazione unilaterale che avviene **durante il periodo designato** (indipendentemente dalla metodologia utilizzata) potrebbe restituire dati errati

La modalità di sincronizzazione **active target** può essere realizzata in due modi:

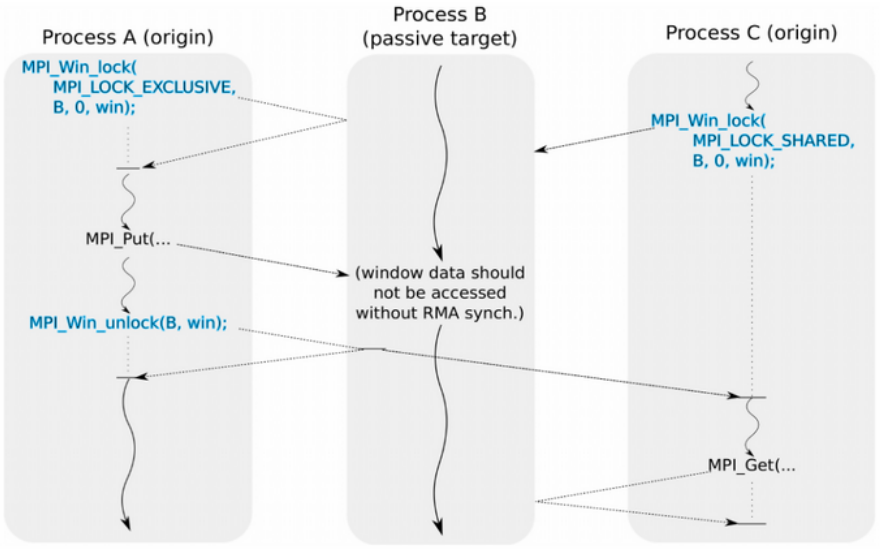
1. Utilizzando la direttiva `MPI_Win_fence`, la quale conferma le modifiche effettuate durante un periodo di esposizione



2. Utilizzando le direttive `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post` e `MPI_Win_wait`, dove le prime due permettono di definire l'era di esposizione e le ultime due permettono di definire l'era di accesso



Per quanto riguarda la modalità **passive target**, invece, vengono fornite le direttive `MPI_WIn_lock` e `MPI_WIn_unlock`, le quali permettono di gestire gli RMA come un **Mutex**. In particolare, il **lock** utilizzato può essere di tipo `MPI_LOCK_EXCLUSIVE`, ossia supportare operazioni di scrittura, o `MPI_LOCK_SHARED`, ossia supportare operazioni di lettura.



Osservazione 11

Poiché il processo target **non è a conoscenza delle comunicazioni in corso**, nella modalità **passive target** è buona prassi che esso acceda al buffer esposto solo tramite le funzioni di coordinazione.

3

POSIX Threads (Pthreads)

3.1 Introduzione ai Pthreads

La **Portable Operating System Interface (POSIX)** è una famiglia di standard definiti dalla società IEEE col fine di mantenere compatibili tra loro i vari sistemi operativi aderenti.

In particolare, i **POSIX threads** (**Pthreads**) sono la modalità standard di utilizzo dei thread fornita dai sistemi aderenti agli standard POSIX (es: i sistemi Unix-like) tramite delle API di sistema per la creazione di programmi multi-thread.

Su sistemi Linux la libreria `pthread` del linguaggio C risulta già installata. Per quanto riguarda la compilazione, viene utilizzato il normale compiler del linguaggio C, includendo però la liberia stessa:

```
gcc -g -Wall -lpthread <source_file> -o <output_file
```

Inoltre, l'esecuzione può essere avviata direttamente tramite l'eseguibile generato.

Lo standard Pthread supporta i programmi basati sul **modello Fork-Join** ed è per tanto basato su quattro direttive principali:

- **pthread_create**: crea un thread che eseguirà la funzione (detta *start routine*) data in input utilizzando gli argomenti dati come input della funzione eseguita

```
int pthread_create (
    pthread_t* thread_p             // out
    const pthread_attr_t* attr_p    // in
    void* (*start_routine)(void)    // in
    void* arg_p                     // in
)
```

- **pthread_self**: restituisce il thread ID (TID) del thread chiamante
- **pthread_equal**: compara due TID tra loro
- **pthread_join**: attende che il thread designato completi la sua esecuzione, per poi terminarlo ed eliminarlo correttamente

```
int pthread_create (
    pthread_t* thread_p     // in
    void**  return_val      // out
)
```

Osservazione 12

Non appena un thread verrà creato, la sua esecuzione verrà subito avviata

Osservazione 13

Le funzioni passate in input alla funzione `pthread_create` devono avere una segnatura della seguente forma:

```
void* function_name(void* args)
```

dove il parametro `args` può esse un puntatore ad un valore singolo o una lista (il casting deve essere effettuato all'interno della funzione stessa)

Per creare un programma multi-thread utilizzando la libreria `pthread`, è possibile utilizzare il seguente **scheletro generale**:

```
#include <pthread.h>
#include <stdlib.h>
#include <alloca.h>

void* fun(void* rank){
    ...
}

int main(int argc, char* argv[]){
    if(argc < 2){
        printf("Too few arguments");
        return 1;
    }

    int thread_num = atoi(argv[1]);
    pthread_t thread_pool[thread_num];
```

```
    for(int rank = 0; rank < thread_num; rank++){
        // Allocate on stack a temporary value
        int* tmp = (int*) alloca(sizeof(int));
        *tmp = rank;

        // Spawn next thread
        pthread_create(&thread_pool[rank], NULL, &fun, (void*) tmp);
    }

    // Close threads
    for(int rank = 0; rank < thread_num; rank++){
        pthread_join(thread_pool[rank], NULL);
    }
}
```

3.2 Busy waiting

Poiché i thread di un programma condividono sempre una porzione della memoria, è necessario gestire il fenomeno del **non determinismo**.

Supponiamo di voler definire un codice che approssimi π con la seguente serie:

$$\pi \approx 4 \sum_{k=0}^{n} (-1)^n \frac{1}{2k+1}$$

Una prima versione seriale di tale codice corrisponde a:

```
// Assuming n is a global variable

double approxPI(){
    double factor = 1.0;
    double sum = 0.0;

    for(int k=0; k < n; k++){
        sum += factor/(2 * k + 1)
        factor = -factor;
    }
    pi = 4.0 * sum;
}
```

A questo punto, definiamo una funzione eseguibile da più thread:

```
// Assuming n, thread_count and sum are global variables

void* thread_approxPI(void* rank){
    long my_rank = *((long*) rank);
    double factor;

    long long my_n = n / thread_count;
    long long my_fist_k = my_n * my_rank;
    long long my_last_k = my_first_k * my_n;

    factor = (my_fist_k % 2 == 0) ? 1.0 : -1.0;
    sum = 0.0;

    for(long long k = my_fist_k; k < my_last_k; k++){
        sum += factor/(2 * k + 1)
        factor = -factor;
    }

    return NULL;
}

// Execute pi = 4.0 * sum after all threads have finished
```

Proviamo quindi ad eseguire il programma multi-thread aumentando la precisione dell'approssimazione (ossia aumentando n) e aumentando il numero di thread:

	n			
	10^5	10^6	10^7	10^8
Tramite serie	3.14159	3.141593	3.1415927	3.14159265
Tramite 1 thread	3.14158	3.141592	3.1415926	3.14159264
Tramite 2 thread	3.14158	3.141580	3.1415692	3.14164886

Notiamo dunque che l'esecuzione con 2 thread tenda a generare un risultato molto impreciso all'aumentare di n. Tale fenomeno è dovuto alla presenza di una **race condition**: i thread potrebbero accedere alla variabile `sum` in contemporanea (**sezione critica**).

Come già accennato nella sezione 1.2.1, un primo metodo atto al gestire le sezioni critiche è il **busy waiting**, dove ogni thread attende in un loop infinito finché non gli venga concesso di poter scrivere.

Implementiamo quindi una versione parallela del codice proposto precedentemente utilizzando il busy waiting:

```
// Assuming n, thread_count, sum and flag are global variables

void* thread_approxPI(void* rank){
    long my_rank = *((long*) rank);
    double factor;

    long long my_n = n / thread_count;
    long long my_fist_k = my_n * my_rank;
    long long my_last_k = my_first_k * my_n;

    factor = (my_fist_k % 2 == 0) ? 1.0 : -1.0;
    sum = 0.0;

    for(long long k = my_fist_k; k < my_last_k; k++){
        while (flag != my_rank);    // busy waiting

        sum += factor/(2 * k + 1)
        factor = -factor;

        flag = (flag + 1) % thread_count;   // update flag
    }

    return NULL;
}

// Execute pi = 4.0 * sum after all threads have finished
```

In tal modo, le iterazioni dei cicli for di ogni thread verranno eseguite in modo **alterno**. Possiamo invece definire una versione analoga alla precedente ma utilizzante una **somma locale** all'interno del ciclo for, per poi effettuare il busy waiting solo quando essa verrà sommata alla somma totale:

```
// Assuming n, thread_count, sum and flag are global variables

void* thread_approxPI(void* rank){
    long my_rank = *((long*) rank);
    double my_sum;
    double factor;

    long long my_n = n / thread_count;
    long long my_fist_k = my_n * my_rank;
    long long my_last_k = my_first_k * my_n;

    factor = (my_fist_k % 2 == 0) ? 1.0 : -1.0;
```

```
        sum = 0.0;

        for(long long k = my_fist_k; k < my_last_k; k++){
            my_sum += factor/(2 * k + 1)
            factor = -factor;
        }

        while (flag != my_rank);    // busy waiting
        sum += my_sum;
        flag = (flag + 1) % thread_count;   // update flag

        return NULL;
    }

    // Execute pi = 4.0 * sum after all threads have finished
```

In tal modo, i cicli for dei vari thread verranno eseguiti del tutto in parallelo, **senza alternarsi** tra di loro.

3.3 Mutex e Semafori

L'utilizzo del **busy waiting**, seppur efficace, presenta alcune problematiche. In particolare, un thread in stato busy waiting utilizzerà continuamente la CPU con il solo scopo di verificare se la sua esecuzione possa proseguire o no, **sprecando** così una notevole capacità di calcolo.

Definizione 13: Mutex

Un **mutex** (da *mutual exclusion*) è uno speciale tipo di variabile che può essere utilizzata per permettere ad un solo thread per volta di accedere ad una sezione critica. In tal modo, viene garantito che il thread attualmente nella sezione critica "escluda" gli altri dal potervi accedere.

La liberia `pthread` permette di utilizzare i mutex tramite le seguenti direttive:

- `pthread_mutex_init`, la quale crea un mutex
- `pthread_mutex_destroy`, la quale elimina un mutex esistente
- `pthread_mutex_lock`, la quale effettua una chiamata bloccante in attesa che il lock del mutex sia prendibile
- `pthread_mutex_trylock`, la quale effettua una chiamata non bloccante che prende il lock del mutex solo se è già libero
- `pthread_mutex_unlock`, la quale restituisce il lock del mutex

Definiamo quindi la versione multi-thread del codice per l'approssimazione di π tramite l'uso dei mutex:

```
// Assuming n, thread_count, sum and mutex are global variables

void* thread_approxPI(void* rank){
    long my_rank = *((long*) rank);
    double my_sum;
    double factor;

    long long my_n = n / thread_count;
    long long my_fist_k = my_n * my_rank;
    long long my_last_k = my_first_k * my_n;

    factor = (my_fist_k % 2 == 0) ? 1.0 : -1.0;
    sum = 0.0;

    for(long long k = my_fist_k; k < my_last_k; k++){
        my_sum += factor/(2 * k + 1)
        factor = -factor;
    }

    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

// Execute pi = 4.0 * sum after all threads have finished
```

Osservazione 14: Deadlock nei mutex

Se due thread condividono due mutex acquisiscono tali mutex in **ordine inverso**, si genererà un **deadlock**

Esempio:

```
// Thread A
pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);    // deadlock

// Thread B
pthread_mutex_lock(&mutex2);
pthread_mutex_lock(&mutex1);    // deadlock
```

Per gestire la possibile creazione di deadlock, la libreria fornisce diverse **tipologie di mutex**:

- `PTHREAD_MUTEX_NORMAL`, il quale non è in grado di rilevare un deadlock:
  - Se un thread tenta di acquisire due volte un mutex senza prima rilasciarlo, si verificherà un deadlock
  - Se un thread tenta di rilasciare un mutex acquisito da un altro thread o di rilasciare un mutex non acquisito, il risultato sarà un undefined behavior
- `PTHREAD_MUTEX_ERRORCHECK`, il quale previene i deadlock generando degli errori
  - Se un thread tenta di acquisire due volte un mutex senza prima rilasciarlo, verrà generato un errore
  - Se un thread tenta di rilasciare un mutex acquisito da un altro thread o di rilasciare un mutex non acquisito, verrà generato un errore
- `PTHREAD_MUTEX_RECURSIVE`, il quale previene i deadlock permettendo di rieseguire alcune operazioni:
  - Se un thread tenta di acquisire due volte un mutex senza prima rilasciarlo, l'operazione andrà a buon fine, ma il thread dovrà rilasciare il lock due volte
  - Se un thread tenta di rilasciare un mutex acquisito da un altro thread o di rilasciare un mutex non acquisito, verrà generato un errore

L'uso dei mutex prevede che l'**ordine di accesso** alle sezioni critiche sia **casuale**, poiché non appena un mutex verrà rilasciato tutti i thread tenteranno contemporaneamente di acquisire il suo lock.

Alcune applicazioni, invece, richiedono che l'**ordine degli accessi sia coerente** e/o imposto dal programmatore, fenomeno che avviene invece all'interno del busy waiting, il quale tuttavia risulta essere generalmente una soluzione peggiore rispetto ai mutex.

Tale problematica può essere risolta tramite l'uso di **più mutex localmente condivisi** al fine di poter alternare i thread tra loro.

Esempio:

- Supponiamo che vi siano tre thread A, B e C, i quali condividono un mutex M. Supponiamo inoltre di voler effettuare gli accessi a tale mutex nell'ordine $A \to B \to C \to A \to \dots$
- Aggiungiamo quindi tre mutex P, Q R, dove il primo sarà condiviso tra i processi A e B, il secondo tra B e C e il terzo tra C e A
- A questo punto, ci basta far si che ogni accesso al mutex M possa essere effettuato solo dal thread che ha acquisito correttamente entrambi i suoi mutex localmente condivisi, per poi rilasciarli entrambi successivamente al rilascio di M

Tale soluzione, tuttavia, richiede l'uso di numerevoli operazioni lock e unlock, influendo sulle prestazioni del programma. Un'altra soluzione, invece, prevedere l'uso di una struttura dati più articolata rispetto ai mutex, ossia i **semafori**, i quali sono anche in grado di regolare l'ordine degli accessi ad una sezione critica.

Definizione 14: Semaforo

Un **semaforo** è una struttura dati dotata di una **coda** in grado di permettere solo ad un determinato numero di thread di accedere ad una sezione critica:

- A differenza dei mutex, non vi è il concetto di possesso del lock
- Sono dotati di un valore, il quale viene inizializzato alla creazione
- **Wait()**: se il valore del semaforo è 0, blocca il thread chiamante,aggiungendolo alla coda. Se invece è maggiore di 0, il valore viene decrementato e il thread accede alla sezione critica
- **Post()** se la coda non è vuota,rimuove e sblocca il primo thread della coda.Se invece la coda è vuota, viene incrementato il valore del semaforo

La libreria `semaphore` permette di utilizzare i semafori tramite le seguenti direttive:

- `sem_init()`: crea il semaforo
- `sem_destroy()`: elimina il semaforo
- `sem_wait()`: esegue Wait() sul semaforo
- `sem_post()`: esegue Post() sul semaforo

3.3.1 Implementazione dei lock

Dopo aver discusso del funzionamento dei mutex e dei semafori, possiamo analizzare il modo in cui essi siano internamente implementati.

In particolare, vogliamo che i lock dei mutex godano delle seguenti proprietà:

- Solo un thread/processo per volta può accedere alla sezione critica
- Le operazioni di accesso ed uscita dalla sezione critica non devono essere interrompibili da altri thread/processi o dal sistema operativo
- Nessuno deve rimanere in attesa perenne
- Devono funzionare per i programmi multi-thread
- La velocità della CPU deve essere irrilevante

Il primo punto risulta già garantito dall'uso delle direttive di acquisizione (**acquire()**) e di rilascio (**release()**) del lock.

Al fine di assicurare il secondo punto della lista, possiamo **disabilitare gli interrupt** di sistema durante l'accesso o uscita dalla sezione critica, per poi riabilitarli a fine operazione, permettendo inoltre che il thread venga interrotto durante l'esecuzione della sezione critica senza creare alcun problema poiché il lock non è stato rilasciato.

```
acquire(lock){
    disable interrupts;

    //spin lock
    while (lock.value != 0);
    lock.value = 1;

    enable interrupts;
}
```

```
release(lock){
    disable interrupts;

    lock.value = 0;

    enable interrupts;
}
```

Tale soluzione, tuttavia, genera alcune problematiche che vanno a violare le altre proprietà desiderate:

- La disabilitazione e abilitazione degli interrupt viene applicata sul core eseguente il thread chiamante, ma non sugli altri core.
- Se si verifica un bug nella sezione critica, l'intero sistema potrebbe bloccarsi a seguito della non riabilitazione degli interrupt
- I processi con priorità minore potrebbero essere schedulati prima (*priority inversion*)

Le prime due problematiche possono essere risolte riabilitando gli interrupt per un breve lasso di tempo all'interno dello *spin lock*, in modo che il processo possa essere bloccato se necessario:

```
acquire(lock){
    disable interrupts;

    //spin lock
    while (lock.value != 0){
        enable interrupts;
        disabile interrupts;
    }
    lock.value = 1;

    enable interrupts;
}
```

```
release(lock){
    disable interrupts;

    lock.value = 0;

    enable interrupts;
}
```

Tale soluzione, tuttavia, fa sì che vengano sprecati numerevoli cicli di clock per la continua disabilitazione e riabilitazione degli interrupt. Una soluzione più sofisticata prevede l'uso di una coda, in modo da evitare il busy waiting generato.

```
acquire(lock){
    disable interrupts;

    //spin lock
    while (lock.value != 0){
        enqueue myself;

        //give control to
        //another thread
        yield();
    }
    lock.value = 1;

    enable interrupts;
}
```

```
release(lock){
    disable interrupts;

    if (!queue.isEmpty()){
        dequeue a thread;
        wake that thread;
    }
    lock.value = 0;

    enable interrupts;
}
```

Sebben perfettamente funzionali, tali direttive risultano troppo complesse per il loro semplice scopo. Di conseguenza, per semplificare la loro implementazione vengono utilizzate direttamente delle **istruzioni atomiche**.

Definizione 15: Istruzione atomica

Un'istruzione è detta **atomica** se non può essere in alcun modo interrotta durante la sua esecuzione.

Esse sono in grado di funzionare anche su sistemi multicore, bloccando momentaneamente il bus della memoria e permettendo solo al processo chiamante di potervi accedere.

In particolare, viene utilizzata l'istruzione atomica **Test&Set (TAS)**:

- Se la variabile in input vale 0, viene impostata ad 1 e viene ritornato 1
- Altrimenti, viene ritornato 0

Tramite tale istruzione, l'implementazione dei mutex viene ridotta al minimo:

```
acquire(lock){
    while (!TAS(lock.value)){

        //give control to
        //another thread
        yield();
    }
}
```

```
release(lock){
    lock.value = 0;
}
```

3.4 Variabili condizionali

Definizione 16: Variabile condizionale

Una **variabile condizionale** è una struttura dati che permette ad un thread di interrompere la sua esecuzione finché un determinato evento non si verifica.

Quando quest'ultimo si verifica, un altro thread può segnalare tale evento al thread in attesa, svegliandolo.

Inoltre, nonostante il nome, una variabile condizionale non può assumere valori, bensì funge solo da "flag" per l'evento

Osservazione 15

Ad ogni variabile condizionale deve essere associato un proprio mutex.

Per utilizzare le variabili condizionali, la libreria `pthread` fornisce le seguenti funzioni:

- `pthread_cond_init()`: crea la variabile condizionale
- `pthread_cond_destroy()`: elimina la variabile condizionale
- `pthread_cond_wait()`: il thread chiamante rimane in attesa dell'evento. Richiede in input anche il mutex associato alla variabile condizionale (deve essere già lockato)
- `pthread_cond_signal()`: segnala e sveglia un thread a caso in attesa dell'evento
- `pthread_cond_broadcast()`: segnala e sveglia tutti i thread in attesa dell'evento

Osservazione 16: Variabili condizionali vs Semafori

La logica dietro le variabili condizionali differisce leggermente da quella dei semafori in quanto:

- `pthread_cond_wait()` blocca sempre l'esecuzione del thread
- Se non vi sono thread in attesa della variabile condizionale, la direttiva `pthread_cond_signal()` viene ignorata e il segnale non viene "conservato"
- Un semaforo può assumere valori maggiori o uguali a zero, mentre una variabile condizionale non può assumere alcun valore

Come per la libreria `mpi`, anche la libreria `pthread` fornisce la possibilità di utilizzare le **barriere**, in particolare tramite le seguenti direttive:

- `pthread_barrier_init()`: crea la barriera
- `pthread_barrier_destroy()`: elimina la barriera
- `pthread_barrier_wait()`: il thread chiamante raggiunge la barriera

In alternativa, è possibile implementare le barriere in due modi:

- Tramite un **mutex** e il **busy waiting**

```
int counter = 0;
int thread_count = ...;
pthread_mutex_t barrier_mutex = ...;
...

void* Thread_work(...){
    ...
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while(counter < thread_count);
    ...
}
```

- Tramite i **semafori**

```
int counter = 0;
int thread_count = ...;
sem_t count_sem = ...;      // initialized to 1
sem_t barrier_sem = ...;    // initialized to 0
...

void* Thread_work(...){
    ...
    sem_wait(&count_sem);

    if(counter == thread_count - 1){
        counter = 0;
        sem_post(&count_sem);
        for(int j = 0; j < thread_count - 1; j++){
            sem_post(&barrier_sem);
        }
    }
    else{
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    ...
}
```

- Tramite le **variabili condizionali**

```
int counter = 0;
int thread_count = ...;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
...

void* Thread_work(...){
    ...
    pthread_mutex_lock(&mutex);
    counter++;

    if(counter == thread_count){
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    }
    else{
        while(pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    ...
}
```

3.5 Strutture dati dinamiche condivise

Supponiamo di voler creare una struttura dati `List` dotata di tre operazioni:

- **Member()**, verifica se un valore è nella lista
- **Insert()**, inserisce un valore nella lista
- **Delete()**, elimina un valore dalla lista

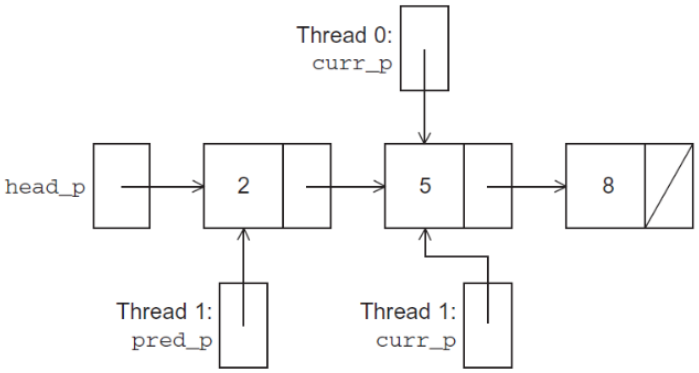
```
struct list_node_s{
    int data;
    struct list_node_s* next;
}

int Member(struct list_node_s* head_p, int value){
    struct list_node_s* curr_p = head_p;
    while(curr_p != NULL && curr_p->data < value){
        curr_p = curr_p->next;
    }
    return !(curr_p == NULL || curr_p->data > value);
}
```

```
int Insert(struct list_node_s** head_p, int value){
    struct list_node_s* curr_p = *head_p;
    struct list_node_s* pred_p = NULL;
    struct list_node_s* temp_p;

    while(curr_p != NULL && curr_p->data < value){
        pred_p = curr_p;
        curr_p = curr_p->next;
    }

    if(curr_p == NULL || curr_p->data > value){
        temp_p = malloc(sizeof(struct list_node_s));
        temp_p->data = value;
        temp_p->next = curr_p;

        if(pred_p == NULL)
            *head_pp = temp_p;
        else
            pred_p->next = temp_p;

        return 1;
    }
    return 0
}

int Delete(struct list_node_s** head_p, int value){
    struct list_node_s* curr_p = *head_p;
    struct list_node_s* pred_p = NULL;

    while(curr_p != NULL && curr_p->data < value){
        pred_p = curr_p;
        curr_p = curr_p->next;
    }

    if(curr_p == NULL || curr_p->data > value){
        if(pred_p == NULL)
            *head_pp = curr_p->next;
        else{
            pred_p->next = curr_p->next;
            free(curr_p);
        }
        return 1;
    }
    return 0
}
```

Supponiamo ora di voler utilizzare tale struttura dati all'interno di un **programma multithread**.

Per condividere l'accesso alla lista, è sufficiente definire il puntatore alla testa della lista come variabile globale. In tal modo, verranno anche semplificati gli header delle tre funzioni, poiché esse potranno accedere direttamente alla lista senza che venga passata in input.

Tuttavia, utilizzando tale lista si verifica una problematica ben più grande: i thread possono **accedere contemporaneamente a nodi diversi della lista**



Osserviamo quindi che:

- Le chiamate **Member** non modificano la lista e potrebbero essere eseguite tutte in parallelo tra loro
- Le chiamate **Insert e Delete** potrebbero generare problemi se eseguite in parallelo (es: un nodo potrebbe essere eliminato mentre un thread stava per accedervi)

Una prima soluzione banale a tale problema prevede l'uso di un mutex che "circondi" ogni chiamata a tali funzioni, equivalente ad eseguire un **lock sull'intera lista** ad ogni operazione.

```
pthread_mutex_lock(&list_mutex);
//call Member, Insert or Delete
pthread_mutex_unlock(&list_mutex);
```

In tal modo, viene **serializzato** l'accesso all'intera lista, implicando che solo un thread per volta possa accedervi. Di conseguenza, tale soluzione penalizza le chiamate Member, ma favorisce le chiamate Insert e Delete.

Una seconda soluzione prevedere che ogni nodo sia dotato di un proprio mutex, equivalente ad eseguire un **lock su ogni nodo** attualmente analizzato da un thread, permettendo ad ogni altro nodo di essere acceduto parallelamente.

```
struct list_node_s{
    int data;
    struct list_node_s* next;
    pthread_mutex_t mutex;
}
```

Sebbene possa inizialmente sembrare una soluzione più versatile rispetto alla precedente, essa risulta in realtà **peggiore**:

- Le tre funzioni richiedono una complessità maggiore
- Ogni volta che un nodo è acceduto bisogna acquisire e rilasciare il suo mutex, peggiorando notevolmente le performance
- Poiché ogni nodo possiede un proprio mutex, è richiesta una grande quantità di memoria

3.5.1 Read-Write locks

Definizione 17: Read-Write lock

Un **Read-Write lock** è una struttura dati simile ad un mutex dotata di due tipi di lock:

- Un **lock in lettura**, il quale può essere acquisito da più thread simultaneamente
- Un **lock in scrittura**, il quale può essere acquisito da un solo thread per volta
- Se un thread tenta di acquisire il lock in scrittura di un Read-Write lock il cui **lock in lettura è stato già acquisito**, tale thread rimarrà in attesa
- Se un thread tenta di acquisire il lock in lettura o scrittura di un Read-Write lock il cui **lock in scrittura è stato già acquisito**, tale thread rimarrà in attesa

La libreria `pthread` fornisce le seguenti direttive per i Read-Write lock:

- `pthread_rwlock_init()`: crea il Read-Write lock
- `pthread_rwlock_destroy()`: crea il Read-Write lock
- `pthread_rwlock_rdlock()`: acquisisce il lock in lettura, bloccandosi se necessario
- `pthread_rwlock_wrlock()`: acquisisce il lock in lettura, bloccandosi se necessario
- `pthread_rwlock_unlock()`: rilascia il lock

Tramite i Read-Write lock, possiamo migliorare la prima soluzione proposta per il problema della linked list condivisa tra thread, differenziando le tipologie di lock acquisite dalle tre funzioni:

```
pthread_rw_lock_rdlock(&rwlock);    // read lock
Member(value);
pthread_rw_lock_unlock(&rwlock);

pthread_rw_lock_wrlock(&rwlock);    // write lock
Member(value);
pthread_rw_lock_unlock(&rwlock);
```

```
pthread_rw_lock_wrlock(&rwlock);    // write lock
Member(value);
pthread_rw_lock_unlock(&rwlock);
```

Confrontiamo quindi le tre soluzioni proposte:

- 100000 operazioni per thread, di cui: 99.9% Member, 0.05% Insert e 0.05% Delete

	Number of threads			
Implementation	**1**	**2**	**4**	**8**
Single Read-Write lock	0.213	0.123	0.098	0.115
Single Mutex	0.211	0.450	0.385	0.457
One Mutex per node	1.680	5.700	3.450	2.700

- 100000 operazioni per thread, di cui: 80% Member, 10% Insert e 10% Delete

	Number of threads			
Implementation	**1**	**2**	**4**	**8**
Single Read-Write lock	2.48	4.97	4.69	4.71
Single Mutex	2.50	5.13	5.04	5.11
One Mutex per node	12.00	29.60	17.00	12.00

3.5.2 Optimistic locking e Lazy list

Metodo 2: Optimistic locking

L'**optimistic locking** è un metodo di gestione degli **accessi concorrenti** ad una struttura dati basata sull'**assenza di lock intermedi**:

1. Viene **scansionata** la struttura dati senza effettuare alcun lock
2. Vengono svolti i **preparativi** per l'operazione da eseguire, **lockando** i dati su cui operare
3. Viene eseguita una **nuova scansione** senza effettuare altri lock.
4. Se gli oggetti sono **ancora lockati** e tutte le **condizioni necessarie** ad eseguire l'operazione sono soddisfatte, quest'ultima viene eseguita, per poi togliere i lock. Altrimenti, viene annullata l'operazione.

Esempio:

- Possiamo modificare l'insert di una linked list affinchè essa utilizzi l'optimistic locking:
  1. Viene scansionata la lista senza effettuare alcun lock in cerca dei due nodi B e D tra cui inserire il nuovo nodo C, dove D è il successore di B nella lista

2. Una volta trovati i nodi B e D, essi vengono lockati
3. Viene nuovamente scansionata la lista.
4. Se B e D sono ancora lockati, B è stato trovato dalla scansione e D è ancora il successore di D, l'inserimento viene effettuato

Osservazione 17

L'optimistic locking risulta **efficace** solo se il costo di due scansioni senza lock è minore del costo di una singola scansione con lock.

Inoltre, poiché è richiesto che tutte le operazioni seguano tale procedura, anche le operazioni che normalmente non richiedono l'uso di lock devono effettuarlo durante l'accesso al dato

Esempio:

- Nel caso delle liste, è richiesto che anche l'operazione Member effettui il lock sul valore da trovare. Essendo essa l'operazione più frequentemente utilizzata sulle liste, ciò potrebbe ridurre notevolmente le prestazioni

Definizione 18: Lazy list

Una **lazy list** è una linked list in grado di gestire le rimozioni in modo lazy (*pigro*):

- Tutti i nodi sono dotati di una flag *removed*
- L'operazione Member **non esegue mai locking**
- L'operazione Delete esegue una scansione senza locking in cerca del nodo da eliminare, marcandolo come *removed* (**rimozione logica**)
- Le operazioni Insert e Delete si comportando come nell'**optimistic locking**, ma senza effettuare la seconda scansione
- Tutti i nodi marcati come *removed* vengono **ignorati** in ognuna delle tre operazioni, passando direttamente al nodo successivo
- Periodicamente, i nodi marcati come *removed* vengono eliminati dalla lista facendo puntare il suo predecessore al suo successore (**rimozione fisica**)

4

Open Multi-processing (OpenMP)