

CALCOLO PARALLELO E DISTRIBUITO – MARCELLINO

UNIVERSITÀ DEGLI STUDI DI NAPOLI
“PARTHENOPE”



Appunti a cura di
FIORENTINO MICHELE

INTRODUZIONE

I seguenti appunti sono stati scritti durante la frequentazione del corso di CALCOLO PARALLELO E DISTRIBUITO tenuto dalla professoressa Livia Marcellino nell' a.a. 2021/2022.

Calcolo parallelo e distribuito - INDICE

- 3. LEZ 1: Introduzione (1)
- 4. LEZ 2: Introduzione (2): tempo richiesto per l'esecuzione di un SW
- 5. LEZ 3: Tipi di parallelismo
- 9. LEZ 4: Somma di N numeri su architetture MIMD (le 3 strategie)
- 13. LEZ 5: Parametri di valutazione di un alg. Parallelo (MIMD SM)
- 16. LEZ 6: Parametri di valutazione di un alg. Parallelo (MIMD DM)
- 18. LEZ 5/6: Come scegliere T_1
- 19. LEZ 7-10: Introduzione a MPI & Somma di N numeri
 - 20. MPI:
 - 20. Funzioni per definire l'ambiente;
 - 21. Funzioni per comunicazioni uno ad uno;
 - 22. Funzioni per comunicazioni collettive;
 - 23. Operazioni collettive.
 - 24. Calcolo della somma di N numeri.
- 27. LEZ 12: Legge di Ware-Amdal
- 29. LEZ 13: Isoefficienza
 - 29. Isoefficienza
 - 31. Calcolo S_p , O_h e E_p per la somma di n numeri (MIMD DM)
 - 34. Calcolo S_p , O_h e E_p per la somma di n numeri (MIMD SM)
- 36. LEZ 15: Applicazione di Ware Amdal e isoefficienza (scalabilità)
 - 36. Ware Amdal Generalizzata
 - 40. Calcolo dell'isoefficienza.
- 41. LEZ 16: Introduzione ad OpenMP
 - 43. Direttive
 - 49. Runtime Library Routines
 - 49. Variabili d'ambiente
- 50. LEZ 21: Richiami di Alg. lineare e Prodotto matXvet
 - 50. Richiami di algebra lineare:
 - 50. Vettori
 - 51. Matrici
 - 56. Prodotto matrice per vettore
 - 56. MIMD DM
 - 58. MIMD SM
- 59. LEZ 22: Prendere i tempi in MPI e OpenMP
 - 61. MPI
 - 62. OpenMP

- 63. LEZ 23: Topologie virtuali: griglie
- 64. LEZ 24-27: Approfondimenti sul prodotto Matrice X Vettore
 - 64. 1 strategia:
 - 64. MIMD DM;
 - 66. MIMD SM.
 - 67. 2 strategia:
 - 67. MIMD DM;
 - 69. MIMD SM.
 - 70. Calcolo dell'isoefficienza
 - 71. Legge di WA

LAB: 11,14,17,18,19,20,28

LEZ 1 – Introduzione (1)

Il **calcolo ad alte prestazioni** (HPC, High Performance Computing) è quella branca dell'informatica che realizza infrastrutture in grado di realizzare enormi quantità di calcoli in brevi tempi.

Il vero **scopo** di un super computer è quello di risolvere problemi. È detto “super” in quanto si tratta del sistema che fornisce le prestazioni più elevate, in quel momento. In particolar modo, potrei volere una macchina ad alte prestazioni per risolvere problemi in “tempo reale (utile)” (es. previsioni meteo), o per risolvere problemi di grandi dimensioni.

Lo sviluppo tecnologico di una nazione può essere misurata attraverso la potenza dei suoi sistemi HPC.

Con **benchmark** si intende un insieme di test software utilizzati per misurare le prestazioni di un sistema.

Con un **Linpack benchmark** risolve un sistema denso di equazioni lineari.

LEZ 2 – Introduzione (2)

In generale, il **tempo richiesto per l'esecuzione di un software** τ (tau) è:

$$\tau = k \cdot T(n) \cdot \mu$$

dove:

- **k**: rappresenta il “tempo sprecato” dalla macchina per fare altre cose. Si tratta dell'elemento più imprevedibile;
- **T(n)**: rappresenta la complessità computazionale dell'algoritmo, inteso come numero di operazioni da fare. Dipende dall'algoritmo;
- μ : rappresenta il tempo di esecuzione di una operazione floating point. Dipende dal calcolatore.

nb: T(n) non deve essere confuso con τ , solo quest'ultimo rappresenta il tempo effettivo.

L' **obiettivo** sarà quello di avere un τ **quanto più ridotto possibile**, e per far ciò si deve cercare di ridurre k , $T(n)$ e μ quanto più possibile:

- **k**: si deve cercare di tenere la macchina quanto più “libera” possibile;
- **T(n)**: si deve riorganizzare l'algoritmo per pervenire ad uno con complessità di tempo inferiore.
Da notare che esiste, per ogni problema, un algoritmo di complessità minima sotto del quale non si può andare. Quindi è possibile ridurre T(n) solo fino ad un certo punto.
Inoltre, il compito del Calcolo Numerico non è solo quello di trovare l'algoritmo più veloce ma anche quello più efficiente dal punto di vista dell'accuratezza (riducendo l'errore quanto più possibile).
Infatti nel Calcolo Parallelo vogliamo fare “presto, ma bene”.
- μ : si devono ridurre le distanze fra la CPU e la memoria. Il processo di miniaturizzazione ha portato a distanze sempre inferiori, che tuttavia stanno trovando oggi limiti fisici.

Dunque è possibile ridurre τ solo fino ad un certo punto, per questo motivo abbiamo bisogno del calcolo parallelo.

Fondamentalmente, decomponiamo il problema in più sottoproblemi di dimensione inferiore, e li risolviamo contemporaneamente con più unità processanti.

Il prezzo da pagare è il tempo richiesto per unire i risultati.

LEZ 3 – Tipi di parallelismo

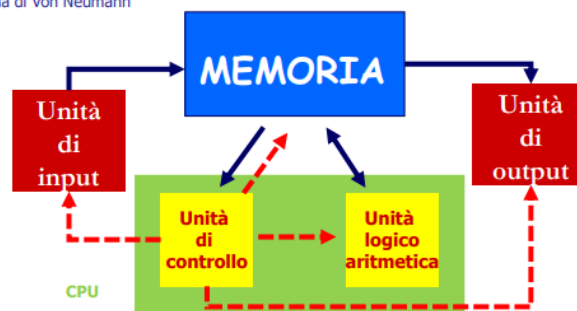
In origine, un calcolatore eseguiva tutte le sue azioni **sequenzialmente**, su un dato alla volta. Dunque, non veniva applicato alcun tipo di parallelismo.

Si tratta dell'architettura **SISD** (Single Instruction Single Data), che altro non è che la classica *architettura di Von Neumann*.

Una CPU contiene una:

- **CU**: esegue le operazioni riguardo il trasferimento di dati o al controllo dell'esecuzione di programmi;
- **ALU**: si occupa di eseguire le operazioni logiche ed aritmetiche.

Macchina di Von Neumann



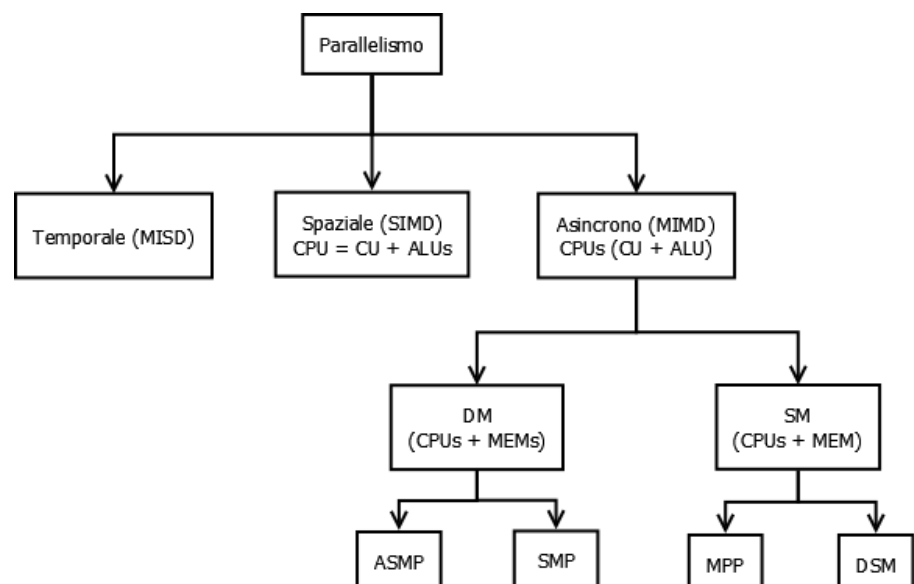
Nel corso del tempo si sono diffusi vari tipi di parallelismo, la cui implementazione avviene a livello hardware. Flynn dunque distinse varie architetture di calcolo, individuati dalla **tassonomia di Flynn**.

Distinguiamo **3 tipologie principali di parallelismo**: *temporale*, *spaziale* e *asincrono*.

Inoltre, distinguiamo due tipi di parallelismo **on-chip**, in base a se il parallelismo è effettuato su unico chip (temporale, spaziale) oppure su più chip (asincrono).

Attualmente, tutti i microprocessori utilizzano forme diverse di parallelismo, e nessun sistema in commercio si può definire puramente sequenziale.

Architetture correlate (Tassonomia di Flynn)		
	Istruzione Singola	Istruzioni Multiple
Dato Singolo	SISD	MISD
Dati Multipli	SIMD	MIMD



Parallelismo temporale

Il parallelismo temporale è realizzato attraverso la tecnica della *catena di montaggio*, detto anche parallelismo **pipeline**.

Consiste nel suddividere un lavoro in più fasi semplici ed affidare ogni fase ad una specifica unità, come se si trattasse di una catena di montaggio.

Questa tipologia di architetture prende il nome di **MISD** (Multiple Instruction Single Data).

Si pensi ad un gruppo di operai che costruiscono un muro: uno mette la calce, uno mette il mattone, un altro livella. Il lavoro è stato diviso in più fasi semplici.

Il parallelismo temporale in un calcolatore è realizzata all'interno della **ALU**, inserendo all'interno di questa **più unità funzionali**.

Ad esempio: l'operazione di **addizione floating point** è divisa in 4 segmenti, dove ciascun segmento è preposto all'esecuzione di una fase dell'operazione:

1. **confronto degli esponenti** (per vedere quale numero è più grande);
2. **shift della mantissa** (per portare il tutto ad un'unica dimensione);
3. **somma delle mantisse**;
4. **normalizzazione**.

Nella somma di N numeri "tradizionale" (cioè senza parallelismo), solo un segmento alla volta è attivo mentre gli altri rimangono inattivi.

Nella somma di N numeri "pipelined", a regime tutti i segmenti sono attivi contemporaneamente (cioè quando arriva per la prima volta alla quarta fase).

Nel primo caso avrò un tempo $4Nt_u$, nel secondo caso avrò un tempo $< 4Nt_u$.

L'uso di unità funzionali pipelined è anche alla base dei **processori vettoriali**, capaci di operare efficientemente su dati strutturati sotto forma di vettori, anche se attualmente **tutti i microprocessori** utilizzano una struttura pipeline per migliorare le loro prestazioni.

nb: (?) le slide fanno vedere che SIDS -> p. temporale, anche se è sequenziale

Parallelismo spaziale

Il parallelismo spaziale consiste nel far eseguire contemporaneamente a più unità la stessa azione ma su parti diverse.

Questa tipologia di architetture prende il nome di **SIMD** (Single Instruction Multiple Data).

Si pensi ad un gruppo di operai che costruiscono un muro: ogni operaio si occupa di mettere la calce, mettere un mattone e livellarlo. Abbiamo più operai che eseguono lo stesso lavoro.

Il parallelismo spaziale in un calcolatore è realizzato inserendo più ALU all'interno della CPU, le quali operano sotto il comune controllo di una CU.

Le architetture SIMD sono nate proprio grazie alla miniaturizzazione (più ALU sulla stessa CPU). Inoltre, siccome abbiamo una sola CU, tale parallelismo è anche detto *parallelismo sincrono*.

Parallelismo asincrono

Il parallelismo asincrono consiste nel far eseguire contemporaneamente a più unità azioni diverse su parti diverse.

Questa tipologia di architetture prende il nome di **MIMD** (Multiple Instruction Multiple Data).

Si pensi ad un gruppo di operai che costruiscono una casa: in questo caso ogni operaio penserà e agirà liberamente in quanto dovrà risolvere uno specifico problema.

Il parallelismo asincrono in un calcolatore può essere realizzato utilizzando più CPU (ALU + CU). In particolar modo, possiamo distinguere **due tipologie** di calcolatori MIMD: *Distributed Memory* e *Shared Memory*.

• DM (Distributed Memory)

I calcolatori DM sono costituiti da più CPU ognuna delle quali ha una propria memoria. Si parla di “cluster” di processori.

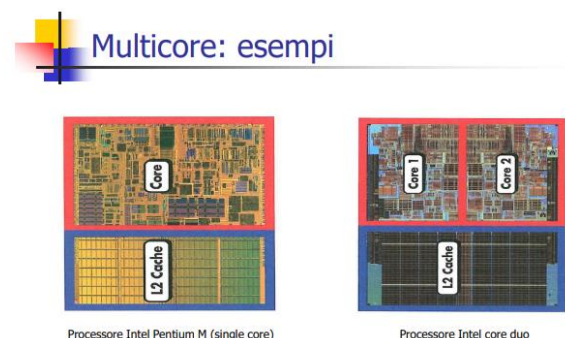
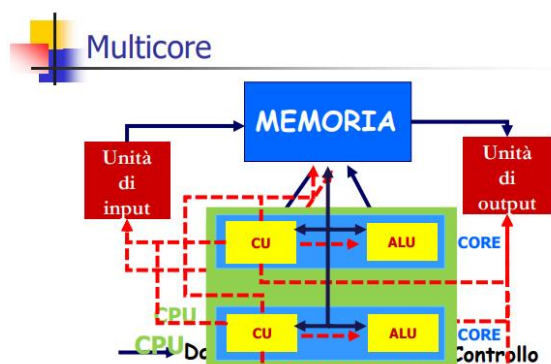
Distinguiamo due categorie di MIMD DM:

- **MMP** (Massive Parallel Processors): architettura con CPU in PC indipendenti che lavorano in parallelo;
- **DSM** (Distributed Shared Memory): architettura con CPU in PC indipendenti che lavorano in parallelo, ma dove uno dei PC ha una memoria condivisa virtuale.

• SM (Shared Memory)

I calcolatori SM sono costituiti da più CPU connesse ad una singola memoria principale in un'unica macchina. Ne distinguiamo due categorie:

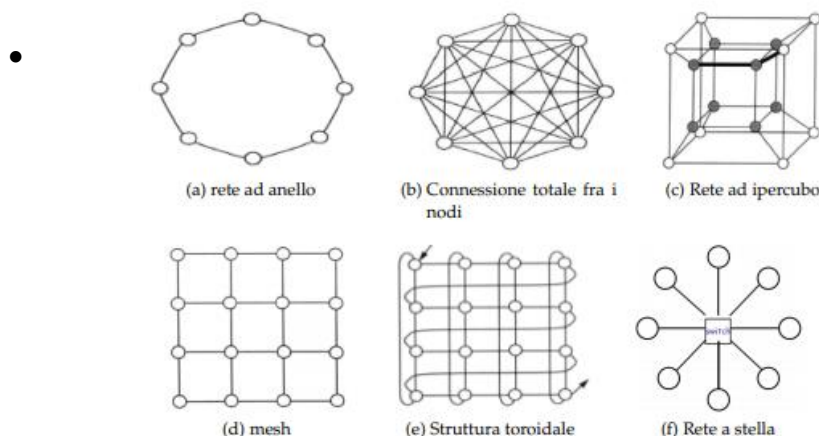
- **SMP** (Symmetric Multi processor): tutti i microprocessori sono **identici**, in quanto tutti i core hanno la stessa funzione (*multicore Intel*);
- **ASMP** (Asymmetric Multi processor): i microprocessori sono **diversi**, in quanto NON tutti i core hanno la stessa funzione (*PS5*).



Nelle macchine MIMD, CPU e memorie possono essere **collegate** in diverso modo a seconda del tipo di problema e applicazione della macchina.

In un calcolatore **MIMD SM**, la memoria e le CPU possono essere collegate con un *bus singolo*, con *bus multipli* oppure *in modo indiretto*.

In un calcolatore **MIMD DM**, le memorie e le CPU possono essere collegate anche con schemi di connessione più complessi, come:



Le **GPU** rappresentano la più sofisticata forma di macchina parallela. Se in un normale processore multicore abbiamo per ogni core 1 CU + 1 ALU, in una GPU abbiamo per ogni core 1 CU + 1 stream di ALU. Sono specializzate in eseguire calcoli semplici ma su un'enorme quantità di dati, e infatti si occupano di risolvere sistemi di equazioni lineari.

Calcolo parallelo VS Calcolo distribuito

Un **ambiente di calcolo parallelo** è un sistema di unità processanti *strettamente collegate* che comunicano per risolvere problemi su larga scala in maniera efficiente.

Un **ambiente di calcolo distribuito** è un sistema di unità processanti *autonome e indipendenti, fisicamente distribuite* che comunicano per risolvere problemi su larga scala in maniera efficiente.

La differenza è nella rete di connessione.

L'**obiettivo principale di un calcolatore parallelo** sono le performance, e infatti si cerca di ridurre il tempo necessario alla risoluzione computazionale di un problema reale.

L'**obiettivo principale di un sistema ad architettura distribuita** è il riuso delle risorse esistenti, e infatti si cerca di riutilizzare "efficacemente" le risorse hardware e software distribuite geograficamente sul territorio.

Infatti il calcolo parallelo mira ad eseguire prima l'algoritmo, mentre il calcolo distribuito mira ad abbassare i costi (le performance peggiorano perché si perde tempo per l'invio di dati, in quanto aumenta la distanza).

Con il tempo il Grid Computing perde piede in conseguenza della miniaturizzazione, che ha reso il calcolo parallelo sempre più economico. Tuttavia di contro, si sviluppa l'infrastruttura Cloud.

LEZ 4 – Somma di N numeri su architetture MIMD

La somma di N numeri, nonostante sia un classico esempio di parallelizzazione, è anche uno degli algoritmi meno parallelizzabili in quanto, per qualsiasi strategia adottiamo, siamo costretti ad effettuare almeno una somma sequenziale.

Dobbiamo **decomporre** il problema in più sottoproblemi e risolverli contemporaneamente su più calcolatori. L'idea è **suddividere** la somma in somme parziali ed assegnare ciascuna somma parziale ad un processore; le somme parziali devono poi essere **combinare** in modo opportuno per ottenere la somma totale.

Le strategie da adottare cambiano in base al sistema MIMD che usiamo:

- un sistema **MIMD DM** avrà la necessità di **organizzare le comunicazioni tra i processi** per trasferire le somme parziali da un processore all'altro. Inoltre, le prime 2 strategie conterranno la somma totale in un solo processore, per cui se abbiamo bisogno che tale somma sia a disposizione anche degli altri processori dovremo fare un'operazione di broadcast.
- un sistema **MIMD SM** ha la possibilità di **collezionare i risultati senza comunicazione**. Essendo tuttavia la memoria condivisa dobbiamo stare attenti a non creare situazioni di race condition (sincronizzare la memoria).

Il risparmio di comunicazione è importante in quanto un'operazione di trasferimento dati è molto più costosa di un'operazione floating point.

nb: NP = numero processori/core.

In un sistema **MIMD DM** dovremo innanzitutto distribuire i dati nelle memorie locali dei processori, e questi calcoleranno concorrentemente le somme parziali.

Possiamo distinguere **3 strategie**:

I Strategia

Ad ogni passo, ciascun processore invia la sua somma parziale ad un unico processore prestabilito. Tale processore contiene la somma totale.

I passi sono $NP-1$. In questa strategia ho una netta separazione tra la fase di calcolo (qui strettamente parallela) e la fase di collezione dei dati (qui strettamente seriale).

Non è la strategia migliore in quanto le altre CPU non lavorano nel mentre che i dati devono essere trasferiti.

II Strategia

Ad ogni passo, coppie distinte di processori comunicano contemporaneamente; in ogni coppia, un processore invia all'altro la propria somma parziale che provvede all'aggiornamento della somma.

Il risultato è in un unico processore prestabilito.

I passi sono $\log_2(NP)$, considerando che NP sia potenza di 2. In questa strategia (anche) la somma parziale avviene in parallelo, ad eccezione dell'ultimo passo che deve necessariamente avvenire sequenzialmente.

Nel caso NP non sia potenza di 2, un'idea è calcolare le somme parziali separatamente e andremo sommarle a quella totale sequenzialmente.

III Strategia

Ad ogni passo, coppie distinte di processori comunicano contemporaneamente; in ogni coppia, i processori si scambiano le proprie somme parziali.

Il risultato è in tutti i processori.

In tal modo tutti i processori sono sempre attivi. Il trasferimento dei dati è sempre da considerarsi per uno, perché avvengono nello stesso passo. Inoltre non avrò bisogno di fare un broadcast (eventualmente) in quanto le somme sono già presenti nelle memorie di tutti i processori.

Sfrutto al meglio i processori e risparmio eventuali spedizioni.

Nel caso NP non sia potenza di 2, un'idea è calcolare le somme parziali separatamente e andremo sommarle a quella totale sequenzialmente.

In un sistema **MIMD SM**, assegneremo un insieme di dati differenti ad ogni core, e questi calcoleranno concorrentemente le somme parziali.
La somma totale può essere ottenuta attraverso **2 strategie**:

I Strategia

Ad ogni passo, ciascun core deve addizionare la propria somma parziale ad una variabile che conterrà la somma finale (es. sumtot).

I core devono accedere a sumtot uno alla volta. Le somme finali avvengono in modo sequenziale in quanto ho un solo core attivo per volta.

II Strategia

Ad ogni passo, la metà dei core (rispetto al passo precedente) calcola un contributo alla somma parziale.

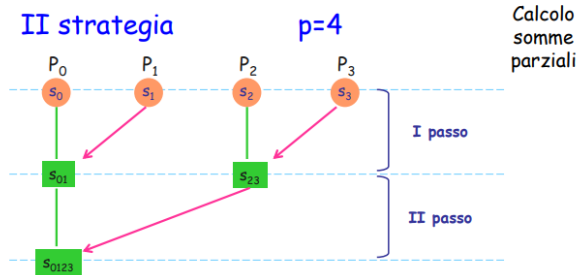
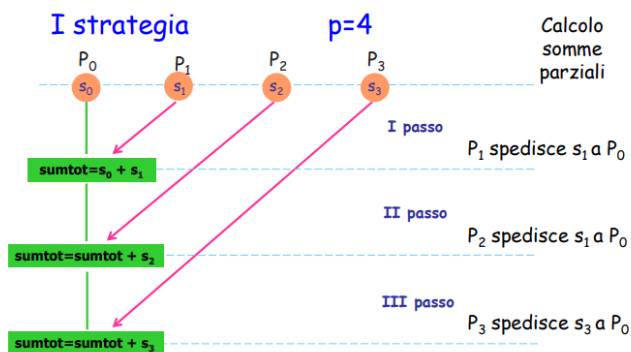
La somma finale (sumtot) sarà data dalla somma degli ultimi due contributi (obv. se NP è potenza di 2).

Non esiste una terza strategia, in quanto il risultato è sempre nell'unica memoria condivisa.

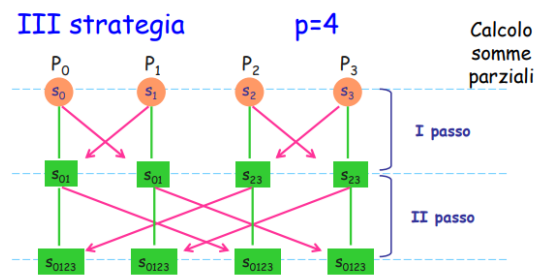
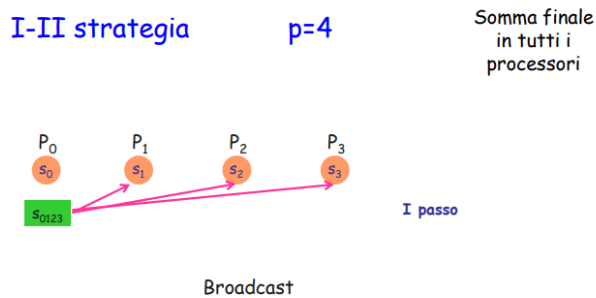
Strumenti software per lo sviluppo di algoritmi in ambienti:

- MIMD DM: MPI (Message Passing Interface);
- MIMD SM: OpenMP, Pthreads, ...

Esempi strategie

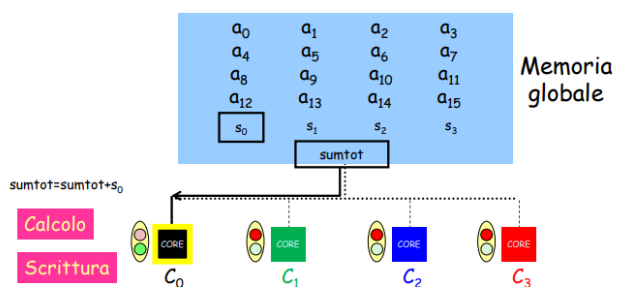


Il numero di "passi" è 2



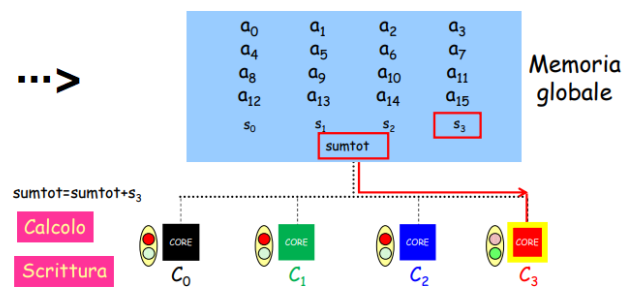
Esempio: Somma 1strategia

- Esempio: $N=16, p=4$



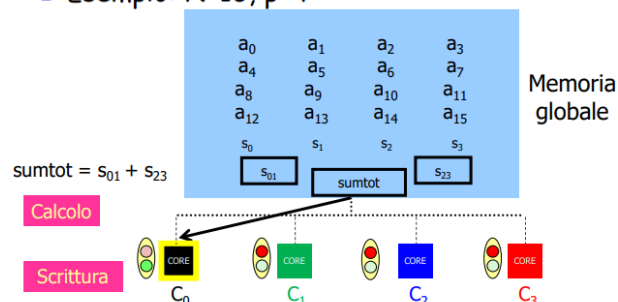
Esempio: Somma 1strategia

- Esempio: $N=16, p=4$



Somma (II strategia MIMD-SM)

- Esempio: $N=16, p=4$



LEZ 5 – Parametri di valutazione di un algoritmo parallelo (MIMD-SM)

L'**efficienza**, in un algoritmo parallelo, ci permette di valutare quanto ci convenga implementare un algoritmo parallelo.

Nel caso di un **algoritmo sequenziale**, l'efficienza di un algoritmo misura l'uso di risorse quali CPU e spazio in memoria da parte dell'algoritmo stesso.

Dobbiamo tener quindi conto di due parametri:

- complessità di tempo $T(n)$;
- complessità di spazio $S(n)$.

Questi due parametri spesso non vanno a braccetto fra di loro, ed anzi, sono spesso inversamente proporzionali.

Nel caso di un **algoritmo parallelo**, un calcolatore parallelo è in grado di eseguire più operazioni concorrentemente, dunque allo stesso passo temporale. Da qui viene che il tempo di esecuzione non è proporzionale alla complessità di tempo.

Dunque, la complessità di tempo $T(n)$ non è adatta a misurare l'efficienza di un algoritmo parallelo.

Nei parametri di valutazione di un software in ambiente parallelo ci sono alcune differenze in base a se si considera un calcolatore MIMD-DM o MIMD-SM;

MIMD SM

Se si esegue l'algoritmo su un **calcolatore MIMD SM**, il tempo di esecuzione dipende solo dal numero di operazioni eseguite in differenti passi temporali

Il *tempo di esecuzione di un software* in un ambiente MIMD SM è anche esprimibile come:

$$\tau_p = k \cdot T_p(N)$$

- dove p rappresenta il numero di processori;
- dove μ è inglobato dentro T_p , e $\mu = t_{\text{calc}}$ (t. addizione).

In un **calcolatore MIMD SM**, T_p è calcolato come:

$$T(p) = \left(\frac{n}{p} - 1 + \log_2 p \right) t_{\text{calc}}$$

nb: da ora potremmo anche incontrare $\tau_1 = k * n * t_{\text{calc}}$, invece di $\tau = k * T(n) * \mu$, dove nel caso della somma di N numeri, n rappresenta il numero di numeri e t_{calc} rappresenta il tempo di esecuzione di un'addizione.

Speed up S_p

Siano T_1 e T_p il tempo di esecuzione di un algoritmo in un ambiente di calcolo dotato rispettivamente di 1 e di p processori/core,

lo **speed up** S_p indica di quanto si riduce il tempo d'esecuzione di un algoritmo su p processori rispetto alla versione sequenziale ed è definito come il rapporto di T_1 su T_p :

$$S_p = \frac{T_1}{T_p}$$

Maggiore è lo speed up, tanto più un algoritmo parallelo è veloce rispetto al corrispondente algoritmo sequenziale. **Idealmente**, ci aspettiamo che un algoritmo in parallelo su p processori sia p volte più veloce della corrispondente versione seriale, ma **in realtà** ogni algoritmo è composto da una parte seriale non parallelizzabile, che dunque restituirà un *numero strettamente minore* di p .

$$S(2)_{ideale} = \frac{T(1)}{T(2)} = 2 \quad \dots \quad S(4)_{ideale} = \frac{T(1)}{T(4)} = 4 \quad \dots \quad S(p)_{ideale} = \frac{T(1)}{T(p)} = p$$
$$S(p)_{reale} = \frac{T(1)}{T(p)} < p$$

Un algoritmo parallelo, quindi, è tanto più veloce quanto più lo *speed up* si avvicina allo *speed ip ideale*.

Overhead totale O_h

L'**overhead totale** $O_h(p)$ misura la differenza fra lo speed up ottenuto e lo speed up ideale. In una situazione ideale $O_h = 0$.

$$S_p^{ideale} = \frac{T_1}{T_p} = p$$
$$\boxed{O_h = (pT_p - T_1) t_{calc}} \quad \text{OVERHEAD totale} \quad \rightarrow \quad T_p = (O_h + T_1) t_{calc} / p$$
$$S_p = \frac{T_1}{T_p} = \frac{T_1}{(O_h + T_1)/p} = \frac{pT_1}{O_h + T_1} = \frac{p}{\frac{O_h}{T_1} + 1}$$

Dalla forma di overhead possiamo inoltre fare un'**osservazione**:

Riscrivendo la formula in funzione di p ed O_h , ci accorgiamo che **al crescere di p , l'overhead aumenta**.

Ciò significa che sebbene usare più processori comporta una riduzione del tempo di esecuzione, comporta anche maggior overhead, e dunque l'algoritmo risulta meno *efficiente*.

Efficienza E_p

L'efficienza E_p di un algoritmo parallelo è definita come il rapporto fra lo speed up e il numero di processori e quantifica lo sfruttamento del parallelismo del calcolatore da parte dell'algoritmo.

Tanto più l'efficienza di un algoritmo si avvicina ad 1, tanto più l'algoritmo parallelo sfrutta le risorse di calcolo (cioè, è efficiente).

Un'efficienza ideale sarebbe uguale a 1, ma nella realtà avremo sempre un numero < 1 .

Es.

Consideriamo l'algoritmo per il "calcolo della somma di N numeri". L'algoritmo su 8 processori è il più veloce. Infatti è 3.75 volte più veloce di quello su 1 processore, ma lo speed up su 2 processori è il "più vicino" allo speed up ideale.

Infatti se si rapporta lo speed up al numero di processori, il maggior sfruttamento dei processori ce l'ho proprio per $p=2$.

p	T_p	T_1/T_p
1	$15 t_{calc}$	1.00
2	$8 t_{calc}$	1.88
4	$5 t_{calc}$	3.00
8	$4 t_{calc}$	3.75

p	Speed-up ottenuto	Speed-up ideale
2	1.88	2
4	3.00	4
8	3.75	8

p	S_p	S_p/p
2	1.88	0.94
4	3.00	0.75
8	3.75	0.47

Rapporto più grande

LEZ 6 - Parametri di valutazione di un algoritmo parallelo (MIMD-DM)

Se si esegue l'algoritmo su un **calcolatore MIMD DM**, il tempo di esecuzione NON dipende solo dal numero di operazioni eseguite in differenti passi temporali, ma anche dal tempo delle comunicazioni.

Possiamo dire che la complessità MIMD DM è data dalla complessità MIMD SM + le comunicazioni.

Il tempo di esecuzione di un software in un ambiente MIMD DM è anche esprimibile come:

$$\tau_p = k \cdot T_p(N)$$

- dove p rappresenta il numero di processori;
- dove μ è inglobato dentro T_p , e $\mu = t_{\text{calc}} + t_{\text{com}}$ (t. addizione + t. comunicazione).

In un **calcolatore MIMD DM** T_p è calcolato come:

$$p=2^k \quad T(p) = \left(\frac{n}{p} - 1 + \log_2 p \right) t_{\text{calc}} + (\log_2 p) t_{\text{com}}$$

nb: vale solo se $p=2^k$.

Le comunicazioni sono tante quanto sono i passi.

Per lo **Speed up** vale lo stesso discorso fatto per i calcolatori MIMD SM.

Overhead di Comunicazione Unitario

L'**Overhead di comunicazione unitario** (OC) misura il rapporto fra il tempo di comunicazione e il tempo di un'operazione f.p.

Dipende dal cluster che sto usando, e in particolar modo mette in relazione la macchina e il cavo che viene usato (quanto tempo ci mette il dato per viaggiare).

$$OC = \frac{t_{\text{com}}}{t_{\text{calc}}}$$

Questo Overhead non è MAI 1, anzi generalmente per spedire un dato impiego il doppio (se non il triplo) del tempo di un'operazione floating point.

Overhead di Comunicazione Totale

L'**Overhead di comunicazione totale** (OC_p) misura il rapporto fra il tempo di comunicazione dell'algoritmo eseguito su p processori e il tempo di calcolo dell'algoritmo eseguito su p processori.

Fornisce una misura del "peso" della comunicazione sul tempo di esecuzione dell'algoritmo, e mi permette di capire se mi conviene parallelizzarlo o meno.

$$OC(p) = \frac{T_{com}(p)}{T_{calc}(p)}$$

In generale, più aumenta il numero di processore e più aumenta tale Overhead.

È importante che l'Overhead sia < 1 , altrimenti avrei che il tempo di comunicazione pesa di più rispetto al tempo di comunicazione (non mi conviene).

...e in tutto ciò, non stiamo ancora considerando il tempo impiegato per la distribuzione iniziale dei dati.

ES.

Supponiamo di avere $n=16$ e $p=8$.

Supponiamo che t_{calc} sia 1, e che t_{com} sia 2. Dunque $OC = 2$.

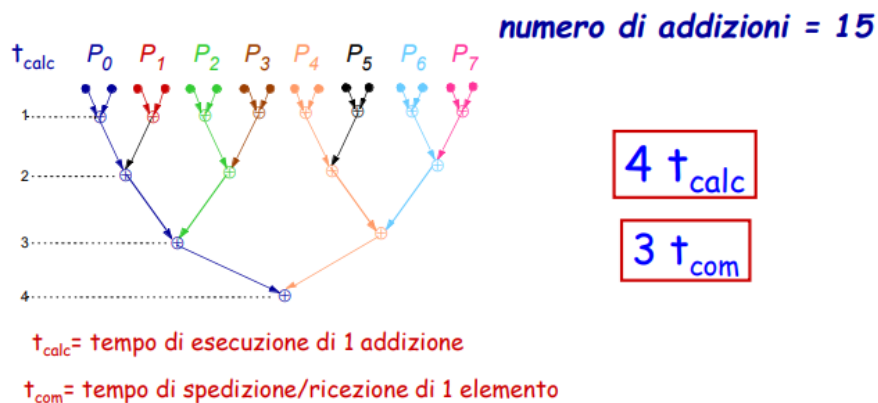
Se sostituiamo n e p nella formula, avremo che: $T_p = (1+3)t_{calc} + 3t_{com} \rightarrow T_p = 4t_{calc} + 3t_{com}$

Dunque, $T_p = 4 + 6$.

Quindi osserviamo che il tempo di comunicazione è maggiore di quello di calcolo. Infatti se calcoliamo l' OC_p ci rendiamo conto che $OC_p = 1.50$.

ALGORITMO PARALLELO

II-III strategia $p=8$



COME SCEGLIERE T_1

Nello speed-up consideriamo $S_p = T_1/T_p$. Tuttavia T_1 può essere misurato in due modi:

- **Prima scelta**

T_1 = tempo di esecuzione dell'**algoritmo parallelo su 1 processore**.

S_p dà informazioni su quanto l'algoritmo **si presta all'implementazione** su un'architettura parallela.

Lo **svantaggio** è che l'algoritmo parallelo su 1 processore potrebbe essere più operazioni del necessario (algoritmo superlineare).

- **Seconda scelta**

T_1 = tempo di esecuzione del **miglior algoritmo sequenziale**.

S_p dà informazioni sulla **riduzione effettiva del tempo** nella risoluzione di un problema con p processori.

Lo **svantaggio** è sta nella difficoltà nell'individuare il "miglior" algoritmo sequenziale, e potrebbe comunque non essere il migliore.

Per **convenzione**, si adotta la **prima scelta con efficienza**. Cioè: si considera l'algoritmo parallelo eseguito con una sola unità processante, ma teniamo conto anche delle informazioni ottenute da S_p e E_p su quanto l'algoritmo si presta all'implementazione su un'architettura parallela.

Inoltre, prima di parallelizzare un algoritmo sequenziale, dobbiamo assicurarci che l'algoritmo sequenziale stesso sia ottimizzato (non ha senso parallelizzare uno "scarso" algoritmo sequenziale, e inoltre la parallelizzazione avviene dopo l'ottimizzazione).

LEZ 7:10 – Introduzione MPI & Somma di N numeri

Ricordiamo che: i calcolatori MIMD SM hanno necessità di sincronizzare gli accessi in scrittura e organizzare il lavoro; i calcolatori MIMD DM hanno necessità di organizzare le comunicazioni tra i processi.

Queste due problematiche verranno gestite con librerie apposite: **MPI** nel caso **MIMD DM**.

Un **thread** è un *flusso di istruzioni indipendente* che deve essere eseguito *sequenzialmente* su una CPU.

Un **processo** è un programma in esecuzione.

Un processo è costituito da almeno un thread (può contenerne più di uno).

Thread diversi possono essere eseguiti indipendentemente su CPU/core diversi, rispettivamente nel caso dei calcolatori MIMD DM e MIMD SM.

Somma di N numeri (1 strategia)

In pseudocodice, con *forall* e *endforall* si intende la porzione di codice che gira su tutti i processori che fanno parte del cluster. Rappresenta la parte parallelizzabile (non confondere con il for, non è un ciclo).

In pseudocodice, possiamo pensare di dare un identificativo *i* ad ogni processore, così da dare un nome ad ogni macchina. Tutti i processori inizializzeranno una variabile sum_i a 0. La *i* ci ricorda a quale macchina appartiene la somma.

Gli elementi da sommare sono presenti in un array A. Tutti dispongono dello stesso array, ma ogni processore lavora su una porzione diversa, che va da:

h to $h+(n/p)-1$, dove $h = i*(n/p)$

Finito il forall, si *collezionano i risultati*. Un processore farà da Master (colui che raccoglie i dati), che convenzionalmente è P_0 .

Il codice è lo stesso per tutti, dunque ogni processore controllerà se è P_0 .

P_0 chiamerà una *receive* per ogni processore, mentre gli altri spediranno con una *send* le loro somme parziali a P_0 .

Algoritmo sequenziale

```
begin
  sumtot := a0;
  for i=1 to N-1 do
    sumtot := sumtot + ai;
  endfor
end
```

I Strategia

```
begin
  forall Pi , 0 ≤ i ≤ p-1 do
    sumi := 0
    h := i * (n/p)
    for j = h to h+(n/p)-1 do
      sumi := sumi + aj
    endfor
  endforall
  if P0 then
    sumtot := sum0
    for k = 1 to p-1 do
      recv(sumk, Pk)
      sumtot := sumtot + sumk
    endfor
  else if Pi then
    send(sumi, P0)
  endif
endif
end
```

MPI

Ogni processore può conoscere i dati in memoria di un altro processore, o far conoscere i propri, attraverso il **trasferimento** di dati (message passing).

Ad ogni processore è associato un *identificativo* (intero fra 0 e p-1). A ciascun gruppo di processori è associato un *communicator*.

Un **communicator** definisce l'ambito in cui avvengono le comunicazioni tra i processori di un stesso gruppo.

Di default, tutti i processore fanno parte del communicator **MPI_COMM_WORLD**; se voglio lavorare con meno processori, dovrò definire un altro communicator.

MPI è una libreria che comprende:

- funzioni per definire l'**ambiente**;
- funzioni per **comunicazioni uno a uno**;
- funzioni per **comunicazioni collettive**;
- funzioni per **operazioni collettive**.

Per comunicazioni che coinvolgono solo due processori, si considerano le funzioni MPI per c. uno a uno; per comunicazioni che coinvolgono più di due processori si considerano funzioni MPI le c. collettive.

• Funzioni per definire l'ambiente

- `MPI_Init(&argc, &argv);`

Inizializza l'ambiente di esecuzione MPI e il communicator **MPI_COMM_WORLD**.
I due dati di Input sono gli argomenti del main.

- `MPI_Finalize();`

È l'equivalente del destroy. Pone fine a tutto ciò che avviene in parallelo.

- `MPI_Comm_rank(MPI_Comm comm, int *menum);`

Permette al processore chiamante, appartenente al communicator `comm`, di memorizzare il proprio identificativo nella variabile `menum`.

- `MPI_Comm_size(MPI_Comm comm, int *nproc);`

Ad ogni processore del communicator `comm`, restituisce in `nproc` il numero totale di processori che fanno parte di quel communicator.

Permette di conoscere quanti processori concorrenti possono essere utilizzati per una determinata operazione.

• Funzioni per comunicazioni uno ad uno

Un dato che deve essere spedito o ricevuto attraverso un messaggio MPI è descritto dalla tripla (address, count, datatype):

- address: indirizzo in memoria del dato;
- count: dimensione del dato;
- datatype: tipo del dato. Ogni tipo di dato di MPI corrisponde univocamente ad un tipo di dato del linguaggio C (es. MPI_INT -> int, MPI_FLOAT -> float, ecc).

Per la **spedizione** e la **ricezione** utilizzeremo le funzioni **bloccanti** MPI_Send e MPI_Recv:

- MPI_Send(void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

Il processore che esegue questa routine spedisce i primi count elementi di buffer (indirizzo del dato da spedire), di tipo datatype, al processore con identificativo dest. L'identificativo tag individua univocamente il messaggio nel contesto comm.

- MPI_Recv(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);

Il processore che esegue questa routine riceve i primi count elementi di buffer (indirizzo del dato in cui ricevere), di tipo datatype, dal processore con id. source. L'identificativo tag individua univocamente il messaggio nel contesto comm. status racchiude informazioni sulla ricezione del messaggio.

Per **verificare se si è presentato qualche problema**, possiamo usare MPI_Get_Count.

- MPI_Get_Count(MPI_Status *status, MPI_Datatype datatype, int *count);

Il processore che esegue questa routine memorizza nella variabile count il numero di elementi, di tipo datatype, che riceve dal messaggio e dal processore indicati nella variabile status.

MPI_Status, in C, è un tipo di dato strutturato composto da 3 campi:

- identificativo del processore da cui ricevere;
- identificativo del messaggio;
- indicatore di errore.

• Funzioni per comunicazioni collettive

Per le **comunicazioni collettive** possiamo distinguere due situazioni:

1. P_0 parla e tutti ascoltano;
2. P_0 ha un vettore X , e distribuisce a tutti gli altri processori un elemento del vettore.

Per fare questo esistono apposite funzioni, come **MPI_Bcast()**, la quale è una funzione bloccante che fa in modo che il master legga un valore (es. n) e che ne spedisca il valore a tutti i processori che fanno parte del communicator.

nb: solo il master dovrà leggere e mettere in memoria il valore, e poi dovrà occuparsi di **distribuire** i dati attraverso un broadcast (o anche una serie di send, volendo).

```
• MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
            int root, MPI_Comm comm);
```

Il processore, con identificativo `root`, spedisce a tutti i processori del communicator `comm` lo stesso dato memorizzato in `*buffer`.

`count`, `datatype`, `comm` devono essere uguali per ogni processore di `comm`.

Un'altra tipologia di **spedizione collettiva** è il **data scatter**.

Il data scatter ci permette di effettuare una gestione dei dati più semplice, in quanto si occupa lui stesso della distribuzione dei dati.

Un processore distribuisce i propri dati agli altri processori, se stesso compreso.

```
• MPI_Scatter(void *send_buff, int send_count, MPI_Datatype send_type,
             void *recv_buff, int recv_count, MPI_Datatype recv_type,
             int root, MPI_Comm comm);
```

Il processore con identificativo `root` distribuisce i dati contenuti in `send_buff`. Il contenuto di `send_buff` viene suddiviso in `nproc` segmenti ciascuno di lunghezza `send_count`, dove l' i -simo segmento verrà affidato all' i -simo processore.

Una routine analoga di **spedizione collettiva**, ma che fa il contrario, è il **data gather**.

Il data gather ci permette di concatenare i dati.

Tutti i processori spediscono i propri dati ad un processore assegnato (es. P_1).

```
• MPI_Gather (void *send_buff, int send_count, MPI_Datatype send_type,
             void *recv_buff, int recv_count, MPI_Datatype recv_type,
             int root, MPI_Comm comm);
```

Ogni processore di `comm` spedisce il contenuto di `send_buff` al processore con identificativo `root`.

Il processore con identificativo `root` concatena i dati ricevuti in `recv_buff`, memorizzando i dati ricevuti dai vari processori.

nb: gli argomenti di `recv_` sono significativi solo per il processore `root`.

L'argomento `recv_count` è il numero di dati da ricevere da ogni processore, NON è il

numero totale dei dati da ricevere (il valore totale è calcolato nella routine stessa).

nb: in caso di non esatta divisibilità in `send_count`, si “aggiusta” da solo. (Sono più da conoscere che da imparare).

• Operazioni collettive

Le operazioni collettive sono eseguite da **tutti** i processori appartenenti ad un communicator.

Hanno diverse **caratteristiche**:

- tutti i processori che eseguono l'operazione collettiva eseguono almeno **una** comunicazione.
- L'operazione collettiva può richiedere una **sincronizzazione**.
- Tutte le operazioni collettive sono **bloccanti**.

Hanno diversi **scopi**:

- la **sincronizzazione** dei processori;
- esecuzione di **operazioni globali** (es. max di un vettore distribuito);
- gestione **ottimizzata degli input/output**, seguendo uno schema ad albero.

Per la **sincronizzazione dei processori** possiamo usare `MPI_Barrier`.

```
• MPI_Barrier(MPI_Comm comm);
```

Ogni processore dell'ambiente `comm` può procedere solo quando tutti gli altri avranno richiamato questa routine.

Una **sottocategoria** delle operazioni collettive sono le **operazioni di riduzione**.

In ciascuna operazione di riduzione *tutti* i processori di un communicator contribuiscono al risultato di un'operazione.

Si trattano di operazioni globali, implementati in maniera efficiente.

```
▪ MPI_Reduce(void *operand, void *result, int count,  
             MPI_Datatype datatype, MPI_Op op, int root,  
             MPI_Comm comm);
```

Tutti i processori di `comm` combinano i propri dati memorizzati in `operand` utilizzando l'operazione `op`.

Il risultato viene memorizzato in `result` di proprietà del processore con id. `root`.

`count`, `datatype`, `comm` devono essere uguali per ogni processore di `comm`.

L'argomento di `op` deve essere un valore predefinito, come: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`...

Calcolo della somma di N numeri

Possiamo individuare **4 fasi** fondamentali:

1. lettura e distribuzione dei dati;
2. calcolo del sottoproblema;
3. eventuali comunicazioni per il calcolo del risultato finale;
4. stampa del risultato.

Teniamo anche in considerazione che il N non sia un multiplo di nproc.

1. Lettura e distribuzione dei dati

La lettura viene effettuata (di solito) dal processore master, che poi si occuperà di distribuirlo a tutti gli altri processori.

La spedizione può avvenire “uno per volta” o “contemporaneamente”.

Vedere “Somma di N numeri – parte di Lettura e Distribuzione dei dati” -> LAB.

2. Calcolo locale

Tutti i processori eseguono le stesse 3 linee di codice:

```
sum = 0;
for(i=0; i<nloc; i++)
    sum = sum+xloc[i]
```

3. Comunicazioni per il calcolo del risultato finale

Le comunicazioni per il calcolo del risultato finale dipendono dalla strategia che vogliamo adottare: I, II, o III strategia (LEZ4, sezione MIMD DM).

I strategia

Il master aspetta di ricevere le somme parziali, mentre gli altri spediscono. Ogni somma parziale che riceve la aggiunge alla somma totale.

II strategia

È più complessa da implementare. Si deve identificare: quanti sono i passi di comunicazione, chi partecipa alla comunicazione e, fra questi, chi riceve e chi spedisce.

- **quanti sono i passi di comunicazione:** calcolati come $\log_2(n_{proc})$.
Es. se ho 4 processori, effettuo 2 passi.
- **chi partecipa alla comunicazione:** ad ogni passo i , i processori che potranno partecipare alla comunicazione saranno quelli con menum (id) che sia interamente divisibile per 2^i .

questo significa che al *primo passo*, tutti partecipano alla comunicazione ($2^0 = 1$, e qualsiasi numero è interamente divisibile per 1).

al *passo successivo*, abbiamo $2^1 = 2$, quindi tutti i processori interamente divisibili per 2 partecipano alla comunicazione.

e così via...

- o **chi riceve:** sono quelli con menum interamente divisibili per 2^{i+1} .
- o **chi spedisce:** sono tutti gli altri partecipanti.

es. 4 processori. Dovrà effettuare 2 passi.

Al primo passo ($i=0$) tutti partecipano alla comunicazione.

Solo 0 e 2 sono interamente divisibili (resto 0) per 2 ($2^{i+1} = 2$), dunque RICEVONO.

Gli altri (1 e 3) no, quindi SPEDISCONO.

Al secondo passo ($i=1$) partecipano alla comunicazione solo 0 e 2.

Solo 0 è interamente divisibile per 4 ($2^{i+1} = 4$), dunque RICEVE.

L'altro (2) SPEDISCE.

III Strategia

La III Strategia è molto simile alla seconda strategia, tuttavia ad ogni passo tutti lavorano, e soprattutto tutti spediscono e ricevono. Bisogna solo decidere le coppie (lo facciamo in base al resto).

nb: di queste strategie è importante ricordare il ragionamento, in quanto disponiamo di routine che già se ne occupano al posto nostro: le *operazioni collettive*.

Se volessimo usare le **operazioni collettive**:

Per la **II Strategia** possiamo usare MPI_Reduce:
(Solo un processore (P0) ha il risultato finale).

```
MPI_Reduce(&sum,&sumtot,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
```

Per la **III Strategia** possiamo usare MPI_Allreduce:
(Tutti i processori hanno il risultato finale).

```
MPI_Allreduce(&sum,&sumtot,1,MPI_INT,MPI_SUM, MPI_COMM_WORLD);
```

*notare che non ci sia 0 in MPI_Allreduce.

4. Stampa del risultato

La stampa del risultato cambia in base se:

- **vogliamo che la stampino tutti i processori:** bastare fare un'operazione di broadcast, nella quale inviamo la somma totale; ogni processore stamperà la somma totale.

```
MPI_Bcast(&sumtot,1,MPI_INT,0,MPI_COMM_WORLD);  
printf("\nSono il processo %d: Somma totale=%d\n", menum,sumtot);
```

- **vogliamo che la stampi un solo processore (P₀):** basta solo controllare se menum == 0.

```
if (menum==0)  
    printf("\nSomma totale=%d\n", sumtot);
```

LEZ 12 – Parametri di valutazione di un algoritmo: legge di Ware-Amdal (11 era lab)

Per stimare l'efficienza di un software parallelo dobbiamo osservare il tempo di esecuzione. In particolar modo è importante suddividere la *complessità di tempo* in 2 parti:

- la parte che deve essere eseguita in sequenziale T_s ;
- la parte che potrebbe essere eseguita in parallelo T_c .
Questa parte verrà suddivisa per i p processori, dunque il tempo per eseguire la parte parallela sarà T_c/p .

Ovviamente il nostro obiettivo è che T_s sia più piccolo possibile.

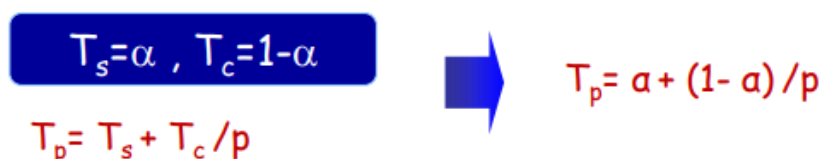
Nel caso di un ambiente MIMD-DM dovrei tener conto anche del tempo totale delle comunicazioni T_{com} , ma per semplicità possiamo supporre che sia integrato in T_c/p .

Legge di Ware-Amdal

La legge di Ware-Amdal è un modo alternativo di scrivere lo Speed Up, che evidenzia in particolar modo il numero di processori, la parte sequenziale, e la parte parallela.

$$S_p = \frac{1}{\alpha + (1 - \alpha)/p}$$

Per arrivare a questa formula, dobbiamo innanzitutto ricordare che il *tempo di esecuzione* T_p è dato dal tempo della parte in sequenziale T_s più il tempo della parte in parallelo T_c . Tuttavia consideriamo $T_s = \alpha$ e $T_c = 1 - \alpha$, cioè α rappresenta tutte le azioni svolte in sequenziale, dunque $1 - \alpha$ sono tutte le azioni che NON sono svolte in sequenziale (ovvero sono svolte in parallelo); dunque, possiamo riscrivere il *tempo di esecuzione* come:


$$T_p = T_s + T_c/p$$
$$T_p = \alpha + (1 - \alpha)/p$$

A questo punto dobbiamo ricordare formula dello Speed Up (T_1/T_p), dove T_1 e T_p sono il tempo di esecuzione di un algoritmo in un ambiente di calcolo dotato rispettivamente di 1 e di p processori/core. Dunque, se sostituisco T_p con la formula prima calcolata avremo:

$$S_p = \frac{T_1}{T_p} = \frac{1}{\alpha + (1 - \alpha)/p}$$

L'andamento dello speed-up, caratterizzato dalla legge di WA, all'aumentare del numero p delle CPU, ovvero per $p \rightarrow \infty$, lo $S_p = 1/\alpha$.

Praticamente, al crescere del numero p di processori, la parte sequenziale è **costante**. Tuttavia Speed Up ed efficienza degradano perché la parte parallela diminuisce.

Dunque, se la dimensione n del problema è **fissata**, al crescere del numero di processori, non solo non si riescono ad ottenere speed up vicini a quello ideale ma le prestazioni peggiorano (e quindi non conviene utilizzare un maggior numero di processori).

Tuttavia, se la dimensione n del problema **cresce** con il numero di processori, possiamo notare come anche la parte parallela sia **costante**.

Infatti, se p è fissato, al crescere della dimensione n del problema, la parte sequenziale $\alpha \rightarrow 0$. Dunque lo $S_p \rightarrow p$.

Al crescere del numero p di processori...

p	α	$(1-\alpha)/p$	S_p	E_p
2	0,032	0,968	1,9	0,95
4	0,032	0,903	3,4	0,85
8	0,032	0,775	5,1	0,6
16	0,032	0,516	6,2	0,3

Speed up ed efficienza degradano perché la parte parallela diminuisce!

la legge di Amdahl con $p=2$ e $n = 8, 16, 32, 64$

Al crescere della dimensione n ...

n	α	$1-\alpha$	$S_2(n)$	$E_2(n)$
8	0,14	0,86	1,75	0,875
16	0,06	0,94	1,8	0,9
32	0,03	0,97	1,9	0,96
64	0,01	0,99	1,96	0,99

Speed up ed efficienza sono "costanti"! E soprattutto l'efficienza tende all'efficienza IDEALE!

per p fissato, notiamo come al crescere di n , $\alpha \rightarrow 0$.

$$S_p = \frac{1}{\alpha + \frac{(1-\alpha)}{p}} \xrightarrow{\alpha \rightarrow 0} p$$

$$S_p \rightarrow p$$

$$(S^{\text{ideale}}(p) = p)$$

Questo significa che se voglio ottenere uno Speed Up quanto più vicino a quello ideale, dovrò fissare il *miglior numero di CPU* e *aumentare le dimensioni del problema*.

Ovviamente, non è possibile aumentare in maniera indefinita la dimensione n del problema in quanto le risorse (hardware) sono limitate.

Aumentando sia n che p , con rapporto fissato, le *prestazioni* dell'algoritmo parallelo non degradano molto velocemente (l'efficienza è "quasi costante"). Ovviamente, il nostro obiettivo sarà trovare una $l=n/p$ tale che l'efficienza resti costante.

In breve, dalla legge di WA possiamo dire che:

1. Fissato il size n del problema e aumentando il numero p di CPU, esiste p_{\max} migliore numero di processori per risolvere il problema con l'algoritmo in esame. Superato questo valore, le prestazioni peggiorano;
2. Fissato il numero p delle CPU e aumentando il size n del problema, esiste n_{\max} massima dimensione che la macchina può memorizzare/elaborare. Superato questo valore, potrei avere problemi con la memoria hardware.

LEZ 13 – Parametri di valutazione di un algoritmo: isoefficienza

Abbiamo visto che il nostro obiettivo è individuare un giusto valore $I = n/p$ tale che $E_p(n)$ sia costante.

Non è possibile trovare “I” ad occhio, soprattutto considerando che deve essere valido per ogni algoritmo. Dunque I non può essere una costante, ma deve essere una **funzione**.

La funzione I è detta **Isoefficienza**, e prende tre valori: n_0 , p_0 , p_1 . Dove n_0 e p_0 sono la coppia iniziale di size e numero di processori, e p_1 sarà il nuovo numero di processori. Lo scopo di questa funzione è individuare il nuovo size n_1 tale che l'**efficienza resti costante**.

$$I = I(n_0, p_0, p_1)$$

Per fare ciò, dobbiamo imporre che l'efficienza dell'algoritmo con size n_0 e p_0 processori sia la stessa che avrei se avessi size n_1 e p_1 processori:

$$E_{p_0}(n_0) = E_{p_1}(n_1)$$

Ricordiamo che l'efficienza non è altro che il rapporto fra lo speed up e il numero di processori. Dunque possiamo riscrivere l'imposizione di prima come:

$$E_p = S_p/p \qquad \frac{S_{p_0}(n_0)}{p_0} = \frac{S_{p_1}(n_1)}{p_1}$$

Tuttavia, possiamo anche riscrivere lo Speed Up in funzione dell'Overhead, avendo:

$$S_p = \frac{p}{\frac{O_h}{T_1} + 1} \qquad \frac{p_0}{p_0(\frac{O_h}{T_1(n_0)} + 1)} = \frac{p_1}{p_1(\frac{O_h}{T_1(n_1)} + 1)}$$

Posso semplificare p_0 e p_1 , e quindi avrò:

$$\frac{1}{\frac{O_h}{T_1(n_0)} + 1} = \frac{1}{\frac{O_h}{T_1(n_1)} + 1} \quad \Rightarrow \quad \frac{O_h(n_0, p_0)}{T_1(n_0)} = \frac{O_h(n_1, p_1)}{T_1(n_1)}$$

Dunque, l'efficienza di un algoritmo con size n_1 sarà:

$$T_1(n_1) = \frac{O_h(n_1, p_1)}{O_h(n_0, p_0)} T_1(n_0)$$

I

Un algoritmo si dice **scalabile** se l'efficienza rimane costante al crescere del numero dei processori e della dimensione del problema, in un rapporto costante pari a:

$$I = \frac{O_h(n_1, p_1)}{O_h(n_0, p_0)}$$

Esempio: isoefficienza della somma, II strategia

Ricordiamo che: $O_h(n, p) = p \log_2 p$; $T_1(n) = n-1$.

Se applichiamo la formula dell'isoefficienza otterremo: $n_1 - 1 = \frac{p_1 \log_2 p_1}{p_0 \log_2 p_0} (n_0 - 1)$

che posso riscrivere come: $\frac{n_1}{n_0} = \frac{p_1 \log_2 p_1}{p_0 \log_2 p_0} \Rightarrow n_1 = \frac{p_1 \log_2 p_1}{p_0 \log_2 p_0} \cdot n_0$

Ci chiediamo se tale algoritmo è **scalabile**. Per fare ciò, mi chiedo se all'aumentare di p e di n l'efficienza rimane costante.

Dunque pongo arbitrariamente le 3 variabili iniziali, ad es. $(n_0, p_0, p_1) = (64, 4, 8)$, ed individuo n_1 .

$$n_1 = ((8 \log_2 8) / (4 \log_2 4)) * 64 = ((8*3) / (4*2)) * 64 = 3*64 = \mathbf{192}.$$

Se provo a rifare lo stesso procedimento anche per n_2, n_3 , ecc. e poniamoli in una tabella:

Efficienza				
$\begin{matrix} n \backslash p \\ \end{matrix}$	1	4	8	16
64	1.0	0.91	0.57	0.33
192	1.0	0.97	0.91	0.79
512	1.0	0.97	0.96	0.91

Osserveremo che l'efficienza, secondo i T_p calcolati, rimane costante al crescere di p e di n con la legge di isoefficienza. Dunque **la somma è scalabile**.

Ricordo che T_p è calcolabile come:

$$T_p(n) = \left(\frac{n}{p} - 1 + \log_2 p \right) t_{calc}$$

I valori ottenuti prima sono i seguenti:

$$p_0 = 4 \quad n_0 = 64 \quad T_4(64) = 17$$

$$p_1 = 8 \quad n_1 = 192 \quad T_8(192) = 26$$

$$p_2 = 16 \quad n_2 = 512 \quad T_{16}(512) = 35$$

Calcolo dello Speed Up, Overhead ed Efficienza dell'algoritmo parallelo della somma di n numeri (MIMD DM)

*i seguenti sono ragionamenti, e non formule da imparare a memoria.

La strategia

Ogni processore calcola la propria somma parziale

Ad ogni passo, ciascun processore invia la sua somma parziale ad un unico processore prestabilito. Tale processore contiene la somma totale.

Le *somme parziali* saranno $(n/p) - 1$, dunque il tempo di calcolo sarà: $(n/p - 1)t_{\text{calc}}$.

Verranno effettuate inoltre (nel secondo passo) $p-1$ spedizioni e $p-1$ somme, dunque il tempo di calcolo sarà: $(p-1)t_{\text{calc}} + (p-1)t_{\text{com}}$.

Ricordiamo che il tempo di una spedizione è 2,3 volte più lenta del tempo di un'addizione, dunque possiamo affermare che $t_{\text{com}} = c \cdot t_{\text{calc}}$, dove $2 \leq c \leq 3$.

Ciò è utile, perché possiamo riscrivere $T_p(n)$ tenendo conto solo di t_{calc} . Infatti:

$$T_p(N) = (N/p - 1)t_{\text{calc}} + (p-1)t_{\text{calc}} + c(p-1)t_{\text{calc}} \\ = [(N/p - 1) + (p-1) + c(p-1)]t_{\text{calc}}.$$

Adesso che ho la formula generale di $T_p(n)$ (**nb**: per la 1 strategia della somma), posso calcolare *speed up*, *Overhead* ed *Efficienza*.

Ricordiamo che il tempo di esecuzione per la *somma di n numeri* in sequenziale è:

$$T_1(N) = N - 1 \cdot t_{\text{calc}} :$$

$$S_p = T_1(N)/T_p(N) = \\ = [N-1] / [(N/p - 1) + (p - 1) + c (p - 1)] < p$$

$$Oh = p \cdot T_p(N) - T_1(N) = \\ p[(N/p - 1) \cdot t_{\text{calc}} + (p - 1) \cdot t_{\text{calc}} + c (p - 1) \cdot t_{\text{calc}}] - (N-1) \cdot t_{\text{calc}}$$

$$E_p = S_p/p = = [N-1] / p [(N/p - 1) + (p - 1) + c (p - 1)]$$

Se n non è esattamente divisibile per p , alcuni processori faranno una somma in più nella fase puramente parallela (**Vale per tutte e 3 le strategie in MIMD DM**).

Ad es. Se ho $n=67$ e $p=8$, mi avanzano 3 numeri che dovrò ridistribuire ai processori. Di norma si danno ai processori che hanno un id minore del resto, dunque p_0 , p_1 e p_2 .

Per calcolare T_p , devo prendere la parte intera delle divisioni:

$$[(\text{intera}(N/p - 1)) + (p-1) + c(p-1)]t_{\text{calc}}.$$

Però la parte di collezione dei risultati non può iniziare finché tutte le somme parziali non sono state completate, dunque è come se ogni processore effettuasse una somma in più! Appunto, i tempi sono corrotti da quei numeri in più.

Per questo si cerca di avere un size che sia divisibile per il numero di processori.

II strategia

Ogni processore calcola la propria somma parziale

Ad ogni passo, coppie distinte di processori comunicano contemporaneamente; in ogni coppia, un processore invia all'altro la propria somma parziale che provvede all'aggiornamento della somma.

Il risultato è in un unico processore prestabilito.

I passi sono $\log_2(NP)$, considerando che NP sia potenza di 2. In questa strategia (anche) la somma parziale avviene in parallelo, ad eccezione dell'ultimo passo che deve necessariamente avvenire sequenzialmente.

Le *somme parziali* saranno $(n/p) - 1$, dunque il tempo di calcolo sarà: $(n/p - 1)t_{\text{calc}}$.

Verranno effettuate inoltre $\log(p)$ spedizioni e $\log(p)$ somme, dunque il tempo di calcolo sarà: $(\log(p))t_{\text{calc}} + (\log(p))t_{\text{com}}$.

Ricordando il discorso di c , e raccogliendo t_{calc} , avremo:

$$T_p(N) = [(N/p - 1) + \log(p) + c \log(p)]t_{\text{calc}}.$$

Ricordiamo che il tempo di esecuzione per la *somma di n numeri* in sequenziale è:

$$T_1(N) = N - 1 \, t_{\text{calc}} :$$

$$S_p = T_1(N)/T_p(N) =$$

$$= [N - 1] / [(N/p - 1) + \log(p) + c \log(p)] < p$$

$$O_h = p \, T_p(N) - T_1(N) =$$

$$p[(N/p - 1) + \log(p) + c \log(p)] \, t_{\text{calc}} - (N - 1) \, t_{\text{calc}}$$

$$E_p = S_p/p = [N - 1] / p [(N/p - 1) + \log(p) + c \log(p)]$$

III strategia

Ad ogni passo, coppie distinte di processori comunicano contemporaneamente; in ogni coppia, i processori si scambiano le proprie somme parziali.

Il risultato è in tutti i processori.

In tal modo tutti i processori sono sempre attivi. Il trasferimento dei dati è sempre da considerarsi per uno, perché avvengono nello stesso passo. Inoltre non avrò bisogno di fare un broadcast (eventualmente) in quanto le somme sono già presenti nelle memorie di tutti i processori.

Sfrutto al meglio i processori e risparmio eventuali spedizioni.

Le *somme parziali* saranno $(n/p) - 1$, dunque il tempo di calcolo sarà: $(n/p - 1)t_{\text{calc}}$.

Verranno effettuate inoltre p *spedizioni* e p somme **contemporaneamente**, dunque è come se fosse un'unica spedizione ed un'unica somma.

Ricordando il discorso di c , e raccogliendo t_{calc} , avremo:

$$T_p(N) = [(N/p - 1) + \log(p) + c \log(p)]t_{\text{calc}}.$$

Ricordiamo che il tempo di esecuzione per la *somma di n numeri* in sequenziale è:

$$T_1(N) = N - 1 \cdot t_{\text{calc}} :$$

$$S_p = T_1(N)/T_p(N) = \\ = [N-1] / [(N/p - 1) + \log(p) + c \log(p)] < p$$

$$O_h = p T_p(N) - T_1(N) = \\ p[(N/p - 1) + \log(p) + c \log(p)] t_{\text{calc}} - (N-1) t_{\text{calc}}$$

$$E_p = S_p/p = [N-1] / p [(N/p - 1) + \log(p) + c \log(p)]$$

Calcolo dello Speed Up, Overhead ed Efficienza dell'algoritmo parallelo della somma di n numeri (MIMD SM)

*i seguenti sono ragionamenti, e non formule da imparare a memoria.

La strategia

Ad ogni passo, ciascun core deve addizionare la propria somma parziale ad una variabile che conterrà la somma finale (es. sumtot).

I core devono accedere a sumtot uno alla volta. Le somme finali avvengono in modo sequenziale in quanto ho un solo core attivo per volta.

Le *somme parziali* saranno $(n/p) - 1$, dunque il tempo di calcolo sarà: $(n/p - 1)t_{\text{calc}}$.

Verranno effettuate inoltre $p-1$ somme per la *somma totale*, dunque il tempo di calcolo sarà: $(p-1)t_{\text{calc}}$.

Essendo la memoria condivisa, non vengono effettuate spedizioni.

Raccogliendo t_{calc} , avremo:

$$T_p(N) = [(N/p - 1) + (p-1)]t_{\text{calc}}$$

Ricordiamo che il tempo di esecuzione per la *somma di n numeri* in sequenziale è:

$$T_1(N) = N-1 t_{\text{calc}}$$

$$S_p = T_1(N)/T_p(N) =$$

$$= [N-1] / [(N/p - 1) + (p-1)] < p$$

$$Oh = p T_p(N) - T_1(N) = p[(N/p - 1) + (p-1)] t_{\text{calc}} - (N-1) t_{\text{calc}}$$

$$E_p = S_p / p = [N-1] / p [(N/p - 1) + (p-1)]$$

Anche in questo caso, se n non è esattamente divisibile per p , alcuni core effettueranno 1 somma in più nella fase puramente parallela.

Per calcolare T_p , devo prendere la parte intera delle divisioni:

$$[(\text{intera}(N/p - 1)) + (p-1)]t_{\text{calc}}$$

II strategia

Ad ogni passo, la metà dei core (rispetto al passo precedente) calcola un contributo alla somma parziale.

La somma finale (sumtot) sarà data dalla somma degli ultimi due contributi (obv. se NP è potenza di 2).

Le *somme parziali* saranno $(n/p) - 1$, dunque il tempo di calcolo sarà: $(n/p - 1)t_{\text{calc}}$.

Verranno effettuate inoltre $\log(p)$ somme per la *somma totale*, dunque il tempo di calcolo sarà: $(\log(p))t_{\text{calc}}$.

Essendo la memoria condivisa, non vengono effettuate spedizioni.

Raccogliendo t_{calc} , avremo:

$$T_p(N) = [(N/p - 1) + \log(p)]t_{\text{calc}}.$$

Ricordiamo che il tempo di esecuzione per la *somma di n numeri* in sequenziale è:

$$T_1(N) = N - 1 \uparrow_{\text{calc}} :$$

$$S_p = T_1(N)/T_p(N) =$$

$$= [N-1] / [(N/p - 1) + \log(p)] < p$$

$$O_h = p T_p(N) - T_1(N) = p[(N/p - 1) + \log(p)] \uparrow_{\text{calc}} - (N-1) \uparrow_{\text{calc}}$$

$$E_p = S_p / p = [N-1] / p [(N/p - 1) + \log(p)]$$

Anche in questo caso, se n non è esattamente divisibile per p , alcuni core effettueranno 1 somma in più nella fase puramente parallela.

Per calcolare T_p , devo prendere la parte intera delle divisioni:

$$[(\text{intera}(N/p - 1)) + (p-1)] t_{\text{calc}}.$$

Non esiste una terza strategia, in quanto il risultato è sempre nell'unica memoria condivisa.

(nb: è un po' un arricchimento della lezione 4).

LEZ 15 – Applicazione di Ware Amdal e isoefficienza (scalabilità) (14 lab)

Formula di WARE AMDAHL GENERALIZZATA

La formula di Ware Amdahl Generalizzata (WAG) viene in nostro soccorso per quegli algoritmi che non hanno una netta separazione fra la parte sequenziale e la parte parallela.

Ricordiamo la formula **base di WA**:

$$\text{BASE} \quad S_p = \frac{1}{\alpha + (1 - \alpha)/p}$$

- α frazione sequenziale
- $(1 - \alpha)$ frazione parallelizzabile
- $(1 - \alpha)/p$ frazione parallela

La formula di **Ware Amdal Generalizzata** sarà:

$$S_p = \frac{1}{\alpha_1 + \sum_{k=2}^{p-1} \frac{\alpha_k}{k} + \frac{\alpha_p}{p}}$$

dove:

- α_1 rappresenta le operazioni eseguite in sequenziale;
- α_k rappresenta le operazioni eseguite con *parallelismo medio*, cioè quelle operazioni eseguite parallelamente su k processori (con $k < p$), ovvero solo su una *porzione* di questi;
- α_p rappresenta le operazioni eseguite con *parallelismo totale*, cioè su tutti i p .

α_k parte da 2 e termina a $p-1$ perché α_1 sarebbe la parte sequenziale e α_p sarebbe la parte completamente parallela.

Esempio, 1 strategia

Supponiamo di avere: $p=8$, $N=32$, $nloc=4$ (n/p , ie quanti numeri deve sommare ogni proc).

La prima domanda che devo pormi è: “questa strategia prevede una netta distinzione fra la parte sequenziale e parallela?”. Se sì, non ho bisogno della WAG ma posso usare la più semplice WA.

Infatti, in questo caso ho una netta distinzione e dunque possiamo usare direttamente la **WA**.

In sequenziale, $T(1) = N-1 = 32-1 = 31$ somme.

Individuiamo le fasi

fase 1

Si tratta di una fase tutta *parallela*, in quanto ogni unità processante effettuerà le proprie somme. In particolare, ogni p farà $nloc-1$ somme, cioè $4-1=3$ somme.

Siccome ogni processore effettua 3 somme, ed abbiamo 8 processori, nella prima fase effettuiamo $3 \times 8 = 24$ somme delle 31 totali.

Dunque, la **parte parallela** $(1-a) = 24/31$.

Possiamo già calcolare $(1-a)/p = 3/31$.

fase 2

Si tratta di una fase tutta *sequenziale*, in quanto il processore Master si occupa di effettuare la somma totale.

Essendo i processori 8, significa che avremo 8 somme parziali. Il processore Master dovrà effettuare la somma totale a partire dalle somme parziali.

In particolare, dovrà effettuare $p-1$ somme (7 somme).

Dunque, la **parte sequenziale** $a = 7/31$.

È **importante** che dalla somma della parte sequenziale e della parte parallela otteniamo 1. Infatti, $24/31 + 7/31 = 31/31 = 1$. Dunque **abbiamo fatto bene i conti**.

Adesso ci rimane da calcolare lo **Speed Up con WA**:

The diagram illustrates the calculation of Speed Up (S_p) using the Work-Aware (WA) model. It starts with the base formula for S_p in a yellow box, then shows the calculation of the sequential part α in an orange box, and finally the final result in a red box, with a blue arrow indicating the flow of the calculation.

$$\text{BASE} \quad S_p = \frac{1}{\alpha + (1-\alpha)/p}$$
$$\alpha = \frac{7}{31}$$
$$\frac{1-\alpha}{p} = \frac{24}{31} \cdot \frac{1}{8} = 3 \cdot \frac{1}{31}$$
$$S_p = \frac{1}{\frac{7}{31} + \frac{3}{31}} = \frac{31}{10} = 3,1$$

NB: in effetti 7 somme sequenziali sono un po' tante.

Esempio, 2/3 strategia

nb: i risultati per la 2 e 3 strategia sono gli stessi nel calcolo di WA perché le somme uguali non sono considerate (infatti la 3 strategia fa più somme, ma servono solo per avere lo stesso risultato in tutti i processori e non fare il broadcast dopo, e non porta alcun vantaggio in termini di velocità dell'algoritmo).

Supponiamo di avere: $p=8$, $N=32$, $nloc=4$.

La prima domanda che devo pormi è: "questa strategia prevede una netta distinzione fra la parte sequenziale e parallela?". Se sì, non ho bisogno della WAG ma posso usare la più semplice WA.

In questo caso, non è possibile separare esattamente la parte sequenziale da quella parallela in quanto ci saranno delle somme parziali nelle fasi successive che saranno effettuate solo da alcuni processori.

Per questo motivo, dobbiamo usare la **WAG**.

In sequenziale, $T(1) = N-1 = 32-1 = 31$ somme.

Individuiamo le fasi, in questo caso vogliamo trovare QUANTI PROCESSORI lavorano per OGNI FASE.

1 fase (è la stessa della 1 strategia)

Si tratta di una fase tutta *parallela*, in quanto ogni unità processante effettuerà le proprie somme. In particolare, ogni p farà $nloc-1$ somme, cioè $4-1 = 3$ somme.

Siccome ogni processore effettua 3 somme, ed abbiamo 8 processori, nella prima fase effettuiamo $3 \times 8 = 24$ somme delle 31 totali.

Dunque, la **parte parallela** $a_8 = 24/31$.

Possiamo già calcolare $a_p/p = a_8/8 = 3/31$.

nb: con la legge di WAG la parte parallela non viene più indicata come $(1-a)$ ma come a_p . Infatti, non indica *tutte* le operazioni parallele ma solo quelle eseguite con *parallelismo totale*.

nb: non c'è nessuna fase in cui lavorano contemporaneamente 7, 6, o 5 processor/core. Quindi possiamo porre i loro valori a 0. Infatti avremo:

$$\frac{\alpha_7}{7} = \frac{\alpha_6}{6} = \frac{\alpha_5}{5} = 0$$

2 fase

Si tratta di una fase che avviene *parzialmente in parallelo*. Infatti come ben ricordiamo, al passo successivo solo metà dei processori del passo precedente lavorano alle somme parziali, e così fino alla fine.

In questo caso, se prima hanno lavorato 8 processori contemporaneamente, ora ne avremo 4. Ogni processore si occuperà di effettuare 1 somma.

Siccome ogni processore effettua 1 somma, ed abbiamo 4 processori, nella seconda fase effettuiamo $1 \times 4 = 4$ somme delle 31 totali.

Dunque, la **parte parallela** $a_4 = 4/31$.

Quindi $a_4/4 = 1/31$.

nb: non c'è nessuna fase in cui lavorano contemporaneamente 3 processori/core. Quindi possiamo porre il suo valore a 0. Infatti avremo:

$$\frac{a_3}{3} = 0$$

3 fase

Anche questa è una fase che avviene *parzialmente in parallelo*. Prima avevamo 4 processori che lavoravano contemporaneamente, dunque ora ne avremo 2.

Siccome ogni processore effettua 1 somma, ed abbiamo 2 processori, nella terza fase effettuiamo $1 \times 2 = 2$ somme delle 31 totali.

Dunque, la **parte parallela** $a_2 = 2/31$.

Quindi $a_2/2 = 1/31$.

4 fase (ultima)

Si tratta di una fase tutta *sequenziale*, in quanto il processore Master si occupa di effettuare la somma totale.

Nello specifico, il processore Master si occuperà di effettuare l'ultima somma.

Dunque, la **parte sequenziale** $a_1 = 1/31$.

È **importante** che dalla somma della parte sequenziale e delle parti parallele otteniamo 1. Dunque: $a_8 + a_7 + a_6 + a_5 + a_4 + a_3 + a_2 + a_1$ deve essere = 1.


Sapendo che $a_7, a_6, a_5, a_3 = 0$, possiamo tener conto solo di a_8, a_4, a_2, a_1 , dunque: $24/31 + 4/31 + 2/31 + 1/31 = 31/31 = 1$. Dunque **abbiamo fatto bene i conti**.

Adesso ci rimane da calcolare lo **Speed Up con WAG**:

$$S_p = \frac{1}{a_1 + \sum_{k=2}^{p-1} \frac{a_k}{k} + \frac{a_p}{p}}$$

$\frac{a_8}{8} = \frac{3}{31}$
 $\frac{a_7}{7} = \frac{a_6}{6} = \frac{a_5}{5} = 0$
 $\frac{a_4}{4} = \frac{1}{31}$

$\frac{a_3}{3} = 0$
 $\frac{a_2}{2} = \frac{1}{31}$
 $a_1 = \frac{1}{31}$



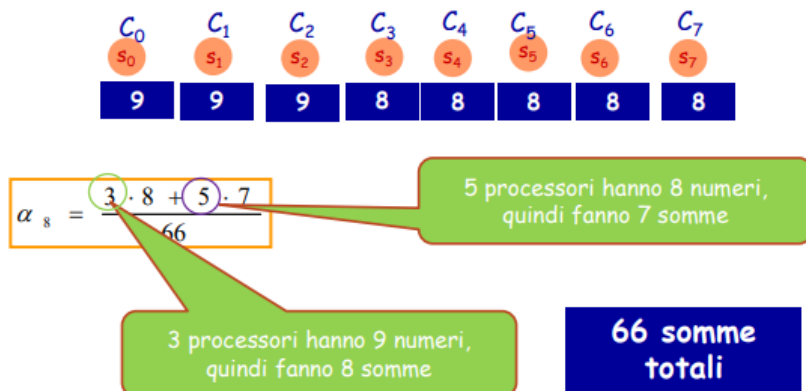
$S_p = \frac{1}{\frac{1}{31} + \frac{1}{31} + \frac{1}{31} + \frac{1}{31}} = \frac{31}{4} = 7.75$

*migliore rispetto alla 1 strategia.

N non esattamente divisibile per p

In questo caso, bisogna stare attenti solo alla parte con *parallelismo totale* (a_p).
I numeri restanti ottenuti dalla divisione n/p saranno poi ridistribuiti a quei processori il quale $id < resto$.

Esempio:



Calcolo dell'ISOEFFICIENZA

Ricordiamo:

La funzione I è detta **Isoefficienza**, e prende tre valori: n_0 , p_0 , p_1 . Dove n_0 e p_0 sono la coppia iniziale di size e numero di processori, e p_1 sarà il nuovo numero di processori. Lo scopo di questa funzione è individuare il nuovo size n_1 tale che l'**efficienza resti costante**.

Viene usata per valutare la **scalabilità** di un algoritmo.

$$I(p_0, p_1, n_0) = \frac{O_h(n_1, p_1)}{O_h(n_0, p_0)}$$

nb: slide da 31 a 47 sembrano essere altri esempi.

LEZ 16 – Introduzione ad OpenMP

Ricordiamo che: i calcolatori MIMD SM hanno necessità di sincronizzare gli accessi in scrittura e organizzare il lavoro;

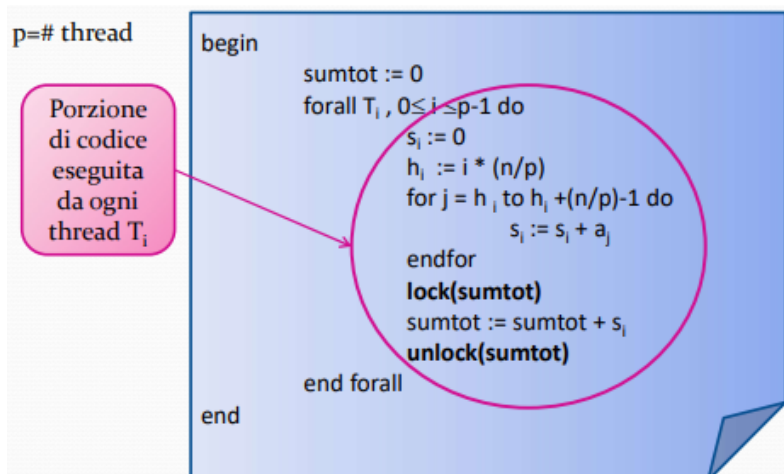
Un **thread** è un *flusso di istruzioni indipendente* che deve essere eseguito *sequenzialmente* su una CPU.

Un **processo** è un programma in esecuzione.

Un processo è costituito da almeno un thread (può contenerne più di uno).

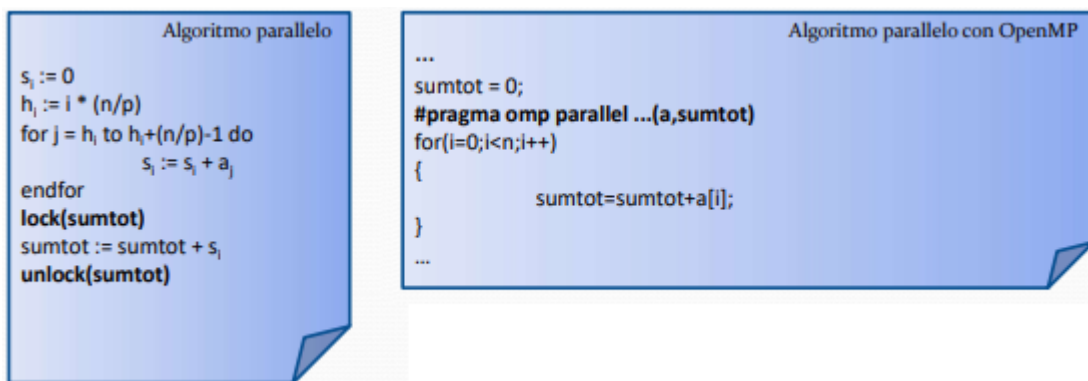
Thread diversi possono essere eseguiti indipendentemente su core diversi.

Normalmente, a basso livello, l'algoritmo della somma di N numeri avrebbe questa forma:



Dobbiamo preoccuparci di diversi aspetti, che invece potremmo gestire molto più facilmente utilizzando **OpenMP**.

Infatti, l'algoritmo di prima diventerà:



OpenMP (Open specifications for Multi Processing) è una libreria che prevede delle API per gestire il parallelismo shared-memory multi-threaded. Consente un approccio ad alto livello, user friendly. Inoltre, è portabile (il codice funziona su qualsiasi SO).

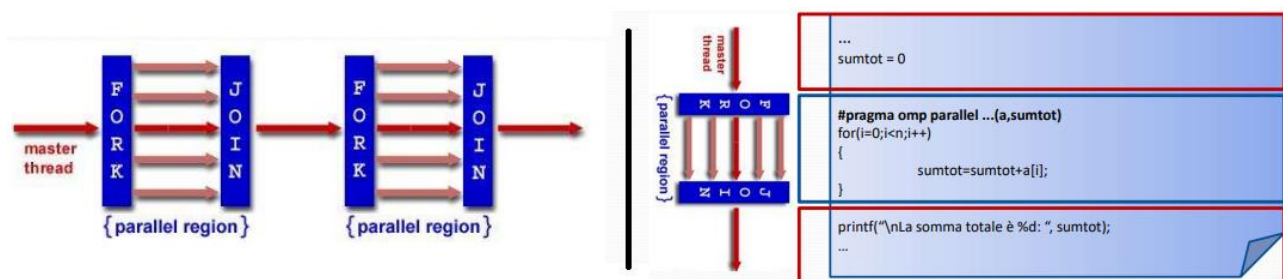
La libreria **OpenMP** ci permette di gestire più **semplicemente** diversi aspetti:

- l'utilizzo di variabili private di appoggio;
- la divisione del lavoro tra i thread;
- la collezione del risultato in una variabile shared in modo sincronizzato.

Il modello d'esecuzione parallela è quello **fork-join**.

Tutti i processi cominciano con un solo thread (*master thread*), ovvero iniziano in maniera sequenziale. Dal processo unico posso dividere il lavoro in più processi (effettuo una *fork*): da qui comincia una *regione parallela* nella quale un team di thread procede parallelamente.

Quando tutti i thread del team hanno terminato le istruzioni della regione parallela, si sincronizzano e terminano, lasciando proseguire solo il master thread.



Non ci sono regole automatiche per la gestione della memoria, dunque devo stare particolarmente attento a quali variabili possono essere shared e quali private. Ad esempio, dichiarare una variabile shared quando non ce n'è bisogno potrebbe non essere una buona idea in quanto rischio che qualche altro thread la sovrascriva.

La libreria OpenMP è fondamentalmente **composta da**:

- **direttive per il compilatore**, completate eventualmente da **clausole** che ne dettagliano il comportamento;
- **Runtime Library Routines**, per intervenire sulle *variabili di controllo interne* a runtime (deve essere incluso il file `omp.h`), es. il numero di thread, informazioni sullo scheduling;
- **variabili d'ambiente**, per modificare il valore delle *variabili di controllo interne* prima dell'esecuzione.

- Direttive

Le **direttive** si usano per creare un team e stabilire quali istruzioni devono essere eseguite in parallelo e come queste devono essere distribuite tra i thread.

Tali direttive saranno inserite nelle zone concorrenti, con le relative **clausole**.

Le direttive sono identificate da “`#pragma omp ...`”.

Per formare il team di thread e avviare l'esecuzione parallela si deve usare il costrutto **parallel**.

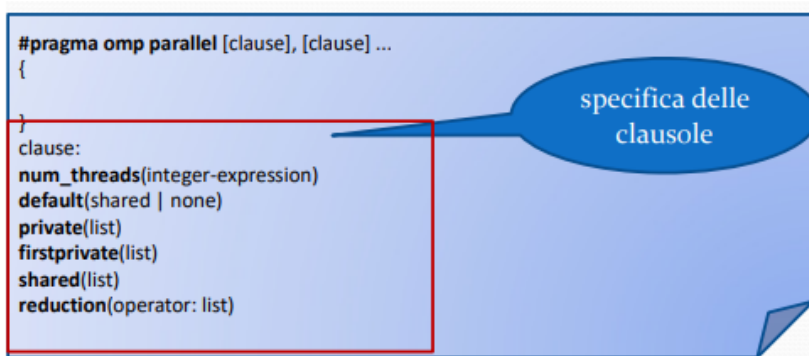
```
#pragma omp parallel [clause], [clause]..  
{  
    structured_block  
}
```

Alla fine del blocco di istruzioni è sottintesa una barriera di sincronizzazione: tutti i thread si fermano ad aspettare che gli altri abbiano completato l'esecuzione, prima di ritornare ad un'esecuzione sequenziale.

Al seguito del blocco, specifichiamo eventualmente le *clausole* che ne dettagliano il comportamento (l'ordine non è importante).

Non tutte le clausole sono valide per tutti i costrutti.

Quasi tutte accettano una lista di argomenti separati.



es.

- **default:** mi permette di impostare automaticamente tutte le variabili a shared (shared), oppure specifico che dovrò essere io ad impostare manualmente se una variabile è shared o private (none).
Di default, la clausola *default* è none.
- **private:** gli argomenti contenuti in *list* sono privati per ogni thread che li utilizza. Ciò significa che ogni thread avrà la propria copia delle variabili private, che saranno tutte inizializzate a 0;
- **shared:** gli argomenti contenuti in *list* sono condivisi tra i thread del team;
- **firstprivate:** come *private*, ma le variabili mantengono il valore che avevano prima di entrare nella regione parallela (solo per *parallel*).
- **lastprivate:** come *private*, ma il valore delle variabili verrà aggiornato una volta usciti dalla regione parallela (solo per *for*).
- **reduction:** gli argomenti contenuti in *list* verranno combinati utilizzando l'operativo associativo specificato.*
- **num:threads:** imposta il numero dei thread che lavoreranno della direttiva.

*Nello specifico, **reduction** si occupa di creare una copia privata della *variabile* (var) per ogni thread. Terminata la regione parallela, la variabile var è ridotta ad una operazione atomica. In altre parole, la variabile che viene esposta è un aggregato di tutte le *var* contenute in ogni thread.

È utile perché evita il verificarsi di race condition sulla variabile (nb: non è necessario inserire la variabile in *shared*, in quanto è già implicitamente considerata shared).

```
thread 1 - private var = compute(x)
thread 2 - private var = compute(y)
thread 3 - private var = compute(z)

//reduction step
public var = var<thread1> + var<thread2> + var<thread3> ...
```

• Costrutti Work Sharing

Oltre al parallel, distinguiamo altri **tre tipi di costrutti** detti **WorkSharing** perché si occupano della distribuzione del lavoro al team di thread: **for**, **sections**, **single**.

Anche all'uscita da un costrutto WorkSharing è sottintesa una barriera di sincronizzazione, se non diversamente specificato dal programmatore.

•• Costrutto FOR

Il costrutto **for** ci permette di parallelizzare un ciclo iterativo, distribuendo le iterazioni tra i thread del team (attraverso un ordine stabilito da OpenMP).

#pragma omp for

Questo costrutto comprende due **clausole** specifiche:

- **nowait**: elimina la barriera implicita alla fine del costrutto. I thread non aspettano che tutti abbiano finito, ma continuano a lavorare.
- **schedule(kind, chunk_size)**: specifica il *modo* (**kind**) con cui distribuire le iterazioni del ciclo; **chunk_size** è il numero di iterazioni contigue da assegnare allo stesso thread (nb: $\text{chunk_size} > 0$).

In particolar modo, *kind* può essere:

- o **static**: chunk assegnati secondo uno scheduling round-robin (es. ho 10 iterazioni e 2 thread? 5 iterazioni per uno, 5 per l'altro);
- o **dynamic**: chunk assegnati su richiesta. Un thread libero richiede l'assegnazione di un'iterazione, quando questo termina gliene viene assegnata un'altra.
- o **guided**: simile a dynamic, tranne per il fatto che la dimensione del chunk cambia durante l'esecuzione del programma. Inizia con chunk di grandi dimensioni, ma se nota che il carico di lavoro è sbilanciato lo riduce.

In alternativa a *guided*, è possibile migliorare le prestazioni utilizzando l'opzione **chunk** con la modalità dynamic. In questo modo, ad ogni thread sarà associato un numero prestabilito di iterazioni, e quando avrà finito gli verrà assegnato un nuovo chunk. Aumentando le dimensioni del chunk, lo scheduling tende alla modalità static; viceversa, tende alla modalità dynamic.

Se non specificato nulla, si adotta quello *static*. Tuttavia, potrebbe non essere ottimale per il tipo di problema che deve essere risolto. Ad esempio, quando differenti iterazioni impiegano un diverso tempo d'esecuzione. In tal caso potrebbe essere conveniente usare la modalità dynamic.

La modalità *dynamic* è migliore quando le iterazioni possono richiedere tempi molto diversi. Tuttavia questa operazione comporta overhead in quanto i thread devono perdere tempo per guardarsi intorno, dunque non sempre si ha un miglioramento delle performance.

Dunque, per quanto riguarda la **modalità da scegliere**, questa dipende dal problema e dalla strategia parallela che vogliamo implementare:

- per i cicli *for* in cui ogni iterazione richiede all'incirca lo stesso tempo, la modalità **STATIC** funziona meglio poiché presenta un overhead minimo;
- per i cicli *for* in cui le iterazioni possono richiedere quantità di tempo diverse, la modalità **DYNAMIC** funziona meglio perché il lavoro verrà suddiviso in modo più uniforme tra i thread, ma ci può essere un ritardo dovuto alla riassegnazione del lavoro;
- se si vuole avere un compromesso fra i due, si può specificare la dimensione del **CHUNK** oppure si può utilizzare la modalità **GUIDED**.

...ulteriori osservazioni sul for

Se ho un numero di thread non perfettamente divisibile per il numero di iterazioni, dei thread si prendono carico di quelle iterazioni aggiuntive.

Inoltre, adesso capiamo perché *lastprivate* si può usare solo per il *for*: potrebbero esserci dei calcoli su alcune variabile di tipo *private* che potrei volere anche uscito dalla direttiva.

Il costrutto *for* può essere combinato con quello *parallel*: praticamente tutti devono fare la stessa cosa MA distribuendosi le iterazioni.

#pragma omp parallel for

Per evitare problemi di sincronizzazione, possiamo usare il costrutto *critical*.

Il costrutto **critical** forza l'esecuzione del blocco successivo ad un thread alla volta (è come se lo facesse in sequenziale): è utile per gestire le *regioni critiche*.

#pragma omp critical

•• Costrutto SECTIONS

Il costrutto **sections** mi da la possibilità di creare una serie di sezioni parallele, dove assegno a più core una specifica sezione. Si tratta di un parallelismo asincrono.

Notar bene che per questo costrutto:

- le diverse sezioni devono poter essere eseguite in ordine arbitrario;
- bisogna stare attenti a bilanciare il carico di lavoro.

La sintassi è del tipo:

```
#pragma omp sections [clause], [clause] ... new-line
{
  [#pragma omp section new-line]

  [#pragma omp section new-line]
  ...
}
clause: private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait
```

•• Costrutto SINGLE

Il costrutto **single** attiva un thread qualsiasi del team per fargli eseguire un blocco di istruzioni in sequenziale, MA gli altri thread devono aspettare.

```
#pragma omp single [clause], [clause] ... new-line
{
}
clause: private(list)
firstprivate(list)
copyprivate(list)
nowait
```

Tutti i costrutti WorkSharing possono essere combinati con il costrutto **parallel**, e le clausole ammesse sono l'unione di quelle ammesse per ognuno. (ovviamente ha poco senso combinare single e parallel).

• MASTER

Il costrutto **master** specifica che il blocco di istruzioni successivo verrà eseguito solo dal thread master.

Non c'è una barriera di sincronizzazione automatica, dunque gli altri thread possono fare altre cose nel frattempo.

```
#pragma omp master  
{  
}  
}
```

• BARRIER

Il costrutto **barrier** forza i thread ad attendere il completamento di tutte le istruzioni precedenti da parte di tutti gli altri.

È l'equivalente dell'MPI_barrier del MIMD DM.

Anche in questo caso non c'è una barriera di sincronizzazione automatica.

```
#pragma omp barrier
```

nb: barrier può essere utile per prendere i tempi.

- Runtime Library Routines

Le **RLR** vengono usate per intervenire sulle *variabili di controllo interne* a runtime (deve essere incluso il file `omp.h`), es. il numero di thread, informazioni sullo scheduling;

Esempi di routine:

- **omp_set_num_threads(scalar-integer-expression)**: definisce il numero di thread da utilizzare;
- **omp_get_max_threads()**: restituisce il numero massimo di thread disponibili per la prossima regione parallela;
- **omp_set_dynamic(scalar-integer-expression)**: permesso (0) o meno (1) al sistema di riadattare il numero di thread utilizzati;
- **omp_get_thread_num()**: restituisce l'id del thread;
- **omp_get_num_procs()**: restituisce il numero di core disponibili per il programma al momento della chiamata;
- **omp_get_num_threads()**: restituisce il numero di thread del team;

- Variabili d'ambiente

Le **variabili d'ambiente** servono per modificare il valore delle *variabili di controllo interne* prima dell'esecuzione (es. numero di thread, informazioni sullo scheduling).

Esempi:

- **OMP_NUM_THREADS**: numero di thread che verranno utilizzati nell'esecuzione/i successiva.

- `OMP_NUM_THREADS(integer)`
- `sh/bash: export OMP_NUM_THREADS=integer`

OMP_DYNAMIC: permesso (true) o meno (false) al sistema di riadattare il numero di thread utilizzati.

OMP_SCHEDULE: stabilisce lo scheduling da applicare nei costrutti *for*.

Tipi possibili: *static*, *dynamic*.

- `OMP_SCHEDULE(type, [chunk])`
- `sh/bash: export OMP_SCHEDULE="type, [chunk]"`

LEZ 17/18/19/20: LAB

LEZ 21 – Richiami di Alg. lineare e Prodotto matrice per vettore in ambiente MIMD

RICHIAMI DI ALGEBRA LINEARE

VETTORI

Un vettore è un segmento orientato caratterizzato da una *direzione*, un *verso* e un *modulo*.

$$\begin{array}{ll} \text{vettore (colonna) } \mathbf{x} & \mathbf{x} = \begin{pmatrix} 1.1 \\ -3.2 \\ \vdots \\ 0.9 \end{pmatrix} \quad \mathbf{x} \in \mathbb{R}^n \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \\ \text{vettore (riga) } \mathbf{x}^T & \mathbf{x}^T = (1.1 \quad -3.2 \quad \dots \quad 0.9) \end{array}$$

Con $\mathbf{x} \in \mathbb{R}^n$ intendiamo che il vettore \mathbf{x} ha n componenti.
Un vettore riga si ottiene *trasponendo* un vettore colonna.

Operazioni di base

scalare per vettore

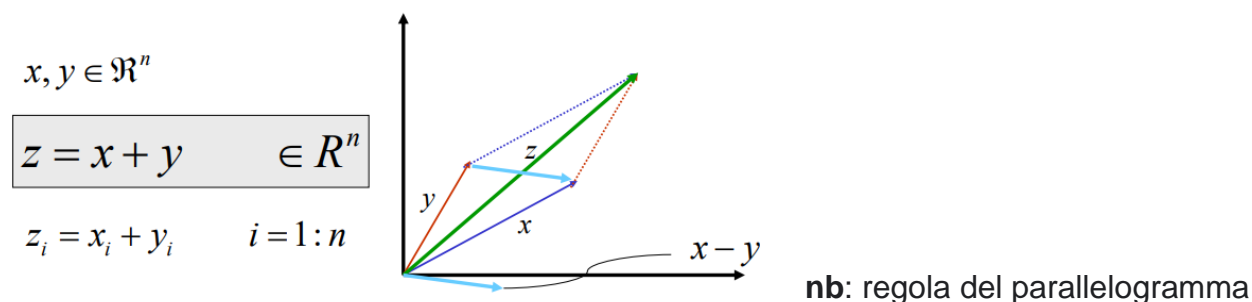
Dati un vettore \mathbf{x} (di n componenti) ed un numero (scalare) α , otteniamo un nuovo vettore \mathbf{y} (di n componenti) come il prodotto fra il vettore \mathbf{x} e lo scalare α .

Più precisamente, la componente i -esima di \mathbf{y} sarà data dal prodotto fra la componente i -esima di \mathbf{x} e α .

$$\mathbf{x} \in \mathbb{R}^n, \alpha \in \mathbb{R} \quad \boxed{\mathbf{y} = \alpha \mathbf{x} \in \mathbb{R}^n}$$
$$y_i = \alpha x_i \quad i = 1:n$$

somma di due vettori

Dati due vettori \mathbf{x} e \mathbf{y} , i quali devono avere lo **stesso numero n** di componenti, otteniamo un nuovo vettore \mathbf{z} (di n componenti) sommando l' i -esima componente di \mathbf{x} con la i -esima componente di \mathbf{y} .



somma saxpy

Si tratta della somma di un multiplo di un vettore a un vettore (è come se fosse una fusione di quei due visti prima).

$$x, y \in R^n, \alpha \in R$$

$$z = \alpha x + y \in R^n$$

$$z_i = \alpha x_i + y_i \quad i = 1:n$$

prodotto scalare di due vettori

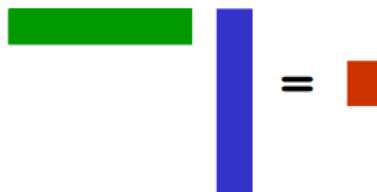
Prodotto scalare in quanto il risultato è uno scalare.

Si calcola facendo il prodotto delle componenti di egual posto dei due vettori, e poi si sommano questi prodotti.

$$x, y \in R^n$$

$$s = x^T y \in R$$

$$s = \sum_{i=1}^n x_i y_i$$



nb: $x^T y = y^T x$ (commuta)

MATRICI

Una matrice è una tabella in cui sono riportati in modo ordinato gli elementi di un dato insieme.

La sua *trasposta* sarà una nuova matrice che si ottiene scambiando le righe con le colonne.

Una matrice può essere *partizionata* per **colonne** o per **righe**:

- per colonne: $A = (a_{\cdot 1}, a_{\cdot 2}, \dots, a_{\cdot n})$
- per righe: $A = (a_{1\cdot}, a_{2\cdot}, \dots, a_{n\cdot})$

Una matrice può essere anche partizionata a blocchi.

Caratteristiche strutturali

Quando una matrice è costituita da moltissimi elementi, potrebbe essere conveniente sapere se la maggior parte degli elementi della matrice sia nulla o meno. In particolar modo distinguiamo due tipi di matrici:

- **matrici piene:** la maggior parte degli elementi sono *diversi* da zero.
- **matrici sparse:** la maggior parte degli elementi sono *uguali* a zero.

Possiamo distinguere vari **tipi comuni** di matrice, quali:

- **diagonali**: gli elementi extradiagonali sono nulli;
- **tridiagonali**: sono nulli tutti gli elementi al di fuori della diagonale principale e della diagonale immediatamente sopra e sotto;
- **triangolari sup**: sono nulli gli elementi del triangolo strettamente inferiore;
- **triangoli inf**: sono nulli gli elementi del triangolo strettamente superiore;
- **di Toeplitz**: elementi uguali sulle diagonali;
- **tridiagonali di Toeplitz**: combina tridiagonale con Toeplitz.

$D = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & x & 0 & 0 \\ 0 & 0 & x & 0 \\ 0 & 0 & 0 & x \end{pmatrix}$ <p>$a_{ij} = 0, i \neq j$</p>	matrice diagonale	$T = \begin{pmatrix} x & x & 0 & 0 \\ x & x & x & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{pmatrix}$ <p>$a_{ij} = 0, i-j > 1$</p>	matrice tridiagonale	$U = \begin{pmatrix} x & x & \dots & x \\ 0 & x & \dots & x \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & x \end{pmatrix}$ <p>$a_{ij} = 0, i > j$</p>	matrice triangolare superiore
$L = \begin{pmatrix} x & 0 & \dots & 0 \\ x & x & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ x & x & \dots & x \end{pmatrix}$ <p>$a_{ij} = 0, i < j$</p>	matrice triangolare inferiore	$V = \begin{pmatrix} d & e & f & g \\ c & d & e & f \\ b & c & d & e \\ a & b & c & d \end{pmatrix}$ <p>elementi uguali sulle diagonali</p>	matrice di Toeplitz	$\begin{pmatrix} 2 & -1 & 0 & 0 \\ 1 & 2 & -1 & 0 \\ 0 & 1 & 2 & -1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$ <p>combinazione di proprietà elementari</p>	matrice Tridiagonale di Toeplitz

Operazioni di base

scalare per matrice (multiplo di una matrice)

Il multiplo di una matrice si ha quando si moltiplica una matrice per uno scalare, e si ottiene una nuova matrice dove ogni componente è stata moltiplicata per quello scalare.

$$A \in R^{m \times n}, \alpha \in R$$

$$B = \alpha A \in R^{m \times n}$$

$$b_{ij} = \alpha a_{ij}, i = 1:m, j = 1:n$$

somma di due matrici

La somma di due matrici si ottiene sommando gli elementi di ugual posto. Le matrici devono avere le stesse dimensioni.

$$A, B \in R^{m \times n}$$

$$C = A + B \quad \in R^{m \times n}$$

$$c_{ij} = a_{ij} + b_{ij} \quad , i = 1:m, j = 1:n$$

prodotto matrice per vettore (IMPORTANTE)

Il prodotto matrice per vettore si ottiene moltiplicando una matrice qualsiasi ad un vettore **colonna**, il quale deve avere tante componenti quante sono le colonne della matrice.

nb: basta ricordare $m \times n * n \times 1$, dove i numeri interni devono essere uguali.

$$A \in R^{m \times n} , v \in R^n$$

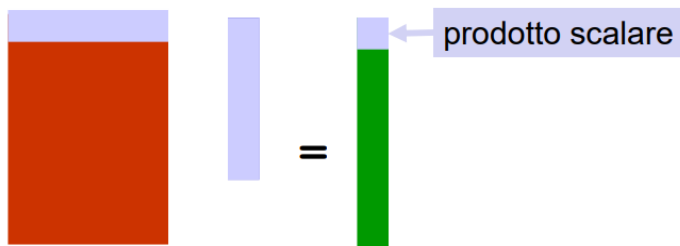
$$z = Av \quad \in R^m$$

$$z_i = \sum_{j=1}^n a_{ij} v_j \quad , i = 1:m$$

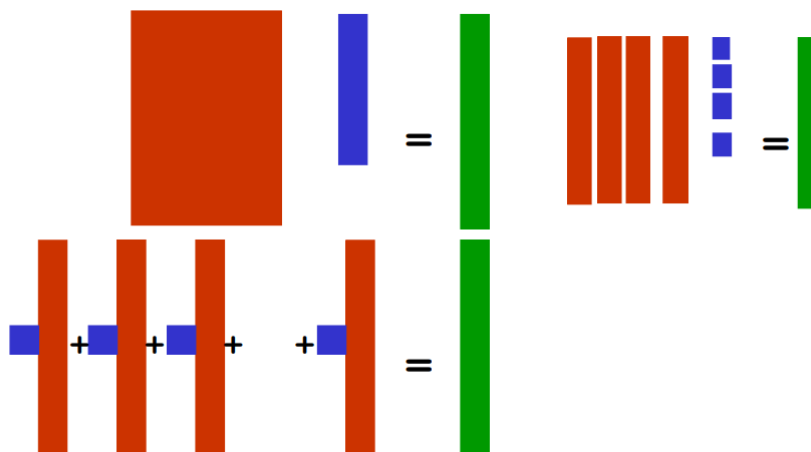
Effettuiamo un prodotto scalare fra la riga i -esima di A e il vettore v , ed otteniamo un nuovo vettore **colonna** z che avrà m componenti.

Questo prodotto è così importante perché è legato alla **trasformazione** di un vettore, e prende il nome di trasformazione lineare.

rappresentazione del primo passaggio: facciamo la somma fra il prodotto della prima riga della matrice con il vettore colonna, poi fra la seconda riga e il vettore colonna, poi la terza, ecc...



Da notare come il vettore risultato è una **combinazione lineare** delle colonne della matrice A.



Prodotto matrice per matrice

Il prodotto matrice per matrice si ottiene moltiplicando due matrici che rispettano il criterio di congruenza delle dimensioni (il numero di colonne della prima matrice = numero righe della seconda).

nb: basta ricordare $m \times n * n \times p$, dove i numeri interni devono essere uguali.

$$A \in R^{m \times n}, B \in R^{n \times p}$$

$$C = AB \in R^{m \times p}$$

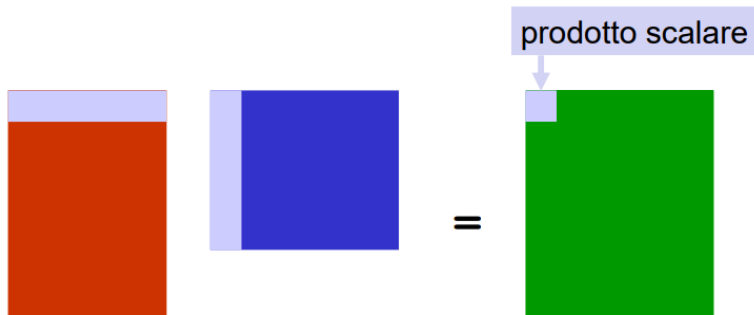
$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}, i = 1:m, j = 1:p$$

Effettuiamo il prodotto scalare della i-esima riga di A per la j-esima colonna di B. Il risultato sarà una nuova matrice **C** che avrà **m*p** componenti.

Il prodotto matrice x matrice può essere INTERPRETATO in 3 modi:

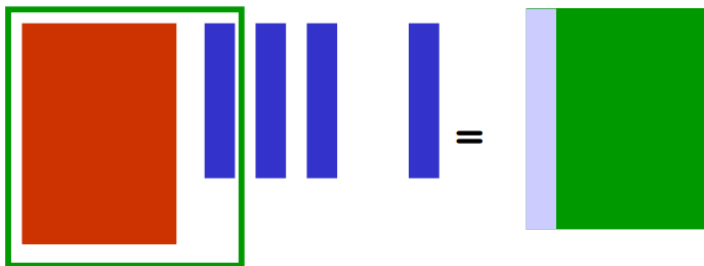
Successione di prodotti scalari di due vettori

Facciamo il prodotto scalare di due vettori: la riga i -esima di A e la colonna j -esima di B . Ripeterà il processo per tutte le righe rimanenti di A . (o colonne di B)



Successione di prodotti matrice x vettore

Ogni **colonna** della matrice C è il prodotto della matrice A per una colonna di B (si può vedere come una serie di *prodotti matrice x vettore*).

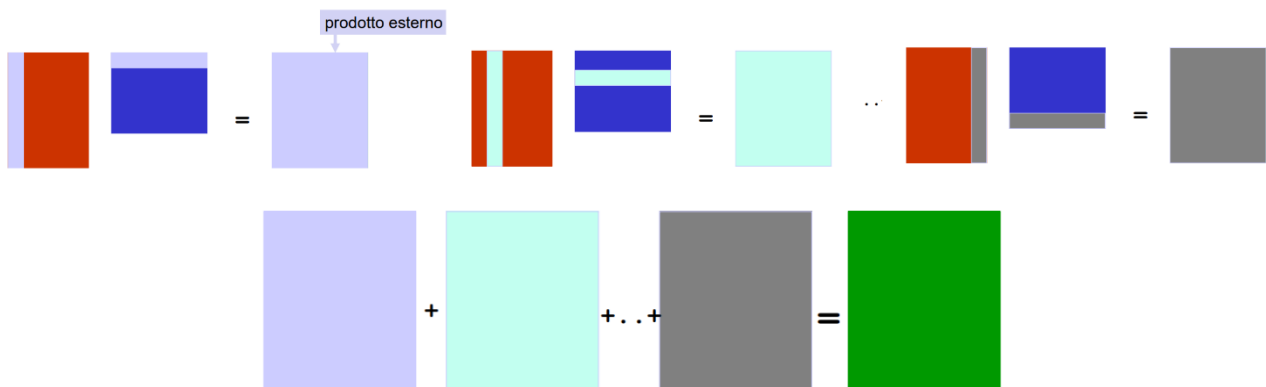


Successione di prodotti esterni

Facciamo il prodotto esterno fra due vettori: la colonna j -esima A e la riga i -esima di B (il risultato sarà una matrice).

Ripeterò il processo per tutte le colonne rimanenti di A (o righe di B).

La matrice risultato è la **somma di n matrici**, ognuna ottenuta come **prodotto esterno** di una colonna di A per una riga di B .



PRODOTTO MATRICE PER VETTORE

Per poter effettuare il prodotto **matrice X vettore** in ambiente parallelo dobbiamo innanzitutto **decomporre** il problema di dimensione N in P sottoproblemi di dimensione N/P e risolverli contemporaneamente su più calcolatori.

L'idea è suddividere la matrice in blocchi e assegnare il lavoro su questi blocchi a più unità processanti, come se fossero algoritmi a blocchi.

Possiamo ovviamente distinguere due casi: MIMD DM e MIMD SM.

MIMD DM

1 Strategia

La 1 strategia suddivide la matrice A in **blocchi di righe**.

Si tratta della strategia migliore in quanto si tratta dell'algoritmo più parallelizzabile.

Per semplicità supponiamo di avere una matrice A di dimensioni $N \times N$.

Suddividiamo la matrice in P blocchi di righe, ed assegniamo ogni blocco ad un processore. Il vettore X viene assegnato **INTERAMENTE** ad ogni processore.

Ogni processore potrà calcolare **SOLO** le prime N/P componenti del vettore Y .

2 Strategia

La 2 strategia suddivide la matrice A in **blocchi di colonne**.

È una variante della prima strategia, che ricordiamo rimanere la migliore.

Per semplicità supponiamo di avere una matrice A di dimensioni $N \times N$.

Suddividiamo la matrice in P blocchi di colonne, ed assegniamo ogni blocco ad un processore. Il vettore X viene **SUDDIVISO** in P parti, ed ogni parte viene assegnata ad un processore.

Ogni processore potrà calcolare **SOLO** un "contributo" del prodotto finale, cioè ogni processore calcola un intero vettore che dovrà tuttavia essere sommato con quelli calcolati dagli altri processori.

La *somma* di questi vettori parziali restituisce il vettore Y .

Tuttavia, effettuare la somma dei contributi significa che i processi devono comunicare fra di loro in quanto questi si trovano in memorie diverse: in tal caso un processore può spedire all'altro il suo contributo, così che quest'ultimo possa effettuare la somma, oppure tutti i processori spediscono a tutti, così che tutti facciano la somma e conservino nella propria memoria il risultato finale.

3 Strategia

La 3 strategia suddivide la matrice A in **blocchi quadrati**.

Per semplicità supponiamo di avere una matrice A di dimensioni $N \times N$.

Suddividiamo la matrice in P blocchi quadrati, ed assegniamo ogni blocco ad un processore. Il vettore X viene **SUDDIVISO** in tante parti quanti sono il numero di righe che

si sono formate in questa griglia, ed ogni parte viene assegnata a tutti i processori che si occupano dei blocchi che si trovano sulla stessa riga.

Ogni processore potrà calcolare SOLO un “contributo” del prodotto finale (che tuttavia è ancora inferiore a quello della 2 strategia), cioè si occuperà del calcolo della propria matrice, da cui otterranno una PORZIONE di un vettore parziale.

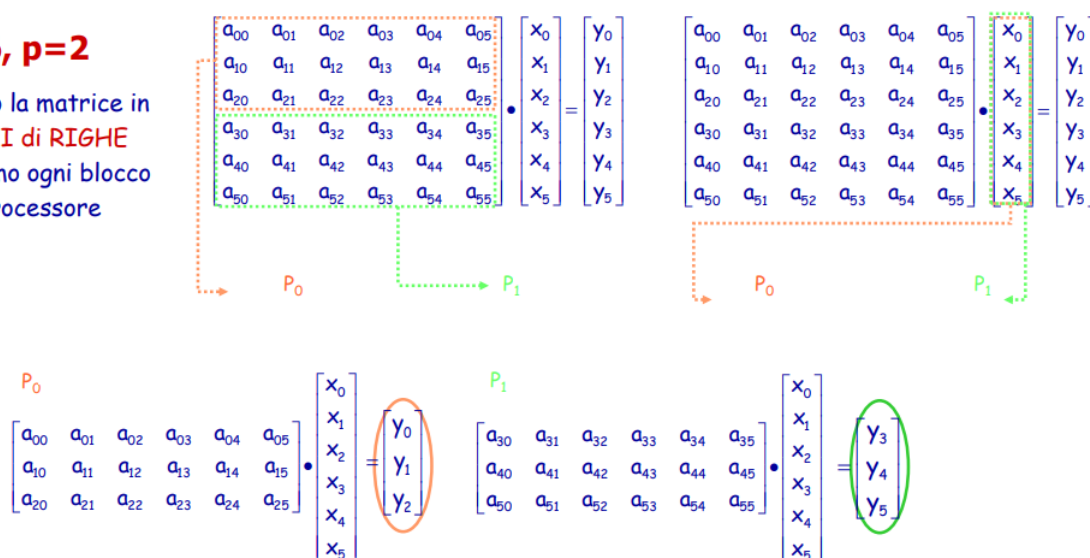
Le porzioni dei diversi vettori parziali vengono sommate fra di loro, ed infine vengono combinate per ottenere un unico vettore Y.

Anche in questo caso i processori devono comunicare fra di loro in quanto la somma dei contributi si trova in memoria diverse.

Es. 1 Strategia

n= 6, p=2

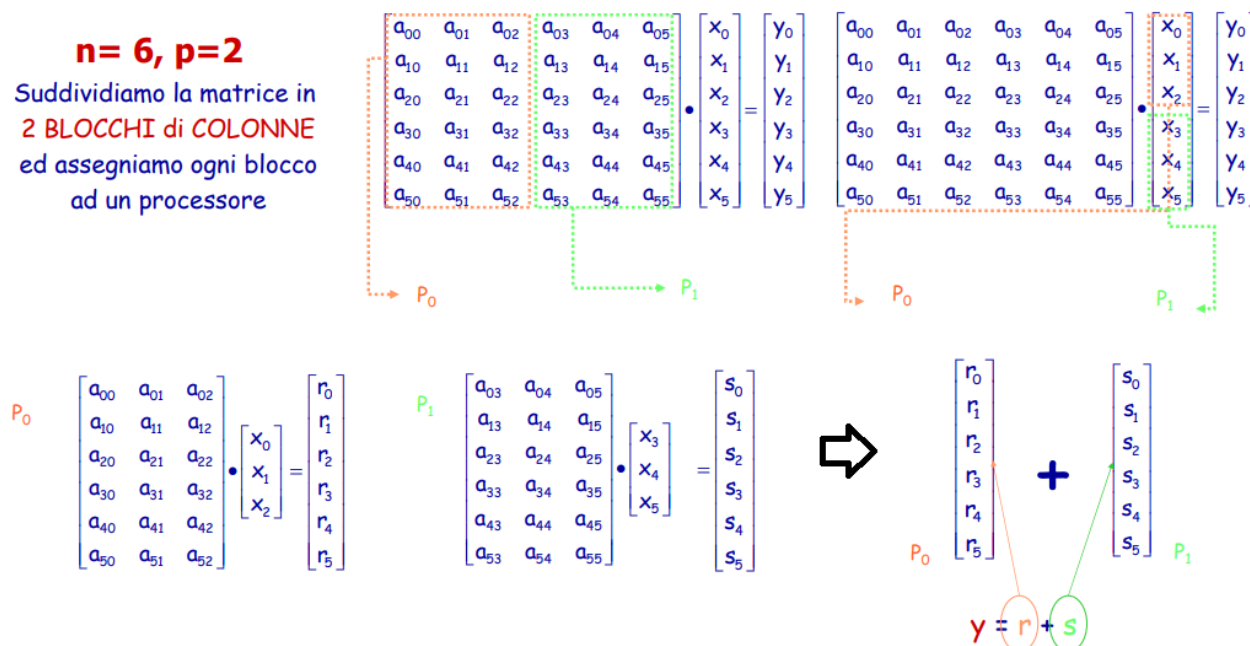
Suddividiamo la matrice in
2 BLOCCHI di RIGHE
ed assegniamo ogni blocco
ad un processore



Es. 2 Strategia

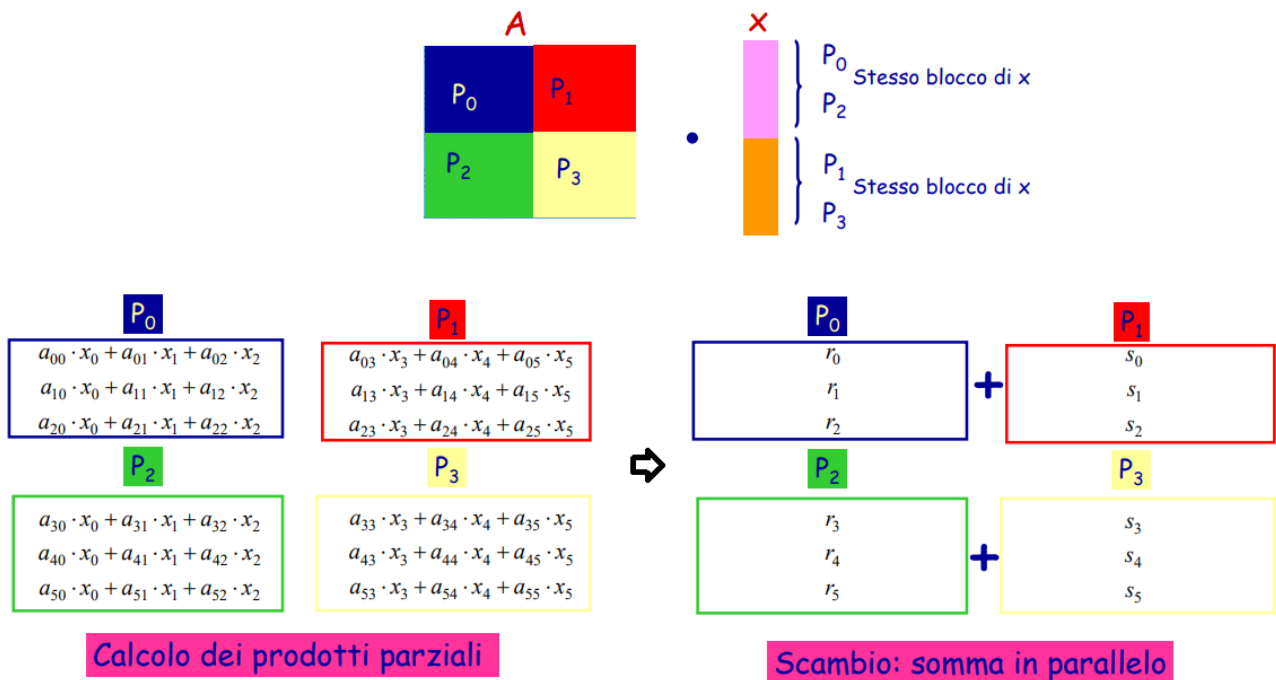
n= 6, p=2

Suddividiamo la matrice in
2 BLOCCHI di COLONNE
ed assegniamo ogni blocco
ad un processore



Es. 3 Strategia

N = 6 , Processori=4



MIMD SM

Per la MIMD SM vale lo stesso identico discorso che abbiamo fatto per la MIMD DM. Le strategie sono le stesse, ma risultano ancora più semplici in quanto non dovremo considerare le spedizioni.

LEZ 22 – Prendere i tempi in MPI e OpenMP

Prendere i tempi ci permette di capire quando si guadagna implementando la parallelizzazione in un software, e dunque se ne vale la pena.

Ricordiamo che il *tempo di esecuzione di un software* è esprimibile come:

$$\tau_p = k \cdot T_p(N)$$

dove per semplicità inglobiamo μ in T_p , e dove:

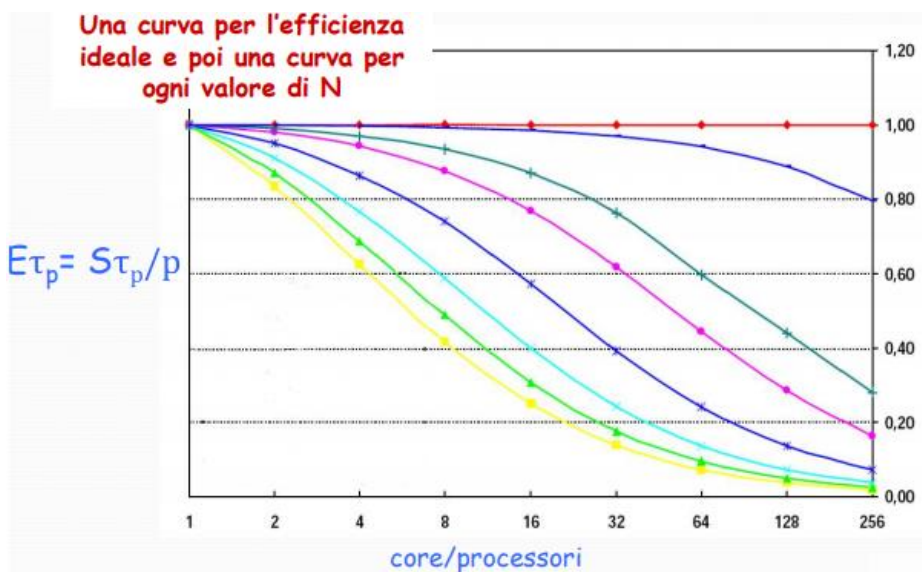
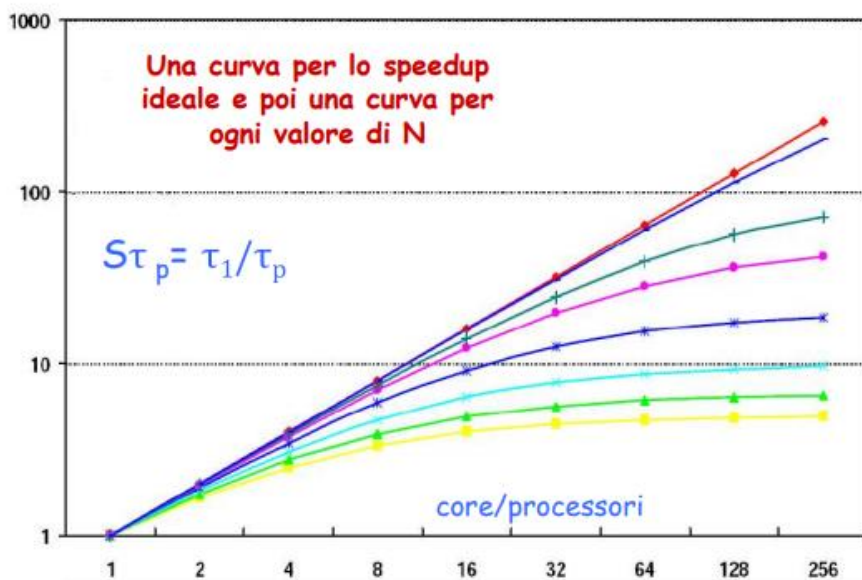
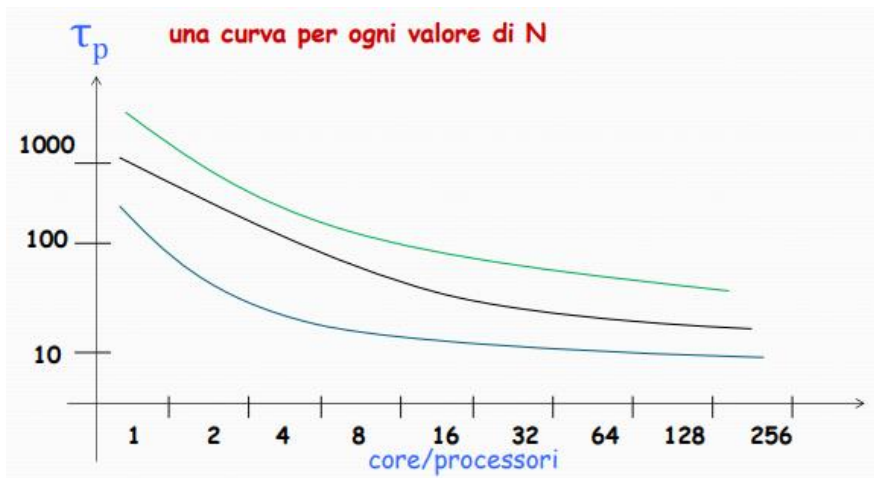
- in un ambiente MIMD-SM, $\mu = t_{\text{calc}}$ (t. 1 addizione).
- in un ambiente MIMD-DM, $\mu = t_{\text{calc}} + t_{\text{com}}$ (t. 1 addizione + t. 1 comunicazione).

Sarebbe bene anche realizzare dei grafici:

- per il **tempo di esecuzione**. Ci aspettiamo una riduzione dei tempi all'aumentare del numero di core, anche se paradossalmente potremmo avere in alcune situazioni una riduzione del tempo usando meno core.
- per lo **speedup**. Ci aspettiamo che i valori siano al di sotto dello speedup ideale (bisettrice rossa), sebbene potremmo trovarci in situazioni di speedup superlineare (di solito questo avviene per via delle ottimizzazioni fatte da MPI o OpenMP, oppure perché l'algoritmo in sequenziale è stato implementato male).
- per l'**efficienza**. Ci aspettiamo che i valori siano al di sotto di 1, e che all'aumentare del numero di core (per un N fissato) l'efficienza diminuisca.

nb: il numero di core deve stare sull'asse delle x, $\tau_p/S_{\tau_p}/E_{\tau_p}$ sull'asse delle y.
Le curve non sono necessariamente così lisce.

Esempi:



MPI

I tempi in MPI si prendono con `MPI_Wtime()`. Ovviamente dovremo prendere un tempo di inizio e di fine, posizionati rispettivamente subito prima e subito dopo l'algoritmo. Ciò verrà fatto per ogni processore.

Prima di prendere il tempo vogliamo sincronizzare i processori del communicator in modo che tutti abbiano raggiunto la stessa linea di codice, per questo vogliamo usare la funzione `MPI_Barrier()`.

Presi i tempi di ogni processore, questi potrebbero leggermente variare fra di loro:

- anche se si usa la stessa CPU, possono verificarsi interrupt;
- possono esserci minuscole differenze di tempo quando se riprende l'esecuzione dopo `MPI_Barrier()`.

A noi interessa solo il tempo massimo, e per questo possiamo considerare **2 possibilità**:

1. usare **MPI_Reduce**:

```
MPI_Reduce(&t, &max, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

solo il processore zero ha il massimo fra tutti i tempi;

2. Usare **MPI_ALLreduce**:

```
MPI_ALLreduce(&t, &max, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
```

tutti i processori hanno il massimo fra tutti i tempi.

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  int main(int argc, char *argv[]){
5      int menum, nproc;
6
7      MPI_Init(&argc, &argv);
8      MPI_Comm_rank(MPI_COMM_WORLD, &menum);
9      MPI_Comm_size(MPI_COMM_WORLD, &nproc);
10
11
12     double t, t1, t2;
13
14     MPI_Barrier(MPI_COMM_WORLD);    //tempo di inizio
15     t1=MPI_Wtime();
16
17     //algoritmo...
18
19     MPI_Barrier(MPI_COMM_WORLD);    //tempo di fine
20     t2=MPI_Wtime();
21
22     t=t2-t1;                        //tempo totale
23
24     MPI_Reduce(&t, &max, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD); //oppure MPI_ALLreduce
25
26
27     MPI_Finalize();
28     return 0;
29 }
```

OpenMP

I tempi in MPI si prendono con `omp_get_wtime()`. Ovviamente dovremo prendere un tempo di inizio e di fine, posizionati rispettivamente subito prima e subito dopo l'algoritmo. Ciò verrà fatto solo dal core master, in sequenziale.

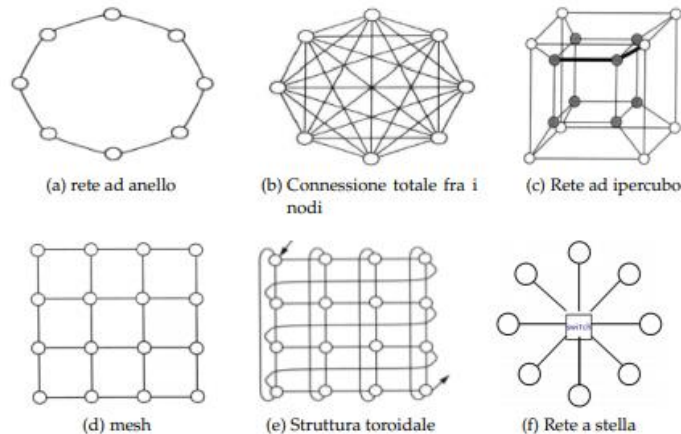
Infatti il tempo di inizio e il tempo di fine dovranno essere presi rispettivamente prima dell'inizio del pragma e dopo la fine del pragma.

```
1  #include <stdlib.h>
2  #include <omp.h>
3
4  int main(){
5      double t, t1, t2;
6
7      t1 = omp_get_wtime();
8      #pragma omp parallel
9      {
10         //istruzioni...
11     }
12     t2 = omp_get_wtime();
13
14     t=t2-t1;
15
16     return 0;
17 }
```


LEZ 23 – Topologie virtuali: griglie

Una topologia è la **geometria “virtuale”** in cui si immaginano disposti i processori. Questa può anche non avere alcun nesso con la loro disposizione “reale”.

Possiamo distinguere diversi esempi di topologie: anello, griglia (mesh), toro... L'utilizzo di una topologia per la progettazione di un algoritmo di ambiente MIMD è spesso legata alla geometria “intrinseca” del problema in esame.



Per la **creazione di topologie** possiamo sfruttare due funzioni:

- MPI_Graph_create: per la creazione di un *grafo* di processori;
- MPI_Cart_create: per la creazione di una *griglia* di processori.

Nello specifico MPI_Cart_create() è un'operazione collettiva che restituisce un nuovo communicator new_comm in cui i processi sono organizzati in una griglia di dimensione dim.

```
MPI_Cart_create(MPI_Comm comm_old, int dim, int *ndim, int *period,
                int *reorder, MPI_Comm *new_comm)
```

Dove comm_old è il communicator di input; dim il numero di dimensioni della griglia; *ndim è il vettore di dimensione dim contenente le lunghezze di ciascuna dimensione; *period è un vettore tale che se period[i]=1, la i-sima dimensione della griglia è periodica, se period[i]=0 invece no; *reorder è un vettore che dà il permesso di riordinare i menu, tale che se reorder[i]=1 il permesso è concesso, se reorder[i]=0 no; *new_comm è il vettore di output.

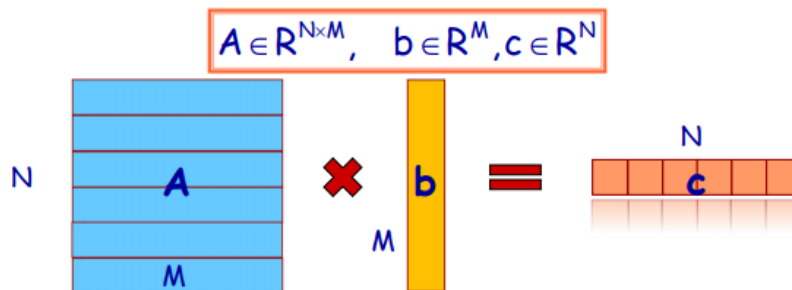
Insieme alla funzione di prima si usa anche la funzione MPI_Cart_coords(), la quale + un'operazione collettiva che restituisce a ciascun processo di comm_grid, con id=menu_grid, le sue coordinate all'interno della griglia predefinita.

```
MPI_Cart_coords(MPI_Comm comm_grid, int menu_grid, int dim,
                int *coord)
```

Dove comm_grid è il communicator della griglia; menu_grid è l'identificativo del processore, dim rappresenta la dimensione della griglia; *coords è un vettore di dimensione dim i cui elementi rapp. le coordinate del processo all'interno della griglia.

LEZ 24 a 27 – Approfondimenti sul prodotto Matrice X vettore

Un algoritmo per il calcolo matrice X vettore, considerata una matrice rettangolare A di dimensioni $N \times M$ e un vettore b di dimensione M , dà come risultato un nuovo vettore c di dimensione N .



In sequenziale, si tratta di effettuare N prodotti scalari di lunghezza M : infatti, per ogni riga di A dovrò fare M moltiplicazioni (prodotto puntuale) e poi sommare tutti questi prodotti (dunque $M-1$ somme).

$$N[M \text{ molt} + (M-1) \text{ add}]$$

Il tempo di una moltiplicazione è circa quello di un'addizione, dunque possiamo considerare $\text{molt} = \text{add}$. Di conseguenza, la **complessità computazionale dell'algoritmo sequenziale** sarà:

$$T_1(N \times M) = N[2M-1] t_{\text{calc}}$$

1 Strategia: Calcolo di speedup ed efficienza (MIMD DM e SM)

Generalizziamo la 1 strategie utilizzando sta volta una matrice $N \times M$ e n processori. Come ben ricordiamo prendiamo in considerazione la matrice A e la dividiamo per righe (ogni blocco riga di A avrà $\text{numRighe} = \text{NumRighe}/\text{numProc}$).

p processori:

$$\dim[A_i] = (N/p) \times M; \dim[b] = M$$

Per l'ambiente **MIMD DM** il vettore b deve essere dato a tutti i processori.

Tutti i processori, contemporaneamente, effettueranno N/p prodotti scalari di lunghezza M :

$$T_p(N \times M) = N/p [2M-1] \uparrow t_{\text{calc}}$$

Nel caso N non fosse perfettamente divisibile per p , si potrebbero aggiungere delle righe inizializzate a 0 in fondo ad A , in modo da poterci trovare sempre in una situazione di esatta divisibilità.

• Calcolo di Speed Up, Overhead ed Efficienza 1 Strat

Ricordiamo che l'algoritmo con cui fare il confronto è T_1 , che NON È l'algoritmo sequenziale, ma l'algoritmo parallelo che gira su un'unica unità processante (tali calcoli valgono sia per l'ambiente MIMD DM che SM).

Proprio perché la 1 strategia del prodotto Matrice x Vettore è un algoritmo completamente parallelizzabile, avremo: Speed Up ideale, Overhead nullo ed Efficienza massima.

$$S_p(N \times M) = T_1(N \times M) / T_p(N \times M) = \\ = N[2M-1] / (N/p [2M-1]) = p \quad N[2M-1] / (N[2M-1]) = p$$

$$Oh = p T_p(N \times M) - T_1(N \times M) = \\ = p(N/p [2M-1]) \uparrow_{calc} - N[2M-1] \uparrow_{calc} = 0$$

$$E_p(N \times M) = S_p(N \times M) / p = p/p = 1$$

Nell'ambiente MIMD DM: S_p si trova nel caso ideale, ovvero è p , solo se si sceglie di non unire il risultato finale. Infatti, in tal caso dovremmo considerare anche il tempo delle spedizioni. Nell'ambiente MIMD SM non abbiamo più nemmeno più questa preoccupazione in quanto tutto è presente nella memoria condivisa, e dunque non ci sono spedizioni.

nb: nella pratica ovviamente non sarà in grado di ottenere tali risultati ideali per via del k nella formula: $\tau = k \cdot T(n) \cdot \mu$, Infatti noi in questo momento **stiamo considerando l'algoritmo**, e non il software.

• Calcolo dell'isoefficienza 1 Strat

Ricordiamo che lo scopo dell'isoefficienza è individuare il nuovo size n_1 tale che l'**efficienza resti costante**. Il tutto si riduceva al calcolo dell'Overhead e a fare il rapporto fra questi:

$$I = \frac{Oh(n_1, p_1)}{Oh(n_0, p_0)}$$

Tuttavia l'Overhead della 1 strategia è sempre 0, dunque il rapporto sarebbe 0/0, ovvero una forma indeterminata. Per questo, per convenzione, l'isoefficienza è posta uguale ad **infinito**, ovvero posso usare qualunque costante moltiplicativa per calcolare n_1 e quindi controllare la scalabilità dell'algoritmo.

• Legge di WA per la 1 strategia

La legge di WA è un modo alternativo per scrivere lo Speed Up, che fa una distinzione fra parte puramente sequenziale e parte puramente parallela. Tuttavia in questo caso ho addirittura solo la parte puramente parallela.

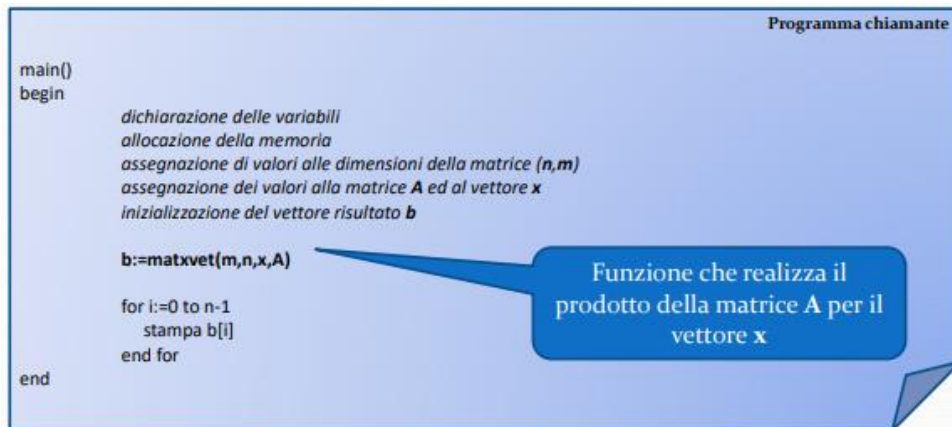
Infatti: $S_p = \frac{1}{\alpha + (1 - \alpha)/p}$ diventa solo: $\frac{1}{p}$.

Algoritmo per il prodotto Matrice x Vettore (MIMD SM) (MIMD DM LEZ 24 saltata)

La **decomposizione** del problema avviene direttamente sulla matrice A. Nessuno invia niente a nessuno perché tutto è già nella memoria condivisa.
Lo stesso vale per il vettore b.

Ognuno dei p core si occuperà di uno dei blocchi di righe della matrice A.
I p core si occuperanno di TUTTO il vettore b.

I **passi** da eseguire sono fondamentalmente questi:

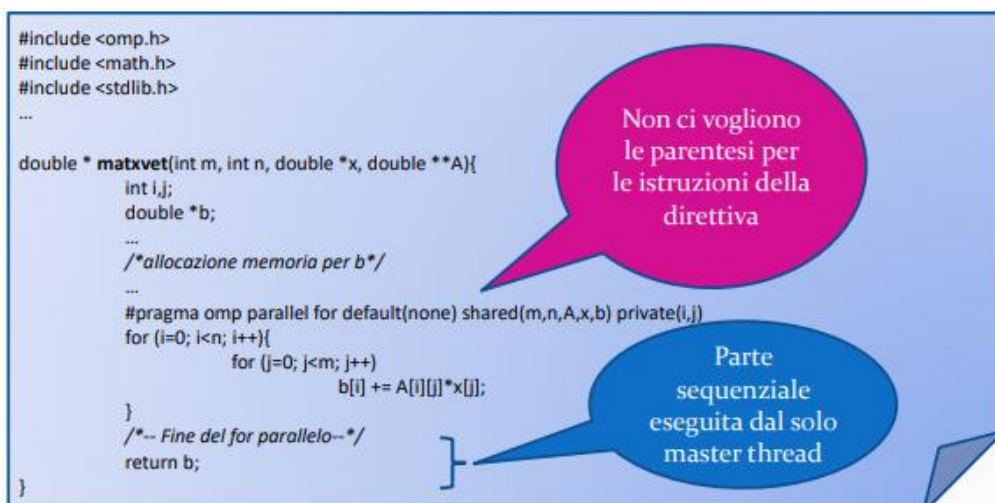


Nella funzione **matxvet()**:

- SOLO il processore Master si occuperà dell'allocazione della memoria per il vettore risultato.
- Il doppio ciclo for che si occupa del prodotto `matXvet` verrà eseguito in parallelo. Vista la situazione possiamo usare insieme alla direttiva `parallel` anche la direttiva `for`: tutti fanno la stessa cosa MA distribuendosi le iterazioni. Se volessimo, si può scegliere la distribuzione usando la clausola `schedule`.
- In linea generale, vogliamo che sia tutto condiviso tranne gli indici (ogni core dovrà averne una sua copia).

Se ponessimo in `private` anche `m`, `n`, `A` e `x` questi verrebbero de-inizializzati.

Il vettore risultato `b` deve essere posto in `shared`, altrimenti non sarebbe accessibile al di fuori della regione parallela.



2 Strategia: Calcolo di speedup ed efficienza (MIMD DM e SM)

Generalizziamo la 2 strategia utilizzando sta volta una matrice $N \times M$ e n processori. Come ben ricordiamo prendiamo in considerazione la matrice A e la dividiamo per colonne (ogni blocco colonna di A avrà $\text{numCol} = \text{NumCol}/\text{numProc}$). A differenza di prima, anche il vettore b deve essere distribuito “a pezzi” (mentre nella 1 strategia veniva inviato l'intero vettore): ogni pezzo avrà dimensioni M/p .

p processori

$$\dim[A_i] = N \times (M/p) \quad \dim[b_i] = M/p$$

Per l'ambiente **MIMD DM** la porzione del vettore b deve essere dato a tutti i processori. Tutti i processori, contemporaneamente, effettueranno N prodotti scalari di lunghezza M/p :

$$T_p(N \times M) = N [2M/p - 1] t_{calc}$$

Nel caso N non fosse perfettamente divisibile per p , si potrebbero aggiungere delle colonne inizializzate a 0 alla destra di A , in modo da poterci trovare sempre in una situazione di esatta divisibilità.

• Come possiamo raccogliere il tutto e calcolare il risultato finale?

Possiamo utilizzare una delle 3 strategie della **somma**:

1. Strategia: i processori spediscono (uno alla volta) il proprio vettore al processore P_0 che fa la somma (componente per componente);
2. Strategia: i processori si accoppiano (due a due), ovvero c'è chi spedisce e chi riceve il vettore somma (componente per componente). Alla fine P_0 avrà il vettore finale;
3. Strategia: come la seconda, ma tutti spediscono e ricevono. In questo modo alla fine tutti hanno il vettore finale.

Il costo computazionale è molto simile a quello della somma, solo che in questo caso non stiamo spedendo un solo valore ma un intero vettore di N elementi. Dunque il **costo**, che comprende **una spedizione** da un vettore ad un altro e **una somma**, sarà:

$$N t_{com} + N t_{calc}$$

(notiamo come sia lo stesso della somma ma moltiplichiamo anche per N . Ovviamente se siamo in ambiente MIMD SM non dobbiamo tener conto di t_{calc}).

Prendiamo in considerazione la **2 strategia**: supponendo di avere p potenza di 2, ad ogni passo ogni processore passerà il proprio vettore parziale al processore alla sua sinistra, che procederà ad effettuare la somma.

Come ben sappiamo, il numero di passi sarà $\log_2(p)$.

Quindi conoscendo il numero di passi e il costo ad ogni passo (N spedizioni + N somme), possiamo dire in generale che il **costo della raccolta e del calcolo finale**, utilizzando la **2 strategia**, per un prodotto Matrice x Vettore sia:

$$N \cdot \log_2(p) t_{com} + N \cdot \log_2(p) t_{calc}$$

Ricordando che t_{com} è dalle 2 alle 3 volte il costo di t_{calc} , possiamo riscrivere il tutto considerando solo t_{calc} :

$$c N \cdot \log_2(p) t_{calc} + N \cdot \log_2(p) t_{calc}$$

$$t_{com} = c t_{calc}$$

$$2 \leq c \leq 3$$

Osservazioni:

Usare la prima strategia per le somme è poco efficiente. Potremmo anche usare la 3, ma i conti rimangono gli stessi della seconda.

Questa fase non è stata necessaria nella prima strategia del prodotto MatxVet semplicemente perché in quel caso abbiamo già i vettori “finiti”, mentre in questo caso abbiamo p vettori parziali che dovranno essere sommati fra di loro così da ottenere quelli “finiti”.

• Calcolo di Speed Up, Overhead ed Efficienza 1 Strat

Ricordiamo che l'algoritmo con cui fare il confronto è T_1 , che NON È l'algoritmo sequenziale, ma l'algoritmo parallelo che gira su un'unica unità processante (tali calcoli valgono sia per l'ambiente MIMD DM che SM).

La 2 strategia del prodotto Matrice x Vettore NON è un algoritmo completamente parallelizzabile (com'era nel caso della 1 strategia), questo perché dovremmo comunque effettuare dopo una serie di somme e spedizioni:

Per questo motivo, per la **somma usando la 2 strategia**:

$$S_p(N \times M) = T_1(N \times M) / T_p(N \times M) = \\ = N[2M-1] / (N [2M/p-1] + cN \log_2(p) + N \log_2(p)) < p$$

$$Oh = p T_p(N \times M) - T_1(N \times M) = \\ = p (N [2M/p-1] + cN \log_2(p) + N \log_2(p)) t_{calc} - N[2M-1] t_{calc}$$

$$E_p(N \times M) = S_p(N \times M) / p = \\ = N[2M-1] / p (N [2M/p-1] + cN \log_2(p) + N \log_2(p))$$

(non è da ricordare a memoria, basta ricordare il ragionamento).

La **somma usando la 1 strategia** è ovviamente più lenta.

I processori spediscono il proprio vettore di N elementi, uno alla volta al processore P₀ che, quindi, aggiorna il proprio vettore effettuando N somme, cioè:

$$p-1(N \, t_{com} + N \, t_{calc}) =$$

$$= p-1(c \cdot N + N) \, t_{calc}$$

Da questo calcoliamo Speed Up, Overhead ed Efficienza:

$$S_p(N \times M) = T_1(N \times M) / T_p(N \times M) =$$

$$= N[2M-1] / (N [2M/p-1] + p-1(cN + N)) < p$$

$$Oh = p \, T_p(N \times M) - T_1(N \times M) =$$

$$= p (N [2M/p-1] + p-1 (cN + N)) \, t_{calc} - N[2M-1] \, t_{calc}$$

$$E_p(N \times M) = S_p(N \times M) / p =$$

$$= N[2M-1] / p (N [2M/p-1] + p-1(cN + N))$$

Tutto ciò ci fa capire che implementare la 2 strategia è, di fatto, inutile. Conviene SEMPRE implementare la 1 strategia.

...in ambiente MIMD SM

In ambiente MIMD SM ovviamente non abbiamo più a che fare con le spedizioni tuttavia dobbiamo preoccuparci a sincronizzare gli accessi in memoria.

Infatti alla fine noi abbiamo p vettori parziali che devono essere sommati fra di loro in un unico vettore c . Ad ogni passo, ciascun core deve addizionare la propria somma parziale a $c[i]$ (elemento i -simo del vettore risultato). I core devono accedere a $c[i]$ uno alla volta.

Tenendo conto di ciò, il calcolo di Speed Up, Overhead ed Efficienza segue praticamente lo stesso ragionamento dell'ambiente MIMD DM a cui, però, rimuoviamo il costo delle spedizioni.

MIMD-SM
p core
II strategia per collezione vettori

$$S_p(N \times M) = T_1(N \times M) / T_p(N \times M) =$$

$$= N[2M-1] / (N [2M/p-1] + cN \log_2(p) + N \log_2(p)) < p$$

$$Oh = p \, T_p(N \times M) - T_1(N \times M) =$$

$$= p (N [2M/p-1] + cN \log_2(p) + N \log_2(p)) \, t_{calc} - N[2M-1] \, t_{calc}$$

$$E_p(N \times M) = S_p(N \times M) / p =$$

$$= p \, N[2M-1] / (N [2M/p-1] + cN \log_2(p) + N \log_2(p))$$

MIMD-SM
p core
I strategia per collezione vettori

$$S_p(N \times M) = T_1(N \times M) / T_p(N \times M) =$$

$$= N[2M-1] / (N [2M/p-1] + p-1(cN + N)) < p$$

$$Oh = p \, T_p(N \times M) - T_1(N \times M) =$$

$$= p (N [2M/p-1] + p-1 (cN + N)) \, t_{calc} - N[2M-1] \, t_{calc}$$

$$E_p(N \times M) = S_p(N \times M) / p =$$

$$= N[2M-1] / p (N [2M/p-1] + p-1(cN + N))$$

• Calcolo dell'isoefficienza 2 Strat

Ricordiamo che lo scopo dell'isoefficienza è individuare il nuovo size n_1 tale che l'**efficienza resti costante**. Il tutto si riduceva al calcolo dell'Overhead e a fare il rapporto fra questi:

$$I = \frac{O_h(n_1, p_1)}{O_h(n_0, p_0)}$$

Per il calcolo dell'isoefficienza è necessario separare i conti tra la 1 e la 2 strategia impiegata nella collezione dei risultati.

Sia nel caso della 1 strategia che della strategia l'Overhead dipende dal numero delle righe e dal numero delle unità processanti:

I strategia per collezione vettori

$$\begin{aligned} O_h &= p (N [2M/p-1] + (p-1) N) - N[2M-1] = \\ &= pN[2M/p] - pN + p^2 N - pN - N[2M-1] = \\ &= 2NM - pN + p^2 N - pN - 2NM + N = \\ &= -2p N + p^2 N + N = N(-2p + p^2 + 1) \end{aligned}$$

$$I(p_0, p_1, n_0) = C = [N_1 (-2p_1 + p_1^2 + 1)] / [N_0 (-2p_0 + p_0^2 + 1)]$$

~~$$\begin{aligned} I_1 M_1 &= C N_0 M_0 = \\ &= I_0 M_0 [N_1 (-2p_1 + p_1^2 + 1)] / [N_0 (-2p_0 + p_0^2 + 1)] \end{aligned}$$~~

II strategia per collezione vettori

$$\begin{aligned} O_h &= p (N [2M/p-1] + N \log_2(p)) - N[2M-1] = \\ &= 2M N - p N + p N \log_2(p) - 2M N + N = \\ &= -p N + p N \log_2(p) + N = \\ &= N (-p + p \log_2(p) + 1) \end{aligned}$$

$$I(p_0, p_1, n_0) = C = [N_1 (-p_1 + p_1 \log_2(p_1) + 1)] / [N_0 (-p_0 + p_0 \log_2(p_0) + 1)]$$

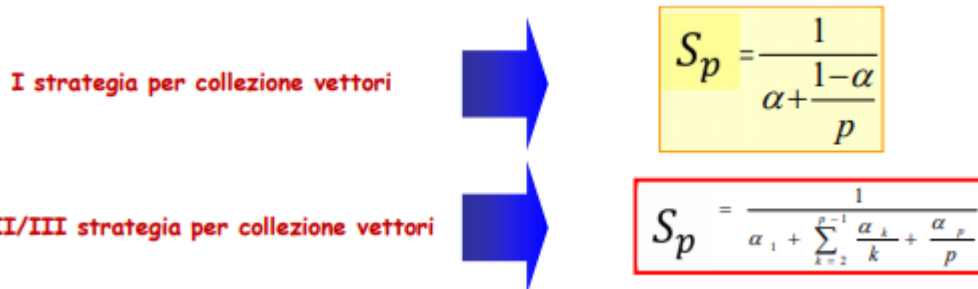
~~$$\begin{aligned} I_1 M_1 &= C N_0 M_0 = \\ &= I_0 M_0 [N_1 (-p_1 + p_1 \log_2(p_1) + 1)] / [N_0 (-p_0 + p_0 \log_2(p_0) + 1)] \end{aligned}$$~~

• Legge di WA per la 2 Strat

La legge di WA è un modo alternativo per scrivere lo Speed Up, che fa una distinzione fra parte puramente sequenziale e parte puramente parallela. In questo caso non sono fortunato come nella prima strategia.

Devo fare innanzitutto una distinzione fra la **fase di calcolo**, nella quale lavorano tutti i processori e dunque è puramente parallela, e la **fase di collezione dei risultati**, che invece avviene in maniera sequenziale.

Nello specifico, come ben ricordiamo nel caso della 1 strategia possiamo usare direttamente WA mentre nella 2/3 dobbiamo usare la forma generalizzata in quanto la parallela e sequenziale non sono nettamente separate.



• 1 Strategia per la collezione dei risultati

Come già detto, in questo caso ho una netta distinzione fra parte sequenziale e parallela e dunque possiamo usare direttamente la **WA**.

Individuiamo le fasi

fase 1

Si tratta di una fase tutta *parallela*, in quanto ogni unità processante effettuerà le proprie somme. Il calcolo dei prodotti parziali, considerati p processori, è: **$N[2M/p - 1]$** ;
Il calcolo dei prodotti parziali, considerato un unico processore, è: **$N[2M-1]$** ;

Siccome i prodotti parziali sono fatti da p processori, verranno effettuate contemporaneamente: **$pN[2M/p - 1]$** delle **$N[2M-1]$** operazioni.

$$1 - \alpha = p \frac{N [2 M / p - 1]}{N [2 M - 1]} \Rightarrow \frac{1 - \alpha}{p} = \frac{p}{p} \cdot \frac{2 M / p - 1}{2 M - 1}$$

fase 2

Si tratta di una fase tutta *sequenziale*, in quanto il processore Master si occupa di effettuare la somma totale.

$$\alpha = \frac{(p - 1)N}{N [2 M - 1]} = \frac{p - 1}{2 M - 1}$$

È **importante** che dalla somma della parte sequenziale e della parte parallela otteniamo **1**, così da capire se **abbiamo fatto bene i conti**.

Adesso ci rimane da calcolare lo **Speed Up con WA**:

$$\frac{1 - \alpha}{p} = \frac{2M / p - 1}{2M - 1} \quad \alpha = \frac{p - 1}{2M - 1}$$

$$S_p = \frac{1}{\frac{2M / p - 1}{2M - 1} + \frac{p - 1}{2M - 1}} = \frac{2M - 1}{2M / p + p - 2}$$

•• 2/3 Strategia per la collezione dei risultati

Come già detto, in questo caso NON ho una netta distinzione fra parte sequenziale e parallela, dunque dobbiamo usare la **WA generalizzata**.
Per semplicità supponiamo che $p=4$.

1 fase (è la stessa della 1 strategia)

Si tratta di una fase tutta *parallela*, in quanto ogni unità processante effettuerà le proprie somme.

$$\alpha_4 \equiv (1 - \alpha) = p \frac{N (2M / p - 1)}{N [2M - 1]} = \frac{4 (M / 2 - 1)}{2M - 1}$$

nb: con la legge di WAG la parte parallela non viene più indicata come (1-a) ma come α_p . Infatti, non indica *tutte* le operazioni parallele ma solo quelle eseguite con *parallelismo totale*.

Da questo punto in poi,

andremo ad analizzare quanto lavoro viene svolto quando lavorano solo 3, 2 ed 1 processori, chiamando tali valori analogamente come: α_3 , α_2 , α_1 .

Non c'è **nessuna fase in cui lavorano solo 3 processori**:

$$\alpha_3 = 0$$

Abbiamo poi una **fase di parallelismo parziale**, dove lavorano solo **2 processori/core**:

$$\alpha_2 = 2 \frac{N}{N [2M - 1]} = \frac{2}{2M - 1}$$

Infine, abbiamo una fase **puramente sequenziale**.

$$\alpha_1 = \frac{N}{N [2M - 1]} = \frac{1}{2M - 1}$$

Non resta che **sostituire**:

$$\alpha_4 = \frac{4(M/2 - 1)}{2M - 1} \quad \alpha_3 = 0 \quad \alpha_2 = \frac{2}{2M - 1} \quad \alpha_1 = \frac{1}{2M - 1}$$

$$S_p = \frac{1}{\frac{1}{4} \cdot \frac{4(M/2 - 1)}{2M - 1} + \frac{1}{2} \cdot \frac{2}{2M - 1} + \frac{1}{2M - 1}}$$



$$S_p = \frac{1}{\frac{(M/2 - 1)}{2M - 1} + \frac{1}{2M - 1} + \frac{1}{2M - 1}}$$



$$S_p = \frac{1}{\frac{(M/2 - 1)}{2M - 1} + \frac{2}{2M - 1}} = \frac{2M - 1}{M/2 + 1}$$