

Appunti Calcolo Parallello

Sommario

Introduzione (Lez.1).....	4
Calcolo ad alte prestazioni	4
Supercomputer	4
Tempo di esecuzione di un software	4
Calcolo parallelo	5
Tassonomia di Flynn (Lez.2).....	5
Tipi di parallelismo	5
Parallelismo temporale	5
Parallelismo spaziale.....	6
Parallelismo asincrono	7
Classificazione delle architetture parallele (tassonomia di Flynn).....	7
Classificazione calcolatori MIMD.....	7
Calcolo parallelo vs calcolo distribuito.....	9
Introduzione alla libreria OpenMP (Lez.3).....	10
Threads e processi	10
Architettura multicore.....	11
Libreria OpenMP	11
Direttive	12
Clausole.....	13
Runtime library routines.....	13
Variabili d'ambiente	14
Parallelismo in ambiente multicore (Lez.4).....	14
Somma di due vettori di dimensione N.....	16
Somma degli elementi di un vettore di dimensione N (somma di N numeri).....	17
I strategia (prima strategia).....	19
II strategia (seconda strategia).....	19
Valutazione di un algoritmo parallelo (Lez.5).....	20
Legge di Ware-Amdahl (Lez.7).....	26
Speed-up prossimi a quelli ideali negli algoritmi non full parallel	26
Legge di Ware-Amdahl	28
Isoefficienza (Lez.8).....	30

Calcolo isoefficienza nel problema della somma di N numeri (II strategia).....	32
Calcolo di speed-up, overhead ed efficienza nell'algoritmo parallelo per la somma di due vettori di dimensione N	33
Calcolo di speed-up, overhead ed efficienza nell'algoritmo parallelo per la somma di N numeri	33
I strategia (prima strategia).....	33
II strategia (seconda strategia).....	34
Valutazione teorica algoritmi paralleli delle somme (Lez.10).....	35
Calcolo dello speed-up con legge di Ware-Amdahl.....	36
Somma di due vettori di dimensione N.....	36
Somma degli elementi di un vettore di dimensione N (somma di N numeri)....	36
N non divisibile per p	38
Calcolo isoefficienza	39
Somma di due vettori di dimensione N.....	39
Somma degli elementi di un vettore di dimensione N (somma di N numeri)....	39
Costrutti work sharing e critical (Lez.12)	40
Costrutto for	40
Costrutto sections (Lez.22).....	41
Costrutto single (Lez.22)	41
Scheduling per costrutto for (Lez.13)	41
Prodotto scalare tra due vettori di dimensione N (Lez.14)	42
Speed-up, overhead ed efficienza dell'algoritmo parallelo per il prodotto scalare	44
Isoefficienza del prodotto scalare di 2 vettori di dimensione N	46
Speed-up calcolato con la legge di Ware-Amdahl.....	47
Ricerca e ordinamento in ambiente multicore (Lez.16)	50
Ricerca binaria (o dicotomica).....	50
Odd-Even sorting.....	52
Algoritmi full-parallel per matrici (Lez.17)	55
Prodotto di uno scalare per una matrice.....	55
BLOCCHI di righe (I strategia).....	56
BLOCCHI di colonne (II strategia)	57
BLOCCHI di righe-colonne (III strategia)	57
Strategie di parallelizzazione per problemi di tipo element-wise.....	58
Valutazione strategie per algoritmi full-parallel per matrici (Lez.18)	58

Calcolo di speed-up, overhead ed efficienza per il prodotto di uno scalare per una matrice	59
I strategia (blocchi di righe)	59
II strategia (blocchi di colonne).....	60
III strategia (blocchi di righe-colonne)	61
Isoefficienza del prodotto di uno scalare ed una matrice	62
Speed-up del prodotto di uno scalare per una matrice caratterizzato con la legge di Ware-Amdahl	62
Codici per algoritmo del prodotto di uno scalare per una matrice (Lez.19).....	65
Collapse.....	65
Prodotto matrice per vettore (Lez.20).....	65
I strategia (blocchi di righe)	67
II strategia (blocchi di colonne).....	68
III strategia (blocchi di righe-colonne).....	69
Valutazione strategie 1-2 del prodotto matrice per vettore (Lez.21).....	70
Calcolo di speed-up, overhead, efficienza ed isoefficienza per la I strategia (blocchi di righe)	71
Speed-up ed efficienza del prodotto matrice per vettore I strategia caratterizzato con la legge di Ware-Amdahl.....	72
Calcolo di speed-up, overhead, efficienza ed isoefficienza per la II strategia (blocchi di colonne).....	72
Speed-up ed efficienza del prodotto matrice per vettore II strategia caratterizzato con la legge di Ware-Amdahl.....	75
Direttive master e barrier (Lez.22).....	77
Valutazione strategia 3 del prodotto matrice per vettore (Lez.23-25).....	78
Calcolo di speed-up, overhead ed efficienza per la III strategia (blocchi di righe-colonne)	78
Isoefficienza per la III strategia (blocchi di righe-colonne).....	81
Calcolo di speed-up, overhead ed efficienza per la III strategia (blocchi di righe-colonne) quando righe e/o colonne non sono esattamente divisibili per il corrispondente numero di core	82
Isoefficienza per la III strategia (blocchi di righe-colonne) quando righe e/o colonne non sono esattamente divisibili per il corrispondente numero di core	86
Problema SAXPY (Lez.26)	89
Strong e weak scalability del problema SAXPY	90
Costrutti critical e atomic	92

Introduzione (Lez.1)

Calcolo ad alte prestazioni

Il calcolo ad alte prestazioni (HPC – **High Performance Computing**) è uno strumento di calcolo scientifico che impiega algoritmi, software e soprattutto hardware (ad es. i **supercomputer**) per fornire soluzioni in tempo reale, o per meglio dire in **tempo utile**, su larga scala, modellando e prevedendo con velocità e precisione un'ampia gamma di proprietà e fenomeni fisici.

Applicazioni di questo strumento possono avvenire nel campo della sicurezza informatica, della meteorologia, dei servizi bancari e finanziari, ecc.

Supercomputer

Il supercomputer è un super calcolatore il cui scopo è quello di risolvere i problemi, facendogli fare i calcoli al nostro posto, nel modo più veloce possibile.

Dal 1993, due volte l'anno, viene stilata la lista dei supercomputer più veloci al mondo, chiamata **TOP 500**. Le prestazioni vengono misurate utilizzando **Rmax**, che è un benchmark (insieme di test software utilizzati per misurare le prestazioni di un sistema) di **LINPACK**, confrontando la capacità di risoluzione di un sistema di equazioni lineari ($Ax = b$).

Tempo di esecuzione di un software

Il tempo di esecuzione di un software è dato da

$$\tau = k \cdot T(N) \cdot \mu .$$

Con τ che rappresenta appunto il **tempo di esecuzione**, k una **costante moltiplicativa** che rappresenta i tempi di I/O o i ritardi relativi alla scrittura in memoria, $T(N)$ la **complessità computazionale dell'algoritmo**, che dipende dal metodo risolutivo scelto, e μ il **tempo di esecuzione di un'operazione floating point**, dipendente dall'ambiente di calcolo.

In generale, quindi, per ridurre il tempo di esecuzione di un software si può procedere alla riduzione di $T(N)$ e μ .

In particolare, **si può ridurre $T(N)$** riorganizzando opportunamente l'algoritmo oppure trovandone un altro con complessità computazionale inferiore, sempre tenendo conto che esistono algoritmi, detti **ottimali**, con complessità computazionale minima.

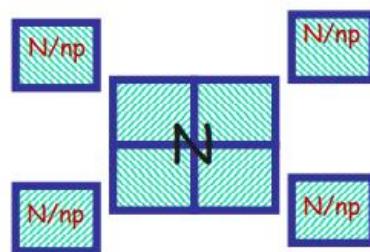
Si può invece ridurre μ migliorando la tecnologia, ad esempio con l'utilizzo di **tecniche di miniaturizzazione** che permettano di ridurre la distanza che i dati devono percorrere tra la CPU e la memoria.

Calcolo parallelo

Si può ulteriormente ridurre la complessità computazionale $T(N)$, che per algoritmi sequenziali equivale a $T_1(N)$ con 1 il numero di processori, **sfruttando un maggior numero di unità processanti** ($T_1(N)$ diventa $T_{np}(N)$, con np il numero di unità processanti appunto).

L'idea del calcolo parallelo si basa sull'impiego simultaneo di risorse di calcolo multiple per risolvere un unico problema, decomponendolo in parti, eseguendo queste ultime in modo seriale su differenti CPU.

In particolare, si tratta di decomporre un problema di dimensione N in np sottoproblemi di dimensione N/np , risolvendoli contemporaneamente utilizzando np CPU.



Tassonomia di Flynn (Lez.2)

Tipi di parallelismo

Esistono diversi tipi di parallelismo, ognuno dei quali si applica in contesti diversi e offre vantaggi specifici:

- **parallelismo temporale**;
- **parallelismo spaziale**;
- **parallelismo asincrono**.

Attualmente tutti i microprocessori utilizzano diverse forme di parallelismo.

Parallelismo temporale

Il parallelismo temporale viene implementato nei **calcolatori tradizionali basati sullo schema di Von Neumann** e si fonda sulla tecnica della catena di montaggio,

o **pipeline**, dove **più entità eseguono contemporaneamente fasi successive dello stesso lavoro**.

Questo tipo di parallelismo prevede infatti la scomposizione di un'istruzione in fasi dette **segmenti**, ovvero operazioni elementari. Ad esempio un'addizione floating point è divisa in 4 segmenti, ciascuno preposto ad una fase dell'operazione:



In processori che non utilizzano questa tecnica nell'operazione, ad esempio, di N somme floating point sarebbe attivo un solo segmento alla volta. Utilizzando invece la pipeline, ogni volta che termina il primo segmento di una somma partirebbe il primo segmento di un'altra (somma). Così facendo, una volta a regime, saranno attivi tutti i segmenti contemporaneamente.

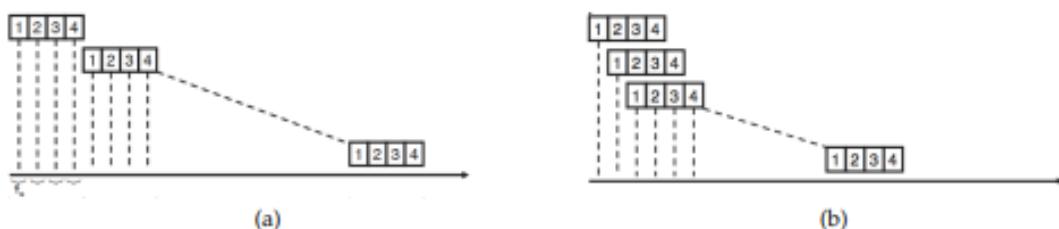
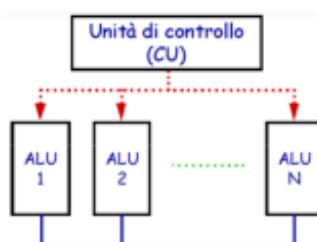


Figura 1.2: (1.2a) Somma di N coppie di numeri su un processore tradizionale. (1.2b) Somma di N coppie di numeri su un processore pipelined.

Parallelismo spaziale

Il parallelismo spaziale è in grado di far **eseguire contemporaneamente, a più unità, le stesse istruzioni su dati diversi**.

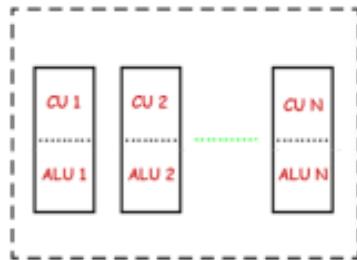
Questo tipo di parallelismo **riguarda l'unità operativa**, infatti avremo **un'unica unità di controllo (CU)** che **impartisce le stesse istruzioni su dati diversi a varie ALU** (Arithmetic Logic Unit).



Parallelismo asincrono

Il parallelismo asincrono è in grado di far **eseguire contemporaneamente, a più unità, istruzioni diverse su dati diversi**.

Questo tipo di parallelismo **riguarda la CPU**, infatti **ne avremo diverse** (di CPU) che eseguono in parallelo istruzioni diverse su dati diversi.



Classificazione delle architetture parallele (tassonomia di Flynn)

Le architetture parallele, secondo la tassonomia di Flynn, **vengono classificate in base al numero di flussi di dati elaborabili simultaneamente e al numero di istruzioni che possono essere eseguite in ogni istante**.

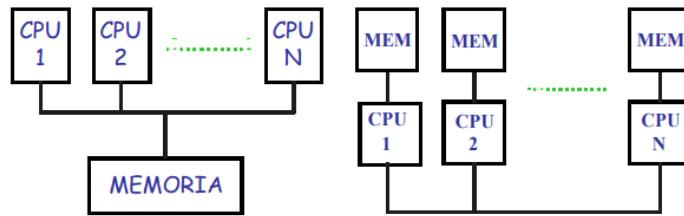
Le **principali classi architetturali** della tassonomia di Flynn sono:

- **SISD (Single Instruction Single Data)**, nella quale rientrano i calcolatori con un **singolo processore**, che di conseguenza elaborano un solo flusso di dati in modo sequenziale con un singolo flusso di istruzioni;
- **MISD (Multiple Instruction Single Data)**, nella quale rientrano i calcolatori in cui viene implementato il **parallelismo temporale**, che permette di eseguire più fasi di un'istruzione contemporaneamente su un unico flusso di dati;
- **SIMD (Single Instruction Multiple Data)**, nella quale rientrano i calcolatori in cui viene implementato il **parallelismo spaziale**, che **permette di eseguire la stessa istruzione su più flussi di dati mediante una CU che controlla diverse ALU**;
- **MIMD (Multiple Instruction Multiple Data)**, nella quale rientrano i calcolatori che implementano il **parallelismo asincrono**, che permette di eseguire istruzioni diverse su flussi di dati diversi grazie alle varie CPU che operano in parallelo.

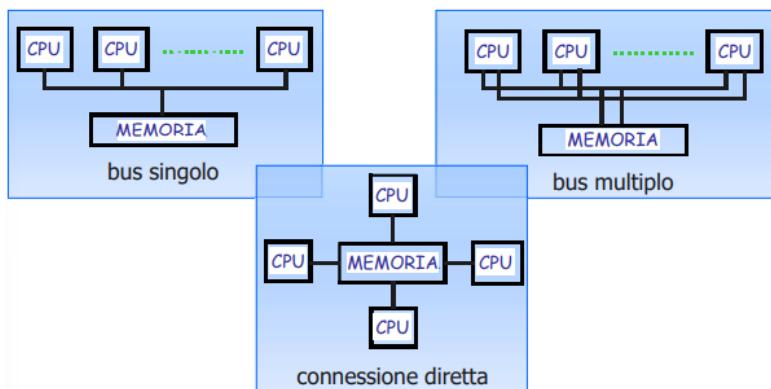
Classificazione calcolatori MIMD

I calcolatori MIMD possono essere ulteriormente suddivisi in quelli a **memoria condivisa (shared-memory, SM)**, se si ha un'architettura in cui **diverse CPU condividono un'unica memoria globale**, e quelli a memoria distribuita

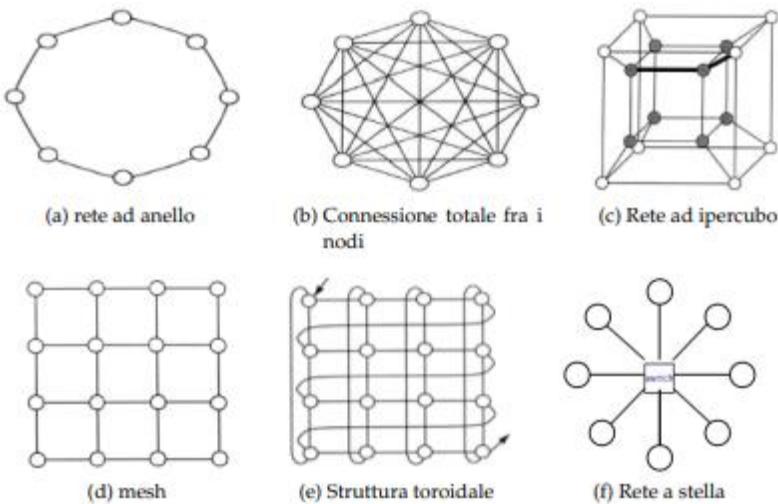
(distributed-memory, DM), se si ha un'architettura in cui **ogni CPU ha una propria memoria locale**.



Nei primi, il **collegamento CPU-memoria** può avvenire mediante **bus singolo**, **bus multipli** o in **modo diretto**.



Nei sistemi a memoria distribuita, invece, vengono utilizzati anche **schemi di collegamento più complessi**, come possono essere



I calcolatori MIMD **distributed-memory** vengono ulteriormente classificati in **Massive Parallel Processors (MPP)**, se si ha un'architettura con **molte CPU che si trovano su calcolatori indipendenti ma che lavorano in parallelo**, oppure **Distributed Shared Memory (DSM)**, se si ha un'architettura con **CPU che si trovano su calcolatori indipendenti che lavorano in parallelo**, in cui **uno di essi** (calcolatori) possiede una **memoria virtuale condivisa** tra tutti i processori.

I calcolatori MIMD **shared-memory**, invece, vengono ulteriormente classificati in **Symmetric MultiProcessor (SMP)**, se si ha un'architettura in cui i **processori**

possiedono lo stesso ruolo e sono connessi ad una singola memoria principale condivisa in un'unica macchina, oppure **ASymmetric MultiProcessor (ASMP)**, se si ha un'architettura con **CPU** aventi ruoli diversi in un'unica macchina.

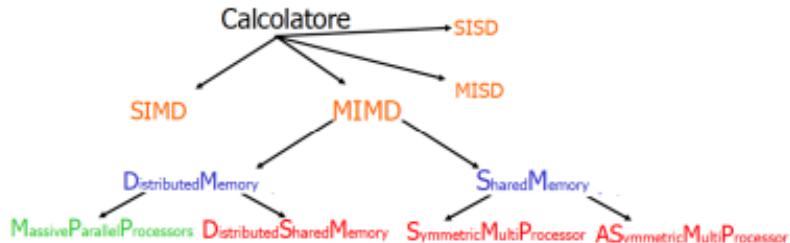
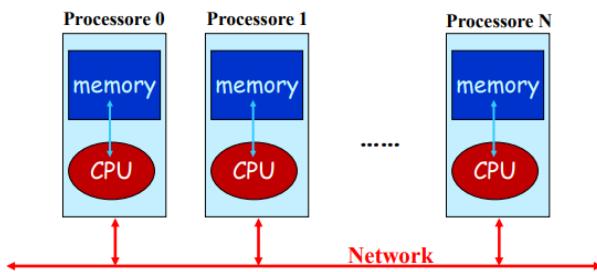


Figura 1.9: Semplificazione della tassonomia di Flynn completa.

Nell'architettura **MIMD distributed-memory**, come abbiamo visto, abbiamo diversi calcolatori autonomi collegati tra loro attraverso reti di connessione. Questi hanno la **necessità di organizzare le comunicazioni tra loro** in quanto ogni processore ha una propria memoria locale alla quale accede direttamente, ma sia per conoscere i dati in memoria di un altro processore, che per far conoscere i propri, c'è bisogno di un trasferimento di dati.



Calcolo parallelo vs calcolo distribuito

Sia negli ambienti di calcolo parallelo, che in quelli di calcolo distribuito, le unità processanti comunicano tra loro per **risolvere problemi su larga scala in maniera efficiente**. Nel calcolo parallelo le unità processanti sono strettamente collegate, mentre nel calcolo distribuito risultano autonome, indipendenti e fisicamente distribuite.

Un **calcolatore parallelo** è caratterizzato da un sistema di nodi omogenei collegati da switch specializzati e dedicati, chiamato **tightly coupled system**, ed il suo obiettivo principale è raggiungere la **miglior performance possibile**, riducendo il tempo necessario alla risoluzione computazionale di un problema reale.

Un **ambiente di calcolo distribuito**, invece, è caratterizzato da un sistema di nodi collegati da reti geografiche, chiamato **loosely coupled system**, ed il suo obiettivo principale è il **riutilizzo di risorse già esistenti** che risultano essere **eterogenee**.

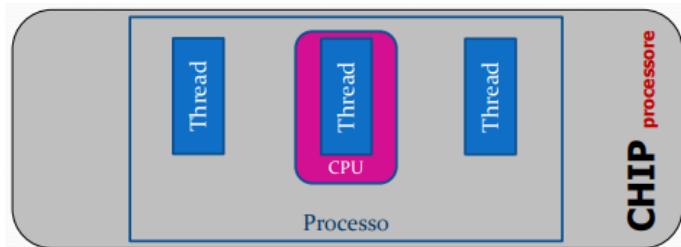
Introduzione alla libreria OpenMP (Lez.3)

Come abbiamo visto, per i calcolatori **MIMD distributed-memory** (DM) si ha la **necessità di organizzare le comunicazioni tra i processi**, mentre invece nei calcolatori **MIMD shared-memory** (SM) si ha la **necessità di sincronizzare gli accessi in scrittura e organizzare il lavoro**.

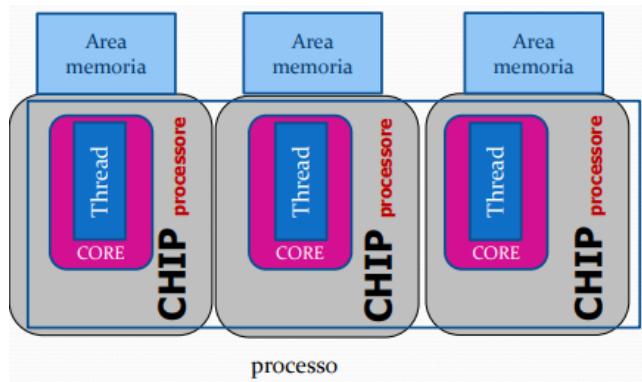
Threads e processi

Per un sistema operativo moderno, l'unità di base di utilizzo della CPU è il thread, il quale **rappresenta un flusso di istruzioni indipendente che deve essere eseguito sequenzialmente su una CPU**.

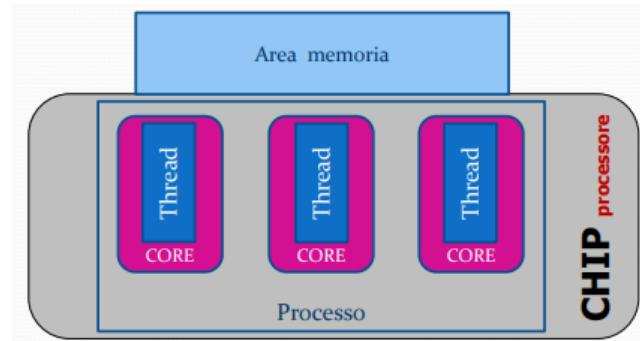
Ogni **processo**, che viene definito come “**programma in esecuzione**”, è **costituito da almeno un thread**, ma può contenerne più di uno (come nella figura seguente).



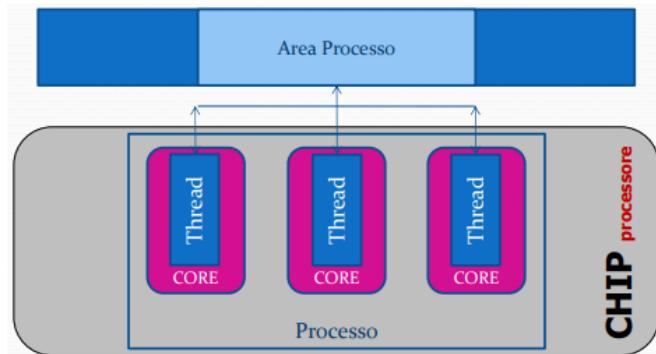
In generale **thread diversi possono essere eseguiti indipendentemente su CPU diverse nei calcolatori MIMD-DM**



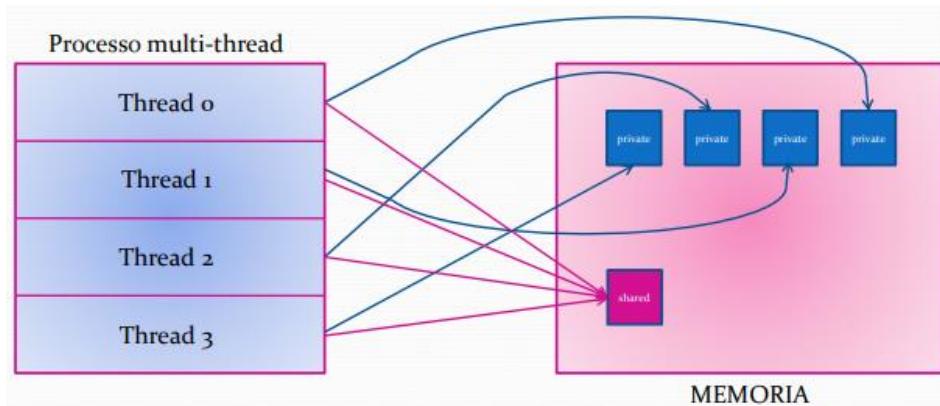
mentre **possono essere eseguiti indipendentemente su core diversi (multicore) nei calcolatori MIMD-SM.**



Architettura multicore



Nelle architetture multicore i **thread di uno stesso processo condividono la stessa area di memoria**, lavorando insieme in maniera naturale. Questo porta ad un **vantaggio** in termini di leggerezza ed efficienza, ma uno **svantaggio** per quanto riguarda la consistenza dei dati. Per questo motivo i thread vengono coordinati attraverso la **sincronizzazione degli accessi alle variabili condivise**.



Libreria OpenMP

Esistono diversi strumenti per lo sviluppo di software in ambiente di calcolo MIMD-SM, ma quello che vedremo noi è la libreria OpenMP che può essere utilizzata in C/C++, sia su Linux che su Windows.

Il modello di esecuzione parallela utilizzato è il **fork-join**, nel quale tutti i processi cominciano con un solo thread, detto **master thread**, che esegue le operazioni in

maniera sequenziale. Una volta raggiunta la **fork** comincia una **regione parallela**, ossia viene creato un team di thread che procede parallelamente. La **join**, invece, viene raggiunta quando tutti i thread del team hanno terminato le istruzioni della regione parallela e si sincronizzano, lasciando proseguire solo il master thread.

La libreria OpenMP è composta da:

- **direttive per il compilatore**, che si utilizzano per creare un team e stabilire quali istruzioni devono essere eseguite in parallelo, oltre a definire la distribuzione delle stesse tra i vari thread del team. Le istruzioni sono rappresentate in un formato speciale, comprensibile solo ad un compilatore dotato di supporto OpenMP, e possono essere innestate per ottenere un **parallelismo multilivello**. Vengono inoltre utilizzate delle **clausole**, per completare le direttive, in modo da dettagliarne il **comportamento**;
- **runtime library-routines**, che sono appunto **routines utilizzate per intervenire sulle variabili di controllo interne a run-time**, come possono essere numero di thread, informazioni sullo scheduling, ecc. (deve essere incluso omp.h);
- **variabili d'ambiente**, che vengono utilizzate per modificare il valore delle variabili di controllo interne prima dell'esecuzione.

Es. di struttura generica del codice:

```
#include <omp.h>
main ()
{
    int var1, var2, var3;
    ...
    Parte sequenziale
    ...
    Inizio della regione parallela:
        si genera un team di thread e si specifica lo scopo delle variabili
        #pragma omp parallel private(var1, var 2) shared(var3)
        {
            Sezione parallela eseguita da tutti i thread
            ...
            Tutti i thread confluiscono nel master thread
        }
    Ripresa del codice sequenziale
    ...
}
```

Direttive

La direttiva principale è la **#pragma omp parallel**, la quale **forma un team di thread avviando un'esecuzione parallela**.

Come abbiamo detto, le direttive vengono accompagnate da clausole che ne specificano il comportamento e alla fine del blocco di istruzioni è sottintesa una barriera di sincronizzazione dove tutti i thread si fermano ad aspettare che gli altri completino la loro esecuzione, in modo da ritornare ad un'esecuzione sequenziale.

Clausole

```
#pragma omp parallel [clause], [clause] ...
{
}
clause:
num_threads(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
reduction(operator: list)
```

Non tutte le clausole sono valide per tutti i costrutti, ma in generale le più diffuse sono:

- **num_threads(int)**, che permette di specificare appunto il **numero di thread** del team;
- **default(shared | none)**, che permette di specificare se di default **tutte le variabili saranno considerate condivise (shared) o se deciderà tutto il programmatore (none);**
- **private(list)**, che permette di specificare una **lista di argomenti privati per ogni thread** che li utilizza;
- **firstprivate(list)**, che permette di specificare una **lista di argomenti privati per ogni thread, il cui valore iniziale sarà quello che avevano originariamente** nel momento in cui è stato incontrato il costrutto in questione. Ciò significa che le copie private vengono pre-inizializzate con il valore che ha la variabile con lo stesso nome prima di incontrare la regione parallela;
- **lastprivate(list)**, che permette di specificare una **lista di argomenti privati per ogni thread, i cui valori originali verranno aggiornati al termine della regione parallela.** Ciò significa che il valore delle variabili con lo stesso nome verrà aggiornato al termine della regione parallela.
- **shared(list)**, che permette di specificare una **lista di argomenti condivisi tra i thread** del team;
- **reduction(operator:list)**, che permette di **combinare una lista di argomenti utilizzando l'operatore associativo specificato.** Ciò significa che ogni thread avrà una copia privata delle variabili specificate come argomenti e **al termine del costrutto si avrà un'unica variabile condivisa.**

Bisogna fare attenzione per quanto riguarda l'utilizzo di firstprivate e lastprivate, in quanto il primo si usa con il costrutto parallel mentre il secondo con il costrutto **for**.

Runtime library routines

Esistono diverse routines per intervenire sulle variabili di controllo interne a runtime:

- **omp_set_num_threads(int)**, per definire il numero di thread da utilizzare;
- **omp_get_max_threads()**, che restituisce il numero massimo di thread disponibili per la prossima regione parallela;
- **omp_set_dynamic(int)**, che specifica se è permesso (1) o meno (0) al sistema di riadattare il numero di thread utilizzati;
- **omp_get_thread_num()**, che restituisce l'id del thread;
- **omp_get_num_procs()**, che restituisce il numero di core disponibili per il programma al momento della chiamata;
- **omp_get_num_threads()**, che restituisce il numero di thread del team.

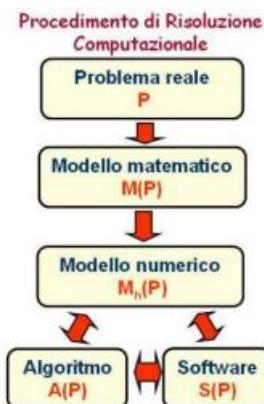
Variabili d'ambiente

Esistono diverse variabili d'ambiente utilizzate per modificare il valore delle variabili di controllo interne prima dell'esecuzione:

- **OMP_NUM_THREADS**, che permette di definire il numero di thread che verranno utilizzati nell'esecuzione successiva
(`export OMP_NUM_THREADS=integer`);
- **OMP_DYNAMIC**, che specifica il permesso (1) o meno (0) del sistema di riadattare il numero di thread utilizzati.

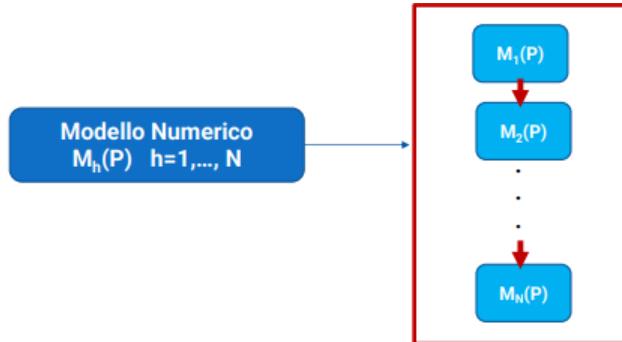
Parallelismo in ambiente multicore (Lez.4)

In generale, nella modellizzazione di problemi su larga scala, viene utilizzato un **procedimento di risoluzione computazionale** che consta di diversi passi.



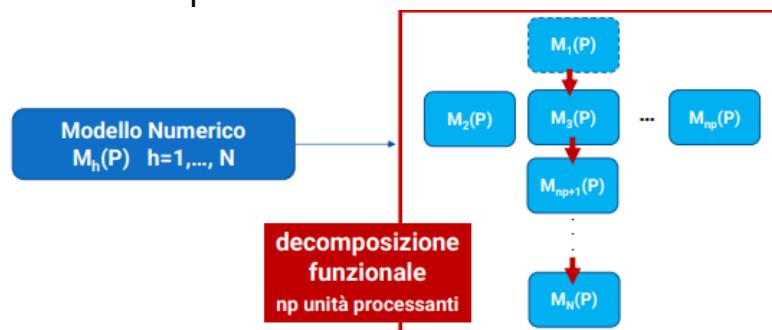
Quando si progetta un algoritmo sequenziale, oltre a concentrarsi sul problema, è importante tenere conto dell'esecutore. **Lo sviluppo del software deve essere infatti ricondotto ad un insieme di passi elementari**, in base allo specifico ambiente di calcolo.

In ambienti multicore risulta utile ripartire dal **modello numerico**, che equivale al **modello matematico suddiviso in N passi**, per poi applicare **tecniche di decomposizione** diverse.



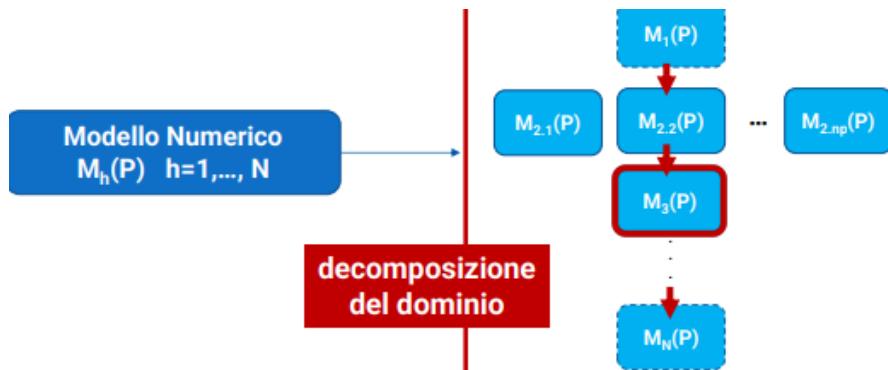
Esistono infatti:

- la **decomposizione funzionale**, nella quale si analizzano i passi per individuare **task indipendenti** che possano essere processati separatamente e contemporaneamente.



Questa tecnica ha come caratteristica fondamentale quella di essere **applicabile ad un modello caratterizzato da più nuclei computazionali**.

- la **decomposizione del dominio**, nella quale l'idea è quella di **suddividere ogni task in sotto-task uguali**, processandoli contemporaneamente, ma riducendo al minimo la collezione dei risultati locali.



La decomposizione del dominio risulta **vantaggiosa solo quando vi è una semplice collezione dei risultati locali oppure quando non vi è proprio alcuna collezione**.

- una **combinazione delle due tecniche precedenti**.

Somma di due vettori di dimensione N

Supponiamo di dover sommare due vettori di dimensione N, il cui output ovviamente sarà un unico vettore dato dalla somma delle componenti di posizione corrispondente.

Input: $a = (a_0, a_1, a_2, \dots, a_{N-1})$, $b = (b_0, b_1, b_2, \dots, b_{N-1})$

Output: $c = (a_0 + b_0, a_1 + b_1, a_2 + b_2, \dots, a_{N-1} + b_{N-1})$

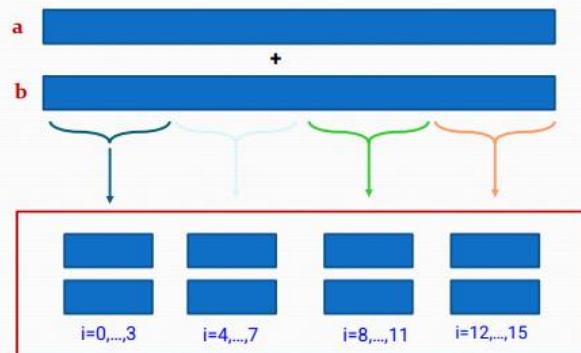
Su un **calcolatore monoprocesso** la somma verrebbe calcolata eseguendo **le N addizioni**, una per volta, seguendo un ordine prestabilito.

```
begin
    for i=0 to N-1 do
        ci := ai + bi;
    endfor
end
```

Se però ho a disposizione **np unità processanti** posso rifarmi all'idea di base del calcolo parallelo, ossia decomporre un problema di dimensione N in np sottoproblemi di dimensione N/np, risolvendo questi ultimi contemporaneamente e utilizzando le np unità processanti.

Applicando questa idea al nostro problema possiamo pensare di **suddividere i due vettori, assegnando l'operazione di somma di sottovettori a diverse CPU**.

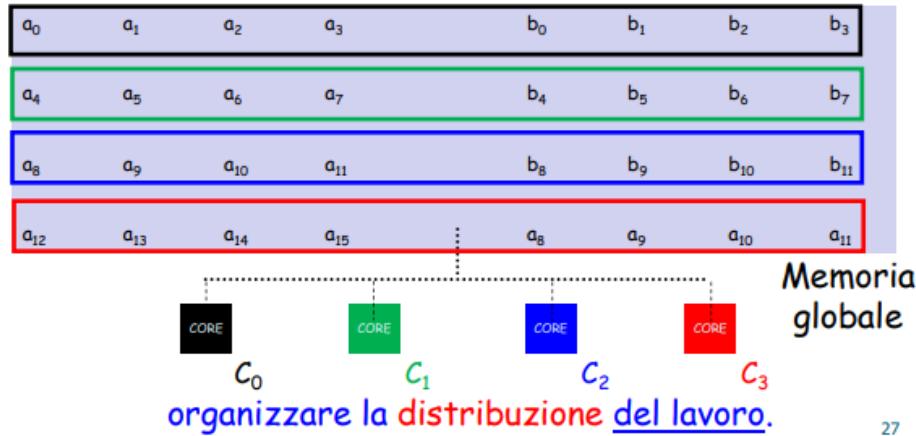
Esempio: N=16, np=4



In questo modo il **lavoro sui due vettori viene distribuito tra i vari core, ogni core si occupa di sommare due sottovettori e i sottovettori sommati possono essere uniti nella memoria condivisa**. Tutto ciò può avvenire senza ulteriori operazioni di collezione dei risultati ed è per questo che si tratta di un **problema full parallel**, ossia completamente parallelizzabile.

I core possono accedere simultaneamente alla memoria globale su dati differenti

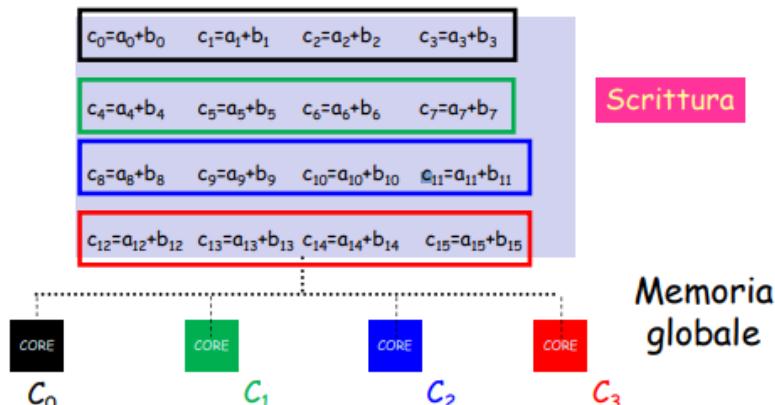
Esempio: N=16, p=4



27

Esempio: N=16, p=4

Calcolo somme parziali



Somma degli elementi di un vettore di dimensione N (somma di N numeri)

Supponiamo di dover sommare tra loro gli elementi di un vettore di dimensione N, il cui output corrispondente sarà un unico valore.

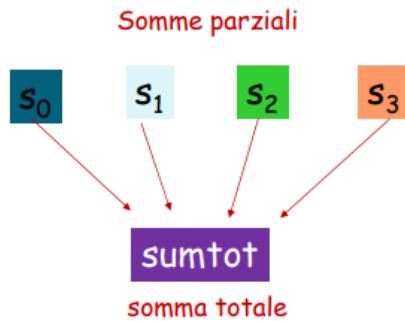
Su un **calcolatore monoprocessoresso** la somma verrebbe calcolata **eseguendo N-1 addizioni**, una per volta, secondo un ordine prestabilito.

```

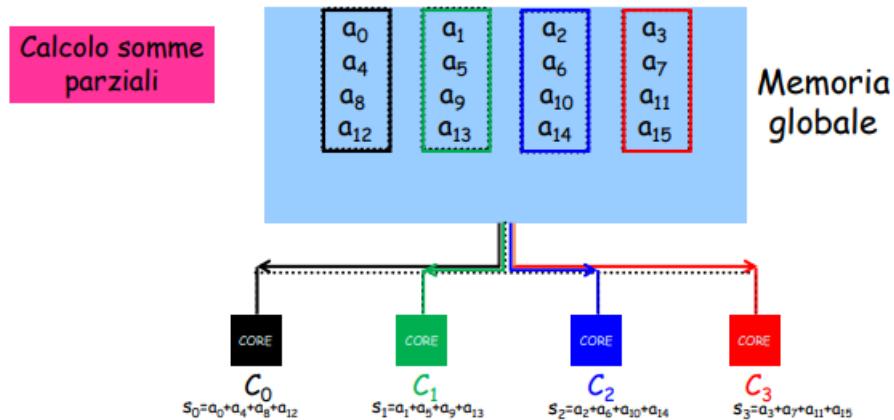
begin
    sumtot := a0;
    for i=1 to N-1 do
        sumtot := sumtot+ai ;
    endfor
end

```

Se però avessi a disposizione **np unità processanti** potrei utilizzare la solita idea alla base del calcolo parallelo per **suddividere la somma in somme parziali ed assegnare ciascuna di esse** (somme parziali) ad una CPU. Le somme parziali dovranno poi essere combinate in modo opportuno per ottenere la **somma totale**.

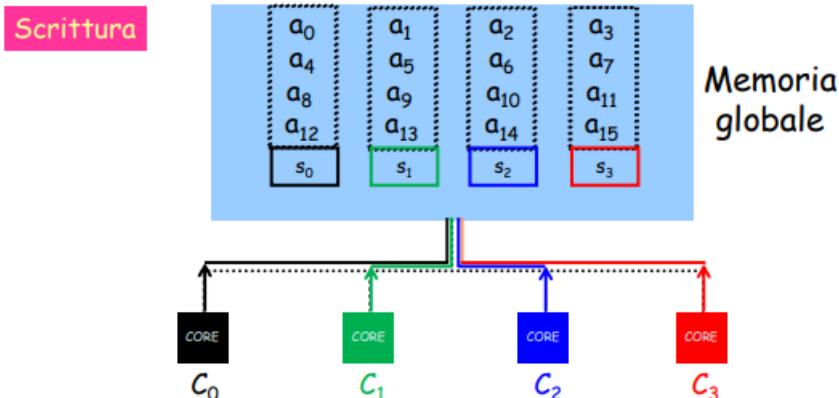


- Esempio: N=16, p=4



- Esempio: N=16, p=4

I core possono accedere simultaneamente alla memoria globale su dati differenti



Per calcolare la somma totale, ogni core, calcolata la propria somma parziale, potrebbe addizionare tale valore ad una variabile che conterrà la somma finale. I core dovranno poi accedere a tale variabile in modo esclusivo, così che non si possano creare inconsistenze nei dati. Per fare ciò è necessaria una corretta **sincronizzazione negli accessi alla variabile condivisa**.

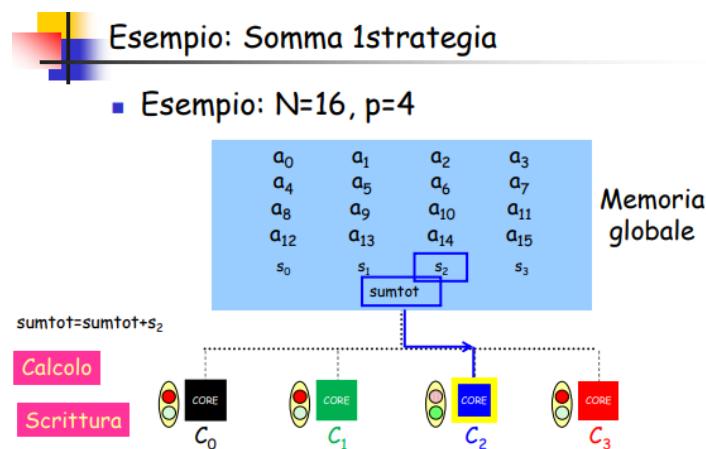
Come si può immaginare, il problema della somma degli elementi di un vettore di dimensione N **non è di tipo full parallel**, a differenza del problema della somma

dei due vettori di dimensione N, poichè richiede delle operazioni di **collezione dei risultati locali**.

Esistono **due strategie** per quanto riguarda la risoluzione di questa problematica in ambito multicore che vedremo nei seguenti paragrafi.

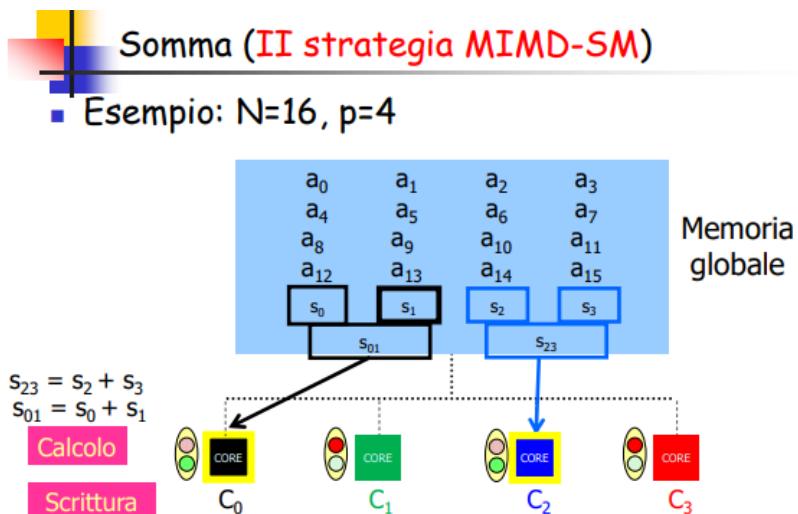
I strategia (prima strategia)

Nella prima strategia di risoluzione **ogni core calcola la propria somma parziale**, dopodichè ad **ogni passo aggiunge la stessa** (somma parziale) ad **un unico valore prestabilito**; in pratica come visto in precedenza. Così facendo il **valore finale si troverà nell'unica memoria condivisa**.



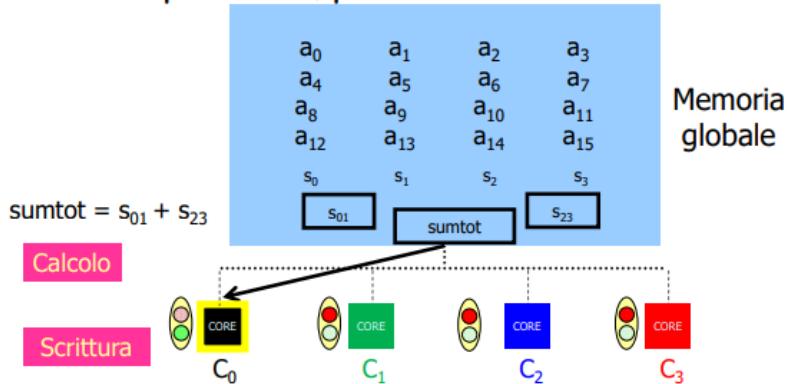
II strategia (seconda strategia)

Anche nella seconda strategia **ogni core calcolerà**, come prima cosa, **la propria somma parziale**. Ad **ogni passo** però, a differenza della prima strategia, **la metà dei core rispetto al passo precedente calcolerà la somma parziale tra la sua somma parziale e quella di un altro, o altri, core**. Il **valore finale**, invece, si troverà comunque **nell'unica memoria condivisa**.



Somma (II strategia MIMD-SM)

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



Valutazione di un algoritmo parallelo (Lez.5)

Per decidere se vale la pena implementare un algoritmo parallelo per la risoluzione di un determinato problema bisogna valutarne l'efficienza.

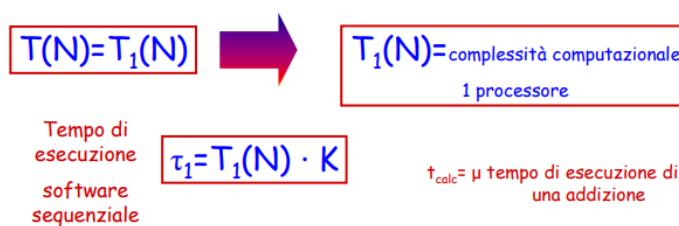
L'**efficienza** di un **algoritmo sequenziale** viene misurata tramite la sua **complessità computazionale $T(N)$** , ossia il numero di operazioni eseguite dall'algoritmo, e la sua **complessità spaziale $S(N)$** , cioè il numero di variabili utilizzate dall'algoritmo.

La complessità computazionale in ambito sequenziale andrebbe in realtà indicata come $T_1(N)$, visto che il numero di processori utilizzati è 1.

Di seguito un recap di come veniva calcolato il tempo di esecuzione di un algoritmo.

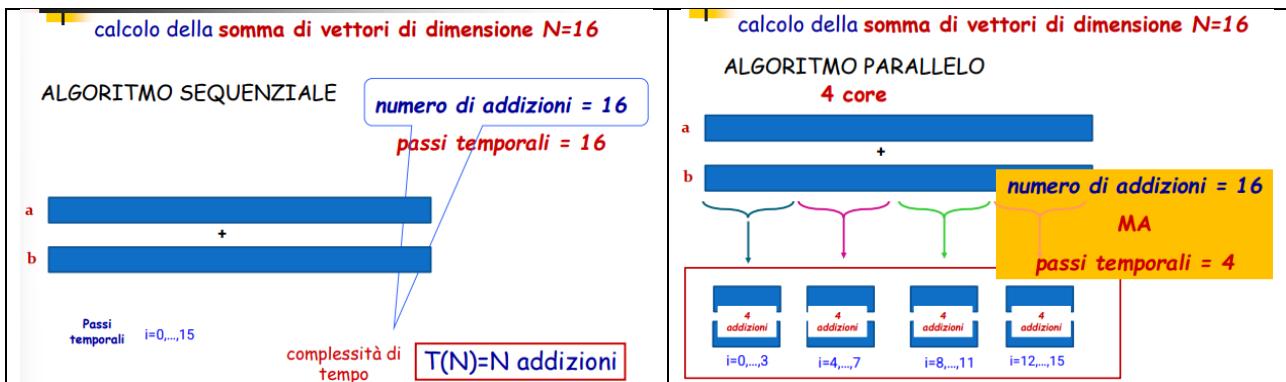
$$\tau = k \cdot T(N) \cdot \mu$$

μ = tempo di esecuzione di 1 op. f.p.



Negli algoritmi sequenziali il numero complessivo di operazioni è uguale al numero dei passi temporali, a differenza degli algoritmi paralleli.

Questo perchè un calcolatore parallelo è in grado di eseguire più operazioni concorrentemente, ossia nello stesso passo temporale. Ne consegue che il tempo di esecuzione non dipende solo dal numero di operazioni floating point effettuate e quindi la complessità di tempo non è adatta a misurare l'efficienza di un algoritmo parallelo.



In generale con p core ci aspettiamo che T_1 sia p volte T_p , ossia **ci aspettiamo di ridurre di p volte il tempo di esecuzione**. In effetti **ciò avviene** per quanto riguarda il problema della somma di due vettori di dimensione N , in quanto **algoritmo full parallel**, ma **non avviene nel caso**, ad esempio, del problema della somma degli elementi di un vettore di dimensione N , poichè **non è full parallel**.

Il tempo di esecuzione di un algoritmo parallelo diventerà

$$\tau = k \cdot T_p(N) \cdot \mu$$

μ = tempo di esecuzione di 1 op. f.p.

$$T_p(N) \rightarrow T_p(N) = \frac{\text{complessità computazionale}}{p \text{ processori}}$$

Tempo di esecuzione software parallelo $t_p = T_p(N) \cdot K$

$t_{\text{calc}} = \mu$ tempo di esecuzione di una addizione

21

considerando p core su cui eseguirlo.

Quindi ricapitolando, per quanto riguarda il **problema della somma di due vettori di dimensione N** , avremo che **all'aumentare del numero di core diminuirà il tempo di esecuzione**, con la complessità computazionale che sarà data dalla dimensione N dei due vettori fratto il numero di core p , moltiplicato per il tempo di esecuzione di un'operazione floating point (T_{calc} o μ).

ALGORITMO PARALLELO della somma di due vettori di dimensione N - posto p processori

$$\begin{aligned} p=1 & \quad T_1=16 t_{\text{calc}} \\ p=2 & \quad T_2=8 t_{\text{calc}} \\ p=4 & \quad T_4=4 t_{\text{calc}} \\ p=8 & \quad T_8=2 t_{\text{calc}} \\ \dots & \end{aligned}$$

$\left. \right\} N = 16$

$$T_p(N) = (N/p) t_{\text{calc}}$$

p	T_p	T_1/T_p
1	$16 t_{\text{calc}}$	1.00
2	$8 t_{\text{calc}}$	2
4	$4 t_{\text{calc}}$	4
8	$2 t_{\text{calc}}$	8

Maggiore riduzione del tempo ovvero maggiore aumento della velocità

L'algoritmo su 8 processori è il più veloce
E' più veloce di 8 volte di quello su 1 processore

Per calcolare qual è l'algoritmo che impiega meno tempo e quanto sono più veloci di quello sequenziale, viene definito il cosiddetto **speed-up** (che nell'immagine soprastante viene calcolato per il problema della somma di due vettori di dimensione N), che equivale al rapporto tra la complessità computazionale dell'algoritmo sequenziale e quella dell'algoritmo utilizzando p core.

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

In pratica lo speed-up misura la riduzione del tempo di esecuzione rispetto all'algoritmo con 1 core.

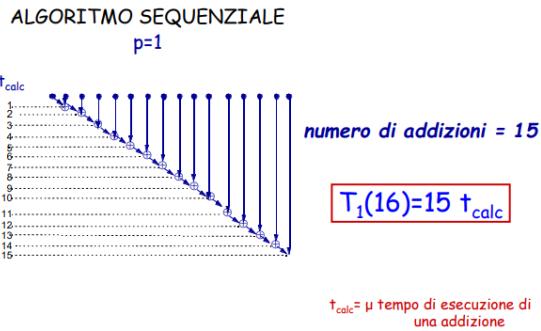
Lo speed-up ideale è uguale a p e si verifica nel caso degli algoritmi full parallel, come il problema della somma di due vettori di dimensione N. Ciò significa che se con un algoritmo parallelo si riduce il tempo di esecuzione dell'algoritmo sequenziale (con 1 core) di p, allora questo algoritmo parallelo avrà speed-up ideale.

SPEEDUP IDEALE
 $S_p^{\text{ideale}} = p$

Puttropo però non tutti i problemi, come sappiamo, sono completamente parallelizzabili e, di conseguenza, il loro speed-up sarà strettamente minore di p.

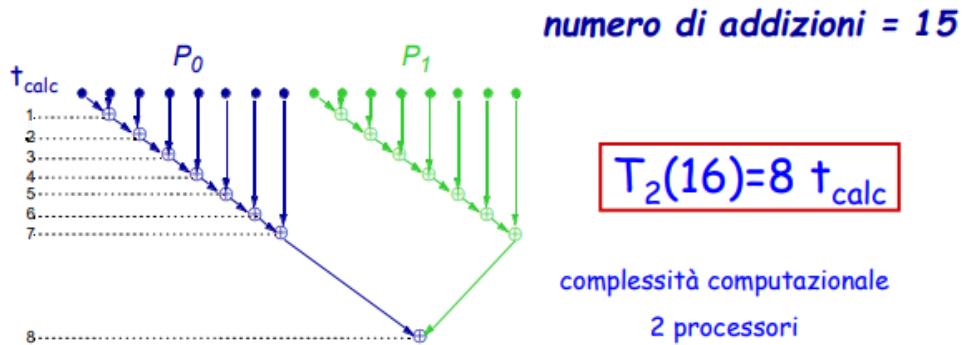
$$S_p < p$$

Consideriamo quindi il problema della somma degli elementi di un vettore di dimensione N, dove la complessità computazionale dell'algoritmo sequenziale è $T_1(N) = N-1$.

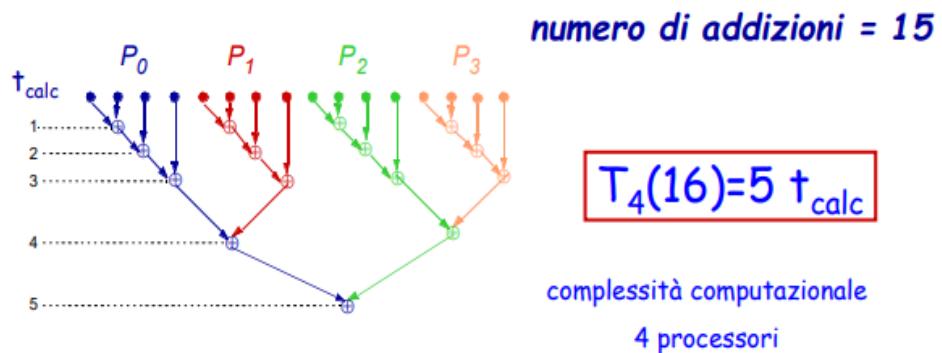


In questo caso (in ambito parallelo), oltre alle addizioni che i vari core dovranno fare, abbiamo anche uno o più passi, in base al numero di core, legati alla **collezione dei risultati locali**, ossia delle somme parziali.

ALGORITMO PARALLELO p=2



ALGORITMO PARALLELO II strategia p=4



Una volta ricavata la complessità computazionale dell'algoritmo su p core

ALGORITMO PARALLELO della somma di N numeri
II strategia - posto p processori

$$\begin{aligned}
 p=1 & \quad T_1 = 15 t_{\text{calc}} \\
 p=2 & \quad T_2 = 8 t_{\text{calc}} = (7+1) t_{\text{calc}} \\
 p=4 & \quad T_4 = 5 t_{\text{calc}} = (3+2) t_{\text{calc}} \\
 p=8 & \quad T_8 = 4 t_{\text{calc}} = (1+3) t_{\text{calc}} \\
 \dots & \quad \dots \\
 T_p(N) &= (N/p-1 + \log_2 p) t_{\text{calc}}
 \end{aligned}
 \quad \left. \begin{array}{l} \\ \\ \\ \\ \end{array} \right\} N = 16$$

si può procedere al **calcolo dello speed-up** in modo da valutare il guadagno in termini di riduzione del tempo di esecuzione rispetto all'algoritmo su 1 processore.

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

Maggiore
riduzione del
tempo ovvero
maggiore
aumento della
velocità

L'algoritmo su 8 processori è il più veloce
E' più veloce di 3.75 volte di quello su 1 processore

Come ricavabile dalla tabella sopra, l'**algoritmo più veloce risulterebbe essere quello su 8 processori**. Introduciamo però il concetto di **overhead totale**, ossia la **misura di quanto lo speed-up differisca da quello ideale**.

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

Se applichiamo la formula per calcolare l'overhead totale nell'algoritmo parallelo per il calcolo della somma degli elementi di un vettore di dimensione N avremo che, **al crescere del numero di processori p , l'overhead aumenta**.

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

$$T_p = (N/p - 1 + \log_2 p) t_{\text{calc}}$$



$$O_h = p T_p - T_1 = [p (N/p - 1 + \log_2 p) - (N-1)] =$$

$$= (\cancel{N} - p + p \log_2 p - \cancel{N} + 1) = O_h(p \log_2 p)$$

p	O_h
2	$2 t_{\text{calc}}$
4	$8 t_{\text{calc}}$
8	$24 t_{\text{calc}}$
2^k	$p \log_2 p t_{\text{calc}}$

Al crescere di p
l'overhead aumenta!

43

Di conseguenza, è vero che l'algoritmo su 8 processori è il più veloce secondo lo speed-up, ma è altrettanto vero che all'aumentare del numero p di core ci si allontana dallo speed-up ideale. Come possiamo notare dalla seguente tabella, infatti, **lo speed-up più vicino a quello ideale si ha con 2 processori**.

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

Da ciò possiamo ricavare che **lo speed-up non basta a fornire informazioni sull'efficienza dell'algoritmo parallelo** in quanto l'utilizzo di un maggior numero di core non è sempre garanzia di sviluppo di algoritmi paralleli efficaci, ossia di algoritmi che sfruttano tutte le risorse nell'ambito di un sistema parallelo.

Per misurare quanto è stato sfruttato il calcolatore parallelo si definisce il concetto di **efficienza**, dato dal rapporto tra speed-up e numero di core p utilizzati.

$$E_p = \frac{S_p}{p}$$

L'**efficienza ideale** si avrà quando il rapporto tra speed-up e numero di core sarà uguale ad 1, ossia quando si avrà lo speed-up ideale (cioè quando si parla di algoritmi full parallel).

EFFICIENZA IDEALE

$$E_p^{\text{ideale}} = \frac{S_p^{\text{ideale}}}{p} = 1$$

Legge di Ware-Amdahl (Lez.7) la lez.6 era di laboratorio

Come abbiamo detto, lo speed-up misura la riduzione del tempo di esecuzione tra un algoritmo sequenziale ed il suo algoritmo parallelo.

Per scegliere, però, l'algoritmo per misurare T_1 , ossia la complessità computazionale dell'algoritmo sequenziale, si possono effettuare due scelte:

- **la prima** consiste nel vedere T_1 come $T_{p=1}$, ossia come **complessità computazionale dell'algoritmo parallelo su 1 processore**. In questo modo lo speed-up darebbe informazioni su quanto l'algoritmo si presta all'implementazione su un'architettura parallela, ma al contempo l'algoritmo parallelo su 1 processore potrebbe eseguire più operazioni del necessario;
- **la seconda** invece consiste nel considerare T_1 come la **complessità computazionale del miglior algoritmo sequenziale**. In questo caso lo speed-up darebbe informazioni sulla riduzione effettiva del tempo nella risoluzione di un problema con p processori, ma al contempo richiederebbe l'individuazione del miglior algoritmo sequenziale e la disponibilità di software che implementino tale algoritmo.

Per convenzione è stata scelta la prima possibilità.

Speed-up prossimi a quelli ideali negli algoritmi non full parallel

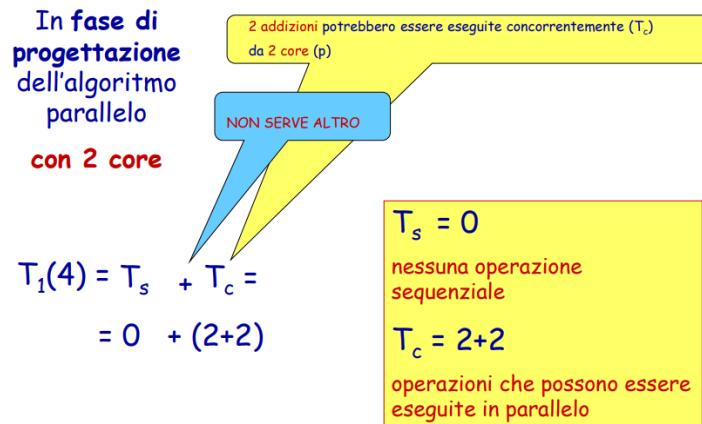
Come sappiamo, negli algoritmi full parallel è sempre possibile ottenere speed-up ideale. Negli algoritmi che richiedono collezione dei risultati locali, però, il discorso è un po' più complicato.

Per analizzare questa situazione ricordiamo che il punto di partenza per scrivere un buon algoritmo parallelo è aver scritto un ottimo algoritmo sequenziale.

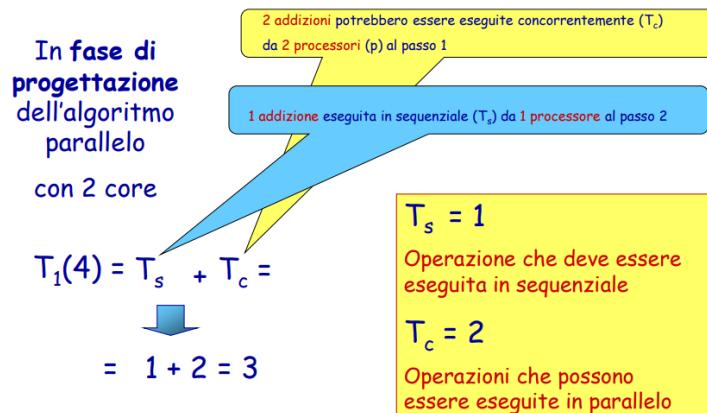
La complessità computazionale T_1 si può decomporre in 2 parti, una relativa alle operazioni che devono essere eseguite esclusivamente in sequenziale T_s e una relativa alle operazioni che potrebbero essere eseguite concorrentemente T_c .

$$T_1 = T_s + T_c$$

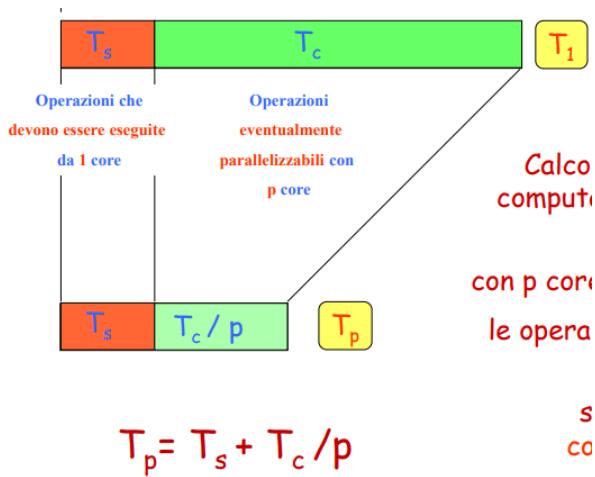
Rifacendoci al problema della somma di due vettori di dimensione N, consideriamo il caso in cui N = 4 e p = 2 (numero di core). In questo caso avremo che non c'è alcuna operazione da svolgere in sequenziale, in quanto è possibile svolgere le somme dividendole tra i p core, che in questo caso sono 2.



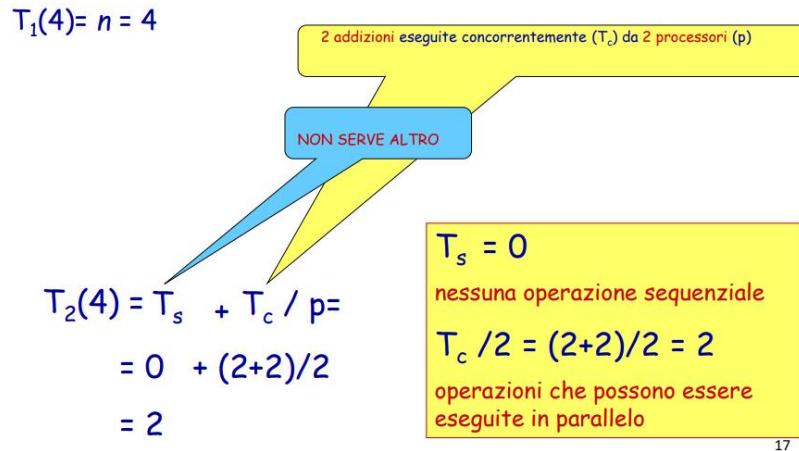
Se prendiamo invece come esempio il problema della somma di N numeri, sempre con $N = 4$ e $p = 2$, avremo un'operazione di collezione dei risultati locali da svolgere obbligatoriamente in sequenziale.



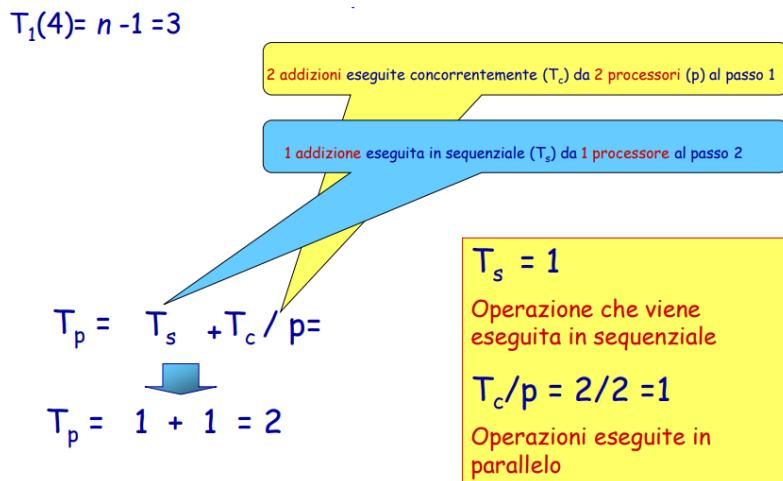
La complessità computazionale di un algoritmo parallelo con p core, sarà quindi uguale alla complessità computazionale delle operazioni da svolgere obbligatoriamente in sequenziale, più la complessità computazionale delle operazioni che possono essere eseguite in parallelo, suddivise tra i p core.



Riprendendo gli esempi precedenti, avremo che per la somma di due vettori di dimensione N , con $N = 4$ e $p = 2$, la complessità computazionale sarà



mentre avremo che per la somma di N numeri, con $N = 4$ e $p = 2$, la complessità computazionale sarà



Legge di Ware-Amdahl

Considerando α come **tutte le azioni svolte in sequenziale**, avremo che la complessità computazionale delle operazioni svolte in sequenziale $T_s = \alpha$ e che la complessità computazionale delle operazioni svolte in parallelo $T_c = 1 - \alpha$. Possiamo quindi procedere al **ricalcolo dello speed-up** sostituendo alla complessità computazionale dell'algoritmo con p core la formula appena ottenuta.

$$T_s = \alpha, T_c = 1 - \alpha$$

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

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

Legge di Ware (Amdahl)

Lo speed-up dei problemi della somma di due vettori di dimensione N e della somma di N numeri, tramite la legge di Ware-Amdhal, sarà dato da

Somma di due vettori di dimensione N

$$\alpha=0, 1-\alpha=1, (1-\alpha)/p$$

$$S_p = \frac{1}{1/p} = p \quad \xrightarrow[p \rightarrow \infty]{} \infty$$

Somma di N numeri

$$S_p = \frac{1}{\alpha + (1 - \alpha)/p} \quad \xrightarrow[p \rightarrow \infty]{} \frac{1}{\alpha}$$

Analizziamo ora le conseguenze di questa legge considerando il **problema della somma di N numeri**, con **N fissato** e uguale a 32, ma con **p variabile** e uguale a 2, 4, 8, 16.

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

Per questo algoritmo con dimensione N del problema fissata e al crescere del numero p di core, non solo non si riescono ad ottenere speed-up vicini a quello ideale, ma addirittura le prestazioni peggiorano.

Il caso migliore è quello con p = 2. Valutiamo quindi cosa succede al crescere della dimensione N, che sarà uguale a 8, 16, 32, 64, con p fissato e uguale a 2.

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

Possiamo notare come la parte sequenziale tende a 0 e la parte parallela è costante, così come speed-up ed efficienza, con quest'ultima che tende all'efficienza ideale.

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

0 0

$S_p \rightarrow p$

Con p fissato, quindi, α tende a 0, e di conseguenza per ottenere uno speed-up quanto più vicino possibile a quello ideale, si deve fissare il miglior numero di core e aumentare la dimensione del problema. Ovviamente non è possibile aumentare in maniera indefinita la dimensione N del problema in quanto le risorse hardware sono limitate.

Ricapitolando, quindi, fissato il size N del problema e aumentando il numero p di CPU, esiste un miglior numero di processori p_{max} per risolvere il problema con l'algoritmo in esame, tale che superato questo valore le prestazioni peggiorano.

Fissato, invece, il numero p delle CPU e aumentando il size N del problema, esiste una massima dimensione n_{max} che la macchina può memorizzare/elaborare, tale che superato questo valore si potrebbero avere problemi con la memoria hardware.

Isoefficienza (Lez.8)

Fino ad ora abbiamo visto cosa succede se aumentiamo il **size n** o il **numero p delle CPU**, mantenendo fissato l'altro valore. Ora vediamo invece **cosa succede se aumentiamo entrambi i valori**, considerando il problema della somma di n numeri che è quello con cui più possiamo notare i problemi, rispetto al problema della somma di due vettori di dimensione N , visto che quest'ultimo è full parallel.

Consideriamo quindi il problema della somma di n numeri risolto con la **prima strategia**, con $n/p = 4$, ossia rapporto tra size del problema e numero di core uguale a 4. **Osserviamo** che la frazione di operazioni eseguite in sequenziale tende a 0, che la frazione di operazioni eseguite in parallelo rimane circa costante, come anche l'efficienza, mentre lo speed-up aumenta.

$$\frac{n}{p} = 4 \quad \text{è costante}$$

n	p	α	$(1-\alpha)/p$	$S_p(n)$	$E_p(n)$
8	2	0,14	0,86	1,75	0,875
16	4	0,06	0,8	3	0,75
32	8	0,03	0,77	5,1	0,64
64	16	0,01	0,76	9	0,56

Aumentando sia n che p, con un rapporto n/p = 4 costante, notiamo che le prestazioni dell'algoritmo parallelo non degradano molto velocemente.

Passando al caso generale, però, **bisogna trovare il valore I = N/p affinchè l'efficienza $E_p(n)$ rimanga costante.**

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

Questo valore I non può essere una costante ovviamente, ed infatti **sarà una funzione**, detta **isoefficienza**, che esprime appunto la legge secondo cui si deve scegliere la nuova dimensione n_1 affinchè si possa verificare che l'efficienza resti costante. In questa funzione, n_0 e p_0 sono la coppia iniziale di size e numero di processori, mentre p_1 sarà il nuovo numero di processori.

Per calcolare n_1 dobbiamo imporre che l'efficienza dell'algoritmo con coppia (n_0, p_0) sia uguale a quella dell'algoritmo con (n_1, p_1) .

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

A questo punto possiamo riscrivere l'efficienza come rapporto tra speed-up e numero di processori

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

ma anche lo speed-up in funzione dell'overhead

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

Si procede ora a semplificare p_0 e p_1 , e considerando solo i termini non costanti avremo

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

A questo punto possiamo ricavare l'efficienza dell'algoritmo con size n_1 , che sarà

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

I

In generale, un algoritmo si dice **scalabile** quando l'efficienza rimane costante al crescere sia del numero dei processori che della dimensione del problema, con un rapporto costante pari all'isoefficienza

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

Calcolo isoefficienza nel problema della somma di N numeri (II strategia)

Ricordando che nel problema della somma di N numeri, con la seconda strategia, abbiamo

$$O_h(n, p) = p \log_2 p$$

$$T_1(n) = n - 1$$

applicando la formula dell'isoefficienza, avremo

$$n_1 - 1 = \frac{p_1 \log_2 p_1}{p_0 \log_2 p_0} (n_0 - 1)$$

Per controllare se l'algoritmo è scalabile pongo arbitrariamente le 3 variabili iniziali, senza considerare le costanti, $(n_0, p_0, p_1) = (64, 4, 8)$ e procedo a calcolare n_1 :

$$n_1 = \left(\frac{8 \log_2 8}{4 \log_2 4} \right) * 64 \rightarrow \frac{24}{8} * 64 = 192.$$

Si può poi riapplicare il procedimento anche con un numero di processori maggiori, in modo da continuare a controllare l'efficienza.

		Efficienza			
		1	4	8	16
n \ p		64	192	512	
	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

Ciò che si osserva è che l'efficienza rimane costante al crescere di p e di n con la legge dell'isoefficienza, di conseguenza il problema della somma di N numeri, utilizzando la seconda strategia, **risulta scalabile**.

Calcolo di speed-up, overhead ed efficienza nell'algoritmo parallelo per la somma di due vettori di dimensione N

Per valutare l'algoritmo parallelo per la somma di due vettori di dimensione N bisogna partire dalla complessità computazionale, che era $T_1(N) = N$ per l'algoritmo sequenziale e $T_p(N) = \frac{N}{p}$ nell'algoritmo parallelo con p core.

Possiamo quindi procedere al calcolo di speed-up, overhead ed efficienza

$$S_p = T_1(N)/T_p(N) = \\ = N / (N/p) : p$$

$$Oh = p T_p(N) - T_1(N) = p(N/p) - N = 0$$

$$E_p = S_p / p = p / p = 1$$

che sono quelli soprastanti nel caso in cui il size dei due vettori N sia divisibile per il numero p di core, essendo l'algoritmo full parallel.

Nel caso in cui invece N non fosse esattamente divisibile per p, alcuni core dovranno fare una somma in più, e di conseguenza la complessità computazionale diventa

$$T_p(N) = \\ = [(\text{int}(N/p) + 1)$$

parte intera

in quanto anche i core che non hanno da effettuare la somma aggiuntiva devono attendere quelli che invece devono eseguire l'operazione.

Soltamente le somme aggiuntive vengono ridistribuite ai processori che hanno id minore del resto della divisione N/p , ed ognuno avrà solo un'operazione in più da eseguire.

Calcolo di speed-up, overhead ed efficienza nell'algoritmo parallelo per la somma di N numeri

I strategia (prima strategia)

Per valutare l'algoritmo parallelo per la somma di N numeri, per quanto riguarda la prima strategia, bisogna partire dalla complessità computazionale,

che è $T_1(N) = N - 1$ per l'algoritmo sequenziale e $T_p(N) = \left(\frac{N}{p} - 1\right) + (p - 1)$ nell'algoritmo parallelo con p core, visto che si ha una fase di calcolo delle somme parziali $\left(\frac{N}{p} - 1\right)$ seguita da una fase di collezione dei risultati locali $(p - 1)$.

Possiamo quindi procedere al **calcolo di speed-up, overhead ed efficienza**

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

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

$$Oh = p T_p(N) - T_1(N) = p[(N/p - 1) + (p - 1)] - (N-1)$$

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

che sono quelli soprastanti **nel caso in cui il size del vettore N sia divisibile per il numero di core p**, visto che l'algoritmo non è completamente parallelizzabile.

Nel caso in cui invece N non fosse esattamente divisibile per p, alcuni core dovranno fare una **somma in più**, e di conseguenza la complessità computazionale diventa

$$T_p(N) =$$

parte intera

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

in quanto anche i core che non hanno da effettuare la somma aggiuntiva devono attendere quelli che invece devono eseguire l'operazione.

Solitamente le somme aggiuntive vengono ridistribuite ai processori che hanno id minore del resto della divisione N/p , ed ognuno avrà solo un'operazione in più da eseguire.

Il strategia (seconda strategia)

Per valutare l'algoritmo parallelo per la somma di N numeri, per quanto riguarda la **seconda strategia, bisogna partire dalla complessità computazionale**, che è $T_1(N) = N - 1$ per l'algoritmo sequenziale e $T_p(N) = \left(\frac{N}{p} - 1\right) + \log_2 p$ nell'algoritmo parallelo con p core, visto che si ha una fase di calcolo delle somme parziali $\left(\frac{N}{p} - 1\right)$ seguita da una fase di collezione dei risultati locali $\log_2 p$.

Possiamo quindi procedere al **calcolo di speed-up, overhead ed efficienza**

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

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

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

che sono quelli soprastanti **nel caso in cui il size del vettore N sia divisibile per il numero di core p**, visto che l'algoritmo non è completamente parallelizzabile.

Nel caso in cui invece N non fosse esattamente divisibile per p, alcuni core dovranno fare una somma in più, e di conseguenza la complessità computazionale diventa

$$T_p(N) = \\ = [\text{int}(N/p) + \log(p)],$$

in quanto anche i core che non hanno da effettuare la somma aggiuntiva devono attendere quelli che invece devono eseguire l'operazione.

Solitamente le somme aggiuntive vengono ridistribuite ai processori che hanno id minore del resto della divisione N/p, ed ognuno avrà solo un'operazione in più da eseguire.

Valutazione teorica algoritmi paralleli delle somme (Lez.10) [la lez.9 era di laboratorio](#)

Nella lez.7 abbiamo visto la legge di Ware-Amdahl in cui venivano considerate una frazione di operazioni eseguite in sequenziale α e una frazione di operazioni eseguite in parallelo $(1-\alpha)/p$.

BASE

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

Non è però sempre possibile distinguere nettamente queste due fasi, infatti la legge di Ware-Amdahl da noi vista è quella **base** (o semplificata). Quella **generalizzata**, invece, **prevede anche una frazione di operazioni eseguite con parallelismo medio**, ossia con $k < p$, in cui ci sono **alcuni core che eseguono operazioni in parallelo**, ma non tutti.

GENERALIZZATA

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

α_1 frazione di operazioni eseguite da un solo processore (sequenziale)

α_k frazione di operazioni eseguite con parallelismo medio $k < p$

α_p frazione di operazioni eseguite con parallelismo totale p

Calcolo dello speed-up con legge di Ware-Amdahl

Somma di due vettori di dimensione N

Considerando di avere numero di core $p = 8$ e dimensione dei vettori $N = 32$, nell'algoritmo parallelo per la somma di due vettori di dimensione N abbiamo un'unica fase parallela, in cui ognuno degli 8 core esegue 4 somme.

Visto che **fase sequenziale** (che non c'è proprio) e **fase parallela** sono **totalmente distinte**, possiamo utilizzare la **legge di Ware-Amdahl** base per calcolare lo speed-up.

In questo caso la **frazione di operazioni eseguite in parallelo** è data da $(1-\alpha) = 32/32 = 1$ da dividere poi per il numero di core p , di conseguenza

$$\frac{1 - \alpha}{p} = \frac{1}{p} = \frac{1}{8},$$

ed essendo un **algoritmo full-parallel** non ci sono ulteriori operazioni da eseguire in sequenziale.

Lo speed-up sarà quindi

$$S_p = \frac{1}{\frac{1}{8} + 0} = 8$$

Il rapporto $32/32$ è dato dal numero di operazioni eseguite in parallelo, fratto il numero di operazioni totali da svolgere (le singole somme tra gli N elementi dei due vettori).

Somma degli elementi di un vettore di dimensione N (somma di N numeri)

Considerando di avere numero di core $p = 8$ e dimensione del vettore $N = 32$, nell'algoritmo parallelo per la somma degli elementi di un vettore di dimensione N , abbiamo da effettuare 31 somme.

Innanzitutto dobbiamo considerare qual è il numero di elementi di cui si occuperà ogni core, che è dato da N/p , quindi in questo caso $nloc = 4$. Il numero di somme sarà poi dato da $nloc - 1$, ossia 3, moltiplicato per il numero di core p che è 8.

La frazione di operazioni da eseguire in parallelo sarà quindi data dal rapporto $24/31$ e dal solito numero di core p .

$$\frac{1 - \alpha}{p} = \frac{24}{31} \cdot \frac{1}{8} = 3 \cdot \frac{1}{31}$$

Le fasi successive dipendono dalla strategia adottata:

- **se utilizziamo la I strategia, la seconda fase sarà completamente sequenziale**, quindi saranno eseguite le 7 somme atte ad addizionare le somme parziali calcolate dagli 8 core. Così facendo, ovviamente, la frazione di operazioni eseguite in sequenziale sarà $\alpha = 7/31$ e quindi lo speed-up che ne consegue è

$$\alpha = \frac{7}{31}$$

$$\frac{1 - \alpha}{p} = \frac{24}{31} \cdot \frac{1}{8} = 3 \cdot \frac{1}{31}$$

$$S_p = \frac{1}{\frac{7}{31} + \frac{3}{31}} = \frac{31}{10} = 3,1$$

Anche in questo caso abbiamo potuto utilizzare la **legge di Ware-Amdahl base**, in quanto la **fase sequenziale e la fase parallela sono ben distinte**;

- **se utilizziamo la II strategia**, però, oltre alla fase sequenziale e alla fase parallela avremo anche una fase di parallelismo medio, e quindi bisognerà usare la **legge di Ware-Amdahl generalizzata**. La **prima fase resta puramente parallela** e quindi, come già visto, avremo gli 8 core che eseguono 3 somme ciascuno in parallelo

$$\frac{\alpha_8}{8} = \frac{24}{31} \cdot \frac{1}{8} = \frac{3}{31}$$

Ci saranno poi le **fasi di parallelismo parziale**, nelle quali lavoreranno prima 4 core e poi 2 core per effettuare la collezione dei risultati locali. Queste due fasi saranno date da

$$\frac{\alpha_4}{4} = \frac{4}{31} \cdot \frac{1}{4} = \frac{1}{31}$$

$$\frac{\alpha_2}{2} = \frac{2}{31} \cdot \frac{1}{2} = \frac{1}{31}$$

A questo punto manca un'unica operazione, da svolgere in **sequenziale**, che è il calcolo della somma finale a partire dalle ultime due somme parziali

$$\alpha_1 = \frac{\alpha_1}{1} = \frac{1}{31}$$

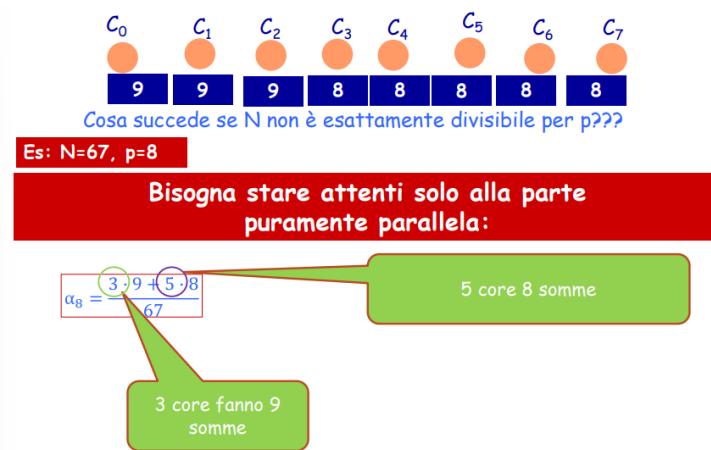
Mettendo insieme i pezzi, quindi, lo speed-up sarà dato da

$$S_p = \frac{1}{\frac{1}{31} + \frac{1}{31} + \frac{1}{31} + \frac{3}{31}} = \frac{31}{6} = 5,1$$

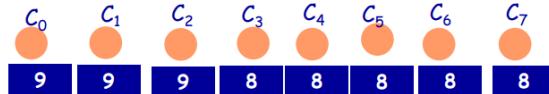
N non divisibile per p

In entrambi i tipi di algoritmi appena visti, **nel caso in cui la dimensione N del vettore, o dei vettori, non sia divisibile per il numero di core p**, bisognerebbe stare attenti solo alla **frazione di operazioni da eseguire puramente in parallelo**.

Considerando il caso in cui abbiamo $p = 8$ e $N = 67$, infatti, notiamo come per la somma di due vettori avremo 3 core che faranno 9 somme e 5 che ne faranno 8, quindi il totale delle operazioni sarà 67



mentre per la somma degli elementi di un vettore avremo che 3 core effettueranno 8 somme, avendo 9 numeri, mentre gli altri 5 ne eseguiranno 7, avendo 8 numeri, con il totale che è 66, viste le 7 operazioni in cui vengono combinate le somme parziali in base alla strategia adottata.



Bisogna stare attenti solo alla parte puramente parallela, come negli altri casi:

$$\alpha_8 = \frac{3 \cdot 8 + 5 \cdot 7}{66}$$

3 core hanno 9 numeri, quindi fanno 8 somme

5 core hanno 8 numeri, quindi fanno 7 somme

66 somme totali

Calcolo isoefficienza

L'isoefficienza, come sappiamo, è una **funzione di tre variabili** p_0, p_1, n_0 che permette di calcolare la nuova dimensione del problema da scegliere n_1 , in modo da **valutare la scalabilità di un algoritmo**.

L'isoefficienza è data dal **rapporto tra gli overhead totali** di un problema risolto con coppie (p_1, n_1) e (p_0, n_0)

$$I(p_0, p_1, n_0) = O_h(p_1, n_1) / O_h(p_0, n_0)$$

Somma di due vettori di dimensione N

Per questo tipo di problema l'**overhead totale è sempre 0**, essendo l'algoritmo **full-parallel**, e quindi l'isoefficienza è

$$I = 0/0$$

che è una **forma indeterminata**, ma per convenzione I può assumere **qualunque valore**.

Come sappiamo, **gli algoritmi full parallel sono naturalmente scalabili**.

Somma degli elementi di un vettore di dimensione N (somma di N numeri)

Il problema della somma di N numeri risulta **scalabile nel caso in cui si utilizzi la II strategia**, mentre assolutamente **non scalabile nel caso in cui si utilizzi la prima**.

Con la seconda, infatti, avremo che l'**overhead totale** è dato da $p \log_2 p$, e di conseguenza **l'isoefficienza è data da**

$$I = (p_1 \log_2 p_1) / (p_0 \log_2 p_0),$$

mentre con la prima l'**overhead totale** è $p^2 - 2p + 1$, e di conseguenza l'**isoefficienza è data da**

$$I = (p_1^2 - 2p_1 + 1) / (p_0^2 - 2p_0 + 1).$$

Costrutti work sharing e critical (Lez. 12) la lezione 11 era di laboratorio e anche questa, ma in questa ci sono delle nozioni nuove

La direttiva #pragma omp, oltre al costrutto parallel, prevede anche tre tipi di costrutti detti **work sharing**, perchè si occupano della distribuzione del lavoro al team di thread: **for**, **sections** e **single**.

Anche all'uscita da un costrutto di questo tipo è sottintesa una barriera di sincronizzazione, a meno che non venga specificato il contrario dal programmatore.

Tutti i costrutti work sharing possono essere combinati con il costrutto parallel, con le clausole specificate che sono l'unione di quelle specificate per ognuno.

Costrutto for

Il costrutto for specifica che le **iterazioni del ciclo contenuto debbano essere distribuite tra i thread del team**.

Come clausole abbiamo:

- **nowait**, che **elimina la barriera implicita alla fine del costrutto**, in modo che i thread che hanno finito possano continuare a lavorare senza aspettare la terminazione di tutti gli altri;
- **schedule (kind[, chunk_size])**, che specifica il modo (**kind**) di distribuire le iterazioni del ciclo seguente e può essere **static**, con i **chunk assegnati secondo uno scheduling round robin** (secondo l'id), **dynamic/guided**, con i **chunk assegnati su richiesta** (quando un thread termina il proprio chunk, ne chiede un altro), o **runtime**, con la **decisione che viene presa a runtime** mediante la variabile d'ambiente **OMP_SCHEDULE**. Il **chunk_size**, invece, è il numero (>0) di iterazioni contigue da assegnare allo stesso thread.

Se si vuole utilizzare il **costrutto parallel** in combinazione con il **costrutto for**, tutto il corpo della regione parallela dovrà essere contenuto tra le parentesi del ciclo for.

Costrutto sections (Lez.22-28)

Il costrutto sections conterrà un **insieme di costrutti section**, ognuno dei quali **verrà eseguito da un thread della regione parallela**.

Le diverse sezioni devono poter essere eseguite in ordine arbitrario, ma tenendo conto del fatto che questo tipo di costrutto espone al **rischio di sbilanciamento del carico**.

Il costrutto sections **permette di implementare la decomposizione funzionale**, diversa da quella classica, ossia la decomposizione del dominio, in quanto **consiste nel far svolgere ad ogni unità processante un passo differente dell'algoritmo** (ad es. potrebbe essere usato nel [problema SAXPY](#)).

```
#pragma omp parallel sections num_threads(3)
{
    #pragma omp section
    {
        printf( Hello world ONE! )
    }
    #pragma omp section
    {
        printf( Hello world TWO! )
    }
    #pragma omp section
    {
        printf( Hello world THREE! )
    }
}
```

In ogni section entrerà un unico thread (all'interno delle section si lavora in sequenziale), di conseguenza **se viene allocato un numero di thread maggiore dei costrutti section** all'interno del blocco sections, **il thread in più non lavorerà**, mentre **se al contrario verranno specificati meno thread dei costrutti section** nella clausola num_threads, **uno dei thread farà doppio lavoro**.

Costrutto single (Lez.22)

Il costrutto single permette di specificare che **il blocco di istruzioni successivo verrà eseguito da un solo thread** qualsiasi della regione parallela.

Scheduling per costrutto for (Lez.13)

Nel caso in cui non si specifichi il tipo di scheduling utilizzato per suddividere le varie iterazioni del ciclo for tra i thread, allora **viene automaticamente assunto** (come default) **che lo scheduling sia di tipo static**.

Lo scheduling **static** è ideale nel caso in cui **ogni iterazione del ciclo for impieghi all'incirca lo stesso tempo**, poiché presenta un **sovraffollamento minimo**.

Nel caso in cui **differenti iterazioni impieghino un diverso tempo di esecuzione**, invece, viene preferito lo scheduling **dynamic**.

Esiste però anche l'altro lato della medaglia, infatti con questo tipo di scheduling i thread, dopo ogni iterazione, devono arrestarsi e riceverne una nuova (di iterazione), ma ciò provoca un **evidente rallentamento**. Addirittura, **in determinati casi**, lo scheduling **dynamic** può risultare **più lento di quello static**.

Per trovare un **compromesso tra questi due tipi** di scheduling esistono due possibilità.

La prima consiste nell'aggiungere allo scheduling **dynamic** l'**opzione chunk**, che permette di **associare ad ogni thread un numero prestabilito di iterazioni**, dopodichè, quando il thread avrà terminato, gli verrà assegnato un nuovo chunk.

La seconda possibilità, invece, consiste nell'utilizzare lo scheduling **guided**, che ha una politica di scheduling molto **simile alla modalità dynamic**, tranne per il fatto che la **dimensione del chunk non deve essere specificata**, in quanto **varia durante l'esecuzione del programma**. Si inizia infatti **con chunk di grandi dimensione, per poi adattarle** (dimensioni del chunk) nel caso in cui il carico di lavoro sia sbilanciato.

In generale, comunque, **la scelta migliore dipende dal problema e dalla strategia parallela da implementare**.

Prodotto scalare tra due vettori di dimensione N (Lez.14)

Come sappiamo, il prodotto scalare tra due vettori viene calcolato effettuando la **somma del prodotto puntuale**, ossia il prodotto tra elementi dei due vettori con indici uguali.

In questo tipo di problema abbiamo quindi **in input due vettori di uguale dimensione N e in output un valore unico**, ossia appunto la somma del prodotto puntuale.

Input: $a = (a_0, a_1, a_2, \dots, a_{N-1})$, $b = (b_0, b_1, b_2, \dots, b_{N-1})$

Output: $c = a_0 \times b_0 + a_1 \times b_1 + a_2 \times b_2 + \dots + a_{N-1} \times b_{N-1}$

In particolare, **su un calcolatore monoprocesso**, il prodotto scalare viene calcolato eseguendo **N moltiplicazioni**, una per volta (prodotto puntuale), e poi effettuando **N-1 addizioni**, sempre una per volta, ossia la somma tra gli elementi risultanti dal prodotto puntuale. La prima parte, ossia quella relativa alle moltiplicazioni, è uguale al problema della somma tra vettori di dimensione N, ma sostituendo l'addizione con la moltiplicazione; al contempo, la seconda parte, quindi quella relativa alle addizioni, è uguale alla somma tra gli elementi di un vettore.

$$\begin{aligned}
 c_0 &:= a_0 \times b_0 \\
 c_1 &:= a_1 \times b_1 \\
 &\dots \\
 &\dots \\
 c_{N-1} &:= a_{N-1} \times b_{N-1} \\
 \\
 \textcolor{red}{c} &:= c_0 \\
 c &:= c + c_1 \\
 c &:= c + c_2 \\
 &\dots \\
 &\dots \\
 c &:= c + c_{N-1}
 \end{aligned}$$

L'**algoritmo sequenziale** che risolve il problema del prodotto scalare tra due vettori di dimensione N sarà quindi

```

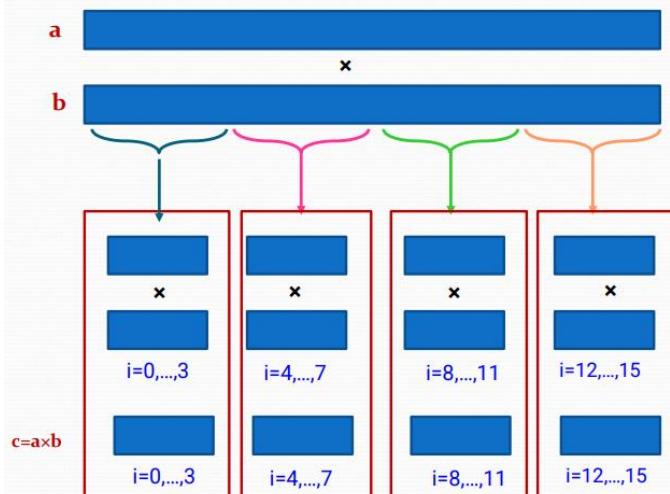
begin
    c := 0;
    for i=0 to N-1 do
        c := c + (ai × bi);
    endfor
end

```

Passiamo ora all'**algoritmo parallelo** che permette di risolvere questo problema, considerando un esempio con dimensione dei vettori N=16 e numero di unità processanti np=4.

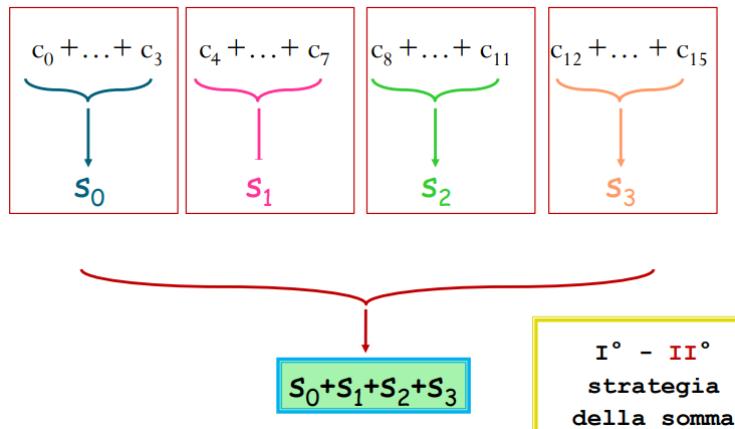
L'algoritmo si divide in **due fasi**:

- nella prima **si suddividono i vettori tra le varie unità processanti**, con ognuna di esse che eseguirà il **prodotto puntuale tra due sottovettori** di quelli di partenza;



- nella seconda fase, invece, **ogni unità processante eseguirà la somma del prodotto puntuale dei sottovettori assegnati**, dopodichè avverrà una

collezione dei risultati locali, ossia la somma delle somme parziali di ogni core.



Speed-up, overhead ed efficienza dell'algoritmo parallelo per il prodotto scalare

La **complessità computazionale sequenziale** dell'algoritmo per il prodotto scalare è data dagli N prodotti più le $N-1$ somme.

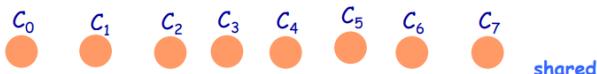
$$N \text{ prodotti} + (N-1) \text{ somme} \quad \text{somme} \sim \text{prodotti}$$



$$T_1(N) = 2N - 1$$

Considerando invece il caso in cui abbiamo **due vettori di dimensione N , con quest'ultima perfettamente divisibile per il numero di core p** , che ad esempio può essere 8, avremo che la **complessità computazionale con p core** sarà data dagli N/p prodotti più le $N/p - 1$ somme, a cui bisognerà aggiungere la collezione dei risultati locali.

$$p=8, \dim[a]=\dim[b]=N$$

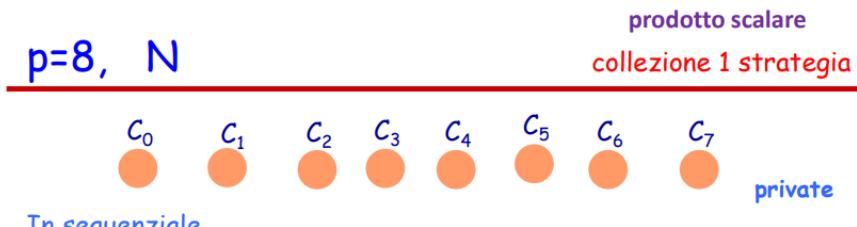


$$\begin{array}{l} \text{Calcolo prodotti} \\ \text{locali} \end{array} \quad \frac{N/p \text{ prodotti}}{+} \quad \frac{N/p}{\text{N/p}} \\ \text{Calcolo somme} \\ \text{parziali} \quad \frac{N/p - 1 \text{ somme}}{\text{(N/p - 1)}} \end{array}$$

$$2N/p - 1$$

La collezione dei risultati locali dipende dal tipo di strategia che si vuole utilizzare. Se si opta per la **prima**, allora bisognerà eseguire $p-1$ operazioni di collezione delle somme parziali di ogni core, mentre se si sceglie la **seconda** bisognerà eseguire un numero di operazioni pari a $\log_2 p$. Di conseguenza la **complessità computazionale su p core** è pari a $\left(2 \frac{N}{p} - 1\right) + (p - 1)$ nella I strategia, mentre è $\left(2 \frac{N}{p} - 1\right) + \log_2 p$ se si utilizza la II strategia.

Conoscendo sia la complessità computazionale in sequenziale che quella su p core, si può procedere al **calcolo di speed-up, overhead e efficienza** come segue

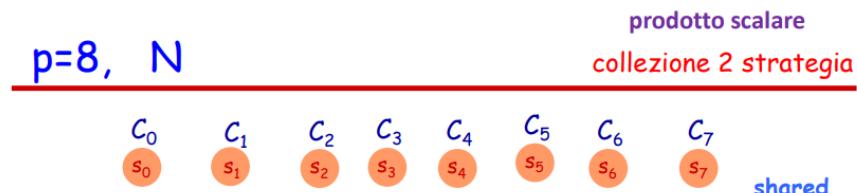


$$T_1(N) = 2N - 1$$

$$\begin{aligned} S_p &= T_1(N)/T_p(N) = \\ &= [2N-1] / [(2N/p - 1) + (p-1)] \end{aligned}$$

$$Oh = p T_p(N) - T_1(N) = p[(2N/p - 1) + (p-1)] - (2N-1)$$

$$E_p = S_p / p = [2N-1] / p [(2N/p - 1) + (p-1)]$$



$$T_1(N) = 2N - 1$$

$$\begin{aligned} S_p &= T_1(N)/T_p(N) = \\ &= [2N-1] / [(2N/p - 1) + \log(p)] \end{aligned}$$

$$Oh = p T_p(N) - T_1(N) = p[(2N/p - 1) + \log(p)] - (2N-1)$$

$$E_p = S_p / p = [2N-1] / p [(2N/p - 1) + \log(p)]$$

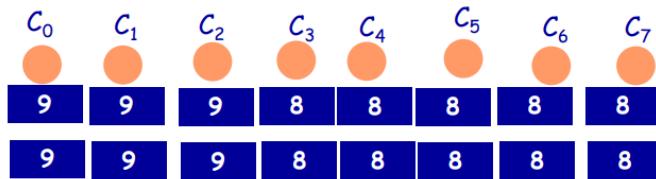
Considerando, invece, il caso in cui la **dimensione dei due vettori N non sia esattamente divisibile per il numero di core p**, ad esempio se abbiamo $N=67$ e $p=8$, alcuni processori dovranno occuparsi di un elemento in più, nel nostro caso 9 invece di 8. Per quanto riguarda la complessità computazionale dell'algoritmo su p

core bisognerà quindi considerare come se tutti i processori dovessero occuparsi di 9 elementi, e questo porta ad avere un prodotto ed una somma in più.

La **complessità computazionale**, in questo caso, sarà data dalla parte intera del rapporto $N/p + 1$, che è il numero di prodotti, e dalla parte intera del rapporto N/p , che è il numero delle somme, a cui poi si deve aggiungere la fase di collezione dei risultati locali, che rimarrà invariata.

$p=8, N=67$

Cosa succede se N non è esattamente divisibile per p???



Fase Locale

$\text{int}(N/p)+1$ prodotti + $\text{int}(N/p)$ somme



La fase di collezione dei risultati, ovviamente resta invariata...
ma attenzione!

Il computo totale sarà quindi $2 \text{int}(N/p) + 1$, più il numero di operazioni di collezione dei risultati locali, che sarà dato dalla strategia scelta.

$T_p(N) = 2 \text{int}(N/p) + 1 + p - 1$ 1 strategia

$T_p(N) = 2 \text{int}(N/p) + 1 + \log(p)$ 2 strategia

Isoefficienza del prodotto scalare di 2 vettori di dimensione N

Per quanto riguarda l'isoefficienza, questa risulta pressoché identica a quella del problema della somma degli elementi di un vettore di dimensione N. In particolare si avrà, per la **1 strategia**

$$\begin{aligned}
 O_h(p, N) &= T_1(N) - p \cdot T_p(N) = && \boxed{\text{I strategia}} \\
 &= p[(2N/p - 1) + (p - 1)] - (2N-1) = \\
 &= 2pN/p - p + p^2 - p - 2N + 1 = \\
 &= 2N + p^2 - 2N + 1 - 2p = p^2 - 2p + 1
 \end{aligned}$$

 $\boxed{I = (p_1^2 - 2p_1 + 1)/(p_0^2 - 2p_0 + 1)}$

È sempre la 1 strategia di collezione della somma

,

mentre per la **II strategia**

$$\begin{aligned}
 O_h(p, N) &= T_1(N) - p \cdot T_p(N) = && \boxed{\text{II strategia}} \\
 &= p[(2N/p - 1) + \log(p)] - (2N-1) = \\
 &= 2pN/p - p + p \log(p) - 2N + 1 = \\
 &= 2N - p + p \log(p) - 2N + 1 = p \log(p) - p + 1
 \end{aligned}$$

 $\boxed{I = (p_1 \log(p_1) - p_1 + 1)/(p_0 \log(p_0) - p_0 + 1)}$

È sempre la 2 strategia di collezione della somma

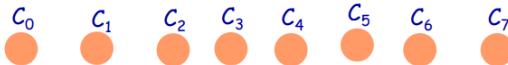
Speed-up calcolato con la legge di Ware-Amdahl

Considerando di avere due vettori di dimensione $N=32$ e numero di core $p=8$, avremo che ogni core dovrà occuparsi di sottovettori di dimensione $nloc=4$.

In sequenziale il numero di operazioni da svolgere sarebbe $2N-1$, che nel nostro caso equivale a 63.

Nella prima fase, ossia quella puramente parallela, ogni core esegue 4 prodotti e 3 somme, e di conseguenza il numero totale delle operazioni svolte in parallelo è $(4+3) * \text{il numero di core}$, nel nostro caso 8, quindi $7*8=56$ sulle 63 totali.

$p=8$, $\text{dim}[a]=\text{dim}[b]=32$, $nloc=4$



In sequenziale $2N-1=63$ operazioni

1 fase (tutta parallela)

$nloc = 4$ prodotti

+

$nloc-1 = 3$ somme

7 operazioni fatte contemporaneamente da 8 core

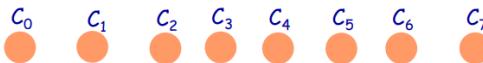
$7 \times 8 = 56$ delle 63 operazioni

$$\alpha_8 = 56/63$$

... a questo punto devo decidere come voglio collezionare le somme parziali!

A questo punto, se per la seconda fase si sceglie la I strategia, si avranno 7 somme effettuate in sequenziale, da un unico core, che portano il totale alle 63 previste.

$p=8, N=32, nloc=4$



In sequenziale

In sequenziale $2N-1=63$ operazioni

$$\alpha_8 = 56/63$$

I strategia

$$\alpha_7 = \alpha_6 = \alpha_5 = \alpha_4 = \alpha_3 = \alpha_2 = 0$$

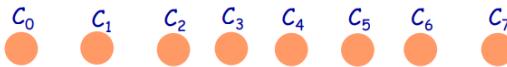
7 somme fatte da 1 solo core

$$\alpha_1 = (7 \cdot 1)/63$$

È sempre la 1 strategia di collezione della somma

Nel caso in cui si scelga la II strategia, invece, si avranno alcune fasi di parallelismo parziale. Infatti avremo una fase in cui 4 core eseguono una somma ciascuno, dopodiché ne avremo un'altra in cui 2 core eseguiranno 1 somma ciascuno, ed infine avremo un'unica somma puramente sequenziale, che porta il totale alle 63 previste.

$p=8, N=32, nloc=4$



In sequenziale

In sequenziale $2N-1=63$
operazioni

$$\alpha_8 = 56/63$$

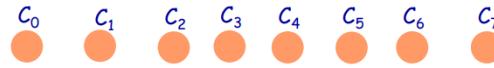
II strategia

$$\alpha_7 = \alpha_6 = \alpha_5 = 0$$

1 somma fatta da 4 core

$$\alpha_4 = (1 \cdot 4)/63$$

$p=8, N=32, nloc=4$



In sequenziale

In sequenziale $2N-1=63$
operazioni

$$\alpha_8 = 56/63$$

$$\alpha_7 = \alpha_6 = \alpha_5 = 0$$

$$\alpha_3 = 0$$

$$\alpha_2 = (1 \cdot 2)/63$$

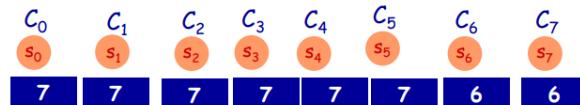
1 somma fatta da 1 core

$$\alpha_1 = (1 \cdot 1)/63$$

È sempre la 2 strategia di collezione della somma

Consideriamo ora di avere due vettori di dimensione $N=54$, che non è esattamente divisibile per il numero di core $p=8$. Qualsiasi sia la strategia per la collezione delle somme parziali, ciò che cambia sarà solo il numero di operazioni da eseguire nella fase puramente parallela.

$p=8, N=54, nloc=6, r=6$



In sequenziale

$2N - 1 = 107$
operazioni
calcolo locale

6 core - $2nloc-1=13$ operazioni
2 core - $2nloc-1=11$ operazioni

13 operazioni fatte contemporaneamente da 6 processori/core



11 operazioni fatte contemporaneamente da 2 processori/core

$$\alpha_8 = (13 \times 6 + 11 \times 2) / 107 = (78 + 22) / 107 = 100 / 107$$

Ricerca e ordinamento in ambiente multicore (Lez. 16) la lez. 15 era di laboratorio, questa lezione non si deve studiare per l'esame probabilmente

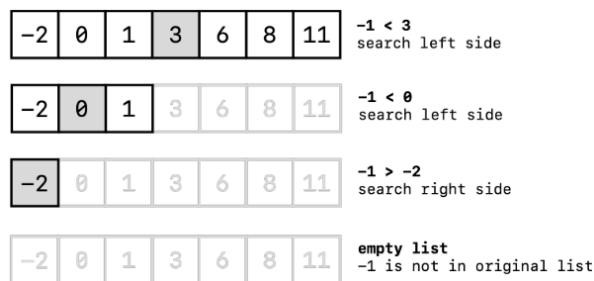
Ricerca binaria (o dicotomica)

L'algoritmo di ricerca binaria è un **algoritmo che individua un determinato elemento (il token) in una sequenza ordinata di elementi.**

Il **funzionamento dell'algoritmo** si può riassumere in:

- suddivisione del vettore in due parti e calcolo dell'indice dell'elemento mediano;
- se il valore token (quello da ricercare) è maggiore rispetto all'elemento di indice mediano, la ricerca si ripete considerando la metà di sinistra del vettore;
- in caso contrario la ricerca procede analizzando la parte di destra del vettore suddiviso.

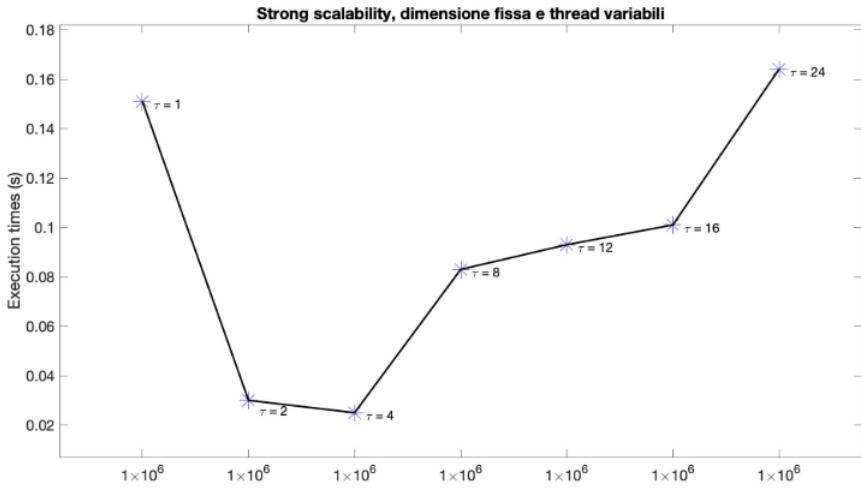
Ad ogni passo, quindi, viene dimezzata la dimensione, il che porta ad avere un limite asintotico **O(log n)**.



Per misurare le performance dell'algoritmo parallelo di ricerca binaria si possono utilizzare **due metriche**:

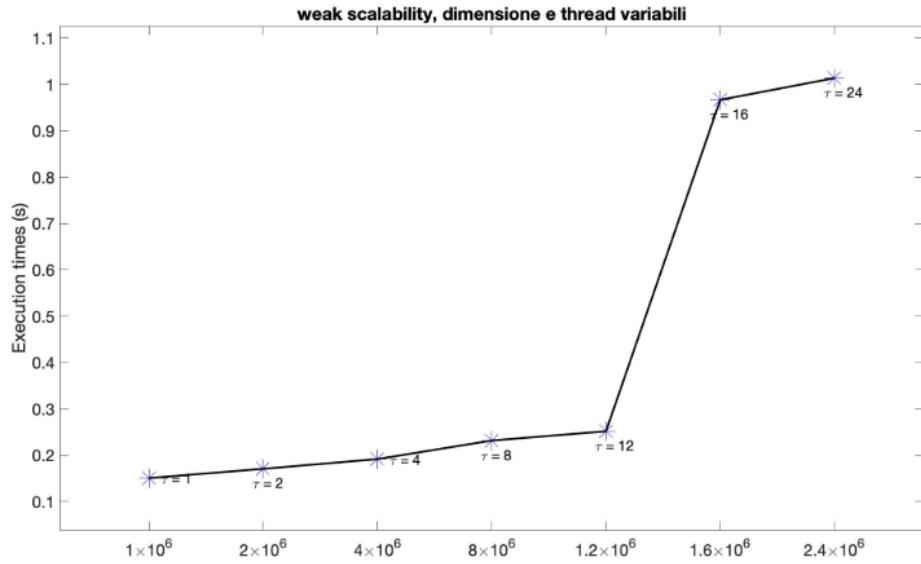
- **strong scalability**, che misura lo **speed-up per una dimensione fissa del problema rispetto al numero di processori che invece si incrementa**, ed è governato dalla legge di Amdahl;

N	τ	Execution time (s)
1×10^6	1	0.151
1×10^6	2	0.030
1×10^6	4	0.025
1×10^6	8	0.083
1×10^6	12	0.093
1×10^6	16	0.101
1×10^6	24	0.162

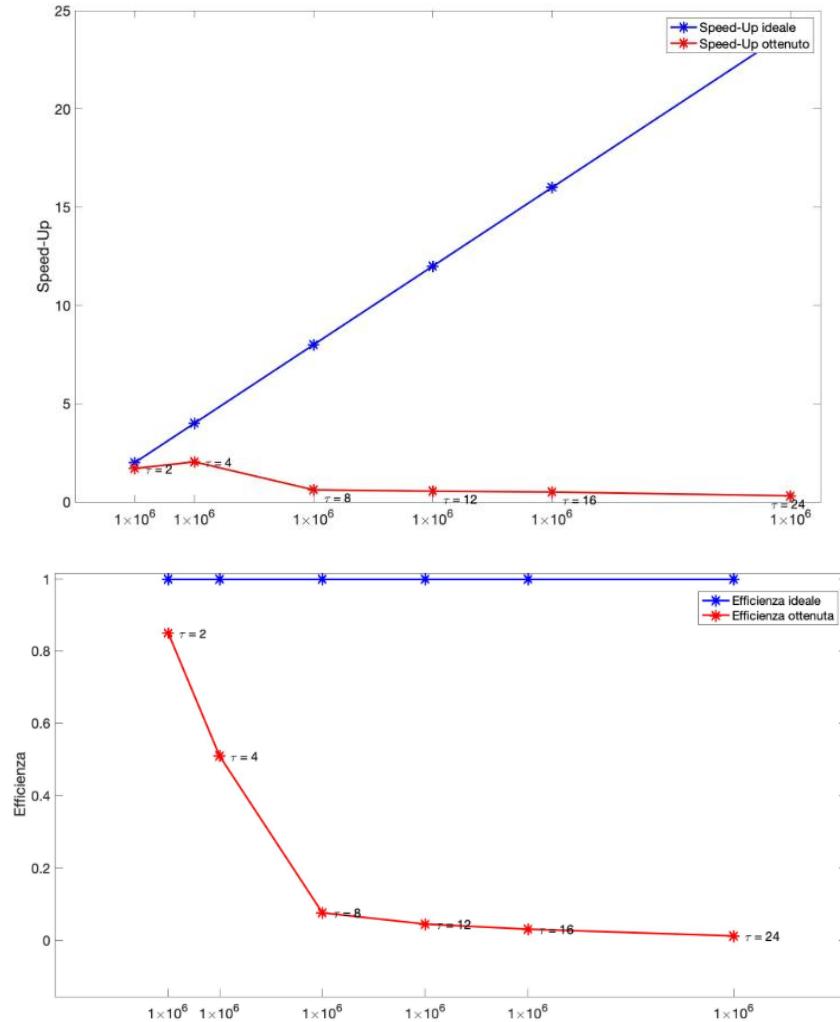


- **weak scalability**, che misura lo **speed-up** per una dimensione del problema ridotta rispetto al numero di processori.

N	τ	Execution time (s)
1×10^6	1	0.151
2×10^6	2	0.171
4×10^6	4	0.192
8×10^6	8	0.232
1.2×10^6	12	0.252
1.6×10^6	16	0.967
2.4×10^6	24	1.014



Di seguito troviamo **speed-up** ed **efficienza** ottenuti, rispetto a quelli ideali.



Odd-Even sorting

L'operazione principale degli algoritmi di ordinamento è il **confronto**, ma bisogna tenere conto che un numero elevato di essi (confronti) genera condizioni di **overhead**.

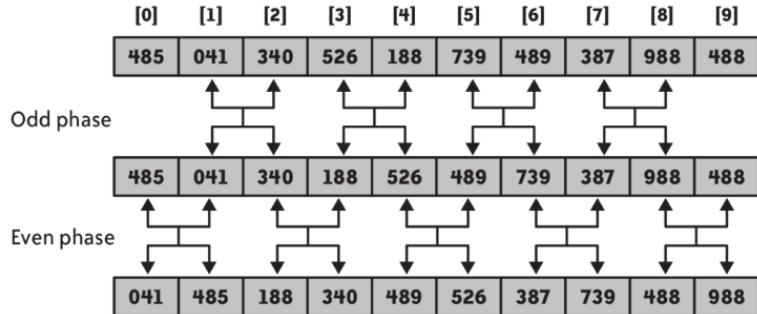
Algoritmo	Complessità di tempo		
	Migliore	Medio	Peggior
Selection sort	$\Omega(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Bubble sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Heap sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\mathcal{O}(n \log(n))$
Quick sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\mathcal{O}(n^2)$
Merge sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$\mathcal{O}(n \log(n))$

L'odd-even sorting è un algoritmo di ordinamento (sotto certi aspetti simile al bubble sort) che **opera**, come suggerisce il nome, **sulla parità degli indici**.

In questo algoritmo **si confrontano tutte le coppie** pari e dispari degli elementi presenti in un vettore, dopodichè, **se una coppia è nell'ordine sbagliato, si**

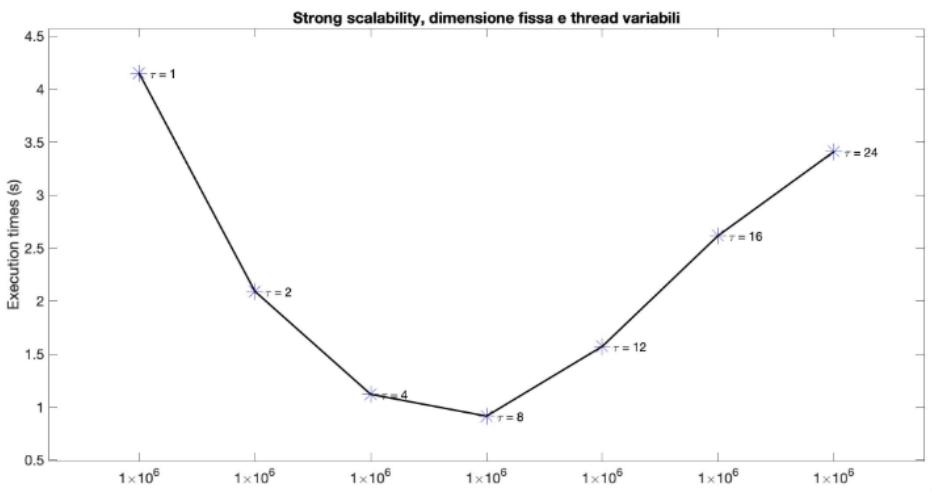
scambiano di posto gli elementi. L'algoritmo di ordinamento continua fino a quando tutte le coppie non sono ordinate.

Visto che l'algoritmo comporta un **elevato numero di confronti** in funzione di n , in particolare **legati alle coppie da analizzare**, la **complessità è $O(n^2)$** .



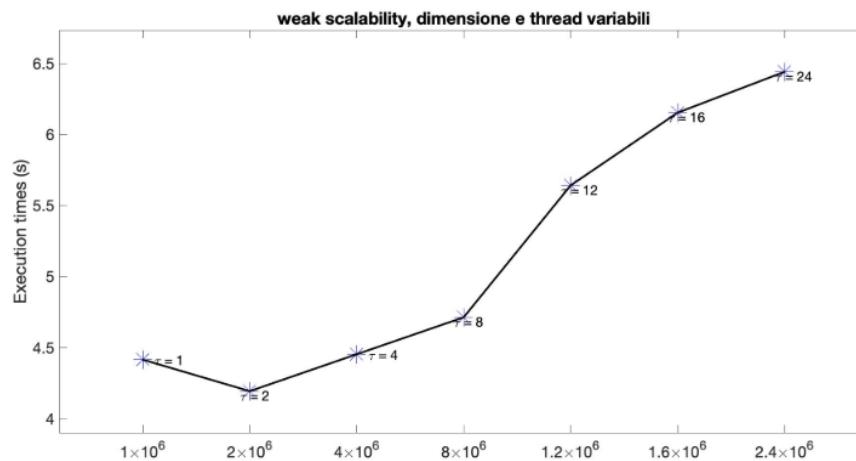
La **strong scalability** di questo algoritmo è

N	τ	Execution time (s)
1×10^6	1	4.151
1×10^6	2	2.092
1×10^6	4	1.120
1×10^6	8	0.914
1×10^6	12	1.572
1×10^6	16	2.617
1×10^6	24	3.410

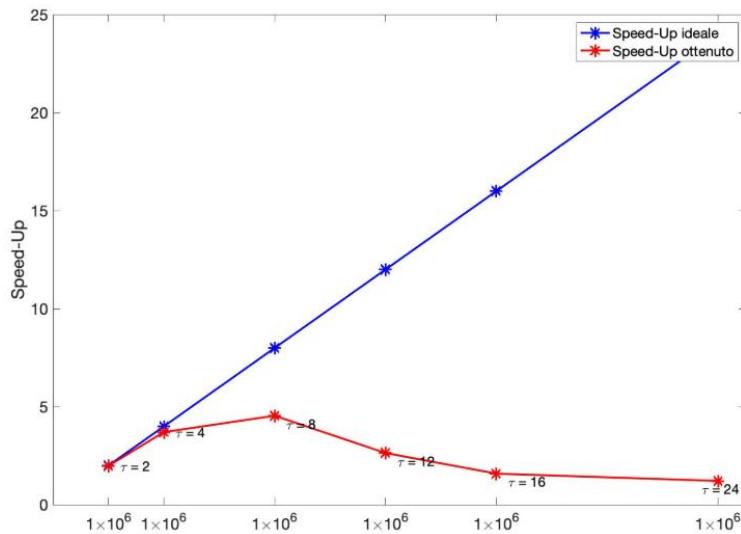


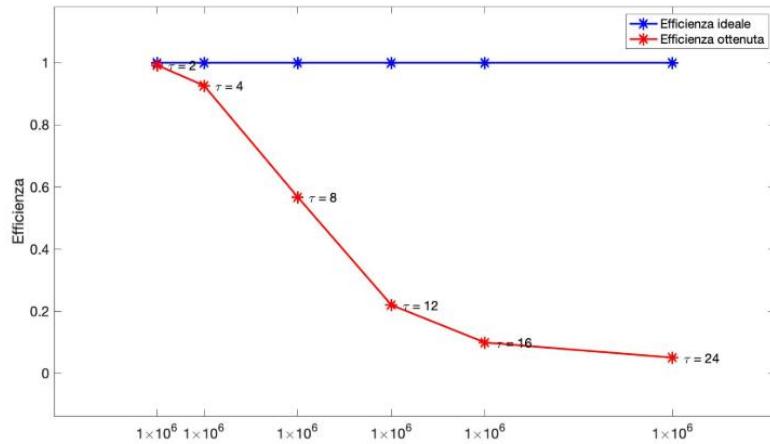
mentre la **weak scalability** è

N	τ	Execution time (s)
1×10^6	1	4.141
2×10^6	2	4.192
4×10^6	4	4.452
8×10^6	8	4.712
1.2×10^6	12	5.641
1.6×10^6	16	6.153
2.4×10^6	24	6.441



Infine **speed-up ed efficienza**, anche in questo caso, si distanziano molto da quelli ideali.





Algoritmi full-parallel per matrici (Lez.17)

Negli algoritmi che prevedono in input matrici, e non vettori, la decomposizione del dominio riguarderà, ovviamente, questo tipo di strutture.

Prodotto di uno scalare per una matrice

Nel problema del prodotto di uno scalare (un numero qualsiasi) per una matrice, si hanno in **input** un qualsiasi valore numerico, che rappresenta lo **scalare**, ed una **matrice**, che nel caso full-parallel che analizzeremo di seguito è **di dimensioni NxN**.

In **output**, invece, avremo **un'unica matrice, di dimensioni NxN**, che equivale semplicemente al **prodotto di ogni elemento della matrice per lo scalare**.

Input: $\beta \cdot A : \dim(A)=N \times N$

Output: $C=\{c_{i,j}\} = \{\beta \cdot a_{i,j}\} \quad i=0, \dots, N-1, j=0, \dots, N-1$

Su un calcolatore monoprocessoresso il prodotto è calcolato eseguendo NxN prodotti, uno per volta, secondo un ordine prestabilito.

```

begin
    for i=0 to N-1 do
        for j=0 to N-1 do
             $c_{i,j} := \beta \cdot a_{i,j};$ 
        endfor
    endfor
end

```

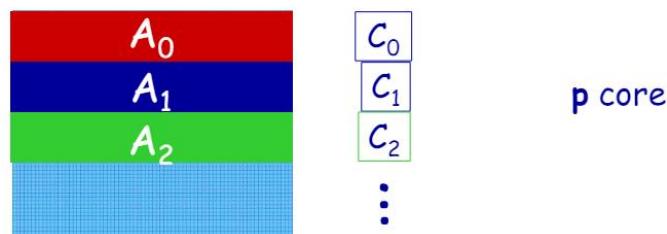
Come qualsiasi algoritmo da parallelizzare, l'idea è quella di decomporre un problema di dimensione N in np sottoproblemi di dimensione N/n_p , risolvendo questi ultimi (i sottoproblemi) contemporaneamente, utilizzando np CPU.

In questo caso, l'algoritmo parallelo sarà dato dal partizionamento della matrice in blocchi, con conseguente distribuzione degli stessi (blocchi) ai 7 diversi processori.

Esistono diverse strategie di partizionamento, e di conseguenza di parallelizzazione:

- blocchi di righe;
- blocchi di colonne;
- blocchi di righe-colonne.

Blocchi di righe (I strategia)



In questa prima strategia, l'idea è quella di suddividere la matrice in blocchi di righe, assegnando il prodotto dello scalare per un blocco riga ad ogni CPU.

Ogni blocco riga avrà come dimensioni il numero totale delle righe fratto il numero di core (N/p) per il numero totale delle colonne (N), ossia $N/p \times N$.

I blocchi riga calcolati possono poi essere uniti per formare la matrice risultato, e, nel caso di matrici $N \times N$, si ha un parallelismo completo.

■ Esempio: $N=6, p=2$

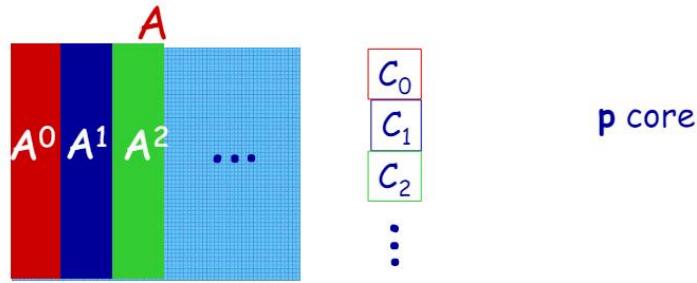
Prodotti locali

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \cdot \beta$$

c_0

c_1

Blocchi di colonne (II strategia)



In questa seconda strategia, l'idea è quella di **suddividere la matrice in blocchi di colonne, assegnando il prodotto dello scalare per un blocco colonna ad ogni CPU.**

Ogni blocco colonna avrà come dimensioni il numero totale delle righe (N) per il numero totale delle colonne fratto il numero di core (N/p), ossia $N \times N/p$.

I **blocchi colonna calcolati possono poi essere uniti** per formare la matrice risultato, e, nel caso di matrici $N \times N$, si ha un **parallelismo completo**.

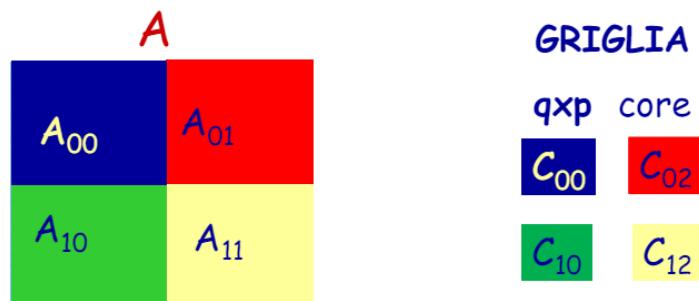
- Esempio: $N=6, p=2$

Prodotti locali

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \\ \cdot \beta & & & \cdot \beta & & \end{pmatrix}$$

$c_0 \qquad c_1$

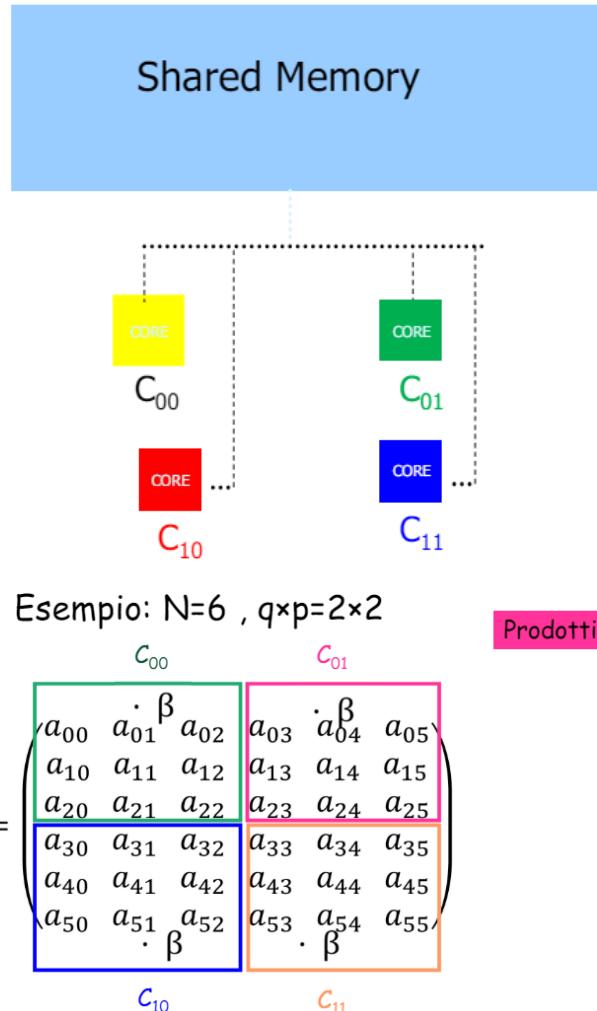
Blocchi di righe-colonne (III strategia)



In questa terza strategia, l'idea è quella di **suddividere la matrice in blocchi di righe e colonne, assegnando il prodotto dello scalare per un blocco riga-colonna ad ogni CPU.**

I blocchi riga-colonna calcolati possono poi essere uniti per formare la matrice risultato, e, nel caso di matrici NxN, si ha un **parallelismo completo**.

La griglia di processori viene gestita tramite **topologia virtuale**, in modo da assegnare ad ognuno dei core un blocco.



Strategie di parallelizzazione per problemi di tipo element-wise

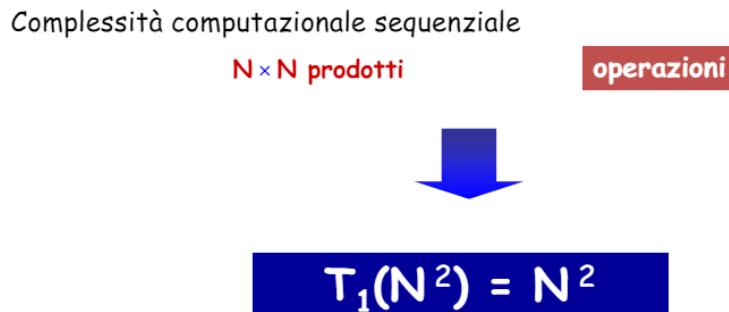
Nelle strategie riguardanti il problema del prodotto tra uno scalare e una matrice, come già detto, abbiamo analizzato problemi full-parallel, quindi senza alcuna collezione dei risultati.

Altri **problematiche element-wise** che possono risolversi in modo completamente parallelizzabile sono la somma, la sottrazione e il prodotto puntuale di matrici, la somma, la differenza e la divisione di uno scalare per una matrice, ma anche il calcolo della trasposta e la riflessione di una matrice.

Valutazione strategie per algoritmi full-parallel per matrici (Lez. 18)

Calcolo di speed-up, overhead ed efficienza per il prodotto di uno scalare per una matrice

Prima di procedere alle considerazioni riguardanti le varie strategie, ricordiamo la complessità computazionale dell'algoritmo sequenziale per il prodotto di uno scalare per una matrice. Considerando un'ipotetica matrice di dimensioni NxN dovranno essere effettuati NxN prodotti, il che porta la **complessità computazionale sequenziale** ad essere



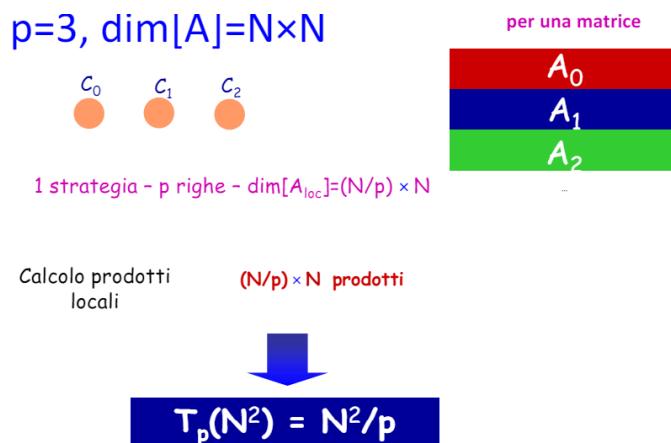
Nei seguenti paragrafi andremo a calcolare speed-up, overhead ed efficienza relativi alle varie strategie utilizzabili per parallelizzare questo problema.

Noteremo che, **fin quando le matrici restano quadrate** (quando il numero di righe e colonne è uguale), **non ci sono particolari differenze tra le tre strategie**.

I strategia (blocchi di righe)

Consideriamo di voler calcolare la **complessità computazionale** dell'algoritmo parallelo per il prodotto di uno scalare per una matrice, di dimensioni NxN, con la I strategia, ossia quella a blocchi di righe.

In questo caso **dovremo quindi suddividere le N righe totali tra i p core, in modo da creare i diversi blocchi riga da assegnare ai vari core**. I prodotti da calcolare saranno quindi



tenendo conto del fatto che **il numero di colonne N da considerare per ogni core rimane invariato**, visto che suddividiamo la matrice tra i vari core per blocchi riga.

A questo punto si può procedere al **calcolo di speed-up, overhead totale ed efficienza** come segue

In sequenziale

$$T_1(N^2) = N^2$$

$$S_p = T_1(N^2)/T_p(N^2) = N^2 / (N^2/p) = p$$

$$Oh = p \cdot T_p(N^2) - T_1(N^2) = p[N^2/p] - N^2 = 0$$

$$E_p = S_p / p = 1$$

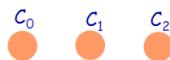
Come possiamo notare avremo speed-up, overhead ed efficienza **ideali, visto che si tratta di un algoritmo full-parallel**.

II strategia (blocchi di colonne)

Consideriamo di voler calcolare la **complessità computazionale** dell'algoritmo parallelo per il prodotto di uno scalare per una matrice, di dimensioni NxN, con la II strategia, ossia quella a blocchi di colonne.

In questo caso **dovremo quindi suddividere le N colonne totali tra i p core, in modo da creare i diversi blocchi colonna da assegnare ai vari core**. I prodotti da calcolare saranno quindi

$$p=3, \dim[A]=N \times N \quad \text{per una matrice}$$



$$2 \text{ strategia - } p \text{ colonne - } \dim[A_{loc}] = N \times (N/p)$$

Calcolo prodotti locali

$N \times (N/p)$ prodotti



$$T_p(N^2) = N^2/p$$

tenendo conto del fatto che **il numero di righe N da considerare per ogni core rimane invariato**, visto che suddividiamo la matrice tra i vari core per blocchi colonna.

A questo punto si può procedere al **calcolo di speed-up, overhead totale ed efficienza** come segue

In sequenziale

$$T_1(N^2) = N^2$$

$$S_p = T_1(N^2)/T_p(N^2) = N^2 / (N^2/p) = p$$

$$Oh = p T_p(N^2) - T_1(N^2) = p[N^2/p] - N^2 = 0$$

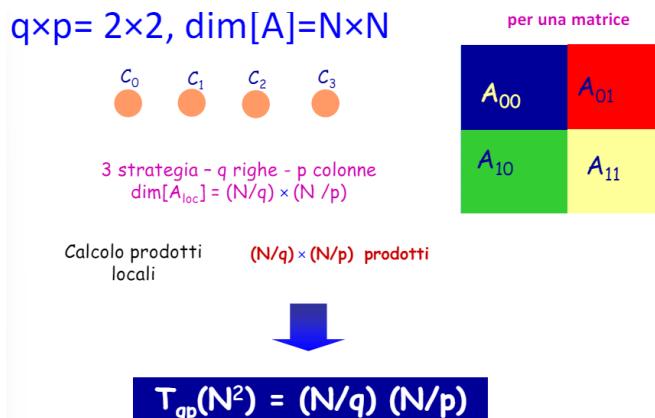
$$E_p = S_p / p = 1$$

Come possiamo notare avremo speed-up, overhead ed efficienza **ideali, visto che si tratta di un algoritmo full-parallel.**

III strategia (blocchi di righe-colonne)

Consideriamo di voler calcolare la **complessità computazionale** dell'algoritmo parallelo per il prodotto di uno scalare per una matrice, di dimensioni NxN, con la III strategia, ossia quella a blocchi di righe-colonne.

In questo caso **dovremo quindi suddividere le N righe totali tra i q core riga e le N colonne totali tra i p core colonna, in modo da creare i diversi blocchi riga-colonna da assegnare ai vari core.** I prodotti da calcolare saranno quindi



A questo punto si può procedere al **calcolo di speed-up, overhead totale ed efficienza** come segue

In sequenziale

$$T_1(N^2) = N^2$$

$$S_{qp} = T_1(N^2)/T_{qp}(N^2) = N^2 / [(N/q)(N/p)] = qp$$

$$Oh = qp T_{qp}(N^2) - T_1(N^2) = qp[N^2/qp] - N^2 = 0$$

$$E_p = S_p / p = 1$$

Come possiamo notare avremo speed-up, overhead ed efficienza **ideali, visto che si tratta di un algoritmo full-parallel.**

Isoefficienza del prodotto di uno scalare ed una matrice

In tutte le strategie del problema del prodotto di uno scalare per una matrice, che consideriamo di dimensione NxN, abbiamo visto che l'overhead totale è 0.

Di conseguenza l'isoefficienza sarà 0/0, che è una forma indeterminata, ma sappiamo che per convenzione significa che può assumere (l'isoefficienza) qualunque valore. **Tutte e tre le strategie** quindi **sono naturalmente scalabili**, visto che stiamo parlando di algoritmi full parallel.

sprodotto di uno scalare per una matrice A di dimensione N^2

1 strategia - p righe 2 strategia - p colonne

$$O_h(p, N^2) = T_1(N^2) - p \cdot T_p(N^2) = N^2 - p(N^2/p) = 0$$

3 strategia - pq blocchi righe&colonne

$$O_h(pq, N^2) = T_1(N^2) - pq \cdot T_p(N^2) =$$
$$N^2 - pq(N^2/pq) = 0$$



$$I = 0/0$$

Forma indeterminata.

Per convenzione: I può assumere qualunque valore

Gli algoritmi full parallel sono naturalmente scalabili

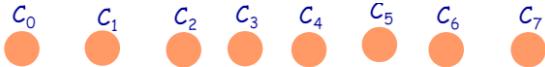
Speed-up del prodotto di uno scalare per una matrice caratterizzato con la legge di Ware-Amdahl

Consideriamo di avere una matrice di dimensioni NxN con N = 32 ed un numero p = 8 di core.

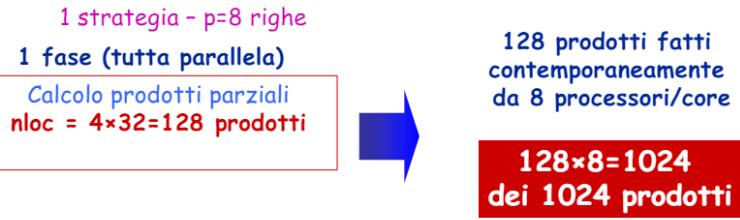
In sequenziale il numero di operazioni, e quindi di prodotti, da effettuare è appunto NxN, che nel nostro caso è uguale a 32x32 = 1024.

Essendo algoritmi full-parallel, in ogni strategia avremo un'unica fase, parallela, che ci permetterà di eseguire tutti e 1024 i prodotti.

Per quanto riguarda la 1 strategia, ogni core dovrà eseguire $32/8 \times 32$ prodotti, ossia $4 \times 32 = 128$. Moltiplicando questo numero per gli 8 core avremo che il numero di operazioni da eseguire nella fase totalmente parallela è 1024 su un totale di 1024, quindi tutte.

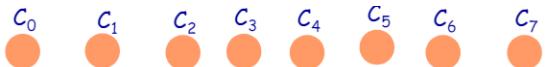


In sequenziale $32 \times 32 = 1024$ prodotti



$$(1-\alpha) = 32 \times 32 / 32 \times 32$$

Similmente, per quanto riguarda la **II strategia**, ogni core dovrà eseguire $32 \times 32/8$ prodotti, ossia $32 \times 4 = 128$. Moltiplicando questo numero per gli 8 core avremo che il numero di operazioni da eseguire nella fase totalmente parallela è 1024 su un totale di 1024, quindi tutte anche in questo caso.

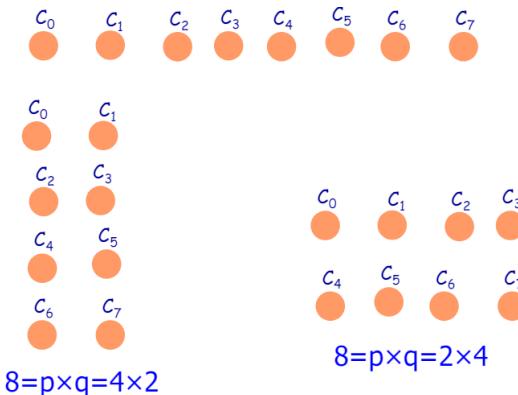


In sequenziale $32 \times 32 = 1024$ prodotti



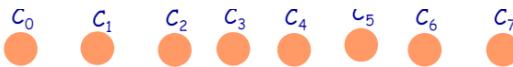
$$(1-\alpha) = 1024 / 1024$$

Per quanto riguarda la **III strategia** bisogna innanzitutto decidere come impostare la griglia di core, che può essere sia una 4×2 che una 2×4 .



Nel nostro caso consideriamo di scegliere la griglia 4×2 , quindi con $p = 4$ e $q = 2$. In questo caso, ogni core dovrà eseguire $32/4 \times 32/2$ prodotti, ossia $8 \times 16 = 128$. Moltiplicando questo numero per gli 8 core avremo che il numero di operazioni da

eseguire nella fase totalmente parallela è 1024 su un totale di 1024, quindi tutte anche in questo caso.



In sequenziale $32 \times 32 = 1024$ prodotti

3 strategia - p=4 righe q=2 colonne

1 fase (tutta parallela)

Calcolo prodotti parziali
nloc = $32/4 \times 32/2 =$
 $=128$ prodotti

128 prodotti fatti contemporaneamente da 8 processori/core

$128 \times 8 = 1024$
dei 1024 prodotti

$$(1 - \alpha) = 1024/1024$$

Analogamente, se scegliessimo la griglia 2x4 avremo



In sequenziale $32 \times 32 = 1024$ prodotti

3 strategia - p=2 righe q=4 colonne

1 fase (tutta parallela)

Calcolo prodotti parziali
nloc = $32/2 \times 32/4 =$
 $=128$ prodotti

128 prodotti fatti contemporaneamente da 8 processori/core

$128 \times 8 = 1024$
dei 1024 prodotti

$$(1 - \alpha) = 1024/1024$$

Anche se avessimo una **situazione con numero di core p = 4**, che porterebbe ad avere p = 2 e q = 2, quindi uguali, nella III strategia, non cambierebbe assolutamente nulla.



In sequenziale $32 \times 32 = 1024$ prodotti

1 strategia - p=4

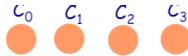
1 fase (tutta parallela)

Calcolo prodotti parziali
nloc = $32/4 \times 32/8 = 8 \times 32$
 $=256$ prodotti

256 prodotti fatti contemporaneamente da 4 processori/core

$256 \times 4 = 1024$
dei 1024 prodotti

$$(1 - \alpha) = 1024/1024$$



In sequenziale $32 \times 32 = 1024$ prodotti

2 strategia - p=4

1 fase (tutta parallela)

Calcolo prodotti parziali
nloc = $32 \times 32 / 4 = 32 \times 8$
= 256 prodotti

256 prodotti fatti contemporaneamente da 4 processori/core

$256 \times 4 = 1024$
dei 1024 prodotti

$$(1 - \alpha) = 1024/1024$$



In sequenziale $32 \times 32 = 1024$ prodotti

3 strategia - p=2, q=2

1 fase (tutta parallela)

Calcolo prodotti parziali
nloc = $32 / 2 \times 32 / 2 = 16 \times 16$
= 256 prodotti

256 prodotti fatti contemporaneamente da 4 processori/core

$256 \times 4 = 1024$
dei 1024 prodotti

$$(1 - \alpha) = 1024/1024$$

Codici per algoritmo del prodotto di uno scalare per una matrice (Lez.19)

Collapse

La clausola collapse(int) viene utilizzata per **indicare al compilatore di espandere un costrutto parallel for sul numero di livelli di annidamento** specificato tra le parentesi della clausola.

In pratica quando si utilizza collapse si vuole trattare un determinato numero di cicli annidati come un singolo livello di iterazione parallela, **assegnando ai vari thread delle combinazioni di iterazioni dei cicli annidati**. Così facendo si avrà una **gestione più efficiente, evitando l'overhead** che potrebbe derivare dalla gestione separata dei diversi livelli di iterazione.

Prodotto matrice per vettore (Lez.20)

Prima di discutere dell'algoritmo parallelo per la risoluzione del problema del prodotto di una matrice per un vettore, ricordiamo com'è definito questo problema. Come suggerisce il nome dobbiamo effettuare il prodotto tra una matrice ed un vettore, che **può essere eseguito solo se il numero di colonne della matrice è uguale al numero di elementi del vettore**. Il risultato di questa operazione sarà

un altro vettore, che avrà come numero di elementi il numero di righe della matrice. In particolare, ogni elemento del vettore risultante sarà dato dal prodotto scalare tra l'i-esima riga della matrice e l'intero vettore di partenza.

Consideriamo di avere una matrice quadrata, quindi con numero di righe e numero di colonne uguali, ed un vettore, rispettivamente di dimensioni $n \times n$ e n , con il vettore risultante che avrà anch'esso un numero di elementi pari a n .

$$Ax = y, \quad A \in \mathbb{R}^{n \times n}, \quad x, y \in \mathbb{R}^n$$

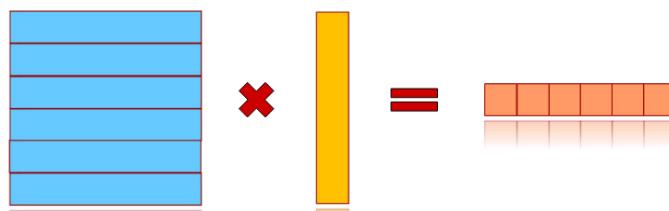
L'algoritmo sequenziale utilizzato per risolvere questo problema richiede ovviamente 2 cicli for, visto che abbiamo una matrice, ed il prodotto scalare di cui abbiamo parlato in precedenza.

```

for i=0,n-1 do
    yi=0
    for j=0,n-1 do
        yi=yi+aij xj
    endfor
endfor

```

Le componenti di y sono calcolate effettuando i prodotti scalari di ciascuna riga di A per il vettore x



Visto che ogni componente del vettore risultante viene calcolata effettuando i prodotti scalari, questi ultimi possono essere calcolati in maniera indipendente l'uno dall'altro, e questa è proprio la base per l'algoritmo parallelo che risolve il problema del prodotto tra matrice e vettore.

Anche in questo caso, come per il prodotto tra uno scalare ed una matrice, la decomposizione dei dati corrisponde ad un partizionamento in blocchi della matrice, ma stavolta, in alcune strategie, si dovrà partizionare anche il vettore. Di conseguenza, il calcolo della soluzione si riconduce al calcolo della soluzione relativa a ciascun blocco.

I strategia (blocchi di righe)

Come abbiamo detto in precedenza, consideriamo il caso in cui **sia le dimensioni della matrice che quella** (la dimensione) **del vettore** siano **uguali a $n = 6$** , con **numero di core** invece **uguale a 2**, suddividendo la matrice in blocchi di righe.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ \hline a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

In questo caso, quindi, **ogni core dovrà effettuare il calcolo di 3 componenti del vettore risultante, che corrispondono a 3 prodotti scalari tra 3 righe della matrice e l'intero vettore di partenza.**

In particolare **il primo core calcola** le prime 3 componenti del vettore risultante, eseguendo il prodotto scalare tra le prime 3 righe della matrice ed il vettore di partenza,

**Il core 0 calcola le prime tre componenti
del vettore y**

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

mentre **il secondo core calcola** le altre 3 componenti del vettore risultante, eseguendo il prodotto scalare tra le altre 3 righe della matrice ed il vettore di partenza.

**Il core 1 calcola
le altre tre componenti
del vettore y**

$$\begin{bmatrix} a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

II strategia (blocchi di colonne)

Anche in questo caso consideriamo $n = 6$ e numero di core uguale a 2, ma stavolta suddividiamo la matrice per blocchi di colonne.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

Con questa strategia, però, **ogni core avrà una sottomatrice 6×3** , di conseguenza, **per eseguire il prodotto matrice per vettore, anche il vettore dovrà avere un numero di elementi pari a 3**. Si procede quindi alla **suddivisione del vettore in due metà**, con il primo core che **dovrà eseguire** il prodotto matrice per vettore con una matrice 6×3 , con le 3 colonne che equivalgono alle prime 3 della matrice di partenza, ed un vettore di 3 elementi, che equivalgono ai primi 3 del vettore.

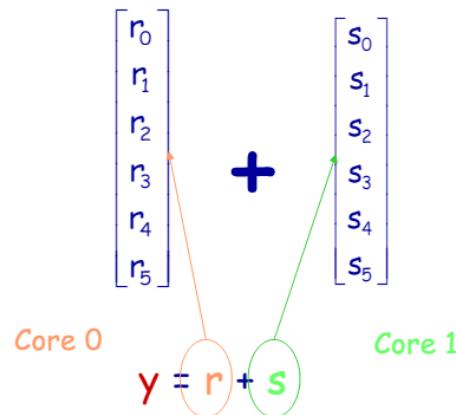
$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \\ a_{50} & a_{51} & a_{52} \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{bmatrix}$$

Anche **il secondo core dovrà eseguire** un prodotto matrice per vettore con una matrice 6×3 , stavolta con le altre 3 colonne della matrice di partenza, ed un vettore di 3 elementi, che corrispondono agli altri 3 del vettore di partenza.

$$\begin{bmatrix} a_{03} & a_{04} & a_{05} \\ a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \\ a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{bmatrix} \bullet \begin{bmatrix} x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

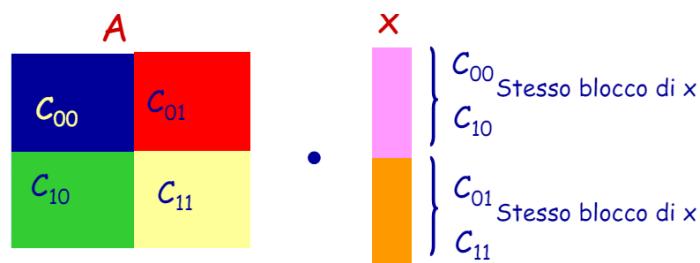
Come possiamo osservare, **il risultato del prodotto matrice per vettore eseguito da ogni core è un vettore di 6 componenti**, in cui **ogni elemento corrisponde al prodotto scalare della metà degli elementi**. Di conseguenza, **uno dei due core**

dovrà poi sommare i risultati parziali, ossia semplicemente eseguire la somma dei due vettori risultanti.



III strategia (blocchi di righe-colonne)

Anche in questo caso consideriamo $n = 6$, ma numero di core uguale a 4, suddividendo la matrice per blocchi di righe-colonne.



Ogni core lavora col pezzo di x che gli serve

Con questa strategia **ogni core si occuperà del prodotto matrice per vettore tra una matrice 3×3 ed un vettore di 3 elementi**, che darà **in output un altro vettore, anch'esso di 3 elementi**.

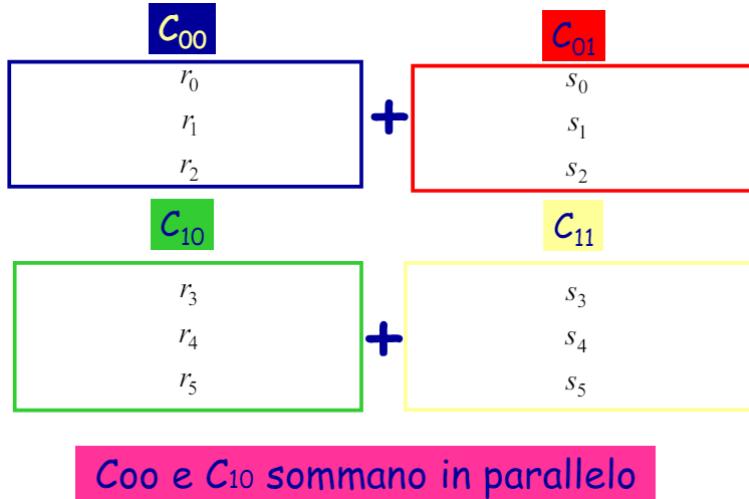
C_{00}	C_{01}
$a_{00} \cdot x_0 + a_{01} \cdot x_1 + a_{02} \cdot x_2$	$a_{03} \cdot x_3 + a_{04} \cdot x_4 + a_{05} \cdot x_5$
C_{10}	C_{11}
$a_{10} \cdot x_0 + a_{11} \cdot x_1 + a_{12} \cdot x_2$	$a_{13} \cdot x_3 + a_{14} \cdot x_4 + a_{15} \cdot x_5$
$a_{20} \cdot x_0 + a_{21} \cdot x_1 + a_{22} \cdot x_2$	$a_{23} \cdot x_3 + a_{24} \cdot x_4 + a_{25} \cdot x_5$

Calcolo dei prodotti parziali

Dopo che ogni core avrà eseguito il proprio prodotto matrice per vettore, **bisognerà sommare il prodotto parziale tra i core che hanno lavorato sulle stesse righe**

della matrice originale, quindi in questo caso il vettore di 3 elementi calcolato dal core C_{00} dovrà essere sommato al vettore calcolato dal core C_{01} , mentre il vettore calcolato dal core C_{10} dovrà essere sommato al vettore calcolato dal core C_{11} .

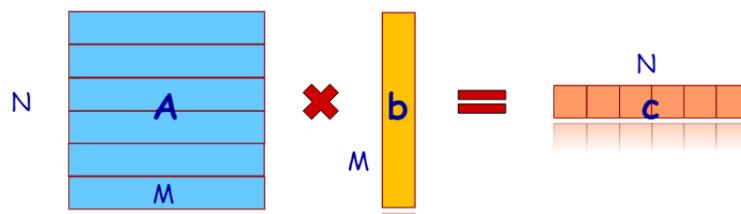
Queste somme possono essere eseguite in parallelo da più core, ad esempio nel nostro caso da 2 core.



Valutazione strategie 1-2 del prodotto matrice per vettore (Lez.21)

Per procedere alla valutazione delle prime due strategie del prodotto matrice per vettore, **consideriamo una matrice rettangolare**, ossia di **dimensioni NxM**, ed un vettore, ovviamente, di **M elementi**.

In sequenziale bisognerà effettuare **N prodotti scalari di lunghezza M**, ma per fare 1 prodotto scalare di lunghezza M in realtà **bisogna eseguire M moltiplicazioni più M-1 addizioni**.



In sequenziale, **N prodotti scalari di lunghezza M**.

Per fare 1 prodotto scalare di lunghezza M, devo fare:

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

Di conseguenza, **in sequenziale, il numero di operazioni da eseguire è pari a**

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

Considerando che il **tempo di esecuzione di una moltiplicazione è circa uguale a quello di un'addizione**, si può semplificare la complessità computazionale dell'algoritmo in **sequenziale come**

$$T_1(N \times M) = N[2M-1]$$

Calcolo di speed-up, overhead, efficienza ed isoefficienza per la I strategia (blocchi di righe)

Conoscendo come funziona la I strategia, ne viene di conseguenza che il **numero di righe della matrice assegnate ad ogni core equivale a N/p** , mentre tutto il resto rimane invariato.

p core:

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

In pratica **tutti i core**, in contemporanea, **eseguono N/p prodotti scalari di lunghezza M .**

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

Come possiamo osservare, questa I strategia permette di avere un algoritmo **full parallel**, poiché tutte le operazioni vengono eseguite parallelamente dai vari core.

Di conseguenza avremo **speed-up, overhead ed efficienza ideali**, come di seguito.

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

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

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

Ovviamente essendo tutto ideale, l'isoefficienza permette di constatare che **l'algoritmo è scalabile**, visto che avremo la solita forma indeterminata 0/0 che per convenzione può assumere qualunque valore, come ad esempio infinito.

p core/processori

$$Oh = 0 \rightarrow I(p_0, p_1, n_0) = 0/0 \text{ forma indeterminata}$$

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.

Speed-up ed efficienza del prodotto matrice per vettore I strategia caratterizzato con la legge di Ware-Amdahl

In questa strategia, **essendo l'algoritmo full-parallel, posso distinguere completamente la parte parallela da quella sequenziale, con quest'ultima che in effetti non c'è proprio.** Di conseguenza posso usare la formulazione della [legge di Ware-Amdahl base](#) per valutare speed-up ed efficienza.

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

Considerando che **in parallelo eseguiamo tutte le operazioni previste, la frazione $1 - \alpha$ di operazioni eseguite in parallelo è proprio uguale a 1**, visto che il numero di operazioni eseguite in sequenziale $\alpha = 0$. Andando a sostituire questi valori nel calcolo dello speed-up, avremo ovviamente speed-up ideale.

In parallelo:

1 fase (tutta parallela)

Calcolo prodotti parziali

$N/p [2M-1]$ operazioni

contemporaneamente

fatto da p processori/core

$p N/p [2M-1]$

delle $N[2M-1]$ operazioni



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

$$\alpha = 0$$

$$\frac{1 - \alpha}{p} = \frac{1}{p}$$



$$S(p) = \frac{1}{\frac{1}{p}} = p$$

Calcolo di speed-up, overhead, efficienza ed isoefficienza per la II strategia (blocchi di colonne)

Conoscendo come funziona la [II strategia](#), ne viene di conseguenza che il **numero di colonne della matrice assegnate ad ogni core equivale a M/p** , e di conseguenza anche il vettore deve essere suddiviso, tra i vari core, **con porzioni di lunghezza M/p** , per rispettare il fatto che per effettuare un prodotto matrice per vettore, il numero di colonne della matrice debba essere uguale al numero di elementi del vettore.

$$\dim[A^i] = Nx(M/p)$$

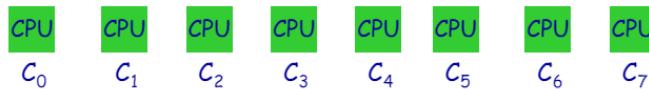
$$\dim[b_i] = M/p$$

p core

In pratica **tutti i core**, in contemporanea, eseguono **N prodotti scalari** di lunghezza **M/p**.

$$N [2M/p-1]$$

Come possiamo osservare, questa **II strategia non permette di avere un algoritmo full-parallel**, a differenza della I strategia, in quanto **ogni core avrà, come output del prodotto matrice per vettore, un vettore di lunghezza N**.



r_i vettori $i=0,7$, di lunghezza N

nell'unica memoria condivisa da sommare tra loro!!!



A questo punto, **per calcolare il risultato finale, si deve utilizzare una strategia per la collezione dei risultati**, quindi l'idea è quella di utilizzare una delle due strategie che venivano usate nel problema della [somma di N numeri](#).

Utilizzando la [I strategia](#) per la somma di N numeri, **ogni core somma, uno alla volta, i vettori calcolati dagli altri core** (componente per componente). In pratica, alle operazioni eseguite nella fase **parallel**a precedentemente calcolate, bisogna aggiungere il numero di operazioni per la **collezione dei vettori**, che equivale al **numero di core p-1 per la lunghezza di ogni vettore (N)**, che rappresenta il numero di somme da effettuare per ogni core.

Di seguito troviamo **speed-up, overhead ed efficienza** nel caso in cui utilizzassimo la **I strategia del problema della somma di N numeri per collezionare i vettori**.

$$\begin{aligned} S_p(N \times M) &= T_1(N \times M) / T_p(N \times M) = \\ &= N[2M-1] / (N [2M/p-1] + (p-1)N) \end{aligned}$$

$$\begin{aligned} Oh &= p T_p(N \times M) - T_1(N \times M) = \\ &= p (N [2M/p-1] + p-1 (N)) - N[2M-1] \end{aligned}$$

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

Utilizzando la **I strategia** per la collezione dei vettori e sviluppando il calcolo dell'overhead che ci servirà per valutare l'**isoefficienza**, osserviamo come esso (l'overhead) dipenda dal numero delle righe e dal numero di unità processanti, visto che noi agiamo sulle colonne.

I strategia per collezione vettori

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

Andando a calcolare l'isoefficienza notiamo come, **nel calcolo delle nuove dimensioni N e M, possiamo fissare il numero di righe, calcolando le colonne.**

I strategia per collezione vettori

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

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

Se invece utilizzassimo la [II strategia](#), considerando un numero di core $p = 8$, i core C_0, C_2, C_4, C_6 sommeranno il vettore di cui si sono occupati nella fase precedente con quelli calcolati dai core C_1, C_3, C_5, C_7 (componente per componente). Dopodichè, dimezzando ancora il numero di core attivi, C_0 e C_4 sommeranno il vettore di cui si sono occupati nella fase precedente con quelli calcolati dai core C_2 e C_6 (componente per componente). Infine, il core C_0 sommerà il vettore di cui si è occupato nella fase precedente con quello calcolato dal core C_4 (componente per componente). Per valutare il numero di operazioni eseguite in questo algoritmo parallelo, bisogna considerare, oltre alle operazioni precedentemente calcolate per la fase completamente parallela, anche le operazioni di collezione dei vettori da eseguire in fasi a parallelismo medio. Visto che ad ogni passo delle fasi a parallelismo medio dimezziamo il numero di core attivi, il numero delle operazioni si riduce a $\log_2 p$ (appunto il numero di core attivi ad ogni passo) per la lunghezza di ogni vettore (N), che rappresenta il numero di somme da effettuare per ogni core.

Di seguito troviamo speed-up, overhead ed efficienza nel caso in cui utilizzassimo la **II strategia del problema della somma di N numeri per collezionare i vettori**.

$$\begin{aligned}S_p(N \times M) &= T_1(N \times M) / T_p(N \times M) = \\&= N[2M-1] / (N [2M/p-1] + N \log_2(p))\end{aligned}$$

$$\begin{aligned}O_h &= p T_p(N \times M) - T_1(N \times M) = \\&= p (N [2M/p-1] + N \log_2(p)) - N[2M-1]\end{aligned}$$

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

Utilizzando la **II strategia** per la collezione dei vettori e sviluppando il calcolo dell'overhead che ci servirà per valutare l'**isoefficienza**, osserviamo come anch'esso (l'overhead) dipenda dal numero delle righe e dal numero di unità processanti, visto che noi agiamo sulle colonne.

II strategia per collezione vettori

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

Di conseguenza anche quando calcoliamo l'isoefficienza, **come nella I strategia** per la collezione dei vettori, notiamo come, **nel calcolo delle nuove dimensioni N e M, possiamo fissare il numero di righe, calcolando le colonne.**

II strategia per collezione vettori

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

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

Speed-up ed efficienza del prodotto matrice per vettore II strategia caratterizzato con la legge di Ware-Amdahl

Per calcolare lo speed-up con la legge di Ware-Amdahl possiamo utilizzare la **definizione base nel caso in cui si utilizzi la I strategia per la collezione dei vettori**, in quanto la fase parallela e quella sequenziale sono esattamente distinguibili, mentre c'è bisogno della **definizione generalizzata nel caso in cui si utilizzi la II strategia per la collezione dei vettori**, visto che abbiamo delle fasi a parallelismo medio.

Nel caso in cui si utilizzi la **I strategia**, si avrà una **prima fase completamente parallela**, nella quale verranno eseguite $N \left[\frac{2M}{p} - 1 \right]$ operazioni da ogni core (che sono p). Di conseguenza avremo che la **frazione di operazioni eseguite in parallelo** sul numero totale di operazioni da eseguire **equivale a**

In sequenziale:

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

I strategia per collezione vettori

In parallelo:

1 fase (tutta parallela)

Calcolo prodotti parziali

$N [2M/p - 1]$ operazioni

contemporaneamente

fatto da p processori/core

$p N [2M/p-1]$

delle $N[2M-1]$ operazioni

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

Nella fase sequenziale, invece, bisogna aggiornare $p-1$ volte il vettore presente nel primo core, commandogli, componente per componente, i vettori calcolati dagli altri core.

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

Avendo sia la frazione delle operazioni eseguite in parallelo che quella (la frazione) delle operazioni eseguite in sequenziale, possiamo calcolare lo speed-up.

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

Anche nel caso in cui si utilizzi la **II strategia**, si avrà una **prima fase completamente parallela**, nella quale verranno eseguite $N \left[\frac{2M}{p} - 1\right]$ operazioni da ogni core (che sono $p = 4$ nel seguente esempio). Di conseguenza avremo che la **frazione di operazioni eseguite in parallelo** sul numero totale di operazioni da eseguire **equivale a**

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

come prima, solo che ho sostituito $p=4$

Dopodichè avremo una **fase di parallelismo parziale** in cui lavora le metà dei core rispetto alla fase precedente, quindi

p=4

Nessuna fase in cui lavorano solo 3 processori/core

$$\alpha_3 = 0$$

Fase di parallelismo parziale (lavorano solo 2 processori/core):

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

solo le somme componenti per componente

Infine abbiamo un'ultima fase puramente sequenziale, nella quale un core si occupa di eseguire l'ultima somma componente per componente con un altro core.

p=4

Fase puramente sequenziale

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

A questo punto, per calcolare lo speed-up, bisogna mettere insieme il numero di operazioni eseguite dai core nelle varie fasi, ossia

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

che semplificando diventa

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

ed infine facendo il minimo comune multiplo

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

Direttive master e barrier (Lez.22)

Il costrutto **master** specifica che il blocco di istruzioni successivo verrà eseguito dal solo **master thread** (il processo principale insomma), mentre il costrutto **barrier** forza i **thread** ad attendere il completamento di tutte le istruzioni precedenti da parte di tutti gli altri **thread**. Per il costrutto barrier

bisogna comunque tenere conto del fatto che **alla fine del costrutto parallel esiste una barriera di sincronizzazione implicita.**

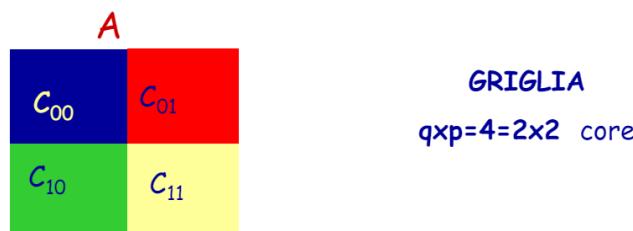
Valutazione strategia 3 del prodotto matrice per vettore (Lez.23-25-27)

la lez.24 era di laboratorio

Ricordando come funziona la strategia dei blocchi riga-colonna per quanto riguarda il problema del prodotto matrice per vettore, andiamo a fare delle valutazioni dal punto di vista computazionale.

Calcolo di speed-up, overhead ed efficienza per la III strategia (blocchi di righe-colonne)

Tenendo conto che la griglia di core potrebbe non essere sempre quadrata (quindi con numero di core riga e core colonna uguali), infatti potrebbe anche essere rettangolare (ad es. $q \times p = 8 = 2 \times 4$ core), **consideriamo il caso in cui abbiamo una griglia $q \times p = 4 = 2 \times 2$ core.**



Ovviamente per parallelizzare il prodotto matrice per vettore con questa strategia, **bisogna dividere anche il vettore, in modo che il numero di elementi del vettore sia uguale al numero di colonne di ogni blocco riga-colonna della matrice.**



12

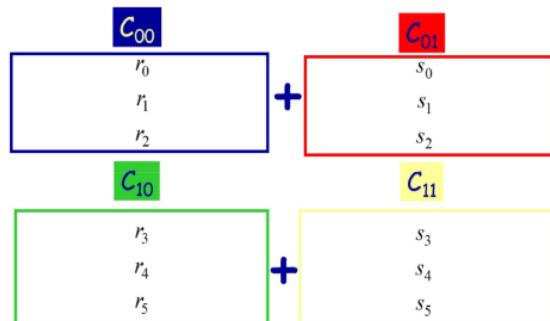
Ogni core esegue contemporaneamente N/q prodotti scalari di lunghezza M/p , e di conseguenza la complessità computazionale di questa strategia è

$$\dim[A_{loc}] = (N/q) \times (M/p)$$

$$\dim[b_{loc}] = M/p$$

$$N/q [2M/p-1] t_{\text{calc}}$$

A questo punto, considerando ogni blocco una matrice 3x3 (quindi con una matrice iniziale 6x6), ogni core avrà un vettore di lunghezza 3, che equivale alla metà del prodotto scalare di 3 righe della matrice con metà del vettore.



Bisogna ora **scegliere la strategia di collezione dei risultati**, la quale dipende dal valore di p. Infatti **se p è potenza di 2 allora** conviene utilizzare la **II strategia** (quella dell'albero), **altrimenti, ed in particolare se è dispari, conviene usare la I strategia** (sequenziale). C'è però da fare un'osservazione riguardo al **caso in cui p = 2**, infatti **I e II strategia**, in tal caso, **risultano equivalenti**, in quanto ogni core con colonna 0 aggiorna il proprio vettore con quello dei core corrispondenti di colonna 1, mediante N/q somme, che in questo caso equivalgono a N/2.

Consideriamo comunque di utilizzare, in questo caso, **la I strategia per la collezione dei risultati**.

$$C_{00} + C_{01}$$

$$C_{10} + C_{11}$$

I passo (e unico)

C_{00}, C_{10} aggiornano il proprio vettore effettuando N/q somme, cioè:

$$N/q = N/2$$

Di conseguenza la **complessità computazionale della fase di collezione dei risultati locali**, in generale, è uguale al numero di core colonna $p-1$ per il numero di righe di cui si occupa ogni blocco riga-colonna, ossia N/q .

$$(p-1) N/q$$

Questo perchè ovviamente se si avesse una griglia rettangolare di 2x3 core, ad esempio, si dovrebbero effettuare 2 volte N/q somme in parallelo.

A questo punto, ricordando che la **complessità computazione dell'algoritmo sequenziale** è

$$T_1(N \times M) = N[2M-1],$$

speed-up, overhead ed efficienza sono uguali a

$$S_{qxp}(NxM) = T_1(NxM)/T_{qxp}(NxM) = \\ = N[2M-1] / (N/q [2M/p-1] + N/q (p-1))$$

$$Oh = qxp T_{qxp}(NxM) - T_1(NxM) = \\ = pxq (N/q [2M/p-1] + N/q (p-1)) - N[2M-1]$$

$$E_{qxp}(NxM) = S_{qxp}(NxM) / (qxp) = \\ = N[2M-1] / [(qxp)(N/q [2M/p-1] + N/q (p-1))]$$

Supponiamo ora di avere una griglia di core rettangolare $qxp = 8 = 2 \times 4$.

Visto che il numero di core colonna p è una potenza di 2, conviene utilizzare la II strategia per la collezione dei risultati.

$$\begin{array}{ccccc} C_{00} & + & C_{01} & + & C_{02} \\ & & & & + C_{03} \\ C_{10} & + & C_{11} & + & C_{12} \\ & & & & + C_{13} \end{array}$$

Al primo passo, i core che avranno come indice di colonna un numero pari aggiornano il proprio vettore, effettuando N/q somme con il core che si occupa del blocco riga-colonna alla propria destra, ossia

$C_{00} C_{10} C_{02} C_{12}$ aggiornano con il proprio vettore effettuando N/q somme, cioè:

$$N/q = N/2$$

come abbiamo già detto.

Dopo aver eseguito il primo passo, rimane da effettuare un ulteriore passo di N/q somme, essendo rimasti 4 core.

$$\begin{array}{ccc} C_{00} & + & C_{02} \\ C_{10} & + & C_{12} \end{array}$$

Nel caso in cui si utilizzi la II strategia per la collezione dei risultati, quindi, la complessità computazionale sarà pari a

$$N/q \cdot \log_2(p)$$

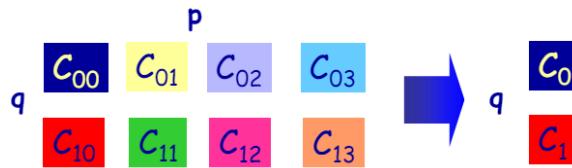
con **speed-up, overhead ed efficienza** che equivalgono a

$$S_{q \times p}(N \times M) = T_1(N \times M) / T_{q \times p}(N \times M) = \\ = N[2M-1] / (N/q [2M/p-1] + N/q \log_2(p))$$

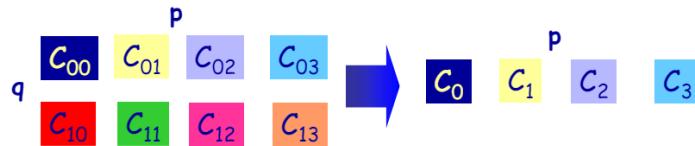
$$Oh = q \times p \cdot T_{q \times p}(N \times M) - T_1(N \times M) = \\ = p \times q (N/q [2M/p-1] + N/q \log_2(p)) - N[2M-1] \cdot t_{\text{calc}}$$

$$E_{q \times p}(N \times M) = S_{q \times p}(N \times M) / (q \times p) = \\ = N[2M-1] / [(q \times p)(N/q [2M/p-1] + N/q \log_2(p))]$$

La I e la II strategia per risolvere il problema del prodotto tra matrice e vettore possono essere ricavate come casi particolari dalla III strategia, rispettivamente quando $p = 1$



e quando $q = 1$



Isoefficienza per la III strategia (blocchi di righe-colonne)

Per il calcolo dell'isoefficienza è **necessario separare i conti tra la I e la II strategia per la collezione dei risultati.**

Nel caso in cui utilizzassimo la **prima strategia**, l'overhead è dato da

I strategia per collezione vettori

$$Oh = q \times p (N/q [2M/p-1] + N/q (p-1)) - N[2M-1] = \\ = 2NM - pN + p^2 N - pN - 2NM + N = \\ = p^2 N - 2pN + N,$$

ed è esattamente **uguale a quello della II strategia del prodotto matrice-vettore con la I strategia per la collezione dei risultati**. Come si può vedere, in questo caso, l'overhead dipende dal numero delle righe (N) e dal numero dei core lungo le colonne (p). **Anche l'isoefficienza**, di conseguenza, è **uguale a quella della II strategia con la I strategia per la collezione dei risultati**, ossia

$$I(q_0 p_0, q_1 p_1, N_0 M_0) = C = [N_1 (-2p_1 + p_1^2 + 1)] / [N_0 (-2p_0 + p_0^2 + 1)]$$

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

Nel calcolo delle nuove dimensioni N e M , come si può vedere, possiamo fissare il numero di righe, calcolando le colonne.

Anche utilizzando la II strategia per la collezione dei risultati notiamo che i conti sono uguali a quelli della II strategia con II strategia anche per quanto riguarda la collezione dei risultati.

L'overhead, anche in questo caso, dipende dal numero delle righe e dal numero dei core colonna.

$$\begin{aligned} \text{II strategia per collezione vettori} \\ Oh &= p (N [2M/p-1] + N \log_2(p)) - N[2M-1] = \\ &= 2M N - p N + p N \log_2(p) - 2M N + N = \\ &= -p N + p N \log_2(p) + N = \\ &= N (-p + p \log_2(p) + 1) \end{aligned}$$

Nel calcolo delle nuove dimensioni, come anche nel caso in cui si utilizzi la I strategia per la collezione dei risultati, si può fissare un qualunque numero di righe e calcolare il più opportuno numero di colonne.

$$I(q_0 p_0, q_1 p_1, N_0 M_0) = C = [N_1 (-p_1 + p_1 \log_2(p_1) + 1)] / [N_0 (-p_0 + p_0 \log_2(p_0) + 1)]$$

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

Calcolo di speed-up, overhead ed efficienza per la III strategia (blocchi di righe-colonne) quando righe e/o colonne non sono esattamente divisibili per il corrispondente numero di core

Andiamo ora a vedere cosa succede nel caso in cui il numero di righe e/o il numero di colonne della matrice non siano esattamente divisibili, rispettivamente, per il numero di core riga e/o per il numero di core colonna. Ovviamente, **in questi casi, alcuni core dovranno occuparsi di blocchi di matrice, e se serve del vettore, di dimensioni maggiori.**

Consideriamo il caso in cui il numero di righe non sia esattamente divisibile per il numero di core riga q . In particolare, ognuna delle righe che avanza, il cui numero totale equivale al resto della divisione, verrà assegnata ad un core riga che ha prima coordinata strettamente minore del resto. Per quanto riguarda il vettore, invece, non ci sarà alcun cambiamento, visto che non cambia nulla sul numero delle colonne. Se consideriamo di avere una matrice di dimensioni

$N \times M = 7 \times 6$, i core che hanno prima coordinata strettamente minore del resto si dovranno occupare di una matrice $4 \times 3 ((N/q + 1) \times (M/p))$, mentre gli altri avranno una matrice $3 \times 3 ((N/q) \times (M/p))$.

Esempio: se $\text{mod}(N, q) \neq 0$

A

A_{00}	A_{01}
A_{10}	A_{11}

GRIGLIA

$$q \times p = 4 = 2 \times 2$$

$$\dim[b_{\text{loc}}] = M/p$$

$$\dim[A_{10}] = (N/q) \times (M/p)$$

$$\dim[A_{11}] = (N/q) \times (M/p)$$

$$\dim[A_{00}] = (N/q + 1) \times (M/p)$$

$$\dim[A_{01}] = (N/q + 1) \times (M/p)$$

Dopo che **ogni core** avrà calcolato il prodotto scalare tra la propria matrice ed il vettore di dimensione M, **avrà come risultato un vettore di dimensione $N/q + 1$ nel caso in cui sia uno dei core con coordinata strettamente minore del resto, oppure N/q nel caso in cui non lo sia.**

$C_{00} \quad C_{01}$

$C_{10} \quad C_{11}$

$$C_{00}: \dim[r_{00}] = (N/q + 1), C_{01}: \dim[s_{01}] = (N/q + 1)$$

$$C_{10}: \dim[r_{10}] = (N/q), C_{11}: \dim[s_{11}] = (N/q)$$

La complessità computazionale nel caso di non esatta divisibilità del numero delle righe rispetto ai core riga aggiunge una riga in più, sia con la I che con la II strategia per la collezione dei vettori. In pratica si dovranno effettuare $N/q + 1$ prodotti scalari di lunghezza $2M/p - 1$ e poi utilizzare una delle due strategie di collezione dei vettori. Nella prima il numero di operazioni equivale a $(N/q + 1) * (p-1)$, mentre nella seconda è $(N/q + 1) * (\log_2 p)$. Ne consegue che lo speed-up sarà uguale a

I strategia per collezione vettori

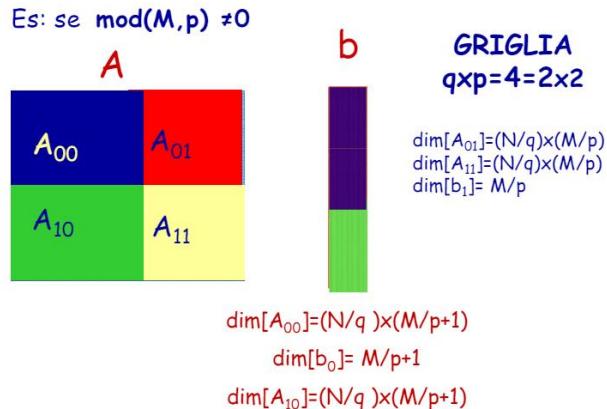
$$S_{q \times p}(N \times M) = T_1(N \times M) / T_{q \times p}(N \times M) = \\ = N[2M-1] / [\boxed{[N/q+1]} [2M/p-1] + \boxed{(N/q+1)(p-1)}]$$

II strategia per collezione vettori

$$S_{q \times p}(N \times M) = T_1(N \times M) / T_{q \times p}(N \times M) = \\ = N[2M-1] / [\boxed{[N/q+1]} [2M/p-1] + \boxed{(N/q+1) \log_2(p)}]$$

Consideriamo ora il caso in cui il numero di colonne non sia esattamente divisibile per il numero di core colonna p. In particolare, ognuna delle colonne che avanza, il cui numero totale equivale al resto della divisione, verrà assegnata ad un core colonna che ha seconda coordinata strettamente minore del resto.

In questo caso, però, anche per quanto riguarda il vettore ci sarà un **cambiamento**, infatti i core colonna che hanno seconda coordinata strettamente minore del resto, avranno anche un elemento in più del vettore. Se consideriamo di avere una matrice di dimensioni $N \times M = 6 \times 7$, i **core che hanno seconda coordinata strettamente minore del resto si dovranno occupare di una matrice $3 \times 4 ((N/q) \times (M/p + 1))$** , con vettore di dimensione 4 ($M/p + 1$), mentre gli altri avranno una matrice $3 \times 3 ((N/q) \times (M/p))$, con vettore di dimensione 3 (M/p).



A prescindere da se si tratti di core con seconda coordinata 0, che calcolano il prodotto scalare tra la propria matrice ed un vettore di dimensione $M/p + 1$, oppure con seconda coordinata 1, che calcolano il prodotto scalare tra la propria matrice ed un vettore di dimensione M/p , si avrà come risultato un vettore di dimensione N/q .

C_{00}	C_{01}
C_{10}	C_{11}

$P_{00}: \dim[r_{00}] = N/q, P_{01}: \dim[s_{01}] = N/q$
 $P_{10}: \dim[r_{10}] = N/q, P_{11}: \dim[s_{11}] = N/q$

La complessità computazionale nel caso di non esatta divisibilità del numero delle colonne rispetto ai core colonna aggiunge una colonna in più, sia con la I che con la II strategia per la collezione dei vettori. In pratica si dovranno effettuare N/q prodotti scalari di lunghezza $(2M/p + 1) - 1$ e poi utilizzare una delle due strategie di collezione dei vettori. Nella prima il numero di operazioni equivale a $(N/q) * (p-1)$, mentre nella seconda è $(N/q) * (\log_2 p)$. Ne consegue che lo speed-up sarà uguale a

I strategia per collezione vettori

$$S_{q \times p}(N \times M) = T_1(N \times M) / T_{q \times p}(N \times M) = \\ = N[2M-1] / (N/q [(2M/p+1)-1] + N/q (p-1))$$

II strategia per collezione vettori

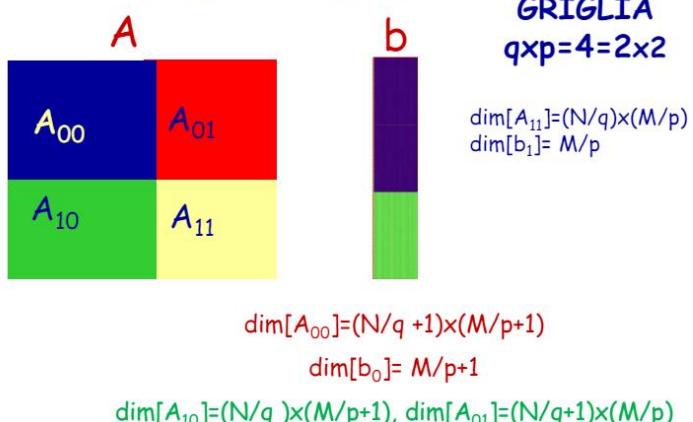
$$S_{q \times p}(N \times M) = T_1(N \times M) / T_{q \times p}(N \times M) = \\ = N[2M-1] / (N/q [(2M/p+1)-1] + N/q \log_2(p))$$

Consideriamo infine il caso in cui sia il numero di righe che il numero di colonne non siano esattamente divisibili, rispettivamente, per il numero di core

riga q e per il numero di core colonna p. In particolare, **ogni core con coordinate riga e colonna strettamente minori del resto riceverà una riga ed una colonna della matrice in più**, ma **anche un elemento del vettore in più**.

Se consideriamo di avere una matrice di dimensioni $N \times M = 7 \times 7$, **il core che ha coordinate riga e colonna strettamente minori del resto si dovrà occupare di una matrice 4×4 ($(N/q + 1) \times (M/p + 1)$)**, con vettore di dimensione 4 ($M/p + 1$), **il core che ha solo prima coordinata strettamente minore del resto si dovrà occupare di una matrice 4×3 ($(N/q + 1) \times (M/p)$)**, con vettore di dimensione 3 (M/p), **il core che ha solo seconda coordinata strettamente minore del resto si dovrà occupare di una matrice 3×4 ($(N/q) \times (M/p + 1)$)**, con vettore di dimensione 4 ($M/p + 1$), mentre l'ultimo avrà una matrice 3×3 ($(N/q) \times (M/p)$), con vettore di dimensione 3 (M/p).

Es: se $\text{mod}(N,q) \neq 0, \text{mod}(M,p) \neq 0$



Dopo che **ogni core** avrà calcolato i propri prodotti scalari, **si avrà come risultato un vettore di dimensione $N/q + 1$ nel caso in cui sia uno dei core con prima coordinata strettamente minore del resto, oppure N/q nel caso in cui non lo sia.**

C_{00}	C_{01}	$P_{00}: \dim[r_{00}] = N/q + 1, P_{01}: \dim[s_{01}] = N/q + 1$ $P_{10}: \dim[r_{10}] = N/q, P_{11}: \dim[s_{11}] = N/q$
C_{10}	C_{11}	

La complessità computazionale nel caso di non esatta divisibilità sia del numero delle righe che del numero delle colonne, rispetto ai core riga e ai core colonna, aggiunge una riga ed una colonna in più, sia con la I che con la II strategia per la collezione dei vettori. In pratica **si dovranno effettuare $N/q + 1$ prodotti scalari di lunghezza $(2M/p + 1) - 1$ e poi utilizzare una delle due strategie di collezione dei vettori**. Nella prima il numero di operazioni equivale a $(N/q + 1) * (p-1)$, mentre nella seconda è $(N/q + 1) * (\log_2 p)$. Ne consegue che lo speed-up sarà uguale a

$$\begin{aligned} S_{q \times p}(N \times M) &= T_1(N \times M) / T_{q \times p}(N \times M) = \\ &= N[2M-1] / (N/q [(2M/p+1)-1] + N/q (p-1)) \end{aligned}$$

$$\begin{aligned} S_{q \times p}(N \times M) &= T_1(N \times M) / T_{q \times p}(N \times M) = \\ &= N[2M-1] / (N/q [(2M/p+1)-1] + N/q \log_2(p)) \end{aligned}$$

Tutti i conti fatti per la III strategia nel caso in cui il numero di righe non sia esattamente divisibile per il numero di core riga e/o il numero di colonne non sia esattamente divisibile per il numero di core colonna **possono essere proiettati per la I strategia, quando il numero di core colonna è 1, e per la II strategia, quando il numero di core riga è 1.**

Isoefficienza per la III strategia (blocchi di righe-colonne) quando righe e/o colonne non sono esattamente divisibili per il corrispondente numero di core

Per valutare l'isoefficienza, in questo caso specifico, si dovrebbero rifare i conti, ma come ogni caso in cui il numero di righe e/o colonne non sia esattamente divisibile per il corrispondente numero di core, **la scalabilità è compromessa**.

Calcolo dello speed-up con la definizione di Ware-Amdahl generalizzata

Nella III strategia per calcolare il prodotto matrice-vettore **si ha una fase a parallelismo medio** e, di conseguenza, **non è possibile utilizzare la definizione di Ware-Amdahl base**.

Sapendo che le **operazioni totali da eseguire** in sequenziale sono

$$T_1(NM) = N[2M-1] \text{ operazioni},$$

nella III strategia **si avrà una prima fase tutta parallela** di calcolo dei prodotti parziali, **nella quale si effettuano**

contemporaneamente
fatto da $q \times p$ processori/core
 $q \times p \frac{N}{q} [2M/p-1]$
delle $N[2M-1]$ operazioni

Di conseguenza, **la frazione di operazioni da eseguire quando si utilizzano $q \times p$ core è**

$$\alpha_{q \times p} = q \times p \frac{\frac{N}{q} [\frac{2M}{p} - 1]}{N[2M-1]}$$

A questo punto, per procedere con i calcoli, bisogna fissare dei valori per q e p, che dipendono da come viene impostata la griglia di core.

Consideriamo di avere una situazione con $q \times p = 2 \times 3 = 6$ core, dove utilizziamo come strategia per la collezione dei risultati la prima, visto che il numero di core non è una potenza di 2.



Andando a sostituire alla formula vista in precedenza i valori q e p, avremo che **nella fase completamente parallela il numero di operazioni è**

$$\alpha_6 = 6 \frac{\frac{N}{2} \left[\frac{2M}{3} - 1 \right]}{N[2M - 1]}$$

Dopodichè osserviamo come **non esista nessuna fase in cui lavorano contemporaneamente 5, 4 e 3 core**,

$$\alpha_5 = \alpha_4 = \alpha_3 = 0,$$

infatti la fase di parallelismo medio che esiste è quella in cui lavorano 2 core, che eseguono 2 somme di vettori parziali ciascuno (2 che è dato da $p-1$ della prima strategia di collezione dei risultati).

$$\alpha_2 = 2 \frac{(3-1) \frac{N}{2}}{N[2M-1]}$$

Infine **non esistono operazioni svolte in sequenziale**, ossia in cui lavora un unico processore.

$$\alpha_1 = 0$$

Per essere sicuro di aver calcolato tutto nel modo giusto, devo sommare tutti i numeratori e ottenere un numero uguale al denominatore comune, che equivale al numero totale delle operazioni da eseguire.

Consideriamo ora di avere una situazione in cui abbiamo $q \times p = 2 \times 4 = 8$ core, dove utilizziamo come strategia per la collezione dei risultati la seconda, visto che il numero di core è una potenza di 2.



Anche in questo caso andiamo a sostituire alla formula vista in precedenza i valori di q e p, avendo come **numero di operazioni da eseguire nella fase totalmente parallela**

$$\alpha_8 = 8 \frac{\frac{N}{2} \left[\frac{2M}{4} - 1 \right]}{N[2M - 1]}$$

In questo caso **non esiste nessuna fase in cui lavorano contemporaneamente 7, 6, 5 e 3 core**

$$\alpha_7 = \alpha_6 = \alpha_5 = 0$$

$$\alpha_3 = 0,$$

poichè le fasi di parallelismo medio sono quelle in cui lavorano 4 e successivamente 2 core.

Nella fase in cui lavorano 4 core, ognuno di essi (di core) **esegue la somma parziale con il proprio vicino**, mentre **nella fase in cui lavorano 2 core** ognuno **esegue un'ulteriore somma con l'altro core della propria riga che al passo precedente aveva effettuato una somma parziale.**

$$\alpha_4 = 4 \frac{\frac{N}{2}}{N[2M - 1]}$$

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

Anche in questo caso, ovviamente, **non esiste nessuna fase in cui lavora un unico core.**

$$\alpha_1 = 0$$

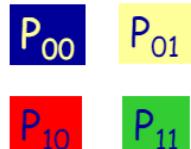
Esempio di calcolo dello speed-up con la definizione generalizzata

Consideriamo di avere una matrice di dimensioni **7x9**, quindi **con un vettore di 9 elementi**. Il numero di operazioni in sequenziale, che ricordiamo essere **$N[2M-1]$** , è, in questo caso,

$$7[2 \cdot 9 - 1] = 119 \text{ operazioni}$$

Consideriamo, inoltre, di avere una griglia di core $q \times p = 2 \times 2 = 4$.

$$4 = q \times p = 2 \times 2 \quad p = 2$$



Andiamo ora a calcolare le dimensioni che ogni core deve trattare

$$\dim[A_{00}] = (7/2+1) \times (9/2+1)$$

$$\dim[b_0] = 9/2+1$$

$$\dim[A_{10}] = (7/2) \times (9/2+1),$$
$$\dim[A_{01}] = (7/2+1) \times (9/2)$$

$$\dim[A_{11}] = (7/2) \times (9/2)$$
$$\dim[b_1] = 9/2$$

ossia

$$\dim[A_{00}] = 4 \times 5$$

$$\dim[b_0] = 5$$

$$\dim[A_{10}] = 3 \times 5, \dim[A_{01}] = 4 \times 4$$

$$\dim[A_{11}] = 3 \times 4$$
$$\dim[b_1] = 4$$

Visto che ognuno dei core effettua un prodotto matrice-vettore con strutture di diverse dimensioni, **la frazione di operazioni da eseguire nella fase puramente parallela si calcola come**

$$\alpha_4 = \frac{1 \cdot 4[2 \cdot 5 - 1] + 1 \cdot 4[2 \cdot 4 - 1] + 1 \cdot 3[2 \cdot 5 - 1] + 1 \cdot 3[2 \cdot 4 - 1]}{119}$$

Non vi è nessuna fase in cui lavorano 3 processori, mentre con 2 processori si ha la fase a parallelismo medio, con la scelta della strategia di collezione dei risultati che in questo caso è indifferente poiché verrà comunque eseguita un'unica somma dei vettori parziali da parte di ogni core.

$$\alpha_2 = \frac{1 \cdot 4 + 1 \cdot 3}{119} = \frac{7}{119}$$

Come sappiamo **non c'è nessuna fase sequenziale**, quindi per controllare di aver fatto bene tutti i calcoli bisogna sommare i numeratori delle varie fasi e vedere se il risultato combacia con il denominatore.

$$\alpha_4 = \frac{112}{119} \quad \alpha_3 = 0 \quad \alpha_2 = \frac{7}{119} \quad \alpha_1 = 0$$

In generale è importante ricordare **come in una griglia qxp = 8 core, la versione 4x2 è migliore della 2x4**, come dimostrerebbero i calcoli se li facessimo. Per esercitarsi svolgere lo stesso esercizio appena visto ma con griglia di core **qxp = 4x2 = 8**.

Problema SAXPY (Lez.26)

Il problema SAXPY prende in input una matrice **A** di dimensioni mxn, due vettori **a** e **b** di dimensioni, rispettivamente, m e n e due scalari **α** e **β** appartenenti all'insieme dei numeri naturali (numeri interi maggiori di 0).

Il risultato di questo problema è uno scalare r dato dalla produttoria degli elementi del vettore risultante dalla seguente operazione

$$r = \alpha A \cdot b + \beta \cdot a;$$

con la produttoria definita come

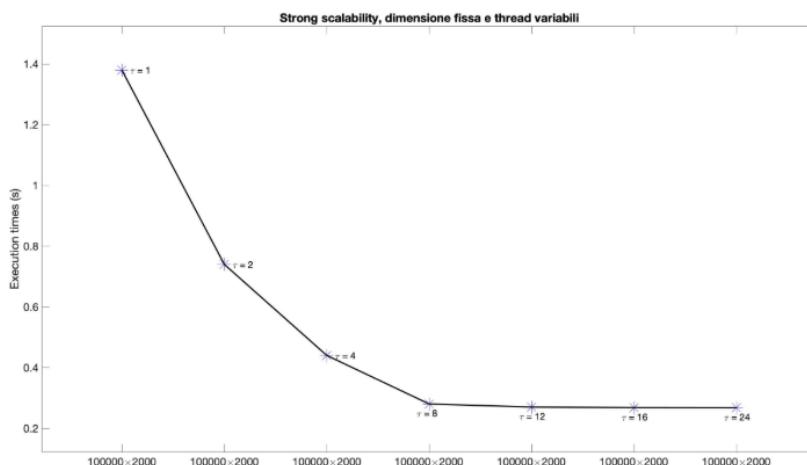
$$\prod_{i=0}^{m-1} r_i$$

Strong e weak scalability del problema SAXPY

Ricordando la definizione di strong e weak scalability, andiamo ad analizzare prima l'una e poi l'altra.

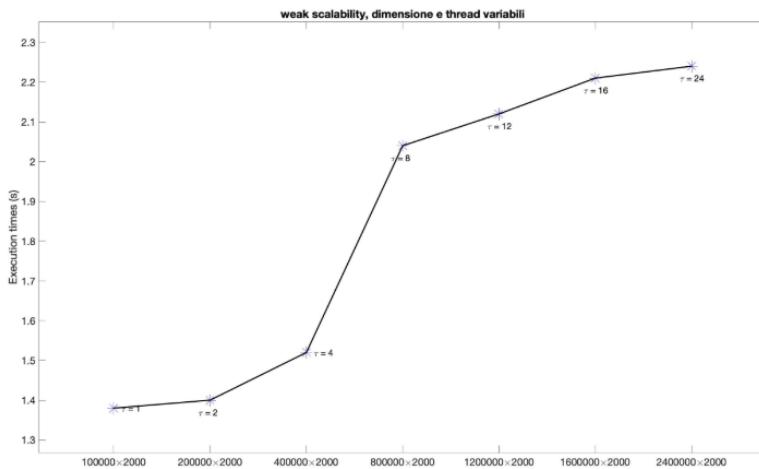
Per la prima si può notare come all'aumentare del numero di processori il tempo di esecuzione diminuisca, fino a stabilizzarsi ad un certo valore.

m	n	τ	Execution Time (s)
1×10^5	2×10^3	1	1.39
1×10^5	2×10^3	2	0.74
1×10^5	2×10^3	4	0.44
1×10^5	2×10^3	8	0.28
1×10^5	2×10^3	12	0.27
1×10^5	2×10^3	16	0.27
1×10^5	2×10^3	24	0.27

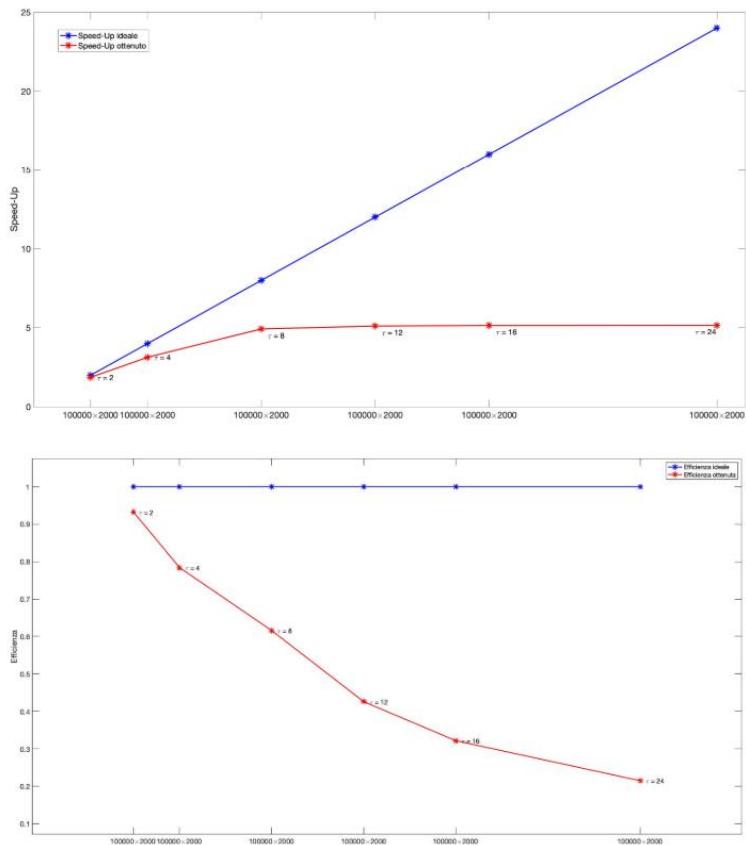


Nella **weak scalability**, invece, il tempo di esecuzione, aumentando sia la dimensione m che i processori, aumenta.

m	n	τ	Execution Time (s)
1×10^5	2×10^3	1	1.39
2×10^5	2×10^3	2	1.40
4×10^5	2×10^3	4	1.51
8×10^5	2×10^3	8	2.02
12×10^5	2×10^3	12	2.10
16×10^5	2×10^3	16	2.21
24×10^5	2×10^3	24	2.24



Di seguito troviamo **speed-up ed efficienza ottenuti**, rispetto a quelli ideali.



Costrutti critical e atomic

Il costrutto critical (#pragma omp critical) forza l'esecuzione del blocco successivo ad un thread alla volta (come farlo in sequenziale).

E' utile per gestire le **regioni critiche**, in modo da evitare che vi siano race condition dovute all'accesso/modifica da parte di vari thread di variabili condivise.

Anche il costrutto atomic funge più o meno nello stesso modo, infatti **entrambi i costrutti assicurano un accesso in mutua esclusione**. La differenza sta nel fatto che il costrutto **critical produce un overhead elevato a causa del meccanismo di lock a livello user**, mentre il costrutto **atomic ne produce una quantità ridotta visto che il meccanismo di lock è implementato a livello hardware**.

Facciamo ora un **confronto riguardo questi due costrutti**, prendendo in considerazione il calcolo del numero di operazioni eseguite in parallelo nel problema SAXPY precedentemente visto. Si può facilmente notare come il tempo di esecuzione del calcolo del numero di operazioni con il costrutto atomic sia di gran lunga minore rispetto a quello del costrutto critical.

m	n	τ	Critical Ex. Time (s)	Atomic Ex. Time (s)
1×10^5	2×10^3	1	7.06	3.28
1×10^5	2×10^3	2	21.56	10.16
1×10^5	2×10^3	4	38.92	11.47
1×10^5	2×10^3	8	71.99	13.17
1×10^5	2×10^3	12	74.96	12.59
1×10^5	2×10^3	16	76.04	12.43
1×10^5	2×10^3	24	70.93	7.16