

Esercitazioni di Calcolo Parallelo

Paolo Avogadro

DISCo, Università di Milano-Bicocca
U14, Id&aLab T36
paolo.avogadro@unimib.it
Aula Lezione T014,
edificio U14

- Martedì 15:30-17:30
- Mercoledì 10:30-12:30

Intro: openMP

- **openMP** = **open** specifications for **M**ulti **P**rocessing
- **openMP** standard nato nel 1997 per il FORTRAN e portato poi a C/C++
- **openMP** e' una **API** per ambienti **multi-thread memory sharing**
- **openMP** definisce:
 - direttive per il compilatore
 - una libreria di funzioni a runtime
 - variabili d'ambiente
- per la **gioia** di grandi e piccoli, openMP e' uno **standard** non una **implementazione**, per questo le implementazioni (per esempio GNU e Intel) possono avere dei comportamenti differenti.
- **openMP** lascia al programmatore il controllo dell'esistenza di **deadlock** e comportamenti **non-deterministici**
- **openMP** non definisce particolari standard di I/O

Un sito di riferimento: <https://computing.llnl.gov/tutorials/openMP/#PRIVATE>

da MPI a openMPI

Visto che abbiamo un'idea di cosa sia MPI, ha senso prendere questo standard come paragone per poter capire cosa sia openMP

- Grossolanamente parlando l'idea dove nasce e': **MPI** ↔ **NUMA** (Non Uniform Memory Access) le **single cpu** accedono solo alla **propria** area di memoria
- con la medesima grossolanità: **openMP** ↔ **UMA** (Uniform Memory Access) tutte le cpu hanno accesso ad una area di memoria condivisa

In realtà le topologie delle memorie sono complicate e, su una macchina NUMA, un core può accedere alla memoria privilegiata di un altro core con costo di latenza (non è quindi più UMA ma openMP funge, perché i processi accedono alla stessa memoria RAM (o anche cache in alcuni casi) solo con tempi diversi).

Nel caso di **MPI**, non esistendo una memoria condivisa, si deve definire un protocollo di comunicazione tra processi. **openMP** viene invece pensato per un ambiente dove la **comunicazione** tra varie unità di calcolo avviene **accedendo alla memoria condivisa**, servono quindi dei protocolli per definire questi accessi.

- ricordiamo che **MPI** si basa (prevalentemente) su una logica **SPMD** (un solo codice gira ed elabora dati differenti)
- anche **openMP** segue il paradigma **SPMD** in cui però è essenziale il concetto di **work sharing** in cui i singoli thread eseguono parti differenti del lavoro.

Un paragone introduttivo tra MPI e openMP:

MPI
tante CPU girano **sempre**
i **processi** necessitano di un **protocollo** per comunicare
il codice va **progettato** parallelo

openMP
utilizzo CPU solo alla **bisogna**
i **thread** accedono alla stessa **memoria**
parallelizzazione progressiva

Cos'è un thread? (differenze rispetto ad un processo)

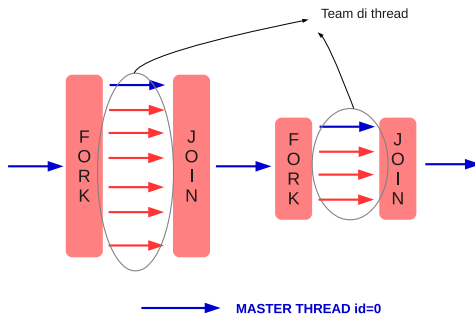
- (ricordi di MPI...) un **processo** è un'istanza di esecuzione del programma (ergo: un processo fa girare tutto un codice)
- un **thread** (o **instruction stream**) è la più piccola unità di calcolo che può essere gestita dal sistema (un thread spesso fa girare solo una **piccola** parte di un codice)
- in **alcune parti** di un **processo** possono girare vari **thread** (non è vero il contrario)
- i thread di un processo possono **condividere la memoria** assegnata al processo.
- invece processi differenti, in genere, non condividono* le risorse.

*in realtà alcune risorse possono essere condivise tra processi differenti, per esempio la memoria di massa.

Fork

L'idea di openMP e' che:

- il codice gira su una macchina con molte "cpu/core" con memoria condivisa
- la parte seriale di codice non ha senso che venga fatta girare su tutte le cpu! (**economia**)
- esiste un **master thread** che fa girare il codice
- al momento della **bisogna** (per esempio quando c'e' un ciclo `for` che puo' beneficiare della parallelizzazione) si inserisce l'apposito costrutto di **openMP**.
- a questo punto c'e'un **fork**: dal **master thread** si crea un **team** di thread che eseguono il calcolo parallelo (in realta' si ha concurrency, o potenziale parallelismo).
- alla fine della sessione parallela tutti i thread tranne il master cessano di esistere e il calcolo riprende in modo seriale, questo viene chiamato una: **join** (dove vi e' una barriera implicita).

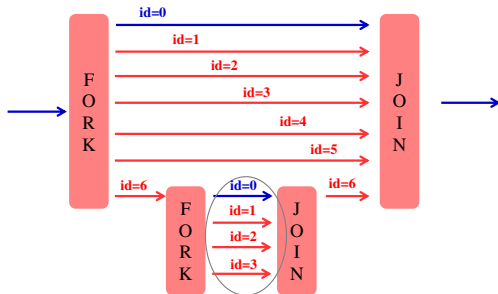


Fork: parallelismo innestato

- E' possibile fare creare un **nuovo team** da un thread di una regione parallela
- Attenzione pero': questo comportamento di default e' **spento** (non avviene nessun fork se si chiama un costrutto parallelo in un costrutto parallelo).
- per **accendere** il nested parallelism si puo' settare una variabile d'ambiente: `OMP_NESTED = true`
- oppure usare la seguente funzione a runtime: `omp_set_nested(1)`

Nested Parallelism

MASTER THREAD id=0



termini

- **parallel region**: parte di codice dove puo' agire piu' di un thread
- una regione parallela e' considerata **attiva** se vi agisce piu' di un thread (se c'e' solo il thread master, la regione non e' attiva).
- a tutti i thread viene assegnato un numero intero di identificazione progressivo chiamato **id** (e' il cugino del rank di MPI)
- il master thread ha **id=0**
- tutti i comandi (**statement**) in una regione parallela sono eseguiti da **tutti** i thread nella regione parallela
- quando **tutti** i thread hanno completato il loro compito si arriva alla fine della regione parallela: c'e' un **JOIN**.
- tutti i thread tranne il master cessano di esistere.

Per compilare e fare girare un codice

- Per un codice in *C* serve un header iniziale: `#include <omp.h>`
- e' possibile (**NON NECESSARIO**), definire delle variabili d'ambiente, che contengano, per esempio il numero di thread che si vuole fare "creare" di default al momento dei **fork**.
- al momento della compilazione con `gcc` si deve usare la flag: `-fopenmp`, per esempio:
`gcc -fopenmp miocodice.c -o miocodice.x`
- per il compilatore `icc`: `icc -qopenmp miocodice.c -o miocodice.x`
- la flag con il compilatore `pg` e': `pgcc -mp miocodice.c -o miocodice.x`
- il codice ottenuto, viene poi lanciato semplicemente con `./miocodice.x` (non si ha una modalita' di lancio particolare, come invece c'era in MPI).

Ciao Mondo!

Il primo codice openMP:

```
#include <omp.h> // header necessario per openMP
#include <stdio.h>
void main()
{
    #pragma omp parallel num_threads(5) // COSTRUTTO parallelo di openMP: c'e' il FORK
    {
        int id; // NON METTERE la GRAFFA sulla linea di #pragma
        id = omp_get_thread_num(); // definisco intero per l'id
        printf("Ciao Mondo, sono il thread (%d)\n", id ); // funzione INTRINSECA di openMP
    } // fine regione parallela JOIN
    printf(" Riciao Mondo\n" ); // Quante volte verra' scritto "Riciao"?
}
```

- # pragma omp sta per *pragmatic information* del tipo **openMP** e serve al compilatore che, nel momento della compilazione, sa di dover fare qualcosa (descritto nel seguito della "frase").
- la **direttiva** parallel dice al compilatore di fare un **FORK** e creare un team di thread (in questo caso, la clausola (clause) num_threads(5) definisce che i thread di **questa** regione parallela devono essere 5).
- Ci possono essere comandi #pragma diversi da quelli omp, in altri ambiti.
- **Occhio**: se si apre la parentesi graffa che definisce il costrutto parallelo sulla stessa linea di #pragma omp parallel si ottiene un **errore**
- **Altro occhio**: se si dimentica di usare la clausola parallel, il fork non avviene e c'e' solo il master thread.

Le direttive

- i comandi sono Case sensitive (in C)
- si può specificare un **solo** nome della direttiva (p.es. `parallel`, `for`,...) (**in realtà** in alcuni casi si hanno delle scorciatoie p.es.: `parallel for` sulla stessa riga è valido)
- dopo il nome della direttiva si possono aggiungere delle *clausole* (p.es. `shared(i)` identifica che la variabile `i` è `shared`)
- ogni direttiva si applica al più ad un blocco strutturato; in un blocco strutturato c'è un solo ingresso e una sola uscita, p.es. non si può entrare/uscire con `goto!` (ma si può uscire con un `exit()` in C o `stop` per il FORTRAN).
- se la direttiva è lunga si può **andare a capo** con `\` e continuare la direttiva.

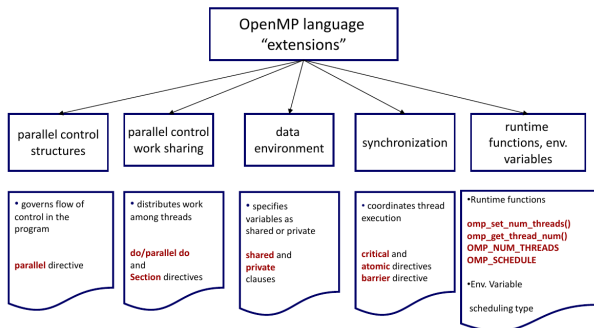
Esempio della struttura di una direttiva:

```

INTESTAZIONE   NOME della DIRETTIVA   Clausole...
#pragma omp      parallel                 shared(i) private (j)

```

Alla fine della direttiva si deve **andare a capo!!!!!!!!!!** (senza ":" in C)



Alcune routine (funzioni) a runtime di openMP

Alcune utili funzioni definite da *openMP* (queste **non** vanno messe dopo un `#pragma`):

- `int omp_get_num_threads();` numero di thread nel team (equivalente alla **size** di MPI nella regione parallela)
- `int omp_get_thread_num();` la funzione restituisce l'**id** del thread chiamante (sotto forma di int) (equivalente del **rank** per i processi di MPI), per esempio: `mioID=omp_get_thread_num();`
- `double omp_get_wtime();` wall time, per esempio: `t1 = omp_get_wtime();`
- `void omp_set_num_threads();` definisce il numero di thread nei successivi costrutti paralleli (a meno che, nel frattempo, intervengano altre specificazioni sul numero di thread).
- `get_num_procs();` trova il numero di CPU presente nella macchina
- `int omp_in_parallel();` restituisce un valore intero che e' 0 se **non** siamo in una regione parallela **attiva**. Restituisce invece 1 se viene chiamato dall'interno di una regione parallela **attiva**.

Quanti thread?

Ci sono vari modi di definire il numero di thread in un blocco parallelo, per esempio:

```
omp_set_num_threads(4); // <= si definisce il numero prima del blocco
//.... altro codice
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf("Io sono: %d", ID);
}
```

Oppure si può definire al momento della creazione della direttiva:

```
int vaiParallelo = 0;
#pragma omp parallel if (vaiParallelo == 1) num_threads( 8) shared (var_b) default ( none)
{ // per fare sì che num_threads( 8) venga preso, meglio mettere vaiParallelo=1;
    int ID = omp_get_thread_num();
    printf("Io sono: %d", ID);
}
```

In ordine di importanza il numero di thread è determinato nel seguente modo (più lontano = meno importante) :

- 1 valutazione della clausola `if` con di seguito il valore all'interno della clausola `num_threads()`
- 2 Utilizzo della funzione `omp_set_num_threads()` (prima del blocco parallelo)
- 3 il valore della variabile d'ambiente `OMP_NUM_THREAD` (settata precedentemente)
- 4 default della implementazione (numero di core di un nodo)

I thread sono identificati da un `id` che va da 0 (per il `Master`) a N-1 (N = numero thread della regione parallela).

Come comunicano i thread? Data sharing

Quando c'è una sessione parallela bisogna definire come vengono gestite le risorse (le variabili, gli array...). In particolare è importante sapere **quale** thread ha accesso a **quale** risorsa, qui indichiamo alcune delle clausole principali:

- `private (i)` **ogni** thread ha la propria versione privata della variabile (non c'è race condition), indipendente da tutte le altre copie. La variabile è **inizializzata** secondo l'inizializzazione standard (come nel main). **Domanda**: può essere un puntatore?
- `shared (v,u)` indica che gli oggetti `v` e `u` possono essere usati da **tutti i thread** all'interno della regione parallela (può causare race condition). Gli oggetti sono condivisi con il resto del codice.
- `firstprivate(i)` è identico a `private` ma il valore di `i` è inizializzato con l'ultimo valore posseduto prima del costrutto parallelo (ergo del master thread).
- `default(shared)` significa che le variabili esterne diventano `shared` nel costrutto parallelo (è il default per un costrutto parallelo).
- `default(private)` indica che tutte le variabili, array sono `private` all'interno del costrutto
- Esiste anche `default(none)`: richiede di specificare per ogni variabile il suo attributo di data sharing: è una **buona pratica** di programmazione!
- `lastprivate(j)` all'uscita dalla regione parallela (JOIN) la variabile viene aggiornata per il master thread (che continua ad esistere) con l'ultimo valore della regione parallela (spiegata più approfonditamente nel seguito).

Attenzione

- 1 le variabili definite **prima** del costrutto `parallel` sono **automaticamente shared**
- 2 Le variabili definite all'**interno** del costrutto `parallel` sono **automaticamente private**

Attenzione

costrutti paralleli differenti (p.es. `#pragma omp task` definiscono in modo diverso dal costrutto `parallel` quali sono le caratteristiche di default delle variabili all'interno della loro regione.

Esempio: shared e private di default

Cosa scrive a video il computer?

```
#include <stdio.h>
#include <omp.h>

int main(){
    int a=5; // a viene definita PRIMA del costrutto parallel, nel costrutto parallelo e' shared
    #pragma omp parallel num_threads(3)
    {
        int id; // id e' definita dentro il costrutto e' PRIVATA di default
        id = omp_get_thread_num();
        printf("thread: %d, a=%d\n", id, a);
    }
}
```

```
thread: 0, a=5
thread: 2, a=5
thread: 1, a=5
```

Se invece avessi inserito:

```
#pragma omp parallel num_threads(3) private(a)
```

il risultato sarebbe stato:

```
thread: 1, a=0
thread: 2, a=0
thread: 0, a=0
```

Attenzione: nelle versioni piu' recenti di openMP le variabili private sono inizializzate automaticamente come lo sarebbero nel main. Nelle versioni piu' vecchie invece non sono inizializzate! Fidarsi dell'inizializzazione di una clausola `private` e' **pericoloso!**

esempio: firstprivate

Cosa scrive a video questo codice?

```
#include <stdio.h>
#include <omp.h>

int main()
{
    int b=5;
    printf("Prima del costrutto parallelo b=%d\n", b);
    #pragma omp parallel num_threads(3) firstprivate(b)
    {
        int id;
        id = omp_get_thread_num();
        b = b+id;
        printf("thread: %d, b=%d\n", id, b);
    }
    printf("Dopo il costrutto parallelo b=%d\n", b);
}
```

Soluzione:

```
Prima del costrutto parallelo b=5
thread: 0, b=5
thread: 2, b=7
thread: 1, b=6
Dopo il costrutto parallelo b=5
```

problemi di default...

Trovare il problema:

```
#include <stdio.h>
#include <omp.h>

int main() {
    int a=5;
    #pragma omp parallel num_threads(3) default(none)
    {
        printf("thread: %d, a=%d\n", omp_get_thread_num(), a);
    }
}
```

Soluzione:

- La variabile `a` e' dichiarata e definita prima della regione parallela.
- Se non ci fosse un `default(none)` nella regione parallela verrebbe trattata come `shared`.
- In questo caso invece la clausola `default(none)` crea problemi al compilatore: non sa come trattare `a`!
- Per risolvere il problema basta aggiungere una clausola tipo `shared(a)`, `private(a)` ... per esempio:

```
#pragma omp parallel num_threads(3) default(none) private(a)
```

- l'uso di `default(none)` e' una **buona pratica** di programmazione, perche' ci obbliga a dichiarare esplicitamente il possesso delle variabili

Cominciamo Bene...

Un esempio di codice volenteroso ma problematico:

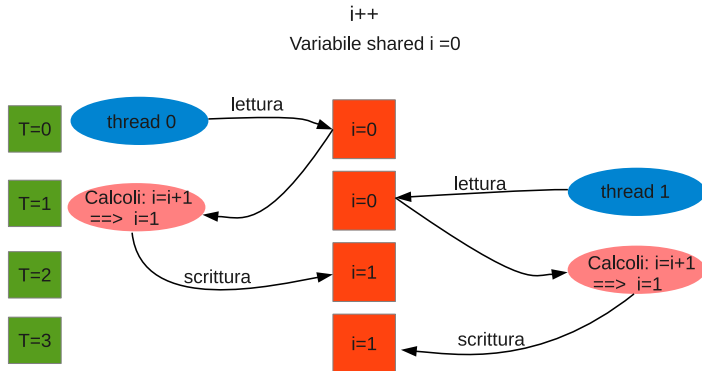
```
#include<omp.h>
#include<stdio.h>
int main()
{
    int i=0,j=0, imax=10, tanto=1000000;
    omp_set_num_threads(imax);
    #pragma omp parallel private (j)
    {
        for (j=0;j<tanto; j++) // questo serve per aumentare i problemi...
            i = i +1;
    }
    printf("teorica= %d, somma con i thread= %d \n", imax*tanto, i);
}
```

Sto utilizzando i thread, ma come?

Problemi

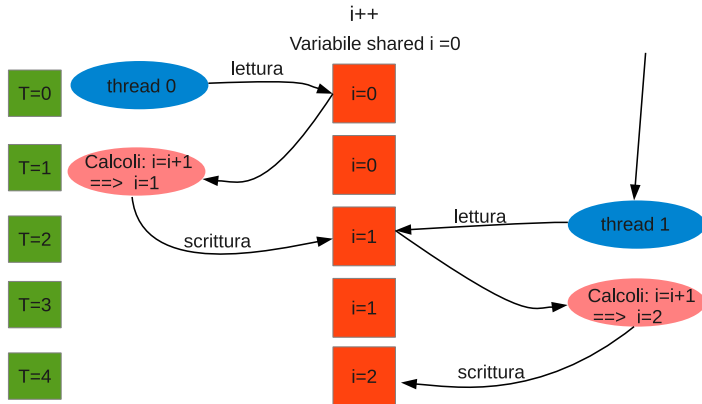
- Tutti i thread cercano di aggiornare la variabile *i*.
- la variabile *i* e' condivisa da tutti i thread: l'aggiornamento genera una **race condition**
- **Attenzione** di solito la race condition **non sussiste** ... se eseguo in locale con una sola cpu, dato che in questo modo i thread vengono simulati e quindi eseguono uno dopo l'altro! Il ciclo su *j* e' stato messo apposta per aumentare la probabilita' che succeda.
- **Occhio** quindi un codice che funziona perfettamente su una macchina virtuale potrebbe essere completamente bacato!

visualizziamo una Race Condition (datarace)



Alla fine il valore della variabile i viene aggiornato dal thread 1, che **NON** tiene conto dei nuovi calcoli fatti dal thread 0.

visualizziamo una Race Condition 2



Fortunatamente, questa volta il thread 1 legge la variabile i **DOPO** che il thread 0 ha aggiornato il valore. Il risultato dipende da quando i thread accedono alla variabile

Un primo esercizio (con sincronizzazione manuale)

- Problema: prendere tutti i numeri da 1 a N e sommarli usando una ragione parallela.
- si definisca il numero di thread tramite il comando `omp_set_num_threads()`
- si ottenga l'id del singolo thread tramite `omp_get_thread_num()`
- il blocco parallelo necessita di `#pragma omp parallel`
- dato che ci sono vari thread e la somma e' associativa, ogni thread eseguirà un pezzo della somma e lo metterà in un ingresso di un array condiviso (`shared`), chiamato `arr_sum` con un ingresso per ogni thread della regione parallela
- alla fine dopo la regione parallela il master thread fa la somma dei vari ingressi dell'array.

Un primo esercizio (lo scheletro del codice)

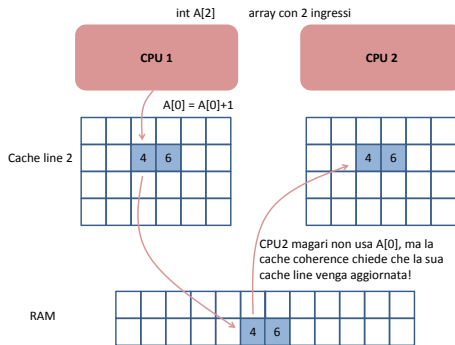
Sommiamo i primi numeri da 1 a N. Usare `omp_set_num_threads()` per decidere il numero dei thread, `omp_get_thread_num()` per conoscere il proprio id, e `#pragma omp parallel` per iniziare il blocco. La parte “difficile” è: suddividere il compito tra i thread correttamente!

```
#include<omp.h> // una somma inutilmente difficile dei numeri da 1 a N
#include<stdio.h>
static long N = 1000; // numero di punti da sommare
void main ()
{
  int i, sum=0;
  int numThreads = N / 10; // numero di thread usato
  //***** diciamo qui ad openMP che vogliamo numThreads
  int arr_sum[numThreads]; // array con somme parziali
  //***** facciamo partire il blocco parallelo
  {
    int par , arr; // partenza e arrivo
    //***** chiediamo a openMP l'id del chiamante, chiamiamolo "me"
    //***** calcoliamo quanti punti sono calcolati da ogni thread
    int local_sum=0 // azzeriamo la somma locale da calcolare
    //***** calcoliamo il punto di partenza della somma (funzione dell'id)
    //***** calcoliamo il punto di arrivo della somma (funzione dell'id)
    for (i= par; i< arr; i++){
      local_sum = local_sum + i; // somma dei valori da "par" a "arr"
    }
    //***** mettiamo la somma locale in un array globale
  }
  for (i=0;i<numThreads; i++) {sum += arr_sum[i];}
  printf("Somma= %d, teorica %d\n", sum, N*(N+1)/2);
}
```

Un primo esercizio: false sharing

Nell'esercizio precedente possono esservi problemi di cosiddetto: **false sharing**

- Supponiamo nel computer in cui gira il codice ci sia **cache coherence** tra i vari core, ognuno dei quali ha una propria **cache**
- Se il codice e' stato scritto a dovere, i vari thread "riempiono" indici diversi dello stesso array
- che, con buona probabilita' sara' su una singola **cache line** quindi il computer dovra' riscrivere la cache per tutti i thread ogni volta che un ingresso dell'array viene aggiornato (passando per la memoria)!



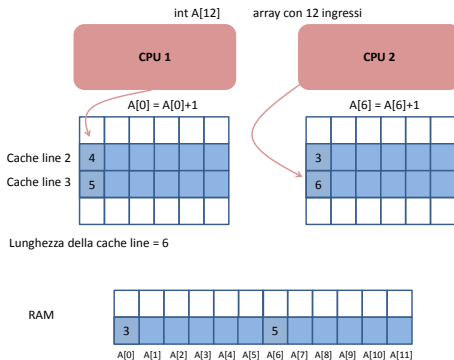
Come evitare il false sharing?

Nella diapositiva precedente abbiamo visto un caso in cui:

- due core accedono e modificano un solo array.
- Il primo core lavora solo con il primo ingresso dell'array
- il secondo core lavora solo con il secondo ingresso dell'array

Per questo motivo se non si sta attenti ci si aspetta che non si "calpestino i piedi"! sfortunatamente il sistema ha la proprietà di **cache coherence**, e quindi non appena uno dei due core scrive su una cache line e l'altro deve accedere alla stessa cache line, il sistema deve aggiornare la cache del secondo core, rallentando il processo.

Consiglio Per evitare questo potrebbe essere possibile scegliere di creare un array, più grande, in modo che i due ingressi su cui lavorano i due core, appartengano a due cache line differenti!



Alcuni costrutti di sincronizzazione: critical

```
#pragma omp critical
```

specifica che nel blocco che la segue puo' entrare solo **un thread** alla volta

- **rallenta** il sistema dato che serializza
- ci sono degli **overhead** dovuti al fatto di fare entrare e uscire un singolo thread, per esempio l'addizione diventa circa 200 volte piu' lenta rispetto alla controparte senza `critical`.
- aiuta ad evitare le **datarace/race condition**
- si puo' dare un nome alla sessione `critical`, per esempio:
`#pragma omp critical (PrimaSessione)`
- se ci sono piu' `critical` senza nome (unnamed), quando un thread entra in uno di questi blocchi, li chiude tutti **a chiave** (nessun processo puo' entrare negli altri blocchi!), questo puo' essere superato se si danno i nomi alle sessioni

altri costrutti di sincronizzazione: *atomic* e *barrier*

`#pragma omp atomic` permette di fare aggiornare **una variabile** ad un thread alla volta.

- c'e' (molto) meno **overhead** rispetto alle critical
- assicura la serializzazione di una **singola** operazione. In pratica il primo thread che arriva, e trova una operazione su una variabile, dice a tutti gli altri: adesso io ho il controllo della variabile!
- l'addizione e' circa 25 volte piu' lenta rispetto ad un sistema non critical
- le funzioni che si possono usare sono: +, *, -, /, &, ' |, <<, >> (occhio che una scrittura tipo `i=i+miafunc()` e' valida, in quanto all'oggetto di cui vogliamo fare l'update ho fatto solo una somma)

Esercizio: Scrivere un codice che aggiorna la variabile `conto` per ogni thread in modo da avere il numero totale di thread della regione parallela:

```
#include <stdio.h>
#include <omp.h>
int main() {
    int conto = 0;
    #pragma omp parallel num_threads(10)
    {
        #pragma omp atomic
        conto++;
    }
    printf("Numero di thread: %d\n", conto);
}
```

Nel caso in cui si necessiti che tutti i thread abbiano **finito di eseguire** il proprio compito prima di continuare l'esecuzione si mette una barriera:

```
#pragma omp barrier
```

(qui non c'e' problema perche' alla fine di una regione parallel, c'e' una barriera implicita)

Altri costrutti di sincronizzazione: *master, single e ordered*



`#pragma omp master` fa eseguire il blocco di seguito solo al master thread.

- gli altri thread saltano il costrutto `master`
- **NON** c'è nessuna barriera implicita né in ingresso né in uscita (quindi gli altri thread semplicemente ignorano il costrutto)
- la mancanza di barriere è pericolosa

Il costrutto `single` assomiglia al `master`, ma:

- non è eseguito dal master, qualunque (**singolo**) thread potrebbe essere l'esecutore
- esiste una **barriera** implicita in uscita (tutti i thread attendono che il costrutto `single` venga eseguito prima di continuare)

```
#include <omp.h>
#pragma omp parallel {
    #pragma omp single nowait    // il nowait elimina la barriera
    {
        printf("Starting calculation\n"); // solo uno scrive, ma tutti lo aspettano
    }

    // continua a lavorare
}
```

il costrutto `ordered` fa eseguire in ordine i vari thread come se fossero in un loop seriale (in qualche modo sospende il costrutto parallelo). Altrimenti se ci fosse una parte che va eseguita in serie dovrei de facto chiudere tutto il costrutto parallelo e perdere le informazioni `private` di ogni thread.

Riduzione

anche in **openMP** e' possibile definire delle operazioni di **riduzione** con la seguente sintassi:

```
#pragma omp reduction(operazione:nomevariabile)
```

- dove le operazioni disponibili sono: +, -, *, &, |, &&, ||
- **Attenzione** si possono definire opportunamente delle funzioni da usare nella riduzione, noi non lo faremo). Una funzione di riduzione deve poter prendere 2 valori e restituirne 1, e deve godere della proprieta' associativa (vedi diapositiva successiva).

Esercizio: scrivere un codice che usi la somma come operazione di riduzione per contare il numero di thread all'interno della regione parallela:

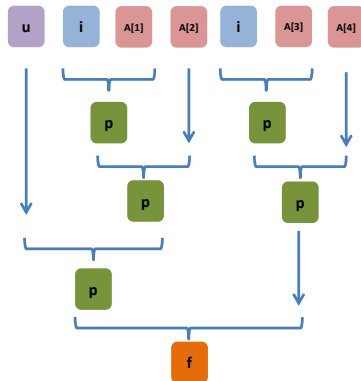
```
#include <omp.h>
#include <stdio.h>

int main ()
{
int  somma=0;

// introduco la regione parallela con clausola di riduzione per "somma"
{
// aggiungo 1 a "somma" per ogni thread della regione parallela
}
printf("Numero di thread presenti = %f\n",somma);
}
```

Occhio alla riduzione

Attenzione: tutte le operazioni di riduzione hanno come default un valore, che in generale (ma non sempre) sarà l'**elemento neutro** dell'operazione (per esempio per la somma lo 0, per il prodotto l'1, etc...). Questo perché la riduzione viene effettuata in un modo non banale, per esempio



- **u**= valore di inizializzazione assegnato dal programmatore alla variabile da ridurre
- **i**= valore di inizializzazione di default dell'operazione
- **A[1],A[2],A[3],A[4]** ... sono i valori che ottengono i thread, il thread(1), il thread(2), etc...
- **f**= valore finale della riduzione

Work-sharing

- All'interno di una regione parallela, le direttive di tipo **work-sharing** consentono ai thread (o ai task, che vedremo piu' avanti) di gestire il modo di lavorare **concorrente**.
- i costrutti **work-sharing** **NON** creano **nuovi** thread.
- un costrutto **work-sharing** deve essere incluso in una regione parallela per poter eseguire in parallelo... (sembra ovvio ma alle volte si dimentica)
- i costrutti **work-sharing** devono essere chiamati da **tutti** i thread o da **nessuno** (assomigliano in questo a delle **collective communications** di MPI)
- le regioni di tipo **worksharing**, hanno una **barriera implicita** alla fine del costrutto, ovvero tutti i thread devono avere completato il proprio lavoro, prima che si possa passare oltre il costrutto stesso.

Due costrutti work-sharing principali per il C:

- 1 `for` per i **loop**
- 2 `sections` per i **task**

La direttiva di Work-sharing: *for*

(in FORTRAN e' sostituita dalla direttiva `do`)

E' una direttiva **utilissima**: fa **suddividere** automaticamente un ciclo `for` tra i thread della regione parallela, de facto rendendo la variabile del loop privata e suddivisa tra thread.

- questa direttiva parallelizza automaticamente il ciclo `for` rendendo privata la variabile del ciclo per ogni thread.
- non ci devono essere **dipendenze** all'interno dei loop, tipo `a[i]=a[i-1]+3`
- Se ho un ciclo da 1 a 100 e ci sono 10 thread, esistono modi differenti per "suddividere" (worksharing) il lavoro

Esempio (con dubbi) implementare ciclo for:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int i, N=20;
    int *a;
    a = (int*) malloc(N*sizeof(int));
    // inizio costruito parallelo con 5 thread
    //
    // inserire un comando openMP che per il ciclo
    for (i=0; i<N; i++)
    {
        a[i]=i; // ogni thread deve riempire un ingresso diverso
    }
    // fine costruito parallelo
    for (i=0;i<N;i++)
    printf("%d\n", a[i]); // controllo
}
```

Non tutti i “for” sono uguali...

Non tutti i cicli for possono essere parallelizzati. Questo e' intuitivo, in quanto il computer deve essere in grado di suddividere il lavoro tra i vari thread e quindi deve avere ben chiaro l'ambito della variabile di loop, ovvero

```
for (inizializzazione; test; incremento)
```

- **inizializzazione** e' del tipo, per esempio: `i=0, i=37, ...`
- **test**, se il test e' vero, il loop continua. In pratica e' un vincolo entro cui si muove la variabile `i`. In un loop generico, basta che la condizione testata sia vera e si passa all' **incremento**. Pero' ci sono delle condizioni per cui e' difficile per il compilatore definire a priori quali valori puo' assumere la variabile `i`. Per esempio una condizione `i != 45` puo' essere difficile da raggiungere (magari `i` non vale mai 45...). Quindi le condizioni di **test** devono essere delle semplici: `<, <=, >, >=`
- per questo motivo anche l'**incremento** deve essere **semplice**: la variabile puo' essere **solo** incrementata (o diminuita) di quantita' costanti! (`i++, i=i+3, --i, ...`)

Con queste condizioni, il compilatore trova facilmente quanti `i` diversi tra loro ci sono e puo' decidere come suddividerli tra i vari thread.

Esempio di lastprivate: chi e' l'ultimo?

- **Attenzione:** una clausola `lastprivate` non puo' essere messa su un costrutto parallelo qualsiasi: deve esserci un modo per definire qual'e' l'**ultimo** thread...
- Per **ultimo** non si intende (necessariamente) dal punto di vista **temporale**, ma ultimo secondo un **univoco** ordine **logico/sintattico**. Nel caso di un ciclo `for` il thread che lavora con gli ultimi indici e' l'ultimo. Quindi per un costrutto `for` e per le `sections` (che vedremo poi) si possono definire variabili `lastprivate`.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main()
{
    int a=5, j;
    omp_set_num_threads(4);           // uso 4 thread nella regione parallela
    #pragma omp parallel for lastprivate(a) // regione for parallela
    for (j=0; j<7; j++)               // j=0,1,2,3,4,5,6
    {
        if (omp_get_thread_num()== 0) sleep(2); // rallento il thread 0
        a = j + 2;
        printf("Il thread =%d lavora con j=%d, il valore di a=%d\n", omp_get_thread_num(), j, a);
    }
    printf("Dopo il costrutto parallelo il valore di a=%d\n", a);
}
```

- Sebbene il thread 0 esegua per ultimo dal punto di vista **temporale** (per colpa dello `sleep(2);`)
- la variabile stampata e' quella del thread 3 ...
- ...perche' openMP, nel ciclo `for`, gli assegna l'**ultimo** valore dell'indice del ciclo (`j=6`).

for collapse

Nel caso in cui ci siano dei loop che siano innestati perfettamente, e' possibile usare questa clausola aggiuntiva, in questo modo:

- il compilatore **crea un unico loop** e poi lo parallelizza
- non c'e' bisogno di scrivere 2 direttive `#pragma omp for`, una per ogni ciclo

```
#pragma omp parallel for collapse (2)
{
  for (i=1; i<10; i++)
  {
    for (j=1, j<10;j++)
    {
      somma = somma +i;
    }
  }
}
```

Come viene suddiviso il lavoro in un for (worksharing)

- clausola `static`: lavoro allocato e assegnato all'inizio del loop
- clausola `dynamic`: il sistema decide come e quando assegnare un lavoro ad thread, non appena il thread ha eseguito e diventa idle, un nuovo compito puo' essere assegnato

openMP Scheduling:

- `schedule(static [, chunksize])`
- `schedule(dynamic[, chunksize])`

Per esempio:

```
#pragma omp parallel for default(none), shared(chunksize) schedule(static, chunksize)
for( int index = 0 ; index < 12 ; index++ )
```

Bell'esempio: <http://jakascorner.com/blog/2016/06/omp-for-scheduling.html>

Prendiamo un loop con 64 iterazioni (i=0,..., 63)

```
schedule(static)
```

```
th1 *****
th2          *****
th2                *****
th4                          *****
```

```
schedule(static,4)
```

```
th1 ****          ****          ****          ****
th2      ****      ****      ****      ****
th3          ****      ****      ****      ****
th4              ****      ****      ****      ****
```

```
schedule(static,8)
```

```
th1 *****
th2          *****
th3              *****
th4                  *****
```

Costrutto Parallel: variabili interne

Un buon motivo per cui ha senso la clausola `private`, all'interno di un `parallel` statement, e' che con openMP si vorrebbe poter aggiungere solo degli statement di tipo `pragma` **SENZA** cambiare la struttura del codice (se stiamo cercando di parallelizzare un codice seriale). Proviamo a vedere come si puo' modificare il codice qui sotto senza la clausola `private`:

```
int i, j;
#pragma omp parallel for private(j)
for(i=0; i<n; i++) {
    for(j=0; j<n; j++) { // la j e' privata per OGNI thread
        //fai qualcosa
    }
}
```

L'unico modo (in assenza di `private` clause) richiede di modificare la forma del codice, per esempio:

```
int i // devo togliere la j da qui
#pragma omp parallel for
for(i=0; i<n; i++) {
    int j; // si deve ridefinire la j all'interno del costrutto!
    for(j=0; j<n; j++) {
        //fai qualcosa
    }
}
```

Entrambi i codici sono validi, ma nel primo caso ho semplicemente dovuto inserire una riga di codice (eventualmente ignorabile), nel secondo caso ho cambiato la struttura!

Esercizio: π in modo poco elegante

Un codice che calcola π , usando il seguente risultato $\int_{-\infty}^{+\infty} \frac{1}{1+x^2} dx = \pi$

```

#include<omp.h>           // una versione di Pi greco con worksharing "MANUALE"
#include<stdio.h>
static long num_steps = 100000;           // numero di punti integrale
double step;
void main ()
{
    int i, tot=100; double x, pi, sum = 0.0;
    omp_set_num_threads(tot);           // def num thread
    step = 40000.0/(double) num_steps;   // grandezza step
    double arr_sum[tot];                // array con somme parziali
#pragma omp parallel private(x,i)
    {
        int par , arr;                  // partenza e arrivo
        int me = omp_get_thread_num();  // numero identificativo del thread
        int stxt = num_steps / tot;     // punti calcolati da ogni thread
        double local_sum=0;
        par = me * stxt;                 // indice di partenza
        arr = par + stxt;                // indice di arrivo
        for (i= par; i< arr; i++){
            x = (i+0.5)*step;           // punto integrazione
            local_sum = local_sum + 2.0/(1.0+x*x);
        }
        arr_sum[me] = local_sum;        // array locale
    }

    for (i=0;i<tot; i++) {sum += arr_sum[i];}
    pi = step * sum;
    printf("Pi Greco= %f\n", pi);
}

```

Esercizio: π

Qui proviamo ad ottenere lo stesso risultato ma:

- 1 usare come scheletro l'esercizio di somma della lezione precedente
- 2 invece che suddividere la somma "manualmente" usare il costrutto `#pragma omp for`
- 3 alla fine usare l'operazione di riduzione "somma"

π con il costrutto for

```

#include<omp.h>           //
#include<stdio.h>
static long num_steps = 1000000;           // numero di punti integrale
double step;
void main ()
{
    int i, tot=100; double x, pi, sum = 0.0;
    omp_set_num_threads(tot);             // def num thread
    step = 40000.0/(double) num_steps;    // grandezza step
    double arr_sum[tot];                  // array con somme parziali
//===== costrutto di openMP
    {
        int me = omp_get_thread_num();    // numero identificativo del thread
        double local_sum=0;
//===== costrutto di openMP
        for (i= 0; i< num_steps; i++){
            x = (i+0.5)*step;              // punto integrazione
            local_sum = local_sum + 2.0/(1.0+x*x);
            arr_sum[me] = local_sum;       // array locale
        }
        sum = 0 ;
//===== qui starebbe bene una riduzione
        for (i=0;i<tot; i++) {
            printf(" %f\n", step*arr_sum[i]);
            sum += arr_sum[i];
        }
        pi = step * sum;
        printf("Pi Greco= %f\n", pi);
    }
}

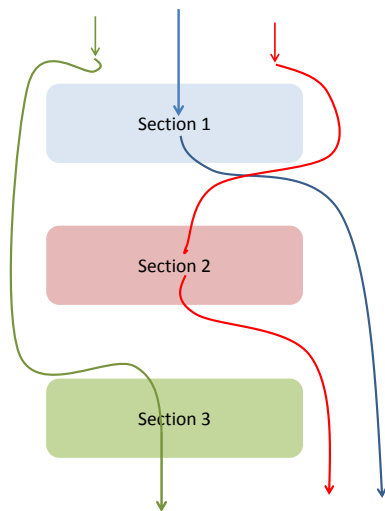
```

Worksharing: Sections

La direttiva `sections` divide esplicitamente il lavoro tra i thread, (anche in questo caso puo' essere utile utilizzare la clausola `nowait` per evitare che ci sia una barriera implicita alla fine di una direttiva `sections`).

```
#pragma omp parallel sections           // <= siamo in una regione parallela con sezioni
{
    #pragma omp section
    {
        fai_qualcosa();                // solo un thread esegue la funzione fai_qualcosa()
    }
    #pragma omp section
    {
        fai_qualcosaltro();           // solo un thread esegue la funzione fai_qualcosaltro()
    }
}                                       // qui c'e' una barriera implicita (eliminabile con nowait)
```

- il primo comando **openMP** inizializza le sezioni
- il secondo e il terzo comando **openMP** esplicitano l'inizio di una sezione
- alla fine delle `sections` c'e' una barriera implicita
- supponiamo che ci siano 4 thread nella regione parallela, ma solo 2 sezioni, cosa succede? solo 2 thread lavoreranno, gli altri saranno inattivi!



Domanda: Cosa succede se ci sono piu' "sections" che thread? **Risposta:** Un thread eseguirà piu' di una section.

Task

In openMP 3 (e migliorato in 4.0) e' stato introdotto il costrutto **Task**. Il nome e' intuitivo: si crea un compito che dovra' poi essere eseguito. L'idea e' quella che i compiti possano necessitare un'esecuzione non semplicemente gestibile con gli altri costrutti. Per esempio se si deve lavorare con delle funzioni ricorsive o con dei "linked set".

Un **task** si compone di:

- codice da eseguire
- Data environment (ovvero degli argomenti di input e di outoput)
- da chi il task viene eseguito (un thread)
- si deve definire **esplicitamente** cosa e' un task (con il task construct)
- un task puo' essere eseguito immediatamente (appena costruito) (**undeferred**) o posticipato (**deferred**) (messo nel task pool ed eseguito poi)
- a **runtime** viene deciso se il momento di esecuzione del task e' *deferred* o *undeferred*
- i task possono essere **innestati** (ricorsivi)

N.B. Anche prima di openMP 3.0 i **task esistevano**, pero' venivano assegnati ai thread automaticamente.

Task 2

il costrutto **task** che funziona in questo modo:

- un thread arriva sul costrutto **task**, tutto quello che trova all'interno del costrutto diventerà un compito (o piu' d'uno) che viene messo nel **task pool** (oppure con una decisione a runtime, il task viene eseguito subito)
- il thread continua nella sua parallel region (per esempio, se incontra nuovamente costrutti task, ne crea di nuovi)
- il thread, arrivato alla **fine** della **parallel region**, comincia a pescare dai task e li esegue; piu' in generale i task cominciano ad essere eseguiti non appena i thread incontrano una barriera (esplicita o implicita).
- un **task** puo' essere **tied** ad un thread o **untied**. Nel primo caso se un thread comincia un task solo quel thread puo' continuarlo. Nel secondo caso, un thread puo' cominciare un task, interrompere il lavoro (con opportune clausole o direttive, o anche perche' l'esecuzione del task richiede un altro task, per esempio se stiamo usando task ricorsivi) e magari un altro thread finisce il lavoro. **tied** e **untied** sono delle clausole che possono essere usate sulla direttiva `task`

I **task** possono essere **ricorsivi** e sono eseguiti secondo la execution tree (prima il genitore, poi il figlio etc...)

L'utente identifica i task e poi lascia che sia il computer ad scegliere l'ordine di esecuzione:

parallelismo irregolare

Generare Task

Per generare i task posso per esempio usare un codice del tipo:

```
#pragma omp parallel                // siamo in una regione parallela
{
    #pragma omp single                // <= perche' uso il costrutto single?
    {
        for (i=0;i<1000; i++)
        {
            #pragma omp task
            {
                miaFunzione(i);
            } // fine del costrutto task
        } // fine del ciclo
    } // fine del costrutto single
} // fine della regione parallela
```

- **quanti task** verranno generati?
- **quando** verranno eseguiti?
- **da chi** verranno eseguiti?
- se sto generando una lista troppo grande di task, l'implementazione puo' decidere di passare dalla **generazione** alla **esecuzione** dei task stessi
- **pericolo** se, prima di avere finito di generare tutti i task, il thread che genera i task si mette ad eseguire un task molto lungo... si arriva in una condizione di **starvation** (i task hanno fame...). Nuovi task non vengono generati, e ci sono dei thread che aspettano!

Task data scoping

Ricordiamo che la **generazione** ed **esecuzione** di un task non coincidono temporalmente.

- Se una variabile e' `shared` in un costrutto `task` il suo valore (referenza) all'interno del costrutto e' riferito al momento in cui il task e' stato generato.
- le referenze ad una variabile `private` all'interno del costrutto sono rispetto ad un nuovo valore non inizializzato e creato al momento dell'esecuzione del task.
- le referenze una variabile `firstprivate`, all'interno del costrutto, sono rispetto ad un nuovo oggetto, **creato al momento dell'esecuzione** con valore di **inizializzazione** uguale a quello della variabile al **momento di creazione** del task.
- ① In un costrutto di tipo `task`, se non e' presente nessuna clausola di tipo `default`, allora una variabile che e' `shared`, nel piu' interno dei costrutti esterni (al `task`), continua ad essere `shared`.
- ② in un costrutto `task`, se non c'e' nessuna clausola `default`, una variabile i cui attributi di data sharing non siano determinati dal punto precedente (ovvero `shared`), e' `firstprivate`.

(Per esempio se un costrutto `task` e' incluso in un costrutto parallelo, allora tutte le variabili che sono `shared` rimangono `shared` anche nel `task`, altrimenti vengono trasformate in `firstprivate`)

Consiglio: per evitare confusione sarebbe meglio che si definisse **esplicitamente** come sono le variabili all'interno del `task`, tramite l'utilizzo di clausole tipo: `private`, `lastprivate`, etc...
(usare sempre `default (none)` non provoca **crampi** alle dita!)

Esempi coi task

```
int a=1;
int b=2;
# pragma omp parallel firstprivate (b)
{
    int c=4;
    # pragma omp task shared(c)
    {
        int d = 5;
        eseguiFunzione(e,b); // esempio di funzione da eseguire
    }
}
```

- a **shared** (non essendoci clausole che la riguardano rimane **shared** in tutto il codice)
- b **firstprivate** nella regione parallela e resta **firstprivate** nel task. **Esiste** fuori dalla regione parallela.
- c **private** nella regione parallela e diventa **shared** nel task. **Non** esiste fuori dalla regione parallela.
- d **private** per ogni task. **Non** esiste fuori dal task.

Variabili e task: esempio

```

#include <stdio.h>
#include <omp.h>

void main()
{
    int tot=6;
    int i, j;
    #pragma omp parallel num_threads(tot)
    {
        int id = omp_get_thread_num();           // id e' private, per ogni thread
        #pragma omp single                       // ne entra solo 1
        {
            for (i=0; i<3; i++)                 // costruisce 3 task
            {
                #pragma omp task
                {
                    for (j=0; j<1000000; j++);   // task deferred
                    printf("Sono il thread=%d\n", id);
                }
            }
        }
    }
}

```

Attenzione: Cosa scrive a video?

- id e' definito dentro il costrutto parallel, quindi e' **privato** per ogni thread
- quando entra nel costrutto task diventa **firstprivate**
- le variabili del task vengono "create" dal thread che e' entrato nel single che quindi definisce il valore di id!

Task....wait!

Una delle caratteristiche del costrutto task e' quello di poter agevolare la scrittura di codici che abbiano delle parti ricorsive. Per questo motivo e' stata introdotto un utilissimo comando:

```
#pragma omp taskwait
```

Questa direttiva specifica che il task deve aspettare il completamento di tutti i task **figli** generati dall'inizio del task corrente **occhio** che vale per i **figli** non tutti i discendenti!

Invece:

```
#pragma omp taskgroup
```

fa bloccare il thread finche' tutti i task **discendenti** all'interno della regione sono completi.

Costuire un codice per calcolare il fattoriale

- creare una funzione fattoriale
- se $i < 2 \Rightarrow \text{fat}(i) = i$ (se $i=0$ $\text{fat}(i)=1$)
- costruire un task $\text{fat}(i-1)$

```
#include <stdio.h>
#include <omp.h>
int fat(int n)
{
    int i, j;
    if (n<1) return 1;
    if (n<2)
        return n;
    else
    {
        // ===== omp =====
        i=fat(n-1);
        // ===== omp =====
        return i*n;
    }
}
int main()
{
    int n = 14;
    omp_set_num_threads(100);
    // ===== omp =====
    {
        // ===== omp =====
        printf ("fat(%d) = %d\n", n, fat(n));
    }
}
```


openMP fine

- Introdotta l'idea di openMP: team di tread con **fork** e **join**
- introdotto come comunicano i thread: data sharing
- definite le clausole principali per il **data** sharing (shared, private, firstprivate, ...)
- introdotti i costrutti di **worksharing**, sections e soprattutto i **task**.

Compito Provare ad impostare un codice che sfrutti openMP per calcolare il prodotto tra matrici.

Domanda: Cannon e' un buon algoritmo per la moltiplicazione tra matrici per openMP? perche'?