

| 01 - Parallel Program Design

| Lezione del 02-10

| 1. Metodologia Foster

| 1.1 Partizionamento

Capire quali parti del programma (operazioni o funzioni) possono essere svolte in contemporanea, dividendole in più funzioni

| 1.2 Comunicazione

Determinare quali comunicazioni devono essere trasportate tra le varie task identificate nello step precedente

| 1.3 Agglomerazione

Combinare le task e le comunicazioni identificate negli steps precedenti in task più grandi.

Per esempio:

Se la task A deve essere eseguita prima della task B, ha senso aggregare in un'unica task grande in modo da non creare problemi all'esecuzione del programma.

| 1.4 Mapping

Assegnare le task identificate nei task precedenti ai processi / threads in base al pid ecc..

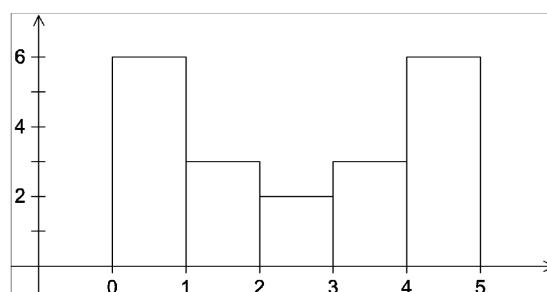
Questo deve essere fatto affinché la comunicazione sia minima tra i processi, cosicché ogni processo/ thread avrà la stessa quantità di lavoro.

Esempio:

| Iistogramma

Voglio creare un istogramma composto da bucket come il seguente.

1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



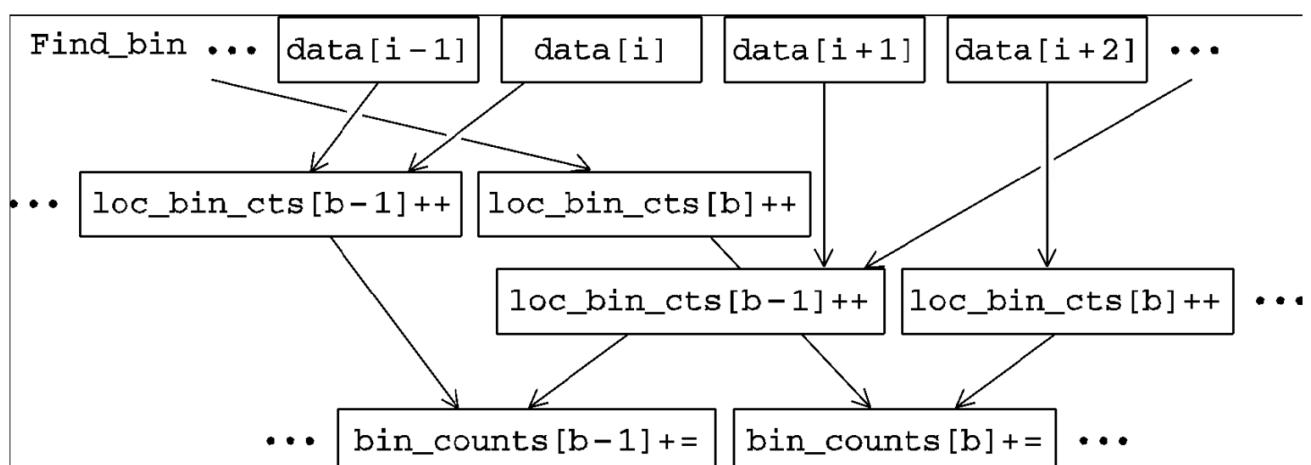
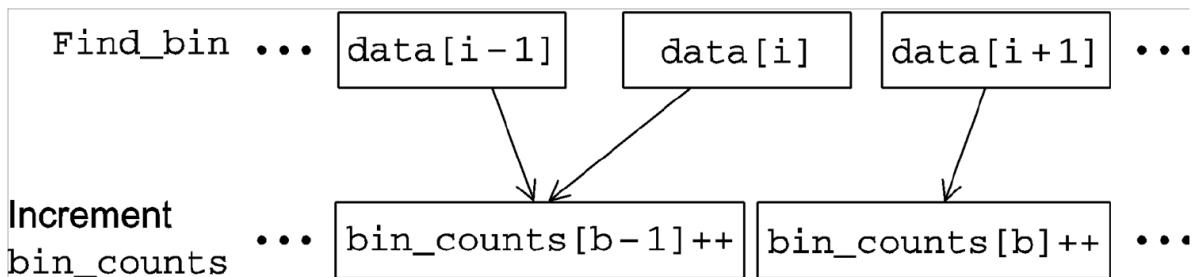
Input *programma seriale*:

- 1 Numero dei misuramenti: data_count
- 2 Un array di data_count float: data
- 3 Il minimo valore per il bucket contenente il valore più piccolo: min_meas
- 4 Il massimo valore per il bucket contenente il valore più grande: max_meas
- 5 Il numero di bucket: buck_count

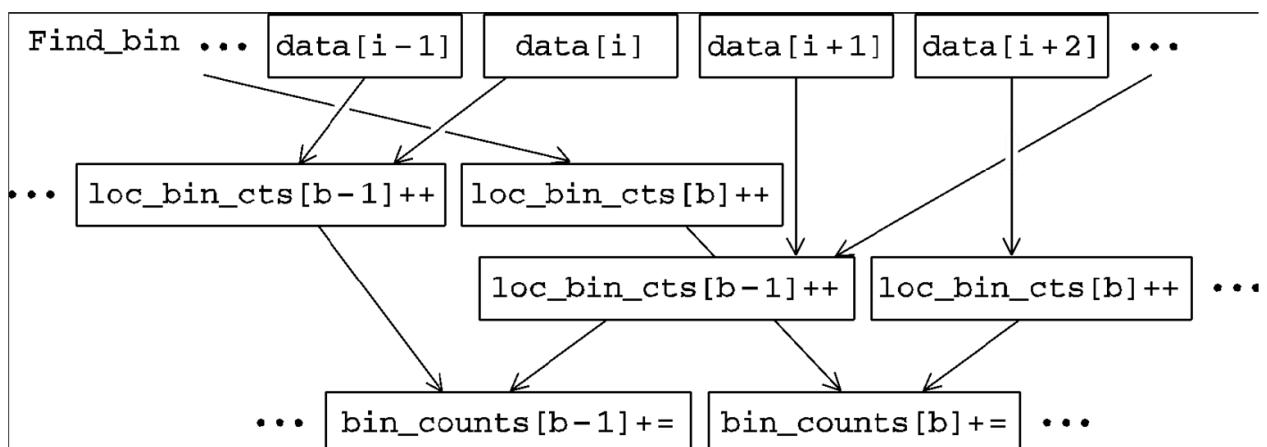
Output *programma seriale*:

- 1 buck_maxes: un array di buck_count floats
- 2 buck_counts: un array di buck_count ints

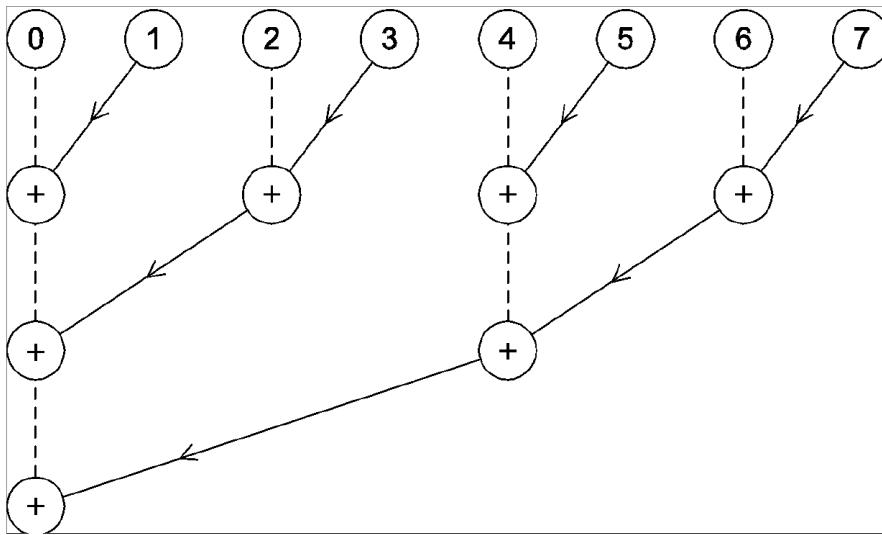
Primi due passaggi del **metodo di Foster**:



Definizione alternativa dei task e della comunicazione:



Adding the local arrays



| Parallel Design Patterns

Possiamo distinguere i parallel program structure patterns in due categorie principali:

- **Globally Parallel, Locally Sequential (GPLS):**

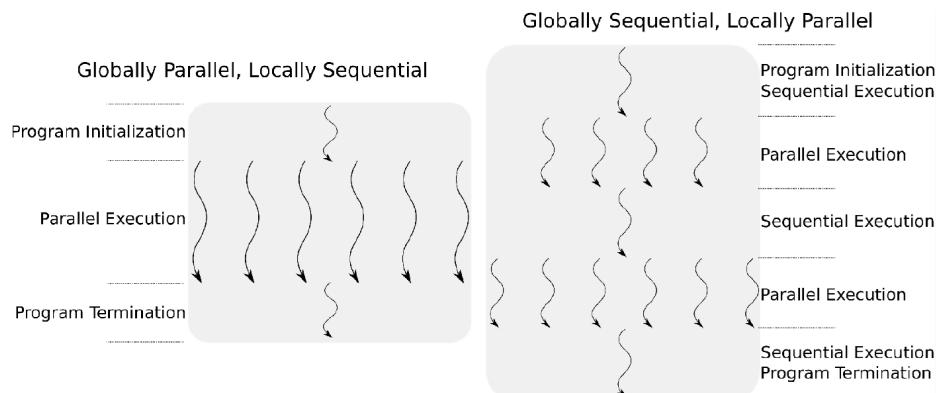
Significa che l'applicazione svolge task multiple concorrentemente con ogni task che runna sequenzialmente, in questa categoria sono inclusi:

- Single-Program, Multiple Data
- Multiple-Program, Multiple Data
- Master-Worker
- Map-reduce

- **Globally Sequential, Locally Parallel (GSLP):**

Significa che l'applicazione runna come un programma sequenziale, con parti individuali che runnano in parallelo quando richiesto, sono inclusi:

- Fork/join
- Loop parallelism



MPI generalmente il primo caso

| Globally Parallel, Locally Sequential (GPLS)

Vediamo nel dettaglio questa tipologia di applicazioni.

| Single-Program, Multiple Data

Tiene tutta la logica dell'applicazione in un singolo programma.

Generalmente la struttura del programma contiene:

- *Inizializzazione* del programma
- *Ottenimento* di un identificatore univoco, sono generalmente numerati a partire da 0
- *Running* del programma, esecuzione dei task a seconda dell'id
- *Terminazione* del programma

| Multiple-Program, Multiple Data

SPMD fallisce quando:

- I requisiti di memoria di ogni nodo sono troppo alti
- Sono coinvolte piattaforme eterogenee

Per il MPMD i processi di esecuzione sono identici a quelli dell' SPMD ma il deployment coinvolge più programmi differenti.

Entrambi i metodi sono supportati da MPI

| Master Worker IMPORTANTE

Come nel programma Hello World con più rank (il master era quello con rank 0).

Due tipi di componenti:

- **Master**
- **Workers**

| Master

E' responsabile per:

- *Conferire* i pezzi di lavoro ai workers
- *Raccogliere* i risultati della computazione dei workers
- *Performare* lavori di I/O , per esempio inviare ai lavoratori dati che devono processare o accedere ad un file ecc..
- *Interagire* con l'utente

Questo metodo è molto buono per il bilanciamento implicito del lavoro, ci sono pochissimi (o nessun) passaggio di dati tra processi worker

Un problema è che un solo master può fare da collo di bottiglia per il sistema, basti pensare a sistemi con 200k GPU, non è possibile che ci sia un solo processo (o nodo) a gestire tutto.

| Map Reduce

E' una variazione del master-worker pattern.

è un vecchio concetto tornato di moda dopo che Google lo ha implementato.

Il master coordina l'intera operazione

I workers hanno due tipi di task:

- **Map**: applica una funzione su dei dati, avendo come risultato un set di risultati parziali
 - **Reduce**: Recupera i risultati della map (i risultati parziali) e crea il risultato finale
- I processi map e reduce possono variare di numero.

Differenze tra Master-Worker e Map Reduce

- Master-Worker: Stessa funzione applicata su differenti data items
- Map Reduce: Stessa funzione applicata su parti differenti di un singolo data item

| Globally Sequential, Locally Parallel (GSLP)

| Fork/Join

Un singolo parent thread di esecuzione

Vi è una creazione dinamica di tasks children in run-time (in esecuzione).

Le task possono essere runnate con lo spawn di threads o usando una threads pool precedentemente creata.

Le task dei processi figli devono terminare per far continuare il processo padre.

Usato da OpenMP / PThreads

Esempio:

```
mergesort(A, lo, hi):
    if lo < hi:                                // at least one element of input
        mid = [lo + (hi - lo) / 2]
        fork mergesort(A, lo, mid) // process (potentially) in
                                    // parallel with main task
        mergesort(A, mid, hi)      // main task handles
                                    // second recursion
        join
        merge(A, lo, mid, hi)
```

| Loop parallelism

E' il più semplice

Usato per la migrazione di legacy/sequential software to multicore

Si basa su breaking loop by manipulating la variabile di controllo del loop.

Un loop deve essere in una forma particolare (for) per poter supportare questa versione di parallelismo

Ha una flessibilità limitata ma anche una facilità di sviluppo

| 02 - Point to Point Communication Models

| 08-10

| Tutte le funzioni fatte

Verranno mostrate a seguire tutte le funzioni fatte in MPI con relativa spiegazione.

| MPI_Send

MPI_Send usa una comunicazione **standard** ovvero in base alla dimensione del buffer decide se bloccare o meno il processo, ovvero se il messaggio è piccolo esso sarà inviato anche se la MPI_Receive non sia stata ancora effettuata, così che il processo non sia bloccante, sarà quindi solo **locally blocked**.

Se il messaggio viceversa è molto grande, questa chiamata risulta bloccante e quindi bisognerà avere anche il Receive prima di poter continuare con l'esecuzione del programma. MPI fornisce 3 modalità di comunicazione aggiuntive (oltre a questo modello standard):

- **Buffered**: l'operazione è sempre **locally blocked** (quindi bloccante solo all'interno del processo), viene quindi creata la copia del buffer e poi il processo continua, bisogna specificare però alla chiamata MPI dove specificare il buffer in cui copiare ciò che voglio mandare.
- **Synchronous**: sempre **globally blocking** anche per messaggi molto piccoli, quindi l'esecuzione viene bloccata finché non c'è la Receive, è importante perché cos' quando la send ritorna so che il processo ricevente è arrivato correttamente fino alla Receive, mentre senza il globally blocking non sono sicuro di cosa sta facendo il processo ricevente, è una sorta di **sincronizzazione** tra processi.
- **Ready**: ha successo solo se dall'altra parte se la Receive è già stata **postata**, ovvero il ricevente ha già specificato la Receive, quindi prima la funzione controlla se il ricevente ha già effettuato l' MPI_Receive , se il ricevente non ha effettuato la funzione allora viene ritornato un errore.

UTILIZZATO SEMPRE MPI_Send o MPI_Isend NONBLOCCANTE

| Non-Blocking Communication

Utilizzata molto spesso, sono comunicazioni non bloccanti

Le comunicazioni tramite buffer (Quindi la semplice MPI_Send) non sono molto performanti poiché bisogna aspettare che il buffer sia copiato.

Con le funzioni immediate posso avere l' **overlap per la comunicazione e computazione**

Posso avere questa comunicazione immediata sia per i Send che per i Receive, normalmente la Receive è bloccante, finché non mi arrivano i dati sono fermo, con questa

funzione posso dichiarare la volontà di ricevere dati, intanto faccio altro e quando ho bisogno di quei dati controllo che mi siano arrivati però intanto sono andato avanti con l'esecuzione.

Uno svantaggio è che queste intenzioni devono essere **esplicite**, vediamo meglio le funzioni.

| MPI_Isend

```
int MPI_Isend( void          *buf,      // Address of data buffer (OUT)
                int           count,     // Number of data items (IN)
                MPI_Datatype datatype, // Same as in MPI_Send (IN)
                int           source,    // Rank of destination proc. (IN)
                int           tag,      // Label identifying the type
                                     //   of message (IN)
                MPI_Comm       comm,     // Identifies the communicator
                                     //   context of 'source' (IN)
                MPI_Request    *req,     // Used to return a handle for
                                     //   checking status (OUT)
            )
```

MPI_Request che è ciò che ritorna la funzione è un handler che permette di richiedere informazioni riguardo lo stato dell'operazione (Posso sapere se è stato fatto l'invio o meno).

| MPI_Irecv

```
int MPI_Irecv( void          *buf,      // Address of receive buff. (OUT)
               int           count,     // Buffer capacity in items (IN)
               MPI_Datatype datatype, // Same as in MPI_Send (IN)
               int           source,    // Rank of sending process (IN)
               int           tag,      // Label identifying the type
                                     //   of message expected (IN)
               MPI_Comm       comm,     // Identifies the communicator
                                     //   context of 'source' (IN)
               MPI_Request    *req,     // Used to return a handle for
                                     //   checking status (OUT)
            )
```

MPI_Request che è ciò che ritorna la funzione è un handler che permette di richiedere informazioni riguardo lo stato dell'operazione (Posso sapere se è stato fatto l'invio o meno).

In MPI_Irecv il parametro MPI_Status è rimpiazzato con MPI_Request.

| Controlli per il Completamento

- Bloccante (distrugge l'handle)

```
int MPI_Wait( MPI_Request *req, // Address of the handle identifying
              //   the operation queried (IN/OUT)
              //   The call invalidates *req by
              //   setting it to MPI_REQUEST_NULL.
              MPI_Status   *st,  // Address of the structure that will
                               //   hold the comm. information (OUT)
            )
```

- Non Bloccante (Distrugge l'handle se l'operazione è avvenuta con successo flag=1

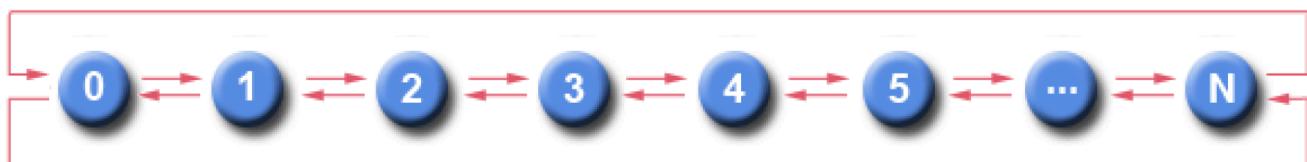
```
int MPI_Test( MPI_Request *req, // Address of the handle identifying
              // the operation queried (IN)
              int        *flag, // Set to true if operation is
              // complete (OUT).
              MPI_Status *st   // Address of the structure that will
              // hold the comm. information (OUT)
)
```

Ci sono tante altre alternative:

- Waitall
- Waitany
- Testany
- ecc..

| Problemi da chiamate Bloccanti

Problem: Ring: Each rank sends something to left/right rank, and receives something from them



Con le chiamate bloccanti ci possono essere problemi, mentre con le non bloccanti non ci sono.

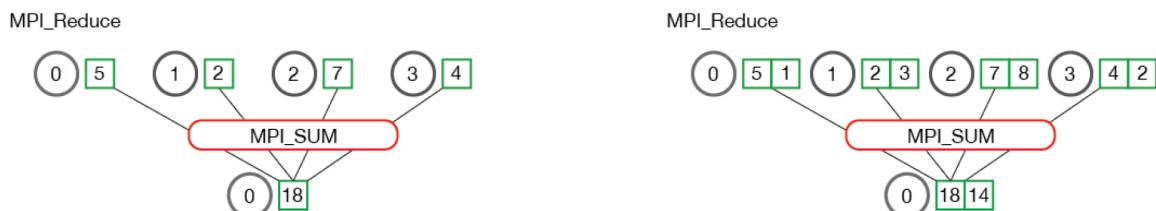
Esempio Trapezoidal Rule per il calcolo dell'integrale.

| Collective communications

Invece di decidere ogni volta come effettuare il passaggio di dati tra più processi (albero, ecc..) decide MPI.

| MPI_Reduce

Specifico che operazione voglio fare e la implementa nel modo migliore



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

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

```
double local_x[N], sum[N];
...
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
            MPI_COMM_WORLD);
```

La MPI_Reduce ha:

- un puntatore per il buffer in ingresso (proprio per ogni chiamata), elementi con cui io contribuisco (es. puntatore ad un array in cui io processo 2 ho salvato i miei numeri)
- un puntatore per il buffer di uscita (importante per il rank 0)
- quanti elementi
- di che tipo
- che operazione
- quale processo riceverà il risultato (es 0)
- Comunicatore (MPI_Comm)

MPI_Reduce è bloccante per il nostro utilizzo.

Operazioni disponibili di base (le posso anche implementare io)

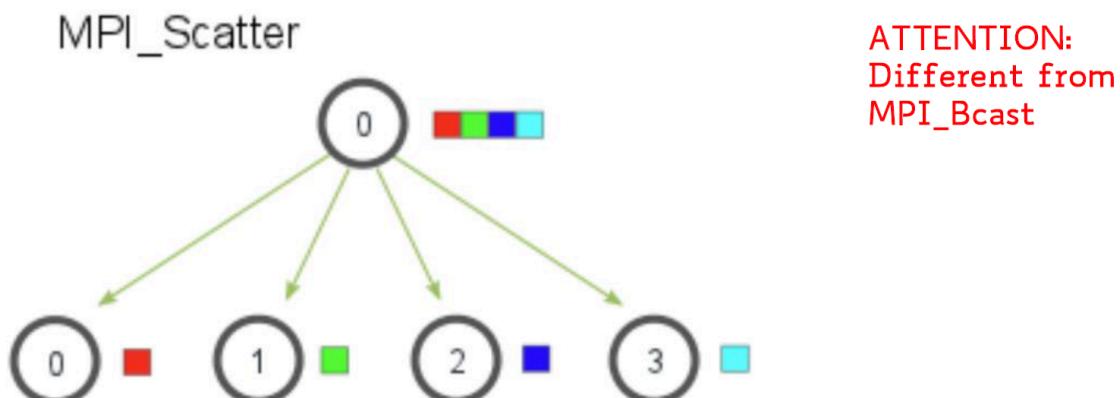
Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Se un'operazione **non** è commutativa e associativa MPI_Reduce manterrà l'ordine da noi definito, se invece è per esempio una somma tra interi non conterà l'ordine dei processi, quando io creo un'operazione posso definire se questa è commutativa e associativa o meno.

Quasi tutte le implementazioni di MPI consentono solo al processo con rank 0 di accettare input da stdin

| Scatter

MPI_Scatter può essere utilizzato in una funzione che legge un intero vettore sul processo 0, ma invia solo i componenti necessari a ciascuno degli altri processi.



```

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

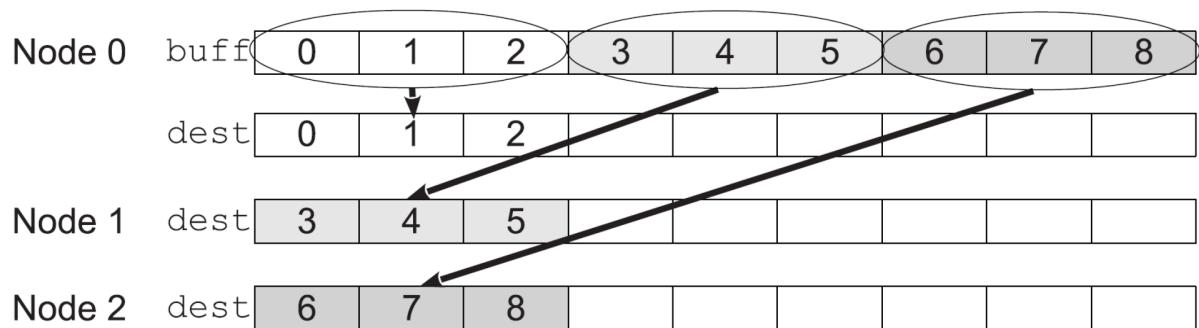
```

Se voglio mandare differenti numeri di elementi ad ogni rank uso: **MPI_Scatterv**

```

MPI_Scatterv(buff, 3, MPI_INT,
                dest, 3, MPI_INT, 0, MPI_COMM_WORLD);

```

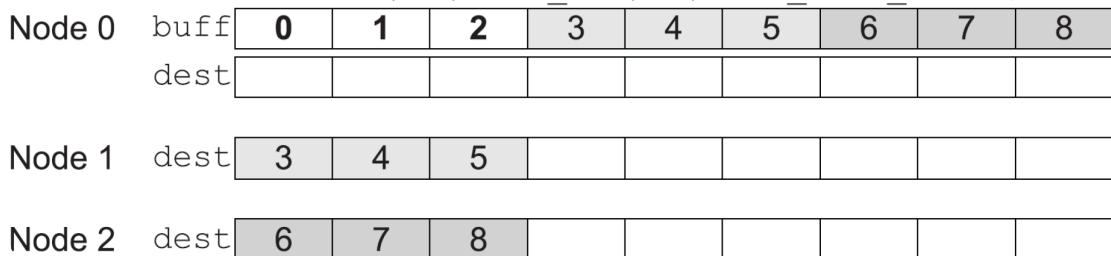


Se non volessi creare un altro buffer per il rank 0 uso **MPI_IN_PLACE**

```

if(rank == 0)
    MPI_Scatter(buff, 3, MPI_INT,
                  MPI_IN_PLACE, 3, MPI_INT, 0, MPI_COMM_WORLD);
else
    MPI_Scatter(buff, 3, MPI_INT,
                  dest, 3, MPI_INT, 0, MPI_COMM_WORLD);

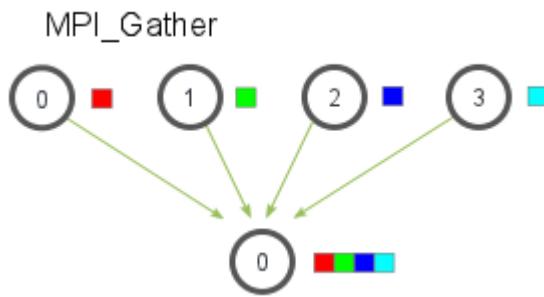
```



| Esempio di lettura di vettore - readvector.c

| Gather

Collezione tutti i componenti del vettore nel processo 0 e poi quest'ultimo può processare i componenti.



```

int MPI_Gather(
    void* send_buf_p /* in */,
    int send_count /* in */ ATTENTION: This is the number of elements that each process sends, not the total number of elements in the final vector!!!
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    int dest_proc /* in */,
    MPI_Comm comm /* in */);
  
```

| Esempio di print di un vettore - printvect.c

| Matrix-Vector Multiplication

Come parallelizzarlo?

Assumiamo che è tutto fatto in un loop e l'output y è usato come input di un vettore per la prossima iterazione:

- Iterazione 0; $y_0 = A \cdot x$
- Iterazione 1: $y_1 = A \cdot y_0$
- ecc..

Prima iterazione:

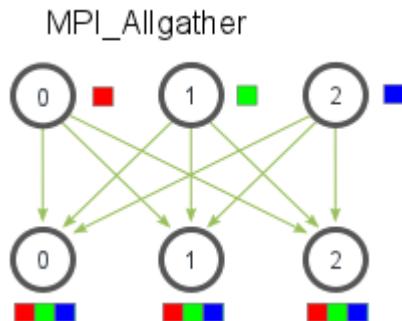
1. Broadcast il vettore x dal rank 0
2. Scatter le righe della matrice A dal rank 0 agli altri processi
3. Ogni processo computa un sottoinsieme di elementi del vettore y_0
4. Gather il vettore finale y_0 al rank 0
5. Broadcast y_0 dal rank 0 a tutti gli altri processi

Tutte le altre iterazioni:

1. Ogni processo computa un sottoinsieme di elementi del vettore y_i usando y_{i-1} ricevuto dallo step precedente
2. GATHER il vettore finale y_i al rank 0
3. Broadcast y_i dal rank 0 agli altri processi

| Allgather

è come un gather + broadcast, molto usata



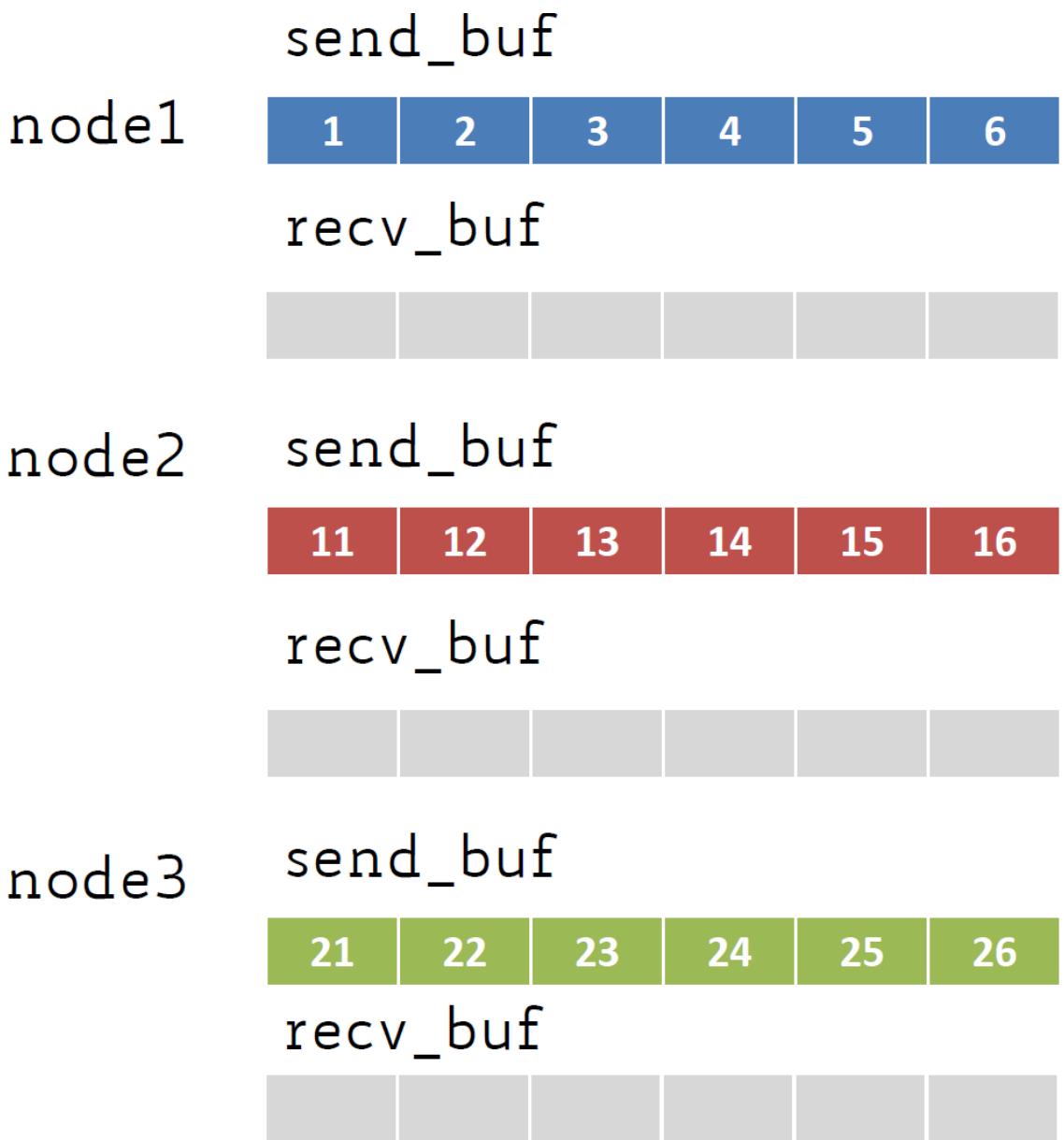
```
int MPI_Allgather(
    void* send_buf_p /* in */,
    int send_count /* in */,
    MPI_Datatype send_type /* in */,
    void* recv_buf_p /* out */,
    int recv_count /* in */,
    MPI_Datatype recv_type /* in */,
    MPI_Comm comm /* in */);
```

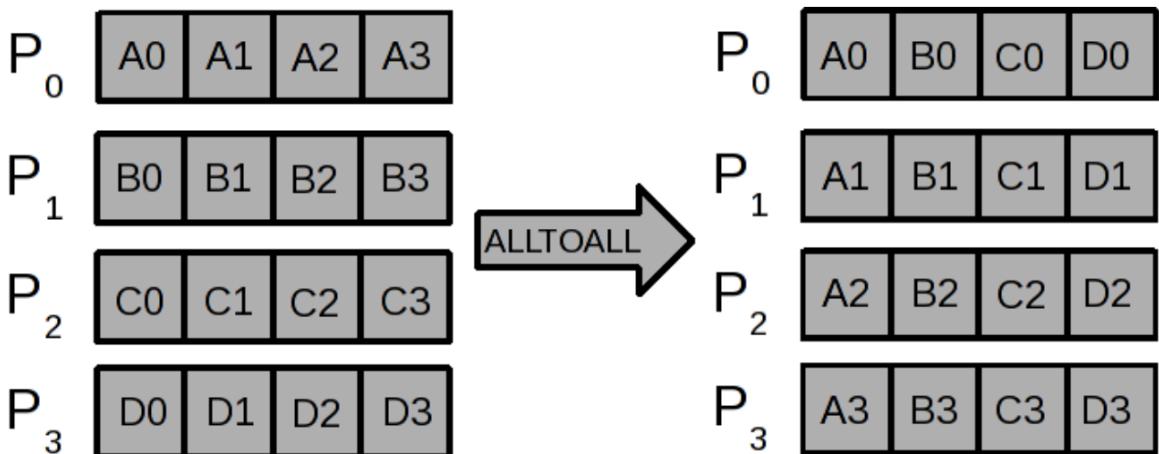
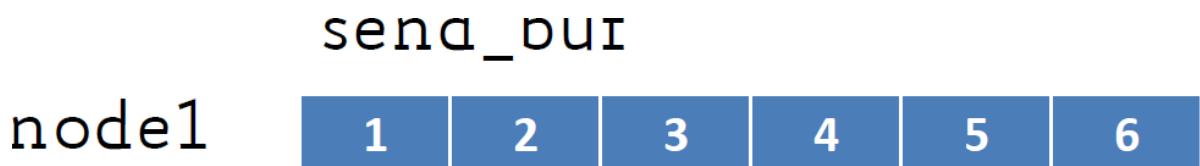
| Reduce-Scatter

Molto usata, fa una reduce (un'operazione comune tra tutti) ma poi ogni processo si tiene solo un blocco degli elementi



| MPI_ALLtoall





Molto utile per la trasposizione di matrici

| 03 - Performance Evaluation

| Lezione del 15-10

| Elapsed parallel time

Abbiamo provato a calcolare il tempo di esecuzione prendendo il tempo all'inizio e alla fine e sottraendoli, però le cose sono più complicate di così, infatti il programma termina quando il più lento termina, magari potrebbe finire prima se cambiassimo il parallelismo.

La soluzione è portare il tempo massimo tra i rank, come?

```
double local_start, local_finish, local_elapsed, elapsed;  
.  
.  
local_start = MPI_Wtime();  
/* Code to be timed */  
.  
.local_finish = MPI_Wtime();  
local_elapsed = local_finish - local_start;  
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,  
            MPI_MAX, 0, comm);  
  
if (my_rank == 0)  
    printf("Elapsed time = %e seconds\n", elapsed);
```

| Ogni rank inizia allo stesso momento?

Non necessariamente

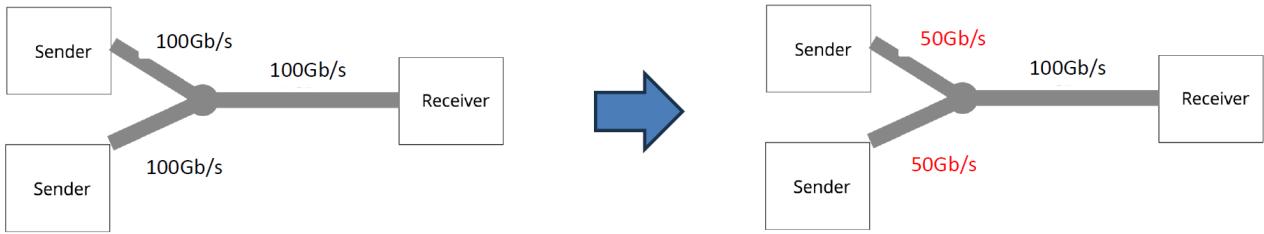
Il tempo riportato da un rank potrebbe essere maggiore non perché più lento ma perché ha iniziato dopo, per poter essere certi che tutti i rank iniziano e finiscono nello stesso momento devo usare le **MPI_Barrier**.

In realtà non posso comunque essere sicuro dell'inizio preciso allo stesso momento, ma per il nostro corso lo consideremo tale.

| Quante misurazioni fare?

Una non è abbastanza

Eseguendo una stessa applicazione 100 volte potrei avere 100 tempi differenti, poichè ci sono molti altri processi che devono essere eseguiti da altri programmi nello stesso momento, inoltre se eseguo il programma su più nodi/server potrei avere interferenze di rete.

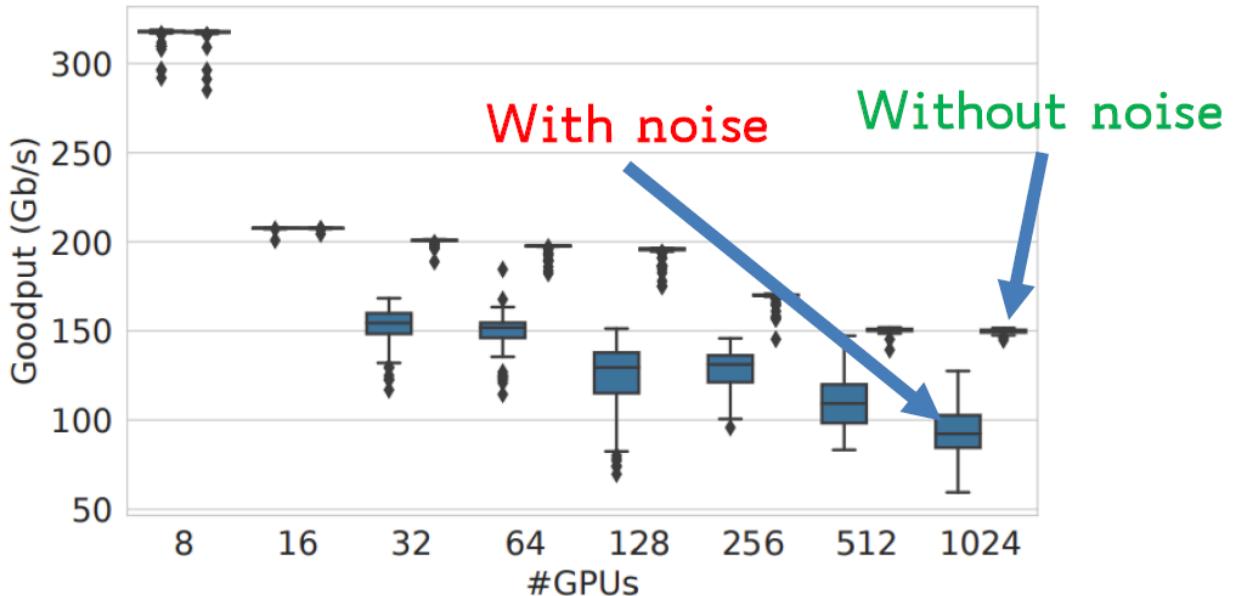


Bisogna quindi runnare molteplici volte l'applicazione e restituire tutta la curva di timings.

| Recap

Creare una barriera all'inizio del programma per far sì che ogni processo parta *quasi* in contemporanea e restituire tutti i risultati di ogni run, magari sottoforma di grafico.

| Impatto del noise



(b) Allreduce

Le performance diminuiscono anche se aumento le gpu, perchè?

Perchè ci saranno tanti più processi e rank e sono affetti dal rumore (noise).

| Esempio

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

In genere il tempo di esecuzione aumenta con il numero di input e diminuisce con il crescere dei processi utilizzati.

Come si può vedere con 1024 esecuzioni con 8 e 16 processi abbiamo lo stesso tempo

| Che aspettative abbiamo?

Idealmente runnando un processo con p processi dovrebbe essere p volte più veloce di runnarlo con 1 processo.

Definiamo come $T_{serial}(n)$ il tempo sequenziale della nostra applicazione su un input n .

Definiamo come $T_{parallel}(n, p)$ il tempo sequenziale della nostra applicazione su un input n con p processi.

Definisco con $S(n, p)$ lo **Speedup** della nostra applicazione parallela che indicamente dovrebbe essere:

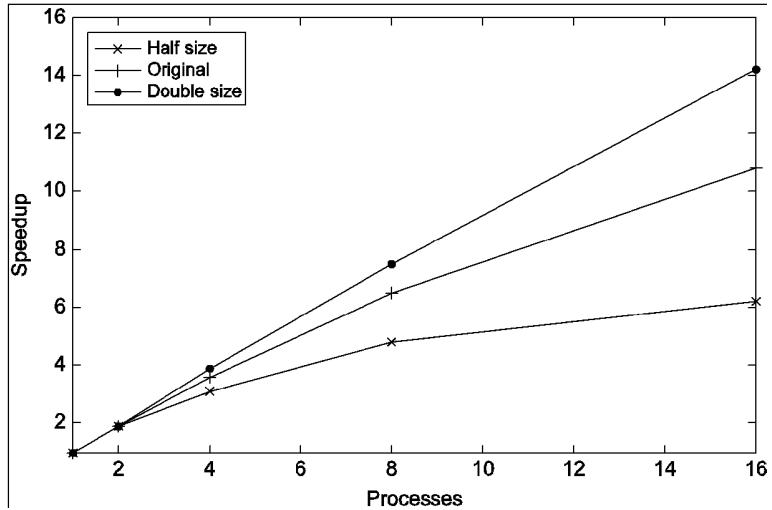
$$S(n, p) = T_{Serial}(n) / T_{parallel}(n, p)$$

Idealmente vorremmo che sia $S(n, p) = p$ ovvero un **speedup lineare**.

- **ATTENTION:** They must be taken on the same type of cores/system. I.e., do not comput the serial time on a CPU core and the parallel time on the GPU cores
- **ATTENTION:** Let's say you implemented a new insertion sort algorithm. Shall T_{serial} be the fastest insertion sort code or the fastest sorting code in general (e.g., quicksort)?

| Le nostre aspettative

What expectations do we have?



In general, we expect the speedup to get better when increasing the problem size n

Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Ci sono modi a partire dal codice seriale quanti processi al max ha senso utilizzare

| NOTE

$$T_{\text{serial}}(n)! = T_{\text{parallel}}(n, 1)$$

$T_{\text{serial}}(n)$ è il tempo di run di un applicazione sequenziale con input n.

$T_{\text{parallel}}(n, 1)$ è il tempo di run di un applicazione parallela di input n con 1 processo, in generale:

$$T_{\text{parallel}}(n, 1) \geq T_{\text{serial}}(n)$$

Speedup

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

Scalability

$$S(n, p) = \frac{T_{\text{parallel}}(n, 1)}{T_{\text{parallel}}(n, p)}$$

Speedup parte dal codice seriale, la scalabilità da un codice parallelo con un processo.

| Efficienza

$$\$ \$ E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p * T_{\text{parallel}}(n, p)} \$ \$$$

Idealmente vorremmo $E(n, p) = 1$, in pratica lo abbiamo ≤ 1 e diventa peggiore con problemi di grandezza minore.

Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

| Scalabilità Forte vs. Scalabilità Debole

- **Strong Scaling**
 - Fixa il problema della grandezza e incrementa il numero dei processi
 - Se riusciamo a tenere alta l'efficienza il nostro programma è fortemente scalabile
- **Weak Scaling**
 - Incrementa il problema della grandezza allo stesso rate dell'incremento dei processi.
 - Incrementando 2x il numero dei processi incremento il problema della grandezza di 2x.
 -

Strong vs. Weak Scaling (From Speedup Data)

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Not strongly scalable

Strong vs. Weak Scaling (From Speedup Data)

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

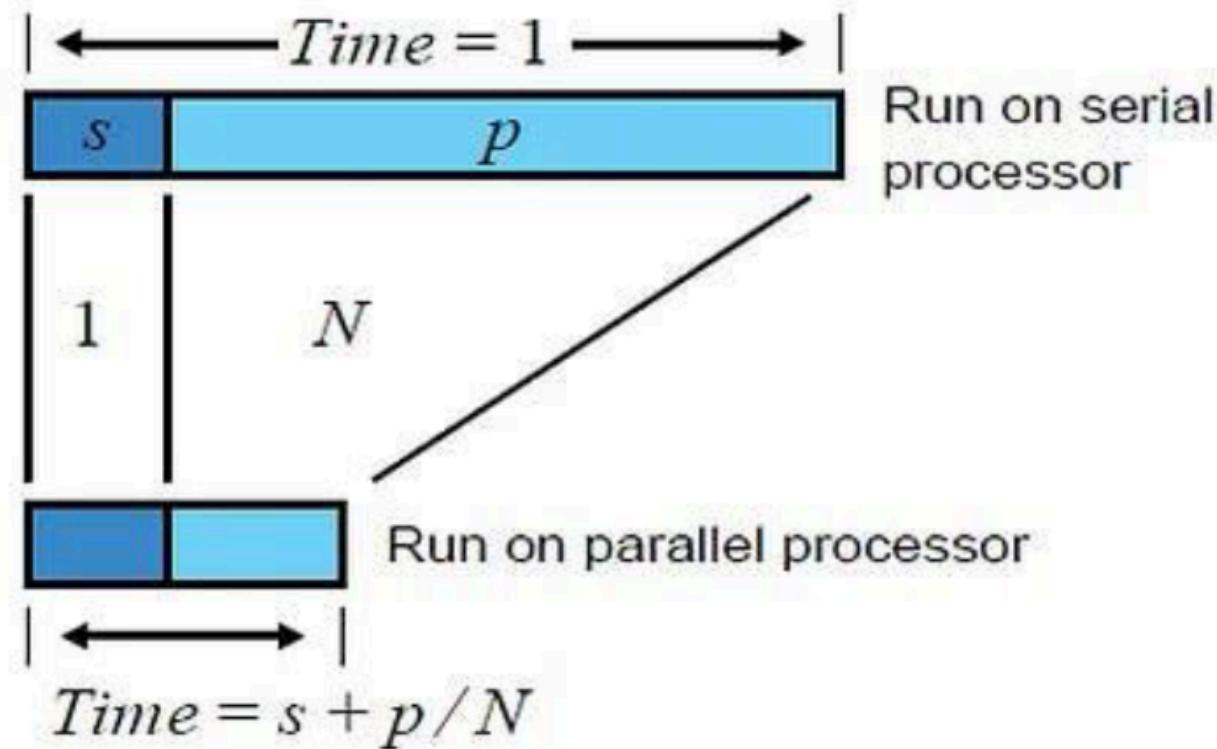
Weakly scalable

| Amdahl's Law per l'estrapolazione delle aspettative

Intuizione: Ogni programma ha alcune parti che non possono essere parallelizzate (**serial fraction $1 - \alpha$**)

Per esempio nella lettura/scrittura di un file sul disco

Amdahl's law dice che lo speedup è **limitato** dal serial fraction.



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

Una frazione $0 <= \alpha <= 1$ può essere parallelizzata, i rimanenti $1 - \alpha$ devono essere fatti sequenzialmente.

Per esempio:

- Se ho $\alpha = 0$ il codice non può essere parallelizzato e $T_{parallel}(p) = T_{serial}$
- Se ho $\alpha = 1$ il codice può essere parallelizzato e ho $T_{parallel}(p) = \frac{T_{serial}}{p}$ (speedup ideale)

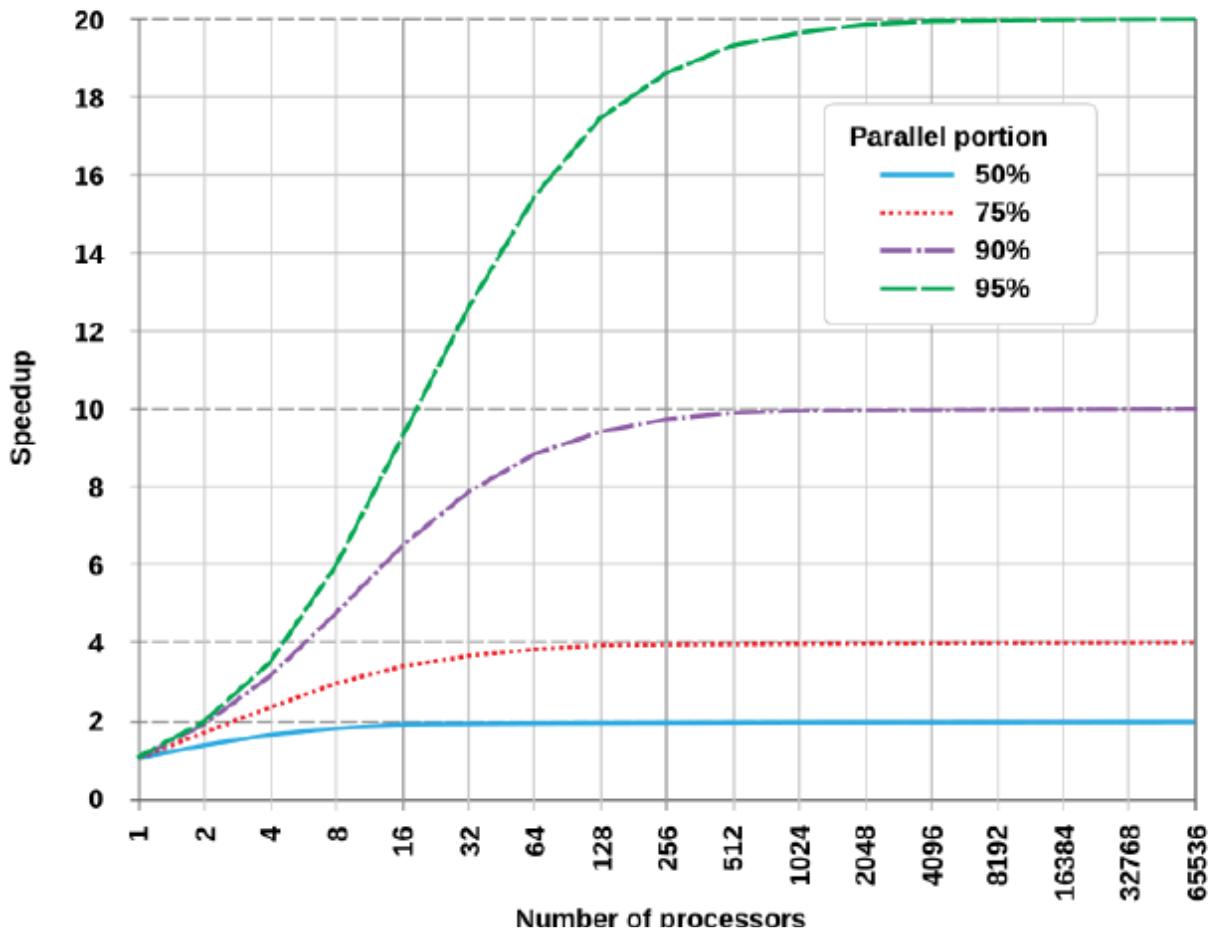
$$S(p) = \frac{T_{serial}}{(1 - \alpha)T_{serial} + \alpha \frac{T_{serial}}{p}}$$

$$\lim_{p \rightarrow \infty} S(p) = \frac{1}{1 - \alpha}$$

I.e.:

- if 60% of the application can be parallelized, $\alpha = 0.6$, which means we can expect a speedup of at most 2.5
- if 80% of the application can be parallelized, $\alpha = 0.8$, which means we can expect a speedup of at most 5
- To be able to scale up to 100000 processes, we need to have $\alpha >= 0.99999$

Amdahl's Law

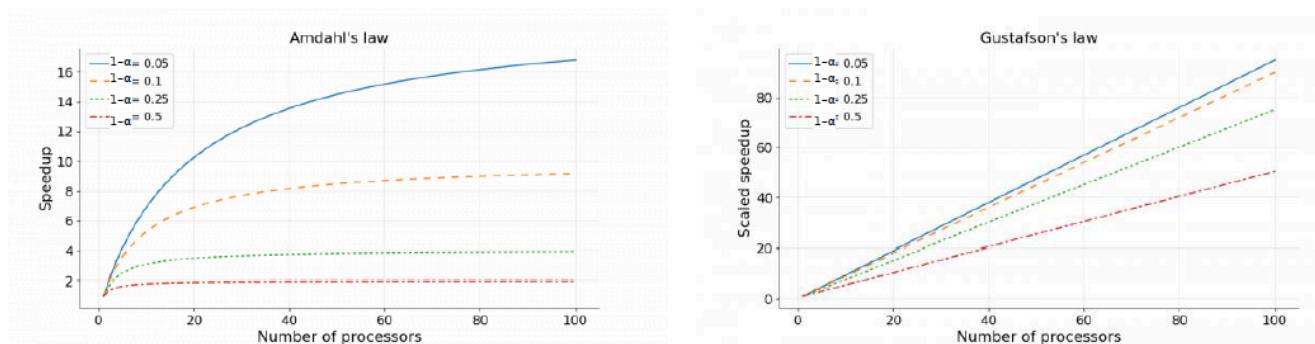


| Gustafson's Law

Se condieriamo la weak scaling, il parallel fraction incrementa con l'incremento della grandezza del problema, il **tempo seriale** rimane **costante** ma il **tempo parallelo aumenta** è conosciuto come **scaled speedup**, speedup scalato

$$S(n, p) = (1 - \alpha) + \alpha p$$

Amdahl's Law vs. Gustafson's Law



| Limitazioni della legge di Amdahl

La serial fraction può diventare più grande quando incremento i numeri di processi (il tempo di esecuzione può incrementare quando incremento il numero di processori).

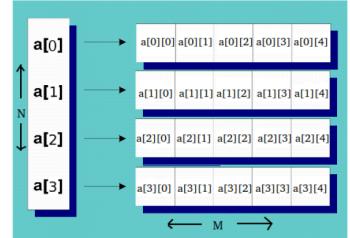
Soluzione: Universal Scalability Law:

<https://wso2.com/blog/research/scalability-modeling-using-universal-scalability-law/>

| 04 - Gather e Broadcast su memoria dinamica

| Lezione del 16-10

```
int** a;
a = (int**) malloc(sizeof(int*)*num_rows);
for(int i = 0; i < num_rows; i++){
    a[i] = (int*) malloc(sizeof(int)*num_cols);
}
```



```
MPI_Reduce(a, recvbuf, num_rows*num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
MPI_Reduce(a[0], recvbuf, num_rows*num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

CORRECT

WRONG!

```
for(int i = 0; i < num_rows; i++){
    MPI_Reduce(a[i], recvbuf[i], num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

```
int* a;
a = (int*) malloc(sizeof(int)*num_rows*num_cols);
...
...
// a[i][j]
a[i * num_cols + j] = ....
```

CORRECT

```
MPI_Reduce(a, recvbuf, num_rows*num_cols, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

| 05 - Parallel Excercises

| Lezione del 16-10

| Parallel Sorting Algorithm

Vogliamo implementare il bubble sort parallelo, useremo una sua versione chiamata Odd-even transposition sort

Che è costituito:

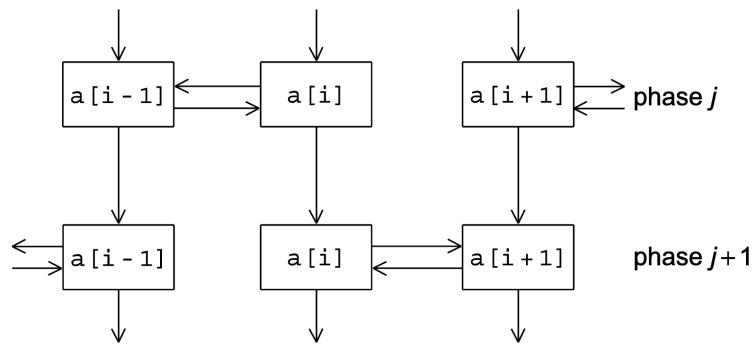
- Una sequenza di fasi
 - Ogni sequenza pari controllo i numeri in indice pari e swappo
 - Ogni sequenza dispari controllo i numeri in indice dispari e swappo

| Algoritmo sequenziale

```
void Odd_even_sort(
    int a[] /* in/out */,
    int n   /* in      */) {
    int phase, i, temp;

    for (phase = 0; phase < n; phase++)
        if (phase % 2 == 0) { /* Even phase */
            for (i = 1; i < n; i += 2)
                if (a[i-1] > a[i]) {
                    temp = a[i];
                    a[i] = a[i-1];
                    a[i-1] = temp;
                }
        } else { /* Odd phase */
            for (i = 1; i < n-1; i += 2)
                if (a[i] > a[i+1]) {
                    temp = a[i];
                    a[i] = a[i+1];
                    a[i+1] = temp;
                }
        }
    } /* Odd-even-sort */
```

Communications among tasks in odd-even sort



Tasks determining $a[i]$ are labeled with $a[i]$.

Idea del codice Parallello

Divido il numero di elementi nei vari processi poichè $n >> p$

Ogni processo si ordina il proprio sottoarray con un algoritmo sequenziale (es quicksort).

Una volta che ogni processo al proprio sottoarray ordinato mi scambio gli elementi con gli altri processi, il processo a sinistra si tiene la metà più piccola quella a dx la metà più grande, processo i scambia con i-1 la metà più piccola e lui tiene quella più grande

In base alla parità o meno della fase decido con chi scambiare

Pseudocodice

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (I'm not idle) {
        Send my keys to partner;
        Receive keys from partner;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```

Attenzione

Visto che ho tanti passaggi di send e recv non posso usare MPI_Send e MPI_Recv perché potrei finire in deadlock, uso:

```

int MPI_Sendrecv(
    void* send_buf_p /* in */,
    int send_buf_size /* in */,
    MPI_Datatype send_buf_type /* in */,
    int dest /* in */,
    int send_tag /* in */,
    void* recv_buf_p /* out */,
    int recv_buf_size /* in */,
    MPI_Datatype recv_buf_type /* in */,
    int source /* in */,
    int recv_tag /* in */,
    MPI_Comm communicator /* in */,
    MPI_Status* status_p /* in */);

```

Così che non si possa verificare un deadlock

| Esercizio Somma Tra Vettori Parallelia

Esercizio sum_vectors.c, importante luso di [Scatter](#) e [Gather](#)

| Esercizio Matrix-Vector Multiplication

Non posso utilizzare una matrice bidimensionale (A), devo usare un singolo array (B) di lunghezza n°righe + n°colonne e per prendere gli indici devo fare:

$$A[i][j] = B[i * N^{\circ} \text{col} + j]$$

a_{00}	a_{01}	\cdots	$a_{0,n-1}$	x_0 x_1 \vdots x_{n-1}	y_0 y_1 \vdots y_{m-1}
a_{10}	a_{11}	\cdots	$a_{1,n-1}$		
\vdots	\vdots		\vdots		
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$		
\vdots	\vdots		\vdots		
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$		

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

Multiply a matrix by a vector

```
/* For each row of A */  
for (i = 0; i < m; i++) {  
    /* Form dot product of ith row with x */  
    y[i] = 0.0;  
  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j]*x[j];  
}
```

Serial pseudo-code

| Come posso implementarlo in parallelo?

1. Broadcast il vettore x dal rango 0 agli altri processi.
2. Scatter le righe della matrice A di rango 0 agli altri processi.
3. Ogni processo calcola un sottoinsieme degli elementi del vettore y risultante.
4. Gather il vettore finale y al rango 0

Now, let's assume that this is done in a loop, and the output vectpr for the next iteration

e.g.:

Iteration 0: $y_0 = A \cdot x$
Iteration 1: $y_1 = A \cdot y_0$
Iteration 2: $y_2 = A \cdot y_1$
...

First iteration:

1. **Broadcast** the vector x from rank 0
2. **Scatter** the rows of the matrix A from rank 0 to the other processes
3. Each process **computes** a subset of the elements of the resulting vector
4. **Gather** the final vector y_0 to rank 0
5. **Broadcast** y_0 from rank 0 to the other processes

} Is there a

All the other iterations:

1. Each process **computes** a subset of the elements of the resulting vector from the previous step
2. **Gather** the final vector y_i to rank 0
3. **Broadcast** y_i from rank 0 to the other processes

} Is there a

Uso [AllGather](#)

IMPLEMENTALO

| 06 - Derived Datatypes

In C posso creare delle strutture specifiche, composte da più tipi di dato combinati, se dovessi inviarle con le funzioni MPI come posso fare?

Devo creare dei tipi di dato derivati che vanno a matchare uno a uno le struct.

Riprendendo l'esempio del trapezio

```
void Get_input()
{
    int      my_rank    /* in   */,
    int      comm_sz    /* in   */,
    double* a_p         /* out */,
    double* b_p         /* out */,
    int*    n_p         /* out */
    int dest;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
    /* Get input */
}
```

Not efficient (3 send, we can do with 1, but how?)

Not efficient (3 recv, we can do with 1, but how?)

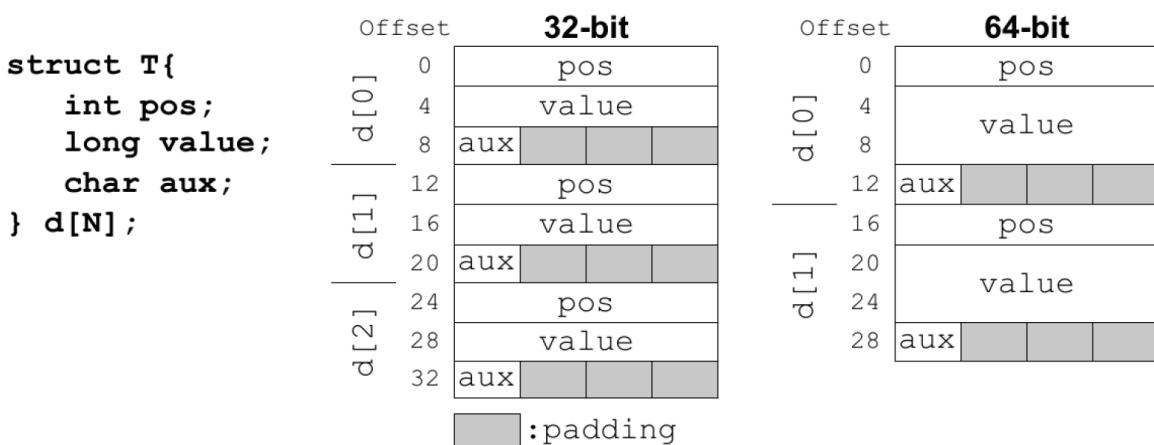
| Potevo usare i Datatype Derivati

Utilizzato per rappresentare qualsiasi raccolta di elementi di dati in memoria, memorizzando sia i tipi di elementi che le relative posizioni in memoria.

L'idea è che se una funzione che invia dati conosce queste informazioni su una raccolta di elementi di dati, può raccogliere gli elementi dalla memoria prima che vengano inviati.

Allo stesso modo, una funzione che riceve i dati può distribuire gli elementi nella loro corretta destinazione in memoria quando li riceve.

I problemi principali nella comunicazione con tipi di dato normali con funzioni normali potrebbe essere se ho dispositivi con diversi bit.



| Derived Datatypes

Formalmente, consiste in una sequenza di tipi di dati MPI di base insieme a uno spostamento per ciascuno dei tipi di dati.

- Esempio di regola trapezoidale:

Variable	Address
a	24d
b	40d
n	48d

{(MPI_DOUBLE,0),(MPI_DOUBLE,16),(MPI_INT,24)}

| MPI_Type_create_struct

Costruisce un tipo di dato derivato che consiste in singoli elementi che hanno tipi di base diversi.

```
int MPI_Type_create_struct(
    int             count          /* in   */,
    int             array_of_blocklengths[] /* in   */,
    MPI_Aint        array_of_displacements[] /* in   */,
    MPI_Datatype    array_of_types[]      /* in   */,
    MPI_Datatype*   new_type_p       /* out  */);
```

e.g., for
struct t{ count = 3
 double a; blocklengths = 1,1,1
 double b; displacements = 0,16,24
 int n; types = MPI_DOUBLE, MPI_DOUBLE, MPI_INT
};

Come ricavo gli offset, in questo caso 0,16,24?

| MPI_Get_address

Restituisce l'indirizzo della posizione di memoria a cui fa riferimento location_p.

Il tipo speciale MPI_Aint è un tipo intero sufficientemente grande per memorizzare un indirizzo sul sistema.

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

Perchè devo specificare sia puntatore che indirizzo?

Perchè in alcuni sistemi sono 2 cose diverse! Non posso usare solamente quindi un puntatore normale.

```
e.g., for                                MPI_Aint a_addr, b_addr, n_addr;  
struct t{  
    double a;  
    double b;  
    int n;  
};  
                                         MPI_Get_address(&a, &a_addr);  
                                         array_of_displacements[0] = 0;  
                                         MPI_Get_address(&b, &b_addr);  
                                         array_of_displacements[1] = b_addr - a_addr;  
                                         MPI_Get_address(&n, &n_addr);  
                                         array_of_displacements[2] = n_addr - a_addr;
```

| Perchè invece di fare questo?

```
MPI_Aint a_addr, b_addr, n_addr;  
  
MPI_Get_address(&a, &a_addr);  
array_of_displacements[0] = 0;  
MPI_Get_address(&b, &b_addr);  
array_of_displacements[1] = b_addr - a_addr;  
MPI_Get_address(&n, &n_addr);  
array_of_displacements[2] = n_addr - a_addr.  
}.
```

| Non facciamo

We don't do:

```
array_of_displacements[0] = 0;  
array_of_displacements[1] = &b - &a;  
array_of_displacements[2] = &n - &a;
```

?

| MPI_Type_commit

Consente all'implementazione MPI di ottimizzare la rappresentazione interna del tipo di dato per l'uso nelle funzioni di comunicazione.

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

| MPI_Type_free

Una volta terminato il nuovo tipo, si libera l'eventuale spazio di archiviazione utilizzato.

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

| Mettendo tutto insieme

```

void Build_mpi_type(
    double*      a_p          /* in */,
    double*      b_p          /* in */,
    int*         n_p          /* in */,
    MPI_Datatype* input_mpi_t_p /* out */ {

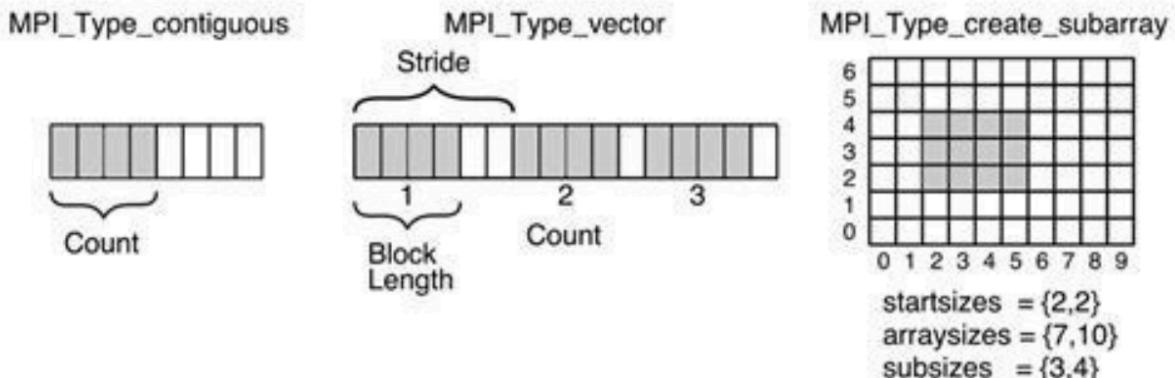
    int array_of_blocklengths[3] = {1, 1, 1};
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
    MPI_Aint a_addr, b_addr, n_addr;
    MPI_Aint array_of_displacements[3] = {0};

    MPI_Get_address(a_p, &a_addr);
    MPI_Get_address(b_p, &b_addr);
    MPI_Get_address(n_p, &n_addr);
    array_of_displacements[0] = b_addr - a_addr;
    array_of_displacements[1] = n_addr - a_addr;
    MPI_Type_create_struct(3, array_of_blocklengths,
                          array_of_displacements, array_of_types,
                          input_mpi_t_p);
    MPI_Type_commit(input_mpi_t_p);
} /* Build_mpi_type */

```

Si chiama **MPI_Type_create_struct**, ma gli elementi non devono necessariamente provenire da una struct (si può usare qualsiasi sequenza di variabili).

| Altre funzioni per nuovi Datatypes



- **MPI_Type_contiguous**: elementi contigui
- **MPI_Type_vector**: consiste in un numero di elementi dello stesso tipo di dato ripetuti con un certo stride (vedi es. 3.17 e 3.18)
- **MPI_Type_create_subarray**: per i subarray multidimensionali
- **MPI_pack /MPI_Unpack**: per impacchettare dati e inviare/ricevere dati impacchettati (vedere es. 3.20).

MPI definisce circa 400 funzioni, di cui circa 40 per la gestione di tipi di dati

| Threads Safety in MPI

Are MPI calls thread-safe? (e.g., can I call MPI_Send from multiple threads at the same time?)

Instead of MPI_Init, use:

```
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
```

The diagram shows four parameters of the MPI_Init_thread function with corresponding text below them. Three blue arrows point from the first three parameters to their respective descriptions. The fourth parameter has no arrow and is aligned vertically with the others.

Same as MPI_Init	Required <threading level> (TN)	Supported <threading level> (OUII)
int *argc	char ***argv	int required
		int *provided

| Threading Levels in MPI

- MPI_THREAD_SINGLE: rank is not allowed to use threads, which is basically equivalent to calling MPI_Init.
- MPI_THREAD_FUNNELED: rank can be multi-threaded but only the main thread may call MPI functions. Ideal for fork-join parallelism such as used in #pragma omp parallel, where all MPI calls are outside the OpenMP regions.
- MPI_THREAD_SERIALIZED: rank can be multi-threaded but only one thread at a time may call MPI functions. The rank must ensure that MPI is used in a thread-safe way. One approach is to ensure that MPI usage is mutually excluded by all the threads, e.g. with a mutex.
- MPI_THREAD_MULTIPLE: rank can be multi-threaded and any thread may call MPI functions. The MPI library ensures that this access is safe across threads. Note that this makes all MPI operations less efficient, even if only one thread makes MPI calls, so should be used only where necessary.
- ATTENTION: Not all the threading levels are supported by all the MPI implementations (e.g. some implementations might not support MPI_THREAD_MULTIPLE)

| 07 - Shared Memory Programming with Pthreads

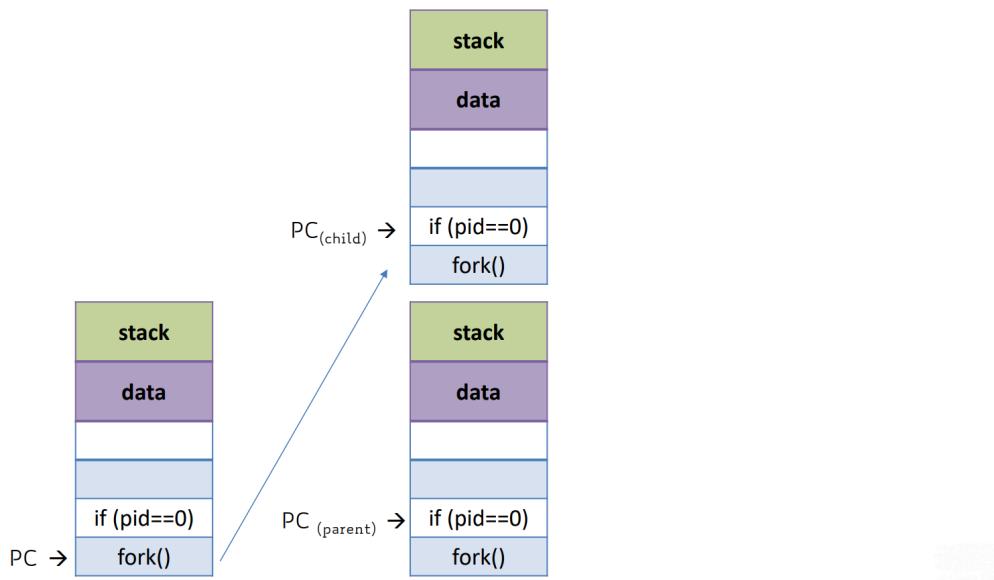
| Processi e Threads

Un processo è un'istanza di un programma in esecuzione (o sospeso).

I thread sono analoghi a un processo “leggero”.

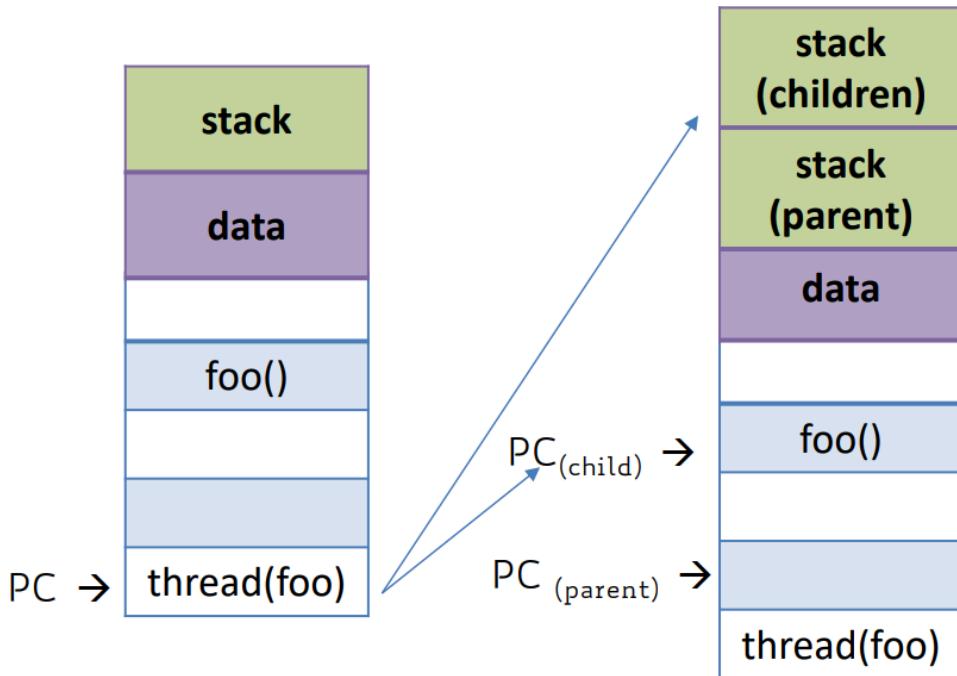
In un programma a memoria condivisa un singolo processo può avere più thread di controllo.

| Memory layout: process



N.B.: modern OSes usually use COW (copy-on-write) policy to optimize memory allocation

| Memory layout: threads



| POSIX THREADS

Conosciuto anche come Pthreads.

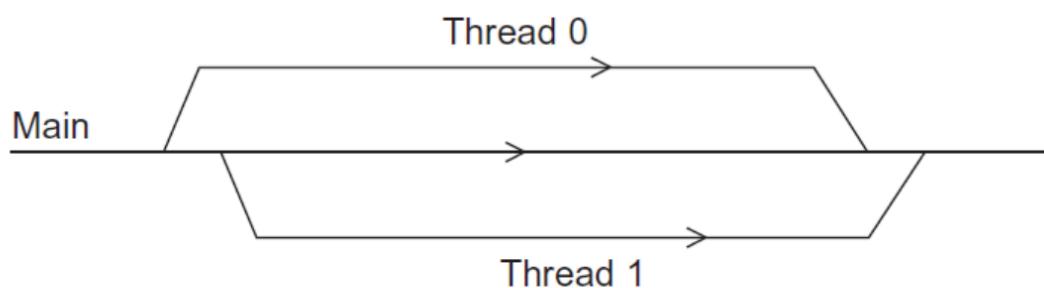
Uno standard per i sistemi operativi Unix-like.

Una libreria che può essere collegata ai programmi C.

Specifica un'interfaccia di programmazione delle applicazioni (API) per la programmazione multi-thread.

L'API Pthreads è disponibile solo su sistemi POSIX - Linux, MacOS X, Solaris, HPUX, ..

| Runnare i threads



Main thread forks and joins two threads.

| Come startare i threads?

I processi in MPI sono avviati da mpirun/mpexec.

In Pthreads i thread sono avviati direttamente dall'eseguibile del programma tramite la

```
funzione pthread_create.  
#include <pthread.h>  
int pthread_create (pthread_t* thread_p /* out */ ,  
                   const pthread_attr_t* attr_p /* in */ ,  
                   void* (*start_routine ) ( void* ) /* in */ ,  
                   void* arg_p /* in */ );
```

Vediamo ogni parametro passo passo:

| **pthread_t* thread_p**

Si tratta di un handle (uno per thread).

Deve essere allocato prima della chiamata

Opaco

I dati effettivi che memorizzano sono specifici del sistema.

I membri dei dati non sono direttamente accessibili al codice utente.

Tuttavia, lo standard Pthreads garantisce che un oggetto pthread_t memorizzi informazioni sufficienti per identificare in modo univoco il thread a cui è associato.

| **const pthread_attr_t* attr_p**

Noi non lo useremo, impostiamolo a NULL

| **void (start_routine) (void*)**

La funzione che il thread sta per eseguire

È un puntatore a una funzione (l'indirizzo di una porzione di memoria contenente codice piuttosto che dati)

In questo caso, abbiamo bisogno di una funzione che restituisca un void *e che prenda come argomento un void*.

| **void* arg_p**

Puntatore ai dati che saranno passati a start_routine

```

void* func(void* a)
{
    int* px = (int*) a;
    int x = *px;
    printf("x=%d\n", x);
}

void main()
{
    ...
    int x = 2;
    pthread_create(..., ..., func, (void*) &x);
    ...
}

```

I puntatori in C (Ripasso)

In C, come i normali puntatori ai dati (*int**, *char**, ecc.), possiamo avere puntatori alle funzioni.

Il nome di una funzione può essere usato per ottenere l'indirizzo della funzione.

```

void func(int a)
{
    printf("a=%d\n", a);
}

void main()
{
    void(*func_ptr)(int) = func;
    *func_ptr(10);
    printf("addr of func is: %p\n", func_ptr);
}

```

Funzione startata da `pthread_create`

Prototipo: `void* thread_function (void* args_p) ;`

`Void*` può essere lanciato a qualsiasi tipo di puntatore in C.

Quindi `args_p` può puntare a un elenco contenente uno o più valori necessari a `thread_function`.

Allo stesso modo, il valore di ritorno di `thread_function` può puntare a un elenco di uno o più valori.

Global variables

Può introdurre bug sottili e confusi!

Limitare l'uso delle variabili globali alle situazioni in cui sono veramente necessarie

```

int g; // Visible both from func and main

void* func(void* a)
{
    int* px = (int*) a;
    int x = *px;
    printf("x=%d\n", x);
}

void main()
{
    ...
    int x = 2;
    pthread_create(..., ..., func, (void*) &x);
    ...
}

```

Una volta aver startato un thread bisogna **ASPETTARLO**

| Aspettare che i threads terminino

La funzione **pthread_join** viene chiamata una volta per ogni thread.

Una singola chiamata a `pthread_join` attenderà che il thread associato all'oggetto `pthread_t` sia completato.

```
int pthread_join(pthread_t thread, void **value_ptr)
```

value_ptr (se non è NULL) ha il valore di ritorno della funzione di thread

| Come runnare ed aspettare più threads

```

for(int i = 0; i < num_threads; i++) {
    pthread_create(...);
}

for(int i = 0; i < num_threads; i++) {
    pthread_join(...);
}

```

VS.

```

for(int i = 0; i < num_threads; i++) {
    pthread_create(...);
    pthread_join(...);
}

```

Correct

Wrong
(everything would be executed sequentially)

| Thred Identification

`pthread_self` fornisce l'ID del thread chiamante

```
pthread_t pthread_self(void);
```

`pthread_equal` compara gli id dei thread

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

| Come passare più argomenti alla funzione del thread?

```
struct thread_args {
    long my_rank;
    char *task_name;
};

void *Hello(void *args) {
    struct thread_args* t_args
        = (struct thread_args *) args;
    printf("Thread %ld is working on task '%s'\n",
        t_args->my_rank, t_args->task_name);
    return NULL;
}
```

```
struct thread_args *t_args
= malloc(sizeof(struct thread_args));

t_args->my_rank = thread;
t_args->task_name = "Hello task";

pthread_create(&thread_handles[thread],
    NULL,
    Hello,
    (void *) t_args);
```



| QUANTI THREADS DOVREMMO RUNNARE?

In linea di principio, si dovrebbe cercare di evitare di avere più thread che core

Ci sono situazioni in cui potrebbe essere sensato (ne parleremo più avanti)

Come verificare quanti core si hanno?

```
$ lscpu | grep -E '^Thread|^Core|^Socket|^CPU\('
CPU(s):                  32
Thread(s) per core:      2
Core(s) per socket:       8
Socket(s):                 2
```

| 08 - Threads Exercises IMP

| 1. Prodotto Matrice - Vettore

Serial pseudo-code

```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

| Come farlo in parallelo?

In linea di principio, si dovrebbe cercare di evitare di avere più thread che core

Ci sono situazioni in cui potrebbe essere sensato (ne parleremo più avanti)

In ogni caso, in generale si partizioneranno le m righe su t thread, con $t < m$

Ciascuno elabora m/t righe

Il thread q elabora le righe a partire da $q \times \frac{m}{t}$ to $(q + 1) \times \frac{m}{t} - 1$

Nota: non è necessario fare scatter/broadcast, ogni thread accede alla stessa memoria/matrice/vettore.

ESERCIZIO MATRICVECTORPROD.C

| SEZIONE CRITICA | CONCORRENZA

Un modo per capire il funzionamento di una sezione critica è tramite la stima del pi con la sommatoria:

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots + (-1)^n \frac{1}{2n+1} + \dots \right)$$

Using a dual core processor

	n			10^8
	10^5	10^6	10^7	
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Note that as we increase n , the estimate with two threads diverge from the real value

Possible race condition

```
y = Compute( my_rank );
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute()	Started by main thread
3	Assign $y = 1$	Call Compute()
4	Put $x=0$ and $y=1$ into registers	Assign $y = 2$
5	Add 0 and 1	Put $x=0$ and $y=2$ into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

Come posso risolvere? BUSY-WAITING

Un thread testa ripetutamente una condizione, ma, di fatto, non svolge alcuna attività utile finché la condizione non assume il valore appropriato.

```
y = Compute( my_rank );
while (flag != my_rank);
x = x + y;
flag++;
flag initialized to 0 by main thread
```

Questo codice può essere riarrangiato dal compilatore come:

```

y = Compute( my_rank );
x = x + y ;
while ( flag != my_rank );
flag++;

```

il compilatore non sa se il codice utilizzerà o meno i thread. Potrebbe riorganizzare il codice in questo modo perché potrebbe ritenere di fare un uso migliore dei registri.

Utilizziamo il **BUSY WAITING** nell'esercizio del Pi greco (piv2.c)

Pthreads global sum with busy-waiting

```

void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        while (flag != my_rank);
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */

```

If I run this with n threads, this is slower than sequential code, why?

Global sum function with critical section after loop

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
        my_sum += factor/(2*i+1);

    while (flag != my_rank);
    sum += my_sum;
    flag = (flag+1) % thread_count;

    return NULL;
} /* Thread_sum */
```

Possiamo usare anche i **MUTEX** nell'esercizio del Pi greco (piv3.c)

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;
    double my_sum = 0.0;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        my_sum += factor/(2*i+1);
    }
    pthread_mutex_lock(&mutex);
    sum += my_sum;
    pthread_mutex_unlock(&mutex);

    return NULL;
} /* Thread_sum */
```

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38

Run-times (in seconds) of π programs using $n = 10^8$ terms on a system with two four-core processors.

In both cases, the critical section is outside the loop

| ESERCIZI

Sono stati svolti i seguenti esercizi in merito alla lezione:

- stimazione del pri greco con metodo montecarlo

| ALTRI ESERCIZI PRESENTI NELLA CARTELLA 06-11/LEC12

| 09 - Gestione Concorrenza e Sincronizzazione

| SEZIONE CRITICA

Come abbiamo visto nella prima versione del estimation of pi si può creare una sezione critica, nel seguente codice semplificato ne mostriamo un'altra.

Possible race condition

```
y = Compute(my_rank);  
x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute()	Started by main thread
3	Assign y = 1	Call Compute()
4	Put x=0 and y=1 into registers	Assign y = 2
5	Add 0 and 1	Put x=0 and y=2 into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

| Come posso risolvere? BUSY-WAITING

Un thread testa ripetutamente una condizione, ma, di fatto, non svolge alcuna attività utile finché la condizione non assume il valore appropriato.

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;  
flag initialized to 0 by main thread
```

Questo codice può essere riarrangiato dal compilatore come:

```
y = Compute(my_rank);  
x = x + y;  
while (flag != my_rank);  
flag++;
```

Il compilatore non sa se il codice utilizzerà o meno i thread. Potrebbe riorganizzare il codice in questo modo perché potrebbe ritenere di fare un uso migliore dei registri.

Per vedere il Busy-Waiting applicato -> [QUI](#)

| Mutex

Un thread in attesa può utilizzare continuamente la CPU senza ottenere alcun risultato.

Mutex (mutua esclusione) è un tipo speciale di variabile che può essere usato per limitare l'accesso a una sezione critica a un solo thread alla volta.

Utilizzato per garantire che un thread “escluda” tutti gli altri thread durante l'esecuzione della sezione critica.

Lo standard Pthreads include un tipo speciale per i mutex: `pthread_mutex_t`.

```
int pthread_mutex_init(  
    pthread_mutex_t*           mutex_p     /* out */  
    const pthread_mutexattr_t* attr_p     /* in  */);
```

Quando un programma Pthreads termina l'utilizzo di un mutex, dovrebbe chiamare

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```

Per ottenere l'accesso a una sezione critica un thread chiama

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p /* in/out */);
```

Quando un thread ha terminato l'esecuzione del codice in una sezione critica, dovrebbe chiamare

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```

La versione non bloccante di lock è:

```
int pthread_mutex_trylock(pthread_mutex_t* mutex_p /* in/out */)
```

ABBIAMO USATO I MUTEX NELL'ESERCIZIO DEL PI [QUI](#)

C'è un altro modo di implementarli:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

| Starvation

La starvation si verifica quando l'esecuzione di un thread o di un processo viene sospesa o non consentita per un periodo di tempo indefinito, sebbene sia in grado di continuare l'esecuzione.

Lo starvation è tipicamente associato all'applicazione delle priorità o alla mancanza di equità nella programmazione o nell'accesso alle risorse.

Se un mutex è bloccato, il thread viene bloccato e inserito in una coda Q di thread in attesa.
Se la coda Q impiegata da un semaforo è una coda FIFO, non si verifica l'inedia.

| Deadlocks

Deadlock: è qualsiasi situazione in cui nessun membro di un gruppo di entità può procedere perché ognuno aspetta che un altro membro membro, incluso se stesso, di intraprendere un'azione, come l'invio di un messaggio o, più comunemente, il rilascio di un blocco. messaggio o, più comunemente, il rilascio di un blocco.

- Ad esempio, bloccando i mutex in ordine inverso.

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

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

| Producer-consumer Synchronization and Semaphores

| Problemi

Il Busy-waiting impone l'ordine di accesso dei thread a una sezione critica.

Utilizzando i mutex, l'ordine è lasciato al caso e al sistema.

Esistono applicazioni in cui è necessario controllare l'ordine di accesso dei thread alla sezione critica.

Un esempio tipico è l'anello di thread per scambio di messaggi:

Example: message exchange in a ring (receive from left, send to right)

```
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main. */
void *Send_msg(void* rank) {
    long my_rank = (long) rank;
    long dest = (my_rank + 1) % thread_count;
    long source = (my_rank + thread_count - 1) % thread_count;
    char* my_msg = malloc(MSG_MAX*sizeof(char));

    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
    messages[dest] = my_msg;

    if (messages[my_rank] != NULL)
        printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
    else
        printf("Thread %ld > No message from %ld\n", my_rank, source);

    return NULL;
} /* Send_msg */
```

Problema

Alcuni thread potrebbero leggere i **messaggi[dest]** prima che l'altro thread vi inserisca qualcosa.

| Come risolvere?

Potremmo risolvere il problema con la busy waiting, ma si verificherebbero gli stessi problemi discussi in precedenza (utilizzo inutile di cpu e riordinamento delle istruzioni da parte del compilatore).

```
while (messages[my_rank] == NULL);
    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```

In linea di principio è possibile risolvere il problema con i mutex (ma in modo complesso).

POSIX fornisce un modo migliore: i **semafori** (non fanno parte di pthread).

Inoltre i mutex possono solo essere Bloccati - Non Bloccati poichè sono binari, i semafori hanno un valore iniziale (i mutex sempre 0) e possono avere più valori

| SINTASSI DELLE FUNZIONI DEI SEMAFORI

```
#include <semaphore.h>
```

Semaphores are not part of
Pthreads;
you need to add this.

```
int sem_init(  
    sem_t*      semaphore_p /* out */,  
    int         shared       /* in  */,  
    unsigned     initial_val /* in  */);
```

Semaphores can be shared also
among processes (shared !=0)

```
int sem_destroy(sem_t*   semaphore_p /* in/out */);  
int sem_post(sem_t*    semaphore_p /* in/out */);  
int sem_wait(sem_t*   semaphore_p /* in/out */);
```

`sem_wait(sem_t*sem)` blocca se il semaforo è 0. Se il semaforo è > 0, decrementa il semaforo e procede

`sem_post(sem_t*sem)` se c'è un thread in attesa in `sem_wait()`, questo thread può procedere con l'esecuzione. Altrimenti, il semaforo viene incrementato.

`sem_getvalue(sem_t *sem, int *sval)` inserisce il valore corrente del semaforo puntato da `sem` nel numero intero puntato da `sval`.

| Ulteriori note

- I mutex sono binari. Le semaforizzazioni sono di tipo `unsigned int`.
- I mutex partono sbloccati. Le semaforizzazioni partono con il valore iniziale.
- I mutex sono solitamente bloccati/sbloccati dallo stesso thread. i Semafori sono solitamente aumentate/diminuite da thread diversi.

Esempio di utilizzo di un semaforo:

Program 4.8: Using semaphores so that threads can send messages

```
1  /* messages is allocated and initialized to NULL in main */
2  /* semaphores is allocated and initialized to 0 (locked) in
   main */
3  void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      char* my_msg = malloc(MSG_MAX*sizeof(char));
7
8      sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9      messages[dest] = my_msg;
10     sem_post(&semaphores[dest])
11         /* ‘‘Unlock’’ the semaphore of dest */
12
13     /* Wait for our semaphore to be unlocked */
14     sem_wait(&semaphores[my_rank]);
15     printf("Thread %d > %s\n", my_rank, messages[my_rank]);
16
17 } /* Send_msg */
```

Note:

Questo problema non ha una sezione critica. Ha un tipo di sincronizzazione nota come produttore-consumatore

Barriers and Condition Variables

Un ulteriore metodo di sincronizzazione sono le barriere (viste con MPI_Barrier)

La sincronizzazione dei thread per assicurarsi che si trovino tutti nello stesso punto del programma è chiamata barriera.

Nessun thread può attraversare la barriera finché tutti i thread non l'hanno raggiunta.

Esempio di utilizzo delle barriere per cronometrare il thread più lento:

```

/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;

```

Possiamo usare le barriere anche per il **debug**

```

point in program we want to reach;
barrier;
if (my_rank == 0) {
    printf("All threads reached this point\n");
    fflush(stdout);
}

```

| Come implemento le Barriere?

| Possiamo usare **BUSY WAITING E MUTEX**

L'implementazione di una barriera utilizzando il busy-waiting e un mutex è semplice.

Utilizziamo un contatore condiviso protetto dal mutex.

Quando il contatore indica che ogni thread è entrato nella sezione critica, i thread possono lasciare la sezione critica.

Esempio:

```

/* Shared and initialized by the main thread */
int counter; /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

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

```

What if we want to use the barrier again?
 Someone must reset counter to 0. Who and when?
 If a thread reset it to 0, this might be reset
 before all the threads see it was equal to
 thread_count

Risposte:

1. Dobbiamo reimpostare il contatore a 0
2. Non posso avere un thread che lo riporta a 0, creerebbe molti problemi poichè altri thread potrebbero perdersi il primo incremento a 1 e potrebbero pensare di essere ancora bloccati

Per rimediare:

| Barriere con Semafori

```

/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */ ←
sem_t barrier_sem;    /* Initialize to 0 */

. . .

void* Thread_work(...) {
    . . .
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count - 1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count - 1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
}

```

Do we have the same
 problem here? Yes, there
 could be a race condition
 (check the book)

| Variabili di Condizione

Una variabile di condizione è un oggetto di dati che consente a un thread di sospendere l'esecuzione finché non si verifica un determinato evento o condizione.

Quando l'evento o la condizione si verifica, un altro thread può segnalare al thread di "svegliarsi".

Una variabile di condizione è sempre associata a un mutex.

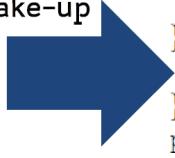
Una variabile di condizione indica un evento; non è possibile memorizzare o recuperare un valore da una variabile di condizione.

Esempio:

```
lock mutex;
if condition has occurred
    signal thread(s);
else {
    unlock the mutex and block;
    /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

Barriere con Variabili di Condizione

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    . . .
}
```

Spurious wake-up 

Ulteriori nozioni sulle Variabili di Condizione

Le variabili di condizione in Pthreads hanno il tipo `pthread_cond_t`.

La funzione sarà:

```
int pthread_cond_signal(pthread_cond_t* cond_var_p /* in/out */)
```

sblocca uno dei thread bloccati e

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);
```

sblocca tutti i thread bloccati. Le funzioni

```
int pthread_cond_init(
    pthread_cond_t* cond_p          /* out */,
    const pthread_condattr_t* cond_attr_p /* in */);
```

```
int pthread_cond_destroy(pthread_cond_t* cond_p /* in/out */);
```

Creare e distruggere la variabile di condizione.

La funzione:

```
int pthread_cond_wait(  
    pthread_cond_t*      cond_var_p /* in/out */,  
    pthread_mutex_t*     mutex_p     /* in/out */);
```

1. sbloccare il mutex;
2. bloccare il thread finché non viene sbloccato dalla chiamata di un altro thread a `pthread_cond_signal` o `pthread_cond_broadcast`
3. quando il thread viene sbloccato, bloccare il mutex.

È come:

```
pthread_mutex_unlock(&mutex_p);  
wait_on_signal(&cond_var_p);  
pthread_mutex_lock(&mutex_p);
```

`pthread_cond_wait()`, `pthread_cond_signal()` differiscono da `sem_wait()` e `sem_post()` perché:

- `pthread_cond_wait()` **Blocca SEMPRE** l'esecuzione del processo
- `pthread_cond_signal()` può essere ignorato. Se non ci sono thread in `cond_wait`, il segnale viene perso.
- `sem_t` può avere valori ≥ 0

| 10 - Locks e finale di Pthread

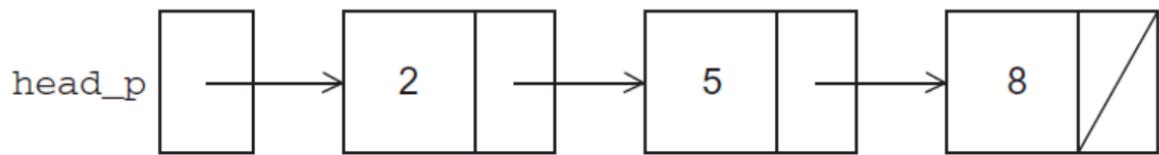
| Read-Write Locks

| Come posso controllare l'accesso a una struttura di dati condivisa di grandi dimensioni?

Partiamo con un esempio:

Supponiamo che la struttura di dati condivisa sia un elenco ordinato di ints e che le operazioni di interesse siano Member, Insert e Delete.

| Usiamo le Linked List



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

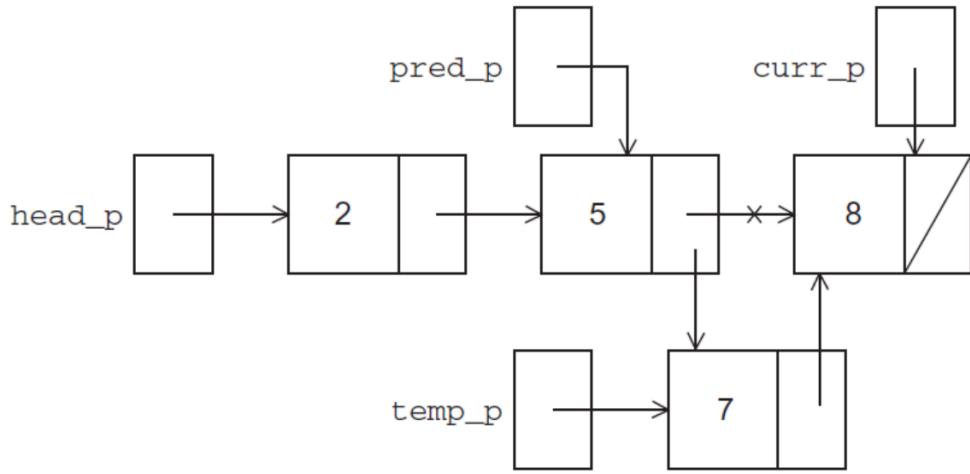
| Membership

```
int Member(int value, struct list_node_s* head_p) {
    struct list_node_s* curr_p = head_p;

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

    if (curr_p == NULL || curr_p->data > value) {
        return 0;
    } else {
        return 1;
    }
} /* Member */
```

| Inserimento



```

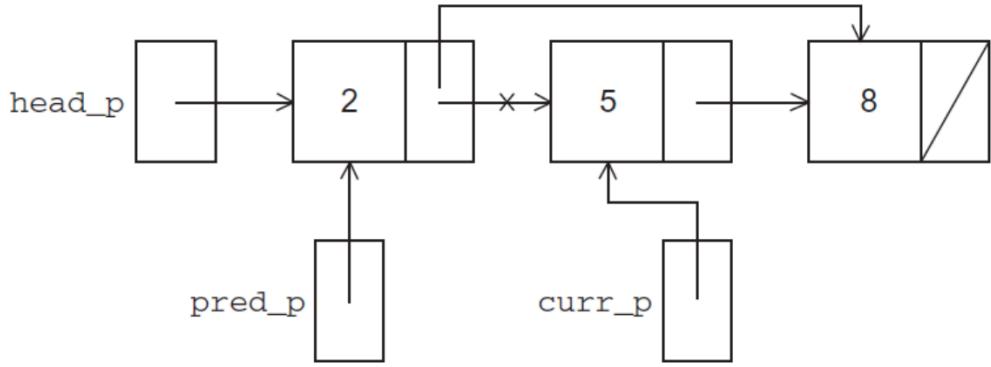
int Insert(int value, struct list_node_s** head_pp) {
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;
    struct list_node_s* temp_p;

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

    if (curr_p == NULL || curr_p->data > value) {
        temp_p = malloc(sizeof(struct list_node_s));
        temp_p->data = value;
        temp_p->next = curr_p;
        if (pred_p == NULL) /* New first node */
            *head_pp = temp_p;
        else
            pred_p->next = temp_p;
        return 1;
    } else { /* Value already in list */
        return 0;
    }
} /* Insert */

```

| Eliminazione



```

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

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

    if (curr_p != NULL && curr_p->data == value) {
        if (pred_p == NULL) { /* Deleting first node in list */
            *head_pp = curr_p->next;
            free(curr_p);
        } else {
            pred_p->next = curr_p->next;
            free(curr_p);
        }
        return 1;
    } else { /* Value isn't in list */
        return 0;
    }
} /* Delete */
    
```

| Multi-Threaded Linked List

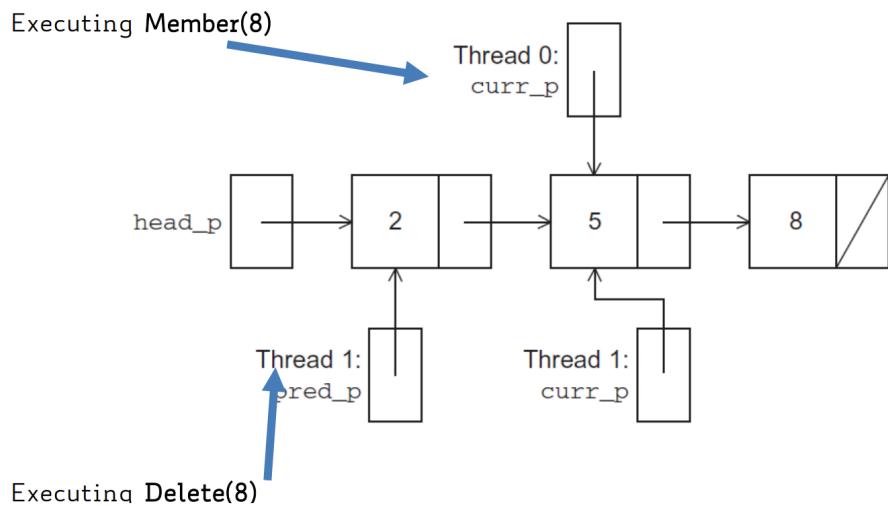
Proviamo a utilizzare queste funzioni in un programma Pthreads.

Per condividere l'accesso all'elenco, possiamo definire **head_p** come una variabile globale. Questo semplificherà le intestazioni delle funzioni Member, Insert e Delete, poiché non dovremo passare né **head_p** né un puntatore a **head_p**: dovremo solo passare il valore di interesse.

Se più thread chiamano Member allo stesso tempo, siamo a posto

E se un thread chiama Member mentre un altro thread sta eliminando un elemento?

Simultaneous access by two threads



Questo genererebbe un errore perchè sto cercando con un thread di eliminare un elemento puntato da un altro thread.

Soluzione n°1

Una soluzione ovvia è quella di bloccare l'elenco ogni volta che un thread tenta di accedervi. Una chiamata a ciascuna delle tre funzioni può essere protetta da un **mutex**.

```
Pthread_mutex_lock(&list_mutex);  
Member(value);  
Pthread_mutex_unlock(&list_mutex);
```

Problemi

- Stiamo serializzando l'accesso all'elenco.
- Se la maggior parte delle operazioni sono chiamate a Member, non riusciremo a sfruttare questa opportunità di parallelismo.
- D'altra parte, se la maggior parte delle operazioni è costituita da chiamate a Insert e Delete, questa potrebbe essere la soluzione migliore, dato che avremo bisogno di serializzare l'accesso all'elenco per la maggior parte delle operazioni e questa soluzione sarà sicuramente facile da implementare.

Soluzione n°2

Invece di bloccare l'intero elenco, si potrebbe provare a bloccare i **singoli nodi**. Un approccio “a grana fine”

```

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

```

| Problemi

- È molto più complesso della funzione Member originale.
- È anche molto più lenta, poiché, in generale, ogni volta che si accede a un nodo, un mutex deve essere bloccato e sbloccato.
- L'aggiunta di un campo mutex a ogni nodo aumenterà notevolmente la quantità di memoria necessaria per la lista

| Read-Write Locks in Pthreads

Nessuna delle nostre liste collegate multi-thread sfrutta il potenziale di accesso simultaneo a qualsiasi nodo da parte di thread che stanno eseguendo membri.

La prima soluzione consente a un solo thread di accedere all'intero elenco in qualsiasi momento.

La seconda soluzione consente a un solo thread di accedere a un determinato nodo in qualsiasi istante.

Un lock di lettura-scrittura è simile a un mutex, tranne per il fatto che fornisce **due funzioni di blocco**.

La prima funzione del lock blocca il blocco lettura-scrittura per la lettura, mentre la seconda lo blocca per la scrittura.

Quindi più thread possono ottenere simultaneamente il blocco chiamando la funzione read-lock, mentre solo un thread può ottenere il blocco chiamando la funzione write-lock.

Pertanto, se un thread possiede il blocco per la lettura, tutti i thread che vogliono ottenere il blocco per la scrittura si bloccheranno durante la chiamata alla funzione write-lock.

Se un thread possiede il blocco per la scrittura, tutti i thread che vogliono ottenere il blocco per la lettura o la scrittura si bloccheranno nelle rispettive funzioni di blocco.

| pthread_rwlock functions

`pthread_rwlock_init` inizializza il rwlock

```
int pthread_rwlock_init(pthread_rwlock_t* rwlock,
                      pthread_rwlockattr_t* attr);
```

ATTENZIONE

Possiamo settare `pthread_rwlockattr_t*` come `NULL`, è utilizzato per specificare se i lettori

devono avere la priorità sugli scrittori o viceversa.

Controllare `pthread_rwlockattr_setkind_np`

`pthread_rwlock_destroy` libera il rwlock

~~int pthread_rwlock_destroy(pthread_rwlock_t* rwlock);~~

| Protecting our linked list functions

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);

. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);

. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

LINKED LIST PERFORMANCE

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

8 (physical) cores

| Take-home message

I lock read-write offrono un **guadagno** in termini di prestazioni, a condizione che il numero di inserimenti/cancellazioni sia ridotto rispetto al numero di operazioni sui membri.

| Thread-Safety

| Esempio

Un blocco di codice è **thread-safe** se può essere eseguito contemporaneamente da più thread senza causare problemi.

Supponiamo di voler usare più thread per “tokenizzare” un file che consiste in un normale testo inglese.

I token sono sequenze contigue di caratteri separate dal resto del testo da spazi bianchi (uno spazio, una tabulazione o una newline).

| Approccio Semplice

Dividere il file di input in righe di testo e assegnare le righe ai thread in modo round-robin.
La prima riga va al thread 0, la seconda al thread 1, . . . , la t-esima va al thread t, la t+1-esima va al thread 0, ecc.

Possiamo serializzare l'accesso alle righe di input utilizzando i semafori.

Perché semafori e non mutex?

Perchè vogliamo che la divisione sia round robin

Dopo che un thread ha letto una singola riga di input, può tokenizzare la riga usando la funzione **strtok**.

| The strtok function

La prima volta che viene chiamato, l'argomento della stringa deve essere il testo da tokenizzare.

- La nostra riga di input.

Per le chiamate successive, il primo argomento **deve essere NULL**

```
char* strtok(  
    char*      string      /* in / out */ ,  
    const char* separators /* in */ );
```

L'idea è che alla prima chiamata, strtok memorizzi nella cache un puntatore alla stringa e per le chiamate successive **restituisca i token successivi presi dal puntatore memorizzato nella cache**.

| Multi-threaded tokenizer

IL CODICE E' NELLA CARTELLA DELLA LEZIONE ODIERNA (5-11)

| Runnare con un thread

Codifica correttamente il flusso di input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

| Runnare con due thread

```
Thread 0 > my line = Pease porridge hot.  
Thread 0 > string 1 = Pease  
Thread 0 > string 2 = porridge  
Thread 0 > string 3 = hot.  
Thread 1 > my line = Pease porridge cold.  
Thread 0 > my line = Pease porridge in the pot  
Thread 0 > string 1 = Pease  
Thread 0 > string 2 = porridge  
Thread 0 > string 3 = in  
Thread 0 > string 4 = the  
Thread 0 > string 5 = pot  
Thread 1 > string 1 = Pease  
Thread 1 > my line = Nine days old.  
Thread 1 > string 1 = Nine  
Thread 1 > string 2 = days  
Thread 1 > string 3 = old.
```

Oops!

| Cos'è Successo?

strtok memorizza il puntatore alla riga di input dichiarando una variabile con classe di memorizzazione statica.

Ciò fa sì che il valore memorizzato in questa variabile persista da una chiamata all'altra. Sfortunatamente per noi, la stringa memorizzata nella cache è condivisa, non privata.

Pertanto, la chiamata del thread 0 a strtok con la terza riga dell'input ha apparentemente sovrascritto il contenuto della chiamata del thread 1 con la seconda riga.

La funzione strtok non è quindi thread-safe.

Se più thread la chiamano contemporaneamente, l'output potrebbe non essere corretto.

| QUALI SONO LE ALTRE LIBRERIE C NON SAFE?

Purtroppo, non è raro che le funzioni della libreria C non siano thread-safe.

Il generatore di numeri casuali random in stdlib.h.

La funzione di conversione del tempo localtime in time.h.

Nelle vecchie versioni dei sistemi POSIX, anche srand/rand non erano thread safe.

| Funzioni "re-entrant"

In alcuni casi, lo standard C specifica una versione alternativa, thread-safe, di una funzione.

```
char* strtok_r(  
    char* string /* in/out */ ,  
    const char* separators , /* in */  
    char** saveptr_p /* in/out */ );
```

In linea di principio “**re-entrant**” != “**thread safe**”

In pratica, le funzioni re-entrant sono spesso anche thread safe

In caso di dubbio, controllare la documentazione!

- `man strtok_r`

| VARIE

| Static Initializers

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
  
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

| Thread Pinning

Cosa succede se si vuole forzare un thread a girare su un core specifico? Come farlo?

```
#define _GNU_SOURCE  
#include <pthread.h>  
  
void* thread_func(void* thread_args){  
    ...  
    cpu_set_t cpuset;  
    pthread_t thread = pthread_self();  
    /* Set affinity mask to include core 3 */  
    CPU_ZERO(&cpuset);  
    CPU_SET(3, &cpuset);  
    s = pthread_setaffinity_np(thread, sizeof(cpu_set), &cpuset);  
    ...  
}
```

| Prendere i tempi dell'esecuzione

```

#include <sys/time.h>

#define GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}

int main(){
    ...
    double start, finish, elapsed;
    GET_TIME(start);
    ...
    // Code to be timed
    ...
    GET_TIME(finish);
    elapsed = finish - start;
    printf("The code to be timed took %e seconds\n", elapsed);
    ...
}

```

la `#define GET_TIME(now)` è una **macro**

| RIPASSO GESTIONE CONCORRENZA E LOCKS

Un **thread** nella programmazione a memoria condivisa è analogo a un processo nella programmazione a memoria distribuita.

Tuttavia, un thread è spesso più leggero di un processo vero e proprio.

Nei programmi Pthreads, tutti i thread hanno accesso alle variabili globali, mentre le variabili locali sono solitamente private del thread che esegue la funzione.

Quando l'indeterminazione deriva da più thread che tentano di accedere a una risorsa condivisa, come una variabile condivisa o un file condiviso, almeno uno degli accessi è un aggiornamento e gli accessi possono dare luogo a un errore, si ha una **race condition**.

| Sezione Critica

Una **sezione critica** è un blocco di codice che aggiorna una risorsa condivisa che può essere aggiornata solo da un thread alla volta.

Quindi l'esecuzione del codice in una sezione critica deve essere eseguita come codice seriale.

| Busy-waiting

Il **Busy-waiting** può essere utilizzato per evitare accessi conflittuali alle sezioni critiche con una variabile flag e un ciclo while con un corpo vuoto.
Può essere molto dispendioso in termini di cicli di CPU.
Può anche essere inaffidabile se l'ottimizzazione del compilatore è attivata.

| Mutex

Un **mutex** può essere utilizzato anche per evitare accessi conflittuali alle sezioni critiche. Consideratelo come un blocco su una sezione critica, poiché i mutex consentono l'accesso reciprocamente esclusivo a una sezione critica.

| Semafori

Il **semaforo** è il terzo modo per evitare accessi conflittuali alle sezioni critiche. È un int senza segno insieme a due operazioni: sem_wait e sem_post. I semafori sono più potenti dei mutex, poiché possono essere inizializzati a qualsiasi valore non negativo.

| Barriera

Una **barriera** è un punto del programma in cui i thread si bloccano finché non lo raggiungono tutti.

| Read-Write Locks

Un **read-write lock** viene utilizzato quando è sicuro che più thread leggano simultaneamente una struttura di dati, ma se un thread deve modificare o scrivere sulla struttura di dati, solo quel thread può accedere alla struttura di dati durante la modifica.

Alcune funzioni C memorizzano i dati tra le chiamate dichiarando le variabili come statiche, causando errori quando più thread chiamano la funzione.

Questo tipo di funzione non è **thread-safe**.

| 11 - Architettura Multicore

| CACHING

Un insieme di posizioni di memoria a cui si può accedere in un tempo minore rispetto ad altre posizioni di memoria.

Una cache della CPU si trova in genere sullo stesso chip o su uno a cui si può accedere molto più velocemente della memoria ordinaria (di solito è fisicamente più vicina).

Utilizza anche una tecnologia più performante (ma anche più costosa)

- (ad esempio, SRAM invece di DRAM).
Quindi sarà più veloce ma più piccola

| Cosa dobbiamo Cachare?

Si presuppone la località (sia per le istruzioni che per i dati).

Cioè, l'accesso a una posizione è seguito da un accesso a una posizione vicina

- **Località spaziale** - accesso a una posizione vicina.
- **Località temporale** - accesso a una posizione vicina.

Locality: example

```
float z [ 1 000 ]; z[i] follows z[i-1] in memory  
          (spatial locality)  
.  
.  
. sum = 0.0; Accesses to z[i] follows access to z[i-1]  
          (temporal locality)  
for ( i = 0; i < 1000; i ++ )  
    sum += z [ i ];
```

| Cache Lineare

I dati vengono trasferiti dalla memoria alla cache in blocchi/linee:

- cioè, quando $z[0]$ viene trasferito dalla memoria alla cache, potrebbero essere trasferiti anche $z[1], z[2], \dots, z[15]$
Effettuare un trasferimento di 16 posizioni di memoria è meglio che effettuare 16 trasferimenti di una posizione di memoria ciascuno
Quando si accede a $z[0]$ è necessario attendere il trasferimento, ma poi si troveranno già gli altri 15 elementi nella cache.

| Livelli di Cache

smallest & fastest



L2

L3



largest & slowest

- I dati memorizzati in L1 possono o meno essere memorizzati anche in L2/L3 (dipende dal tipo di cache)
- La CPU controlla innanzitutto se i dati sono in L1, in caso contrario controlla in L2, ecc.

| Cache Hit

Quando il sistema operativo trova il dato cercato in cache

| Cache Miss

Quando il dato cercato non è in memoria

| Perchè ci interessa come funziona?

Per scrivere codice parallelo efficiente/performante:

- Le sue parti sequenziali devono essere efficienti/performanti
 - Provate a pensare a come la vostra applicazione accede ai dati. Gli accessi casuali sono molto peggiori degli accessi lineari
- La coordinazione tra queste parti sequenziali deve essere fatta in modo efficiente

| GLI ESERCIZI DELLA LEZIONE SONO IN /MultiCore/EserciziC/pthreads/06-11/lec12

| CONSISTENZA

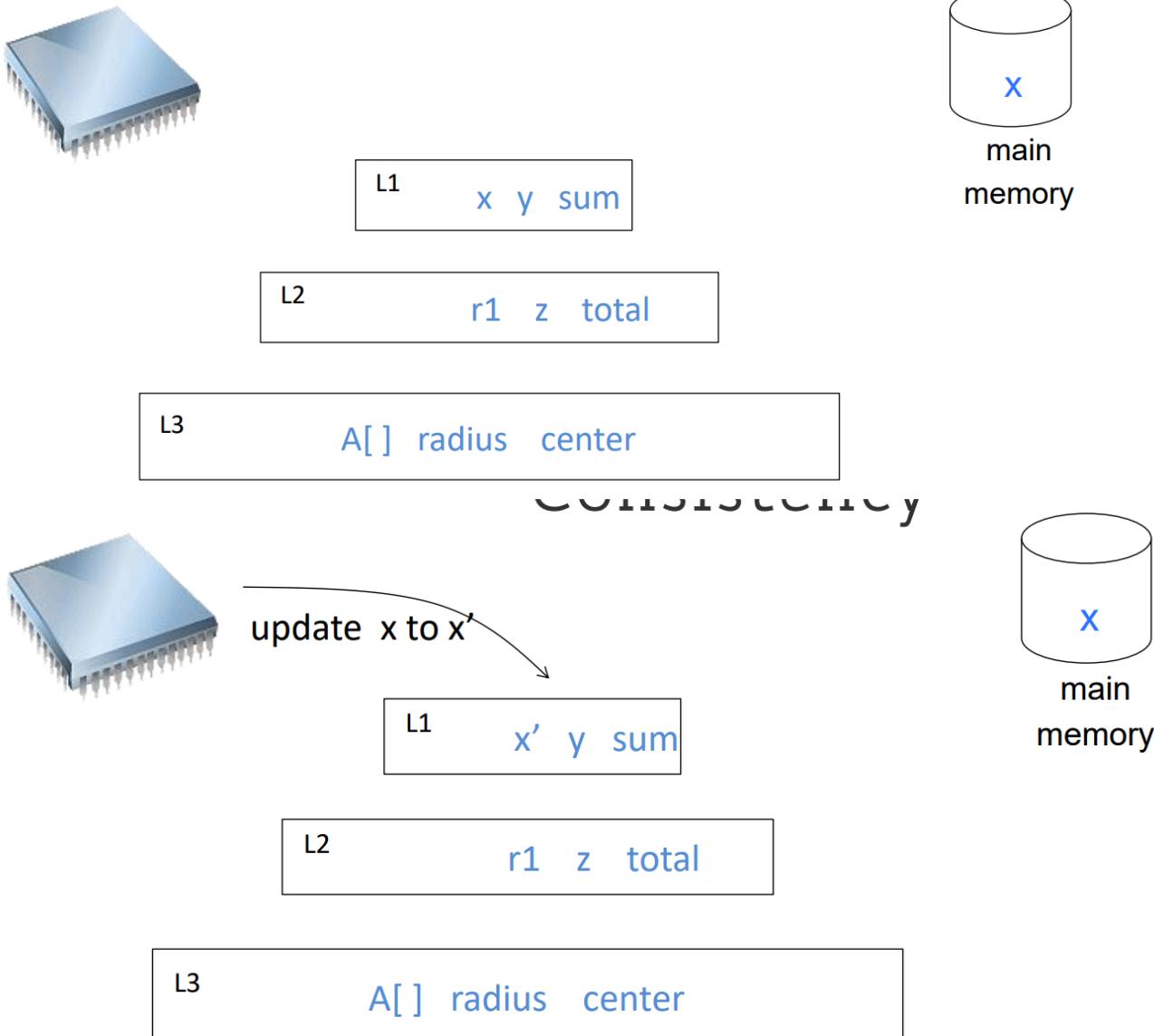
Quando una CPU scrive dati nella cache, il valore nella cache può essere incoerente con il valore nella memoria principale.

Le cache Write-through gestiscono questo problema aggiornando i dati nella memoria

principale nel momento in cui vengono scritti nella cache.

Le cache Write-back contrassegnano i dati nella cache come sporchi.

Quando la riga della cache viene sostituita da una nuova riga dalla memoria, la riga sporca viene scritta in memoria.



| ESERCIZI SULLE CACHE

Si trovano nella cartella della lezione odierna (06-11)

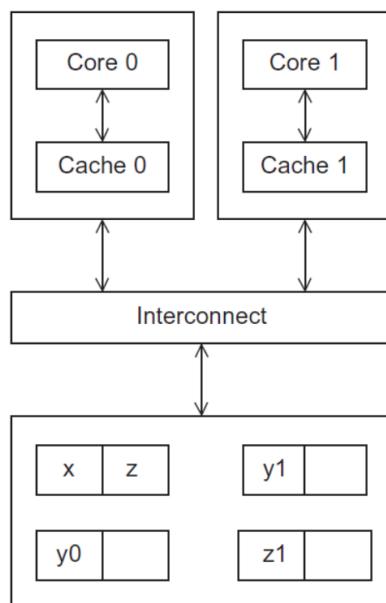
- Github, lec12, cache_01_*.c
- Use "perf" to measure number of cache misses. What's happening? How to fix cache_01_slow.c?
- You can do that at a finer grain from the code itself, by using a library called PAPI (<https://icl.utk.edu/papi/>)

- Similar effects can happen when swapping from memory to disk

| CACHING SU SISTEMI MULTICORE

| Coerenza della cache

I programmati non hanno alcun controllo sulle cache e su quando vengono aggiornate.



Un sistema con 2 core e 2 cache

```

y0 privately owned by Core 0
y1 and z1 privately owned by Core 1
x = 2; /* shared variable */

```

Time	Core 0	Core 1
0	y0 = x;	y1 = 3*x;

Core 0 cache:

x = 2

y0 = 2

Core 1 cache:

x = 2

y1 = 6

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

Time	Core 0	Core 1
0	y0 = x;	y1 = 3*x;
1	x = 7;	Statement(s) not involving x

Core 0 cache:

x = 7

y0 = 2

Core 1 cache:

x = 2

y1 = 6

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

Time	Core 0	Core 1
0	y0 = x;	y1 = 3*x;
1	x = 7;	Statement(s) not involving x
2	Statement(s) not involving x	z1 = 4*x;

Core 0 cache:

x = 7

y0 = 2

Core 1 cache:

x = 2

y1 = 6

z1 = 4*2 or 4*7?

N.B.: occurs for both WT and WB policies

| Coerenza della cache basata su Snooping

I core condividono un bus.

Qualsiasi segnale trasmesso sul bus può essere “visto” da tutti i core collegati al bus.

Quando il core 0 aggiorna la copia di x memorizzata nella sua cache, trasmette anche questa informazione sul bus.

Se il core 1 “snobba” il bus, vedrà che x è stato aggiornato e potrà contrassegnare la sua copia di x come non valida.

Non è più molto usato: il broadcast è costoso, oggi abbiamo multicores con 64/128 core.

| Coerenza della cache basata su Directory

Utilizza una struttura di dati chiamata directory che memorizza lo stato di ogni linea di cache. Quando una variabile viene aggiornata, la directory viene consultata e i controllori della cache dei core che hanno la linea di cache di quella variabile nella loro cache vengono **invalidati**.

| Ulteriori esercizi cache

Sempre presenti nella cartella della lezione odierna

| FALSE SHARING

I dati vengono recuperati in memoria per la cache in righe.

Ogni riga può contenere diverse variabili

- Ad esempio, se una cache è lunga 64 byte, può contenere 16 interi da 4 byte (che sono consecutivi in memoria)

Quando i dati vengono invalidati, l'intera riga viene invalidata

Anche se due thread accedono a due variabili diverse, se queste si trovano sulla stessa riga di cache, ciò causerebbe comunque un'invalidazione

| SOLUZIONI al False Sharing

Cercate di forzare le variabili a cui accedono thread diversi a stare su linee di cache diverse

Padding (ma attenzione, il compilatore potrebbe cambiare l'ordine dei campi in una struct).

Per evitarlo, fate qualcosa di simile (per GCC, assumendo linee di cache da 64 byte):

```
struct alignTo64ByteCacheLine {  
    int _onCacheLine1 __attribute__((aligned(64)))  
    int _onCacheLine2 __attribute__((aligned(64)))  
}
```

Come ottenere la dimensione della linea di cache?

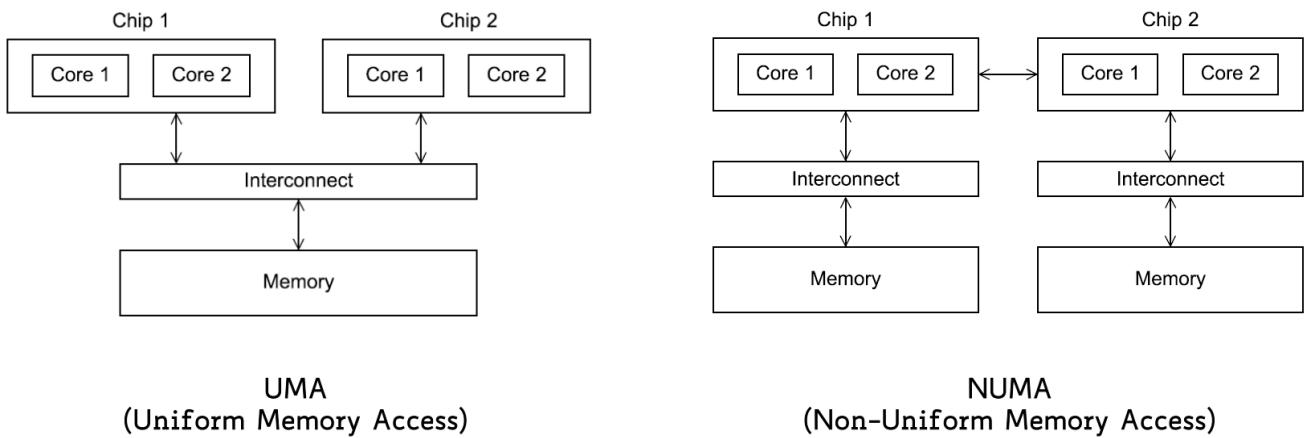
Dal codice:

- `sysconf (_SC_LEVEL1_DCACHE_LINESIZE)`
Dalla shell:

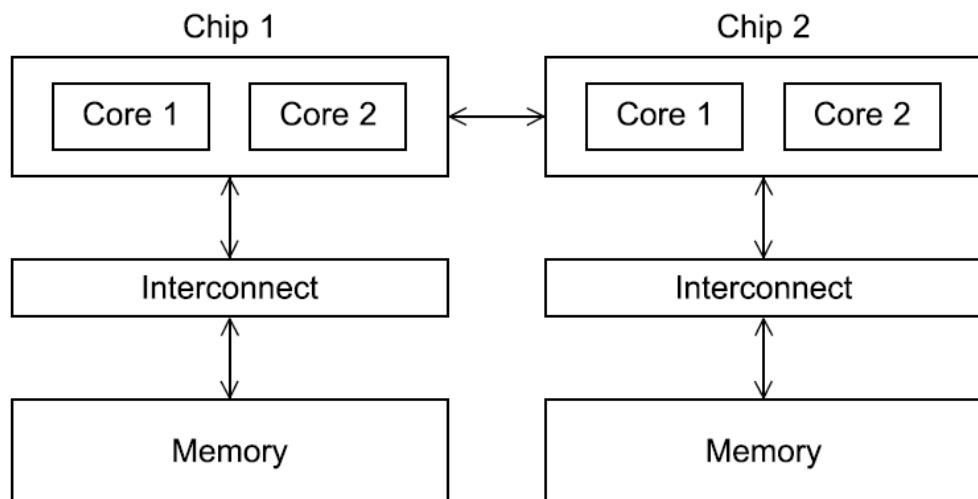
- `getconf LEVEL1_DCACHE_LINESIZE`

Eseguire tutti gli aggiornamenti su una variabile locale al thread (ad esempio, allocata sullo stack, e poi scritta sull'heap solo alla fine)

| ORGANIZZAZIONE DELLA MEMORIA



| NUMA systems



Il costo dell'accesso alla memoria cambia a seconda di dove vengono allocati i dati

Accedere a una memoria “locale” è più economico che accedere a una memoria “remota”. È possibile specificare dove devono essere allocati i dati (si veda la libreria numa.h)

È possibile avere un certo controllo dalla riga di comando (si veda numactl)

| Problematiche

Alcuni core potrebbero essere più “vicini” alla NIC, quindi potrebbe avere senso che siano quelli che eseguono MPI_Send/MPI_Recv/ecc.

| VARIE

| Profiling

Supponiamo di avere un pezzo di codice con molte funzioni: come sapere dove si spende la maggior parte del tempo e, quindi, dove avrebbe più senso ottimizzare?

(Ricordate: "L'ottimizzazione prematura è la radice di tutti i mali")

Potremmo cronometrare esplicitamente ogni funzione (noioso, richiede molto tempo)

Possiamo usare un profiler che lo faccia per noi:

- (vedere https://hpc-wiki.info/hpc/Gprof_Tutorial).

| 12 - OpenMP

Un'API per la programmazione parallela a memoria condivisa.

MP = multiprocessing

Progettato per sistemi a memoria condivisa.

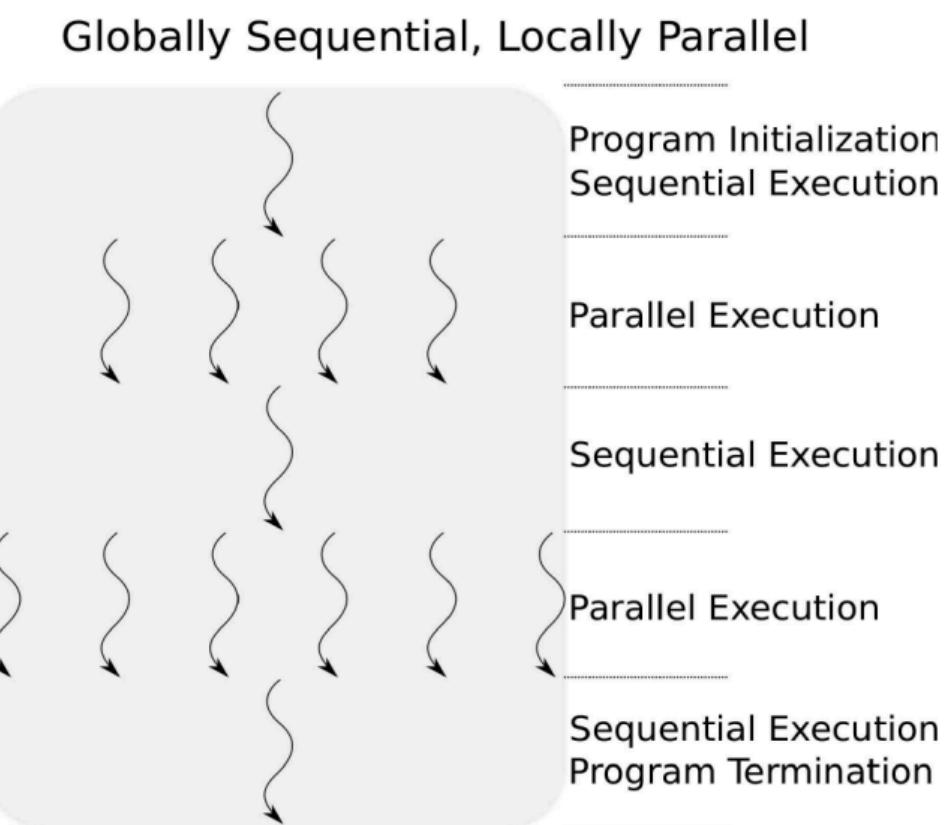
Il sistema è visto come un insieme di core o CPU, tutti con accesso alla memoria principale.

OpenMP mira a scomporre un programma sequenziale in componenti che possono essere eseguiti in parallelo.

OpenMP consente una conversione “incrementale” di programmi sequenziali in programmi paralleli, con l'assistenza del compilatore.

OpenMP si affida alle direttive del compilatore per la scelta delle porzioni di codice che il compilatore tenterà di parallelizzare.

I programmi OpenMP sono globalmente sequenziali, localmente paralleli. I programmi seguono il paradigma fork-join:



| Pragmas

Istruzioni speciali del preprocessore.

In genere vengono aggiunte a un sistema per consentire comportamenti che non fanno parte della specifica C di base.

I compilatori che non supportano i pragmi li ignorano.

```
# pragma omp parallel
```

Direttiva parallela di base.

Il numero di thread che eseguono il seguente blocco di codice strutturato è determinato dal sistema di runtime.

ESEMPIO:

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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */

gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4
```

running with 4 threads

possible outcomes

compiling

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4

| Thread Team Size Control

| Universalmente

Tramite MP_NUM_THREADS environmental variable:

```
$ echo ${OMP_NUM_THREADS} # to query the value
$ export OMP_NUM_THREADS=4 # to set it in BASH
```

| Livello di Programma

Tramite la funzione `omp_set_num_threads` fuori da un costruttore OpenMP

| Livello di Pragma

Tramite il `num_threads`

Precedenze:

- Universally/env. Variable
- Program level

- Pragma level

```

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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

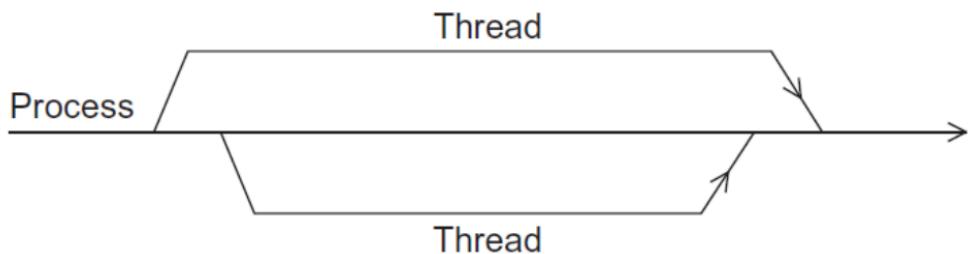
    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
}
    
```

| Esempio di process forking and joining two threads



| Clausola

Testo che modifica una direttiva.

La clausola num_threads può essere aggiunta a una direttiva parallela.

Consente al programmatore di specificare il numero di thread che devono eseguire il seguente blocco

| Note

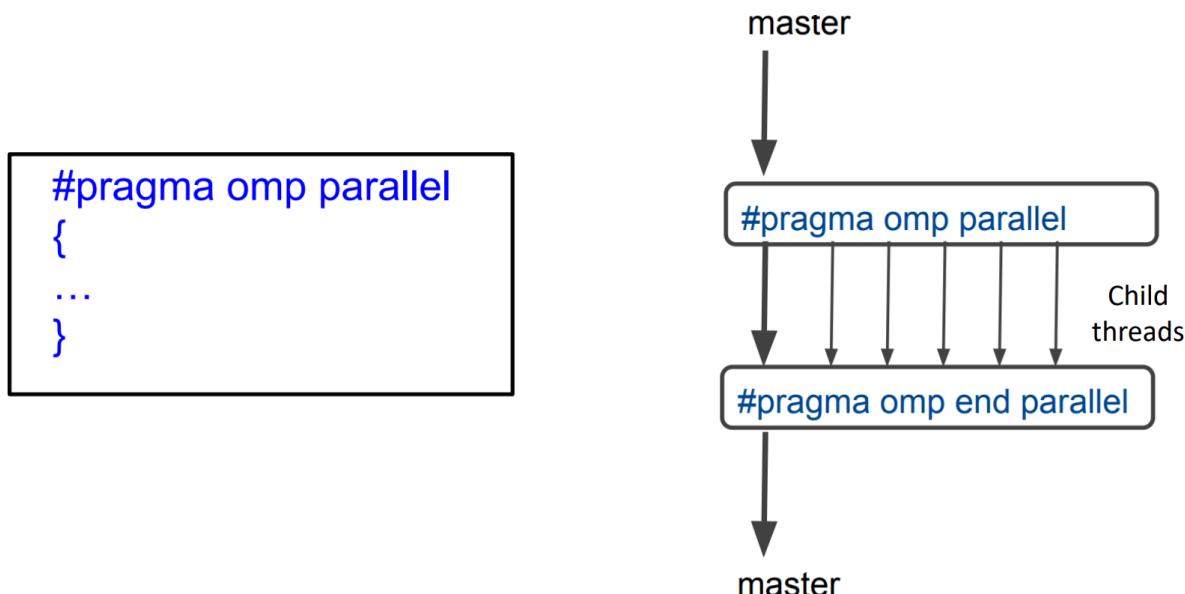
- Potrebbero esserci limitazioni definite dal sistema sul numero di thread che un programma può avviare.
- Lo standard OpenMP non garantisce che il programma avvii effettivamente thread_count.
- La maggior parte dei sistemi attuali può avviare centinaia o addirittura migliaia di thread.
- A meno che non si voglia avviare un numero elevato di thread, si otterrà quasi sempre il numero di thread desiderato.
- Dopo il completamento di un blocco, c'è una barriera implicita.

| Terminologia

Nel linguaggio OpenMP l'insieme dei thread che eseguono il blocco parallelo - il thread originale e i nuovi thread
è chiamato team

- **master**: il thread originale di esecuzione
- **parent**: thread che ha incontrato una direttiva parallela e ha avviato un team di thread.
In molti casi, il genitore è anche il thread master.
- **figlio**: ogni thread avviato dal genitore è considerato un thread figlio.

| Parallel Construct



```
1 #include<stdio.h>
2 #include<omp.h>
3
4 int main()
5 {
6     printf("Only master thread here \n");
7
8     #pragma omp parallel
9     {
10         int tid = omp_get_thread_num();
11         printf("Hello I am thread number %d \n", tid);
12     }
13
14     printf("Only master thread here \n");
15 }
```

1

2

3

Only master thread here
Hello I am thread number 0
Hello I am thread number 1
Hello I am thread number 2
Hello I am thread number 3
Only master thread here

int tid

4

*Each thread has a
private copy of
variable*

- Specifying number of threads
 - `export OMP_NUM_THREADS=4`
- Execute:
 - `./basic.exe`

ESEMPIO DI UTILIZZO:[12.1 - Regola del Trapezio con OpenMP](#)

| OpenMP vs Pthreads

OpenMP fornisce un'interfaccia di astrazione superiore

- Sui sistemi POSIX, molto probabilmente è implementato sopra Pthreads
- Sui sistemi non POSIX (ad esempio, Windows), sarà implementato sopra un'altra astrazione di threading - Pertanto, è possibile che OpenMP sia implementato sopra un'altra astrazione di threading, Windows), sarà implementato sopra un'altra astrazione di threading
- Pertanto, è più portabile di Pthreads (si scrive il codice OpenMP e lo si esegue "ovunque" senza modificarlo)
- Alcuni costrutti OpenMP possono essere usati per eseguire codice su GPU (non lo vedremo) -

Pthreads offre un controllo molto più fine

- Come al solito è una questione di compromessi tra facilità d'uso e flessibilità/prestazioni Mescolare OpenMP e Pthreads nello stesso codice dovrebbe funzionare, ma potrebbero esserci casi particolari in cui non è così. Di solito è meglio usare l'uno o l'altro, a meno che non ci siano ragioni molto forti per mischiarli.

| Scope delle Variabili

| Scope

Nella programmazione seriale, l'ambito di una variabile è costituito dalle parti del programma in cui la variabile può essere utilizzata.

In OpenMP, l'ambito di una variabile si riferisce all'insieme dei thread che possono accedere alla variabile in un blocco parallelo.

| Scope in OpenMP

Una variabile a cui possono accedere tutti i thread del team ha scope condiviso.

Una variabile a cui può accedere solo un singolo thread ha scope privato.

L'ambito predefinito per le variabili dichiarate prima di un blocco parallelo è **condiviso**.

```

...
int x;           ← Shared
#pragma omp parallel
{
    int y;       ← Private
    ...
}
...

```

| La clausola predefinita

Consente al programmatore di specificare l'ambito di ciascuna variabile in un blocco.

`default(None)`

Con questa clausola il compilatore richiederà di specificare l'ambito di ogni variabile utilizzata nel blocco e dichiarata al di fuori del blocco.

| Scope modifying clauses

- **shared**: il comportamento predefinito per le variabili dichiarate al di fuori di un blocco parallelo. Deve essere usato solo se viene specificato anche `default(None)`.
- **reduction**: viene eseguita un'operazione di riduzione tra le copie private e l'oggetto "esterno". Il valore finale viene memorizzato nell'oggetto "esterno".
- **private** : crea una copia separata di una variabile per ogni thread del team.
Le variabili private non sono inizializzate, quindi non ci si deve aspettare di ottenere il valore della variabile dichiarata al di fuori del costrutto parallelo.

ESEMPIO:

```

int x = 5;
#pragma omp parallel private(x)
{
    x = x+1; // dangerous (x not initialized)
    printf("thread %d: private x is
          %d\n",omp_get_thread_num(),x);
}
printf("after: x is %d\n",x); // also dangerous (prints the x declared in the first line)

```

```

int x = 5;
#pragma omp parallel private(x)
{
    x = x+1; // dangerous (x not initialized)
    printf("thread %d: private x is
           %d\n",omp_get_thread_num(),x);
}
printf("after: x is %d\n",x); // also dangerous (prints the x declared in the first line)

```

output:

```

thread 0: private x is 1
thread 1: private x is 1
thread 3: private x is 1
thread 2: private x is 1
after: x is 5

```



| Scope modifying: clausole

- **firstprivate**: si comporta come la clausola private, ma le copie delle variabili private sono inizializzate al valore dell'oggetto “esterno”.
- **lastprivate**: si comporta come la clausola private, ma il thread che termina *l'ultima iterazione* del blocco sequenziale copia il valore del suo oggetto nell'oggetto “esterno”. Qual'è l'ultima iterazione?

Quando dichiariamo una variabile come “privata”, essa non esisterà più al di fuori dell'ambito.

Come dichiarare una variabile “privata” che persista tra diverse sezioni parallele?

- **threadprivate** : crea una memoria persistente e specifica del thread (cioè per la durata del programma) per i dati globali. La variabile deve essere una variabile globale o statica in C, o un membro statico della classe in C++.
- **copyin**: usato insieme alla clausola threadprivate per inizializzare le copie threadprivate di un gruppo di thread dalle variabili del thread master.

ESEMPIO THREADPRIVATE:

```

int tp;

int main(int argc,char **argv) {
#pragma omp threadprivate(tp)

#pragma omp parallel num_threads(7)
    tp = omp_get_thread_num();

#pragma omp parallel num_threads(9)
    printf("Thread %d has %d\n", omp_get_thread_num(), tp);
    return 0;
}

```

Thread 3 has 3
 Thread 2 has 2
 Thread 8 has 0
 Thread 7 has 0
 Thread 5 has 5
 Thread 4 has 4
 Thread 6 has 6
 Thread 1 has 1
 Thread 0 has 0

Se i dati privati del thread iniziano in modo identico in tutti i thread

```

#pragma omp threadprivate(private_var)
...
private_var = 1;

```

```

#pragma omp parallel copyin(private_var)
    private_var += omp_get_thread_num()

```

Se un thread deve impostare tutti i dati privati del thread sul suo valore:

```

#pragma omp parallel
{
...
#pragma omp single copyprivate(private_var)
    private_var = 4;
...
}

```

```

#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)

void default_none(int a) {
    const int c = 1;
    int i = 0;

#pragma omp parallel default(none) private(a) shared(z)
{
    int j = omp_get_num_threads();
        /* O.K. - j is declared within parallel region */
    a = z[j];    /* O.K. - a is listed in private clause */
        /* - z is listed in shared clause */
    x = c;        /* O.K. - x is threadprivate */
        /* - c has const-qualified type */
    z[i] = y;    /* Error - cannot reference i or y here */

#pragma omp for firstprivate(y)
        /* Error - Cannot reference y in the firstprivate clause */
    for (i=0; i<10 ; i++) {
        z[i] = i; /* O.K. - i is the loop iteration variable */
    }

    z[i] = y;    /* Error - cannot reference i or y here */
}
}

```

| Reduction Clause

Nella versione che abbiamo mostrato, usiamo global_result_p come parametro di uscita, dove ogni thread accumula il risultato.

```
void Trap(double a, double b, int n, double* global_result_p);
```

Si può fare in modo che la funzione restituisca l'area calcolata? Ad esempio:

```
double Local_trap(double a, double b, int n);
```

Se lo sistemiamo in questo modo...

```

global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
#pragma omp critical
    global_result += Local_trap(double a, double b, int n);
}
...
```

forziamo l'esecuzione sequenziale dei thread.

Si può evitare questo problema dichiarando una variabile privata all'interno del blocco parallelo e spostando la sezione critica dopo la chiamata di funzione.

```

global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
    double my_result = 0.0; /* private */
    my_result += Local_trap(double a, double b, int n);
# pragma omp critical
    global_result += my_result;
}

```

Indipendentemente dal modo in cui lo facciamo, si tratta di una Reduce (simile a quella che abbiamo visto in MPI) OpenMP fornisce un modo nativo per farlo?

| Operazioni di Riduzione

Un operatore di riduzione è un'operazione binaria (come l'addizione o la moltiplicazione). Una **riduzione** è un calcolo che applica ripetutamente lo stesso operatore di riduzione a una sequenza di operandi per ottenere un unico risultato. Tutti i risultati intermedi dell'operazione devono essere memorizzati nella stessa variabile: la variabile di riduzione

Una clausola di riduzione può essere aggiunta a una direttiva parallela

reduction(<operator>: <variable list>)

+, *, -, &, |, ^, &&, ||

 global_result = 0.0;
pragma omp parallel num_threads(thread_count) \
reduction(+: global_result)
global_result += Local_trap(double a, double b, int n);

ATTENTION: If I do not specify the reduction clause I would have a race condition

la \ serve per andare a campo nell'istruzione pragma

| Riduzione

Le variabili private create per una clausola di riduzione sono inizializzate al valore di identità dell'operatore. Ad esempio, se l'operatore è la moltiplicazione, le variabili private vengono inizializzate a 1.

La riduzione alla fine della sezione parallela accumula il valore esterno e i valori privati calcolati all'interno della regione parallela.

| CICLI

| Parallel for

Forks un gruppo di thread per eseguire il seguente blocco strutturato. Tuttavia, il blocco strutturato che segue la direttiva parallel for deve essere un **ciclo for**.

Inoltre, con la direttiva parallel for il sistema parallelizza il ciclo for dividendo le iterazioni del ciclo tra i thread.

RIPRENDENDO [12.1 - Regola del Trapezio con OpenMP](#)

| Caveats

- L'indice della variabile deve essere di tipo intero o puntatore (ad esempio, non può essere un float).
- Le espressioni start, end e incr devono avere un tipo compatibile. Ad esempio, se index è un puntatore, incr deve essere di tipo intero.
- Le espressioni start, end e incr non devono cambiare durante l'esecuzione del ciclo.
- Durante l'esecuzione del ciclo, la variabile index può essere modificata solo dalla "espressione di incremento" nell'istruzione for.

Esempi:

```
for (i=0; i<n; i++) {  
    if (...) break; //cannot be parallelized  
}
```

```
for (i=0; i<n; i++) {  
    if (...) return 1; //cannot be parallelized  
}
```

```
for (i=0; i<n; i++) {  
    if (...) exit(); //can be parallelized  
}
```

```
for (i=0; i<n; i++) {  
    if (...) i++; //CANNOT be parallelized  
}
```

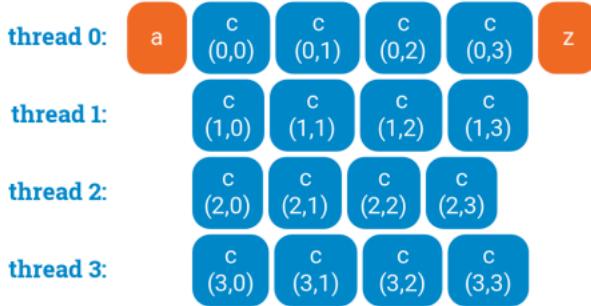
ESEMPIO [12.2 - Odd- Even Sort](#)

| CICLI ANNIDATI

| Cicli for annidati

Se abbiamo cicli for annidati, spesso è sufficiente parallelizzare semplicemente il ciclo più esterno

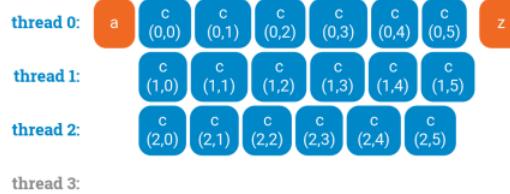
```
a();
#pragma omp parallel for
for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
        c(i, j);
    }
}
z();
```



A volte il ciclo più esterno è così corto che non vengono utilizzati tutti i fili:

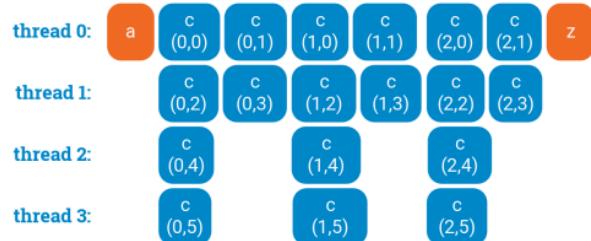
3 iterations, so it won't have sense to start more than 3 threads

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



Si potrebbe provare a parallelizzare il ciclo interno, ma non è garantito che l'utilizzo dei thread sia migliore.

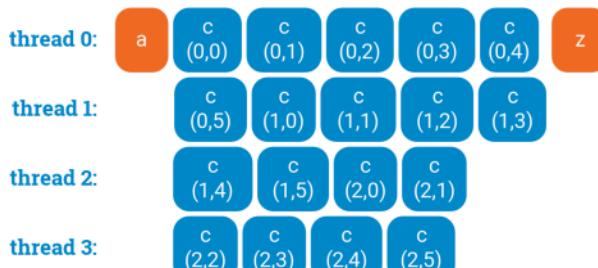
```
a();
for (int i = 0; i < 3; ++i) {
    #pragma omp parallel for
        for (int j = 0; j < 6; ++j) {
            c(i, j);
        }
    }
z();
```



La soluzione corretta è quella di **comprimere il ciclo in un unico ciclo** che esegue 18 iterazioni.

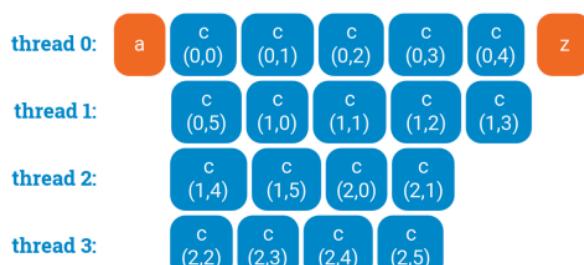
Possiamo farlo manualmente:

```
a();
#pragma omp parallel for
for (int ij = 0; ij < 3*6; ++ij) {
    c(ij / 6, ij % 6);
}
z();
```



Oppure possiamo chiedere a OpenMP di farlo per noi:

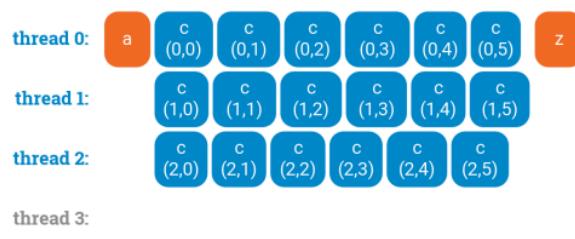
```
a();
#pragma omp parallel for
collapse(2)
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j)
        c(i, j);
}
z();
```



Modo sbagliato:

Il “parallelismo annidato” è disabilitato in OpenMP per impostazione predefinita (ovvero, il parallelismo interno per i pragma viene ignorato).

```
a();  
#pragma omp parallel for  
for (int i = 0; i < 3; ++i) {  
#pragma omp parallel for  
    for (int j = 0; j < 6; ++j)  
        c(i, j);  
}  
z();
```



Modo sbagliato:

Se il “parallelismo annidato” è abilitato, creerà 12 thread su un server con 4 core ($3 * 4$)!

```
a();  
#pragma omp parallel for  
for (int i = 0; i < 3; ++i) {  
#pragma omp parallel for  
    for (int j = 0; j < 6; ++j)  
        c(i, j);  
}  
z();
```

| 12.1 - Regola del Trapezio con OpenMP

| PARALLEL FOR

```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```



```
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
# pragma omp parallel for num_threads(thread_count) \  
    reduction(+: approx)  
for (i = 1; i <= n-1; i++)  
    approx += f(a + i*h);  
approx = h*approx;
```

Legal forms for parallelizable for statements

for	index++	
	++index	
	index < end	index--
	index <= end	--index
	index = start ; index >= end ; index += incr	
	index > end	index -= incr
		index = index + incr
		index = incr + index
		index = index - incr

Why? It allows the runtime system to determine the number of iterations prior to the execution of the loop

| 13 - Data Dependencies

Data dependencies

```
fibo[ 0 ] = fibo[ 1 ] = 1;  
for (i = 2; i < n; i++)  
    fibo[ i ] = fibo[ i - 1 ] + fibo[ i - 2 ];  
  
fibo[ 0 ] = fibo[ 1 ] = 1;  
# pragma omp parallel for num_threads(2)  
for (i = 2; i < n; i++)  
    fibo[ i ] = fibo[ i - 1 ] + fibo[ i - 2 ];  
  
1 1 2 3 5 8 13 21 34 55  
this is correct  
  
1 1 2 3 5 8 0 0 0 0  
but sometimes  
we get this
```

| Cos'è successo?

1. I compilatori OpenMP non controllano le dipendenze tra le iterazioni in un ciclo che viene parallelizzato con una direttiva direttiva parallel for.
2. Un ciclo in cui i risultati di una o più iterazioni dipendono da altre iterazioni non può, in generale, essere dipendono da altre iterazioni non possono, in generale, essere correttamente parallelizzato da OpenMP. Diciamo che abbiamo una dipendenza trasportata dal ciclo

| Data dependencies

Assumiamo di aver eun loop nella forma:

```
for (i = ...  
{  
    S1 : operate on a memory location x  
    ...  
    S2 : operate on a memory location x  
}
```

- Esistono quattro modi diversi in cui S1 e S2 sono collegati, a seconda che stiano leggendo o scrittura su x.
- Esiste un problema se la dipendenza attraversa il ciclo iterazioni: dipendenza trasportata dal ciclo

| Tipi di Dipendenze

- Flow dependence : RAW

```
x = 10;           // S1
y = 2 * x + 5;  // S2
```

- Anti-flow dependence : WAR

```
y = x + 3;      // S1
x ++ ;          // S2
```

- Output dependence : WAW

```
x = 10;           // S1
x = x + c ;      // S2
```

- Input dependence : RAR (it's not an actual dependence)

```
y = x + c ;    // S1
z = 2 * x + 1; // S2
```

| Flow Dependence Removal (RAW)

| Data Dependency Resolution

6 tecniche:

1. reduction/induction variable fix
2. Loop skewing
3. Partial parallelization
4. Refactoring
5. Fissioning
6. Algorithm change

| Flow Dependence : Reduction, Induction Variables

Esempio:

```
double v = start;
double sum=0;
for( int i = 0; i < N; i++)
{
    sum = sum + f(v);    // S1
    v = v + step;        // S2
}
```

- RAW (S1) causato dalla variabile di riduzione sum.
- RAW (S2) causato dalla variabile di induzione v (la variabile di induzione è una variabile che viene aumentata/diminuita in misura costante a ogni iterazione).
- RAW (S2->S1) causato dalla variabile di induzione v.
- Variabile di induzione: funzione affine della variabile del ciclo.

N.B.: I RAW sono compresi tra la variabile i e le i+1 iterazioni. Ad esempio, la somma viene letta in l'i (i+1)-esima iterazione dopo essere stata scritta nella i-esima iterazione

Remove RAW (S2) and RAW (S2->S1)

```

double v = start;
double sum = 0;
for(int i = 0; i < N ; i++)
{
    sum = sum + f(v); // S1
    v = v + step;     // S2
}

```



```

double v;
double sum = 0;
for(int i = 0; i < N ; i++)
{
    v = start + i*step;
    sum = sum + f(v);
}

```

```
i = 0 -> v = start
i = 1 -> v = start + step
i = 2 -> v = (start + step) + step
...

```

Remove RAW (S1)

```

double v;
double sum = 0;
#pragma omp parallel for reduction(+ : sum) private(v)
for(int i = 0; i < N ; i++)
{
    v = start + i*step;
    sum = sum + f(v);
}

```

| Flow Dependence: Loop Skewing

Un'altra tecnica prevede la riorganizzazione del corpo del ciclo del corpo del ciclo.

Esempio:

```

for (int i = 1; i < N; i++)
{
    y[i] = f( x[i-1] ); // S1
    x[i] = x[i] + c[i]; // S2
}

```

- RAW (S2->S1) su x

- Soluzione: assicurarsi che le affermazioni che consumano i valori calcolati che causano la dipendenza, utilizzino valori generati durante la stessa iterazione.

Flow Dependence : Loop Skewing (2)

```
for (int i = 1; i < N; i++)
{
    y[ i ] = f( x[ i-1 ] );
    // S1
    x[ i ] = x[ i ] + c[ i ];
    // S2
}
```



```
y[ 1 ] = f( x[ 0 ] );
for (int i = 1; i < N - 1; i++)
{
    x[ i ] = x[ i ] + c[ i ];
    y[ i + 1 ] = f( x[ i ] );
}
x[ N - 1 ] = x[ N - 1 ] + c[ N - 1 ];
```

Come fare lo skewing del loop?

- Suggerimento: srotolate il loop e osservate il modello di ripetizione.

```
y[1]= f(x[0]);
for (int i = 1; i< N-1; i++)
{
    x[i]= x[i]+c[i];
    y[i+1]= f(x[i]);
}
x[N-1]= x[N-1]+ c[N-1];
```

```
y[1]= f(x[0]);
x[1]= x[1]+c[1];
y[2]= f(x[1]);
x[2]= x[2]+c[2];
...
y[N-2]= f(x[N-3]);
x[N-2]= x[N-2]+ c[N-2];
y[N-1]= f(x[N-2]);
x[N-1]= x[N-1]+ c[N-1];
```

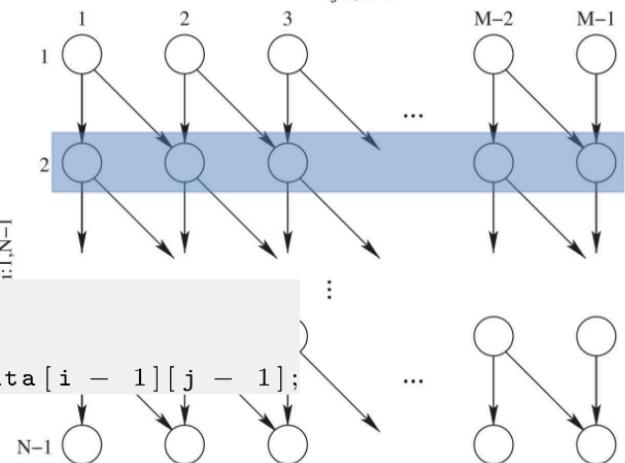
Iteration Space Dependency Graph

ISDG è costituito da nodi che rappresentano una singola esecuzione del corpo del ciclo e da spigoli che rappresentano le dipendenze.

Esempio:

- Example:

```
for (int i = 1; i < N; i++)
    for (int j = 1; j < M; j++)
        data[i][j] = data[i - 1][j] + data[i - 1][j - 1];
```



No edges/dependencies between nodes on the same row.

I.e., we can parallelize the j-loop

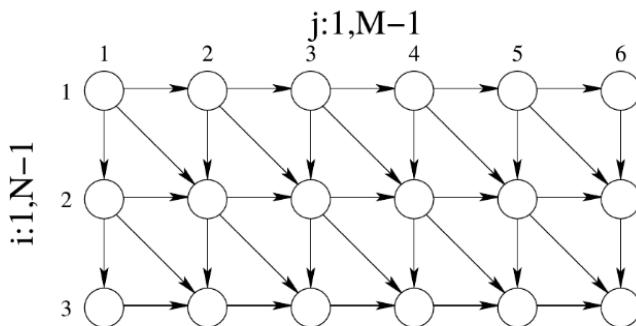
```
for (int i = 1; i < N; i++)
#pragma omp parallel for
    for (int j = 1; j < M; j++)
        data[i][j] = data[i - 1][j] + data[i - 1][j - 1];
```

Flow Dependencies: Refactoring

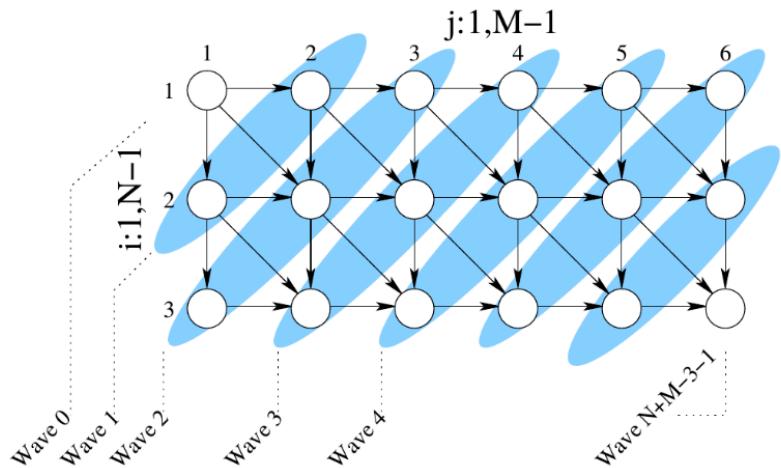
Il refactoring si riferisce alla riscrittura dei cicli loop in modo da esporre il parallelismo.

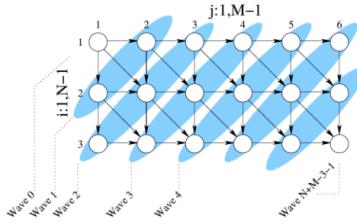
L'ISDG per il seguente esempio:

```
for (int i = 1; i < N; i++)
    for (int j = 1; j < M; j++)
        data[i][j] = data[i - 1][j] + data[i][j - 1] + data[i - 1][←
            j - 1]; // S1
```



Gli insiemi di diagonali possono essere eseguiti in parallelo (non ci sono bordi/dipendenze tra i nodi dello stesso stesso insieme diagonale):





```

for(wave=0 wave<NumWaves; wave++) {
    diag=F(wave);
    #pragma omp parallel for
    for(k=0; k<diag; k++) {
        int i = get_i(diag, k);
        int j = get_j(diag, k);
        data[i][j] = data[i-1][j] + data[i][j-1] + data[i-1][j-1];
    }
}

```

(Intuition, full code on the book)

L'esecuzione in onde richiede una modifica delle variabili del ciclo dall'originale i e j

| Flow Dependencies : Fissioning

Fissionare significa suddividere il ciclo in una parte sequenziale e una parallelizzabile
Esempio:

```

s = b[ 0 ];
for (int i = 1; i < N; i++)
{
    a[ i ] = a[ i ] + a[ i - 1 ]; // S1
    s = s + b[ i ];
}

```

// sequential part
`for (int i = 1; i < N; i++)
 a[i] = a[i] + a[i - 1];`

// parallel part
`s = b[0];
#pragma omp parallel for reduction(+: s)
for (int i = 1; i < N; i++)
 s = s + b[i];`

| Flow Dependencies: Algorithm Change

Se tutto il resto fallisce, cambiare l'algoritmo forse la risposta.

Ad esempio, la sequenza di Fibonacci:

```

for (int i = 2 ; i < N; i++)
{
    int x = F[ i - 2 ]; // S1
    int y = F[ i - 1 ]; // S2
    F[ i ] = x + y; // S3
}

```

can be parallelized via Binet's formula:

$$F_n = \frac{\varphi^n - (1 - \varphi)^n}{\sqrt{5}}$$

| Antidependence Removal (WAR)

- E.g.:

```
for(int i = 0; i < N-1; i++)
{
    a[ i ] = a[ i + 1 ] + c;
}
```
- Simple solution, make a copy of a before starting to modify it:

```
for(int i = 0; i < N-1; i++)
{
    a2[ i ] = a[ i + 1 ];
}

#pragma omp parallel for
for(int i = 0; i < N-1; i++)
{
    a[ i ] = a2[ i ] + c;
}
```
- ATTENTION: Space and time tradeoffs must be carefully evaluated!

| Output Dependence Removal (WAW)

- E.g.:

```
for(int i = 0; i < N; i++)
{
    y[i] = a * x[i] + c; // S1
    d = fabs( y[i] );      // S2
}
```
- Where is the WAW dependency?
 - On variable d.
- How to guarantee that at the end of the execution the computed d is the one computed in the last iteration?

```
#pragma omp parallel for shared( a, c ) lastprivate( d )
for(int i = 0; i < N; i++)
{
    y[i] = a * x[i] + c;
    d = fabs( y[i] );
}
```

14- Scheduling Loops

We want to parallelize
this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

In practice, how are iterations assigned to threads?

Thread	Iterations
0	0, 1, 2, ..., $n/t - 1$
1	$n/t, n/t + 1, \dots, 2n/t$
...	...
$t-1$	$n(t-1)/t, \dots, n-1$

Thread	Iterations
0	$0, n/t, 2n/t, \dots$
1	$1, n/t + 1, 2n/t + 1, \dots$
:	:
$t-1$	$t-1, n/t+t-1, 2n/t+t-1, \dots$

Default partitioning.

Cyclic partitioning.

```
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
} /* f */
```

- i.e., $f(i)$ calls the \sin function i times.

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

- assume the time to execute $f(2i)$ requires approximately twice as much time as the time to execute $f(i)$.
- How are iterations assigned to threads?
What's the best way of doing it?

Risultati

#Threads	1	2 (default scheduling)	2 (cyclic scheduling)
Runtime	3.67	2.76	1.84
Speedup	1	1.33	1.99

La clausola di programmazione

- Default schedule:

```
    sum = 0.0;  
#   pragma omp parallel for num_threads(thread_count) \  
    reduction(+:sum)  
  for (i = 0; i <= n; i++)  
    sum += f(i);
```

- Cyclic schedule:

```
    sum = 0.0;  
#   pragma omp parallel for num_threads(thread_count) \  
    reduction(+:sum) schedule(static,1)  
  for (i = 0; i <= n; i++)  
    sum += f(i);
```

| schedule (type , chunkszie)

- Il tipo può essere:

- **statico**: le iterazioni possono essere assegnate ai thread prima dell'esecuzione del ciclo. prima dell'esecuzione del ciclo.
- **dinamico** o **guidato**: le iterazioni sono assegnate ai thread durante l'esecuzione del ciclo. thread durante l'esecuzione del ciclo.
- **auto**: il compilatore e/o il sistema di run-time determinano il programma.
- **runtime**: la programmazione viene determinata in fase di esecuzione.

Il chunkszie è un numero intero positivo.

| The Static Schedule Type

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,1)
```

Thread 0: 0,3,6,9

Thread 1: 1,4,7,10

Thread 2: 2,5,8,11

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,2)
```

Thread 0 : 0,1,6,7

Thread 1 : 2,3,8,9

Thread 2 : 4,5,10,11

e.g., twelve iterations, 0, 1, . . . , 11, and three threads

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

Thread 0 :	0, 1, 2, 3
Thread 1 :	4, 5, 6, 7
Thread 2 :	8, 9, 10, 11

| The Dynamic Schedule Type

- Anche le iterazioni sono suddivise in parti iterazioni consecutive di dimensioni di blocchi (**chunksize**).
- Ogni thread esegue un pezzo, e quando un thread termina un pezzo, ne richiede un altro al sistema di run-time.
- Questo continua finché non vengono completate tutte le iterazioni completato.
- La dimensione del blocco può essere omessa. Quando viene omesso, a viene utilizzata la dimensione del blocco pari a 1.
- Migliore bilanciamento del carico, ma costi generali più elevati pianificare i blocchi (può essere ottimizzato tramite il file dimensione del pezzo)

| The Guided Schedule Type

- Ogni thread esegue anche un pezzo e quando un thread finisce un pezzo, ne richiede un altro.
- Tuttavia, in una pianificazione guidata, man mano che i blocchi vengono completati la dimensione dei nuovi blocchi diminuisce.
- I pezzi hanno una dimensione num_iterazioni/num_thread, dove num_iterazioni è il numero di iterazioni non assegnate
- Se non viene specificata la dimensione dei blocchi, la dimensione dei blocchi diminuisce fino a 1.
- Se viene specificata la dimensione del blocco, diminuisce fino alla dimensione del blocco, con l'eccezione che l'ultimo pezzo può essere più piccolo rispetto alla dimensione del pezzo.
- Pezzi più piccoli verso la fine per evitare sbandati

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

| The Runtime Schedule Type

Il sistema utilizza la variabile d'ambiente `OMP_SCHEDULE` per determinare in fase di esecuzione come pianificare il ciclo.

La variabile d'ambiente `OMP_SCHEDULE` può assumerne qualsiasi dei valori che possono essere utilizzati per un'operazione statica, dinamica o programma guidato.

Esempio: `export OMP_SCHEDULE=" static,1 "`

Un altro modo per specificare il tipo di pianificazione è impostarlo con la funzione `omp_set_schedule(omp_sched_t kind, int Chunk_size);`

| Come selezionare una opzione di schedule?

- **Static:** se le iterazioni sono *omogenee*
- **Dynamic/Guided:** Se il costo di esecuzione *varia*
 - Se in dubbio settare:
 - `#pragma omp parallel for schedule (runtime)`
 - **Non vi è alcuna garanzia** che selezionerà di più esecuzione dell'opzione di pianificazione.
 - Misurare/Provare diverse opzioni (la scelta migliore potrebbe essere diverso a seconda dell'input nel file programma)

| Mutual Exclusion

| Sezioni critiche nominate

OpenMP offre la possibilità di aggiungere un nome a una direttiva critica:

```
#pragma omp critical (name)
```

- In questo modo, due blocchi protetti con direttive critiche con direttive critiche con nomi diversi possono essere eseguiti simultaneamente (cioè, è come agire su due diversi

blocchi)

- Tuttavia, queste direttive devono essere impostate in fase di compilazione.
- Cosa succede se vogliamo avere più blocchi/sezioni critiche, ma non sappiamo quanti ma non sappiamo quanti sono in fase di compilazione? tempo di compilazione? (ad esempio, un elenco collegato con un blocco per ogni nodo)

| Locks in OpenMP

```
omp_lock_t writelock;
omp_init_lock(&writelock);

#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
    // some stuff
    omp_set_lock(&writelock);
    // one thread at a time stuff
    omp_unset_lock(&writelock);
    // some stuff
}

omp_destroy_lock(&writelock);
```

| critical, atomic, o locks?

1. In generale, la direttiva atomica è potenzialmente il metodo più veloce per ottenere la mutua esclusione.
2. Tuttavia, le specifiche di OpenMP consentono alla direttiva atomica di applicare la mutua esclusione a tutte le direttive atomiche del programma. cioè, le seguenti potrebbero essere eseguite in modo mutuamente esclusivo (dipende dall'implementazione).

```
#pragma omp atomic x++; #pragma omp atomic y++;`
```

3. L'uso dei lock dovrebbe probabilmente essere riservato a situazioni in cui la mutua esclusione è necessaria per una struttura di dati piuttosto che per un blocco di codice.

| Some Caveats

1. Non si devono mischiare i diversi tipi di mutua esclusione per una singola sezione critica.
2. Non c'è garanzia di equità nei costrutti di mutua esclusione.

```
#pragma omp atomic x+=f(y); #pragma omp atomic x=g(x);`
```
3. Può essere pericoloso "annidare" i costrutti di mutua esclusione.

| Sezioni critiche annidate

```

int main() {

    # pragma omp critical
    y = f(x);
    ...
}

double f(double x) {

    # pragma omp critical
    z = g(x); /* z is shared */
    ...
    return z;
}

```

Soluzione

Le sezioni critiche annidate andranno in stallo.

In questo esempio, è possibile risolvere il problema utilizzando sezioni critiche denominate.

```

int main() {

    # pragma omp critical(one)
    y = f(x);
    ...
}

double f(double x) {

    # pragma omp critical (two)
    z = g(x); /* z is shared */
    ...
    return z;
}

```

| Costrutti di sincronizzazione

| Direttive master/singole

master, single:

Entrambi forzano l'esecuzione del seguente blocco strutturato da parte di un singolo thread.

C'è una differenza significativa: single implica una barriera all'uscita dal blocco.

Ci sono altre differenze (ad esempio, con master, il blocco viene garantito che il blocco

venga eseguito dal thread master).

```
int examined = 0;
int prevReported = 0;
#pragma omp for shared( examined, prevReported )
for( int i = 0 ; i < N ; i++ )
{
    // some processing

    // update the counter
#pragma omp atomic
    examined++;

    // use the master to output an update every 1000 newly finished iterations
#pragma omp master
{
    int temp = examined;
    if( temp - prevReported >= 1000)
    {
        prevReported = temp;
        printf("Examined %.2lf%\n", temp * 1.0 / N );
    }
}
```

| Direttive barriera

barriera

blocca finché tutti i thread della squadra non raggiungono quel punto.

```
int main( )
{
    int a[5], i;
#pragma omp parallel
{
    // Perform some computation.
#pragma omp for
    for (i = 0; i < 5; i++)
        a[i] = i*i;

    // Print intermediate results.
#pragma omp master
    for (i = 0; i < 5; i++)
        printf("a[%d] = %d\n", i, a[i]);

    // Wait.
#pragma omp barrier

    // Continue with the computation.
#pragma omp for
    for (i = 0; i < 5; i++)
        a[i] += i;
}
}
```

| Le direttive di sezione/sezioni

Come possiamo inviare diversi task in parallelo?

```
#pragma omp parallel
switch (omp_get_thread_num())
{
    case 0: {
        //concurrent block 0
    }
    break;
    case 1: {
        //concurrent block 1
    }
    break;
}
```

Le singole voci di lavoro sono contenute in blocchi decorati da direttive di sezione

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // concurrent block 0
    }
    ...
    #pragma omp section
    {
        // concurrent block M-1
    }
}
```

La direttiva `omp parallel sections` combina le direttive `omp parallel` e `omp sections`.

Alla fine di un costrutto di sezione c'è una barriera implicita, a meno che non sia specificata una clausola `nowait`.

| Synchronization Constructs

ordinato

usato all'interno di un parallelo for, per garantire che un blocco sarà eseguito come se fosse in ordine sequenziale.

```
double data[ N ];
#pragma omp parallel shared( data, N )
{
    #pragma omp for ordered_schedule( static, 1 )
    for( int i = 0; i < N; i++ )
    {
        // process the data
        // print the results in order
    }
    #pragma omp ordered
    cout << data[i];
}
```

ordered clause
is required

| False sharing

Condivisione fittizia

- condivisione di linee di cache senza condividere effettivamente i dati.
Come risolvere il problema:
 - Imbottire i dati (Pad the data)
 - Modificare la mappatura dei dati ai thread/cores (Data mapping change)
 - Utilizzare variabili private/locali (Using private variables)

| Padding the data

- Original

```
double x[N];
#pragma omp parallel for schedule(static, 1)
for( int i = 0; i < N; i++ )
    x[ i ] = someFunc( x[ i ] );
```

- Padded:

```
double x[N][8];
#pragma omp parallel for schedule(static, 1)
for( int i = 0; i < N; i++ )
    x[ i ][ 0 ] = someFunc( x[ i ][ 0 ] );
```

- Can kill cache effectiveness.
- Wastes memory.

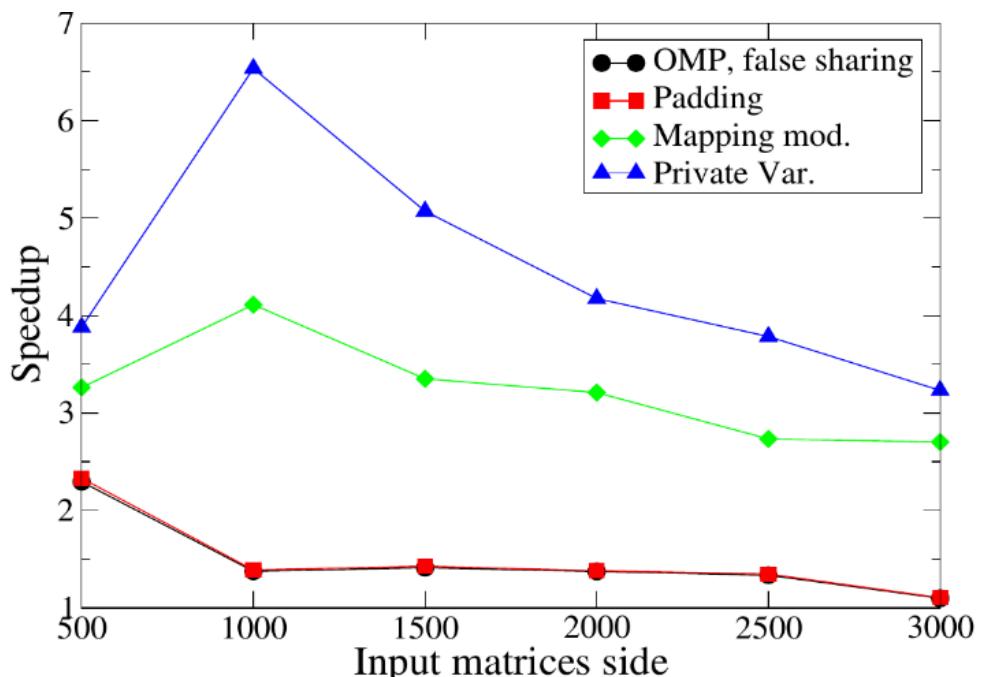
| Data mapping change

```
double x[N];
#pragma omp parallel for schedule(static, 8)
for( int i = 0; i < N; i++ )
    x[ i ] = someFunc( x[ i ] );
```

| Using private variables

```
// assuming that N is a multiple of 8
double x[N];
#pragma omp parallel for schedule(static, 1)
for( int i = 0; i < N; i += 8 )
{
    double temp[ 8 ];
    for( int j = 0; j < 8; j++ )
        temp[ j ] = someFunc( x[ i + j ] );
    memcpy( x + i, temp, 8 * sizeof( double ) );
}
```

| Impatto del false sharing nella moltiplicazione tra matrici



| OpenMP + MPI

MPI definisce **4 livelli di sicurezza dei thread**:

- **`MPI_THREAD_SINGLE`**: esiste un solo thread nel programma
- **`MPI_THREAD_FUNNELED`**: solo il thread master può effettuare chiamate MPI. chiamate MPI. Master è quello che chiama `MPI_Init_thread()`
- **`MPI_THREAD_SERIALIZED`**: Multithread, ma solo un thread può effettuare chiamate MPI alla volta
- **`MPI_THREAD_MULTIPLE`**: Multithread e ogni thread può effettuare chiamate MPI in qualsiasi momento.

Più **sicuro** (più semplice) utilizzare **`MPI_THREAD_FUNNELED`**

- Si adatta bene alla maggior parte dei modelli OpenMP
- Loop costosi parallelizzati con OpenMP

- Comunicazione e chiamate MPI tra i loop

OpenMP/Pthreads + MPI

```
$ ./a.out 4  
$ Time: 0.40 seconds
```

```
$ mpirun -n 1 ./a.out 4  
$ Time: 1.17 seconds
```

Why?

Open MPI maps each process on a core. Thus, all the threads created by the process will run on the same core (i.e., 4 threads will run on the same core).

How to fix it?

```
$ mpirun --bind-to-none -n 1 ./a.out 4  
$ Time: 0.40 seconds
```

How to check how Open MPI is binding processes?

```
$ mpirun --report-bindings -n 1 ./a.out 4 1024 1024 1024  
MCW rank 0 bound to socket 0[core 0[hwt 0-1]]:  
[BB/..../..../..../..../..][..../..../..../..../..][..../..../..../..../..][..../..../..../..../..]
```

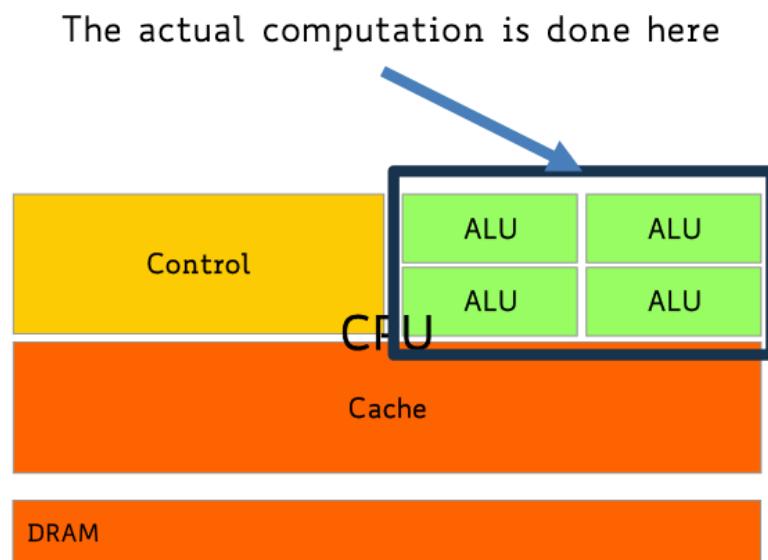
| 15 - GPU Programming

| CPUs: Latency Oriented Design

Alta frequenza di clock

Cache di grandi dimensioni

- Convertire gli accessi alla memoria a lunga latenza di memoria a lunga latenza in accessi alla cache a breve latenza
Controllo sofisticato
- Previsione dei rami per ridurre latenza dei rami
- Esecuzione fuori ordine
- ecc.
- Potente ALU
- Riduzione della latenza delle operazioni



| GPUs: Throughput Oriented Design

Frequenza di clock moderata

Cache di piccole dimensioni

- Per aumentare il throughput della memoria
Controllo semplice
- Nessuna predizione di ramo
- Esecuzione in-order
ALU ad alta efficienza energetica
- Molte, a lunga latenza, ma pesantemente pipelined per elevato throughput
Richiedono un numero massiccio di thread per tollerare le latenze

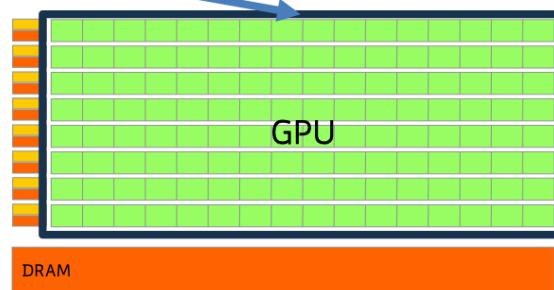
Utilizzare interfacce ad alta larghezza di banda per scambio di dati con la memoria e con l'host

The actual computation is done here

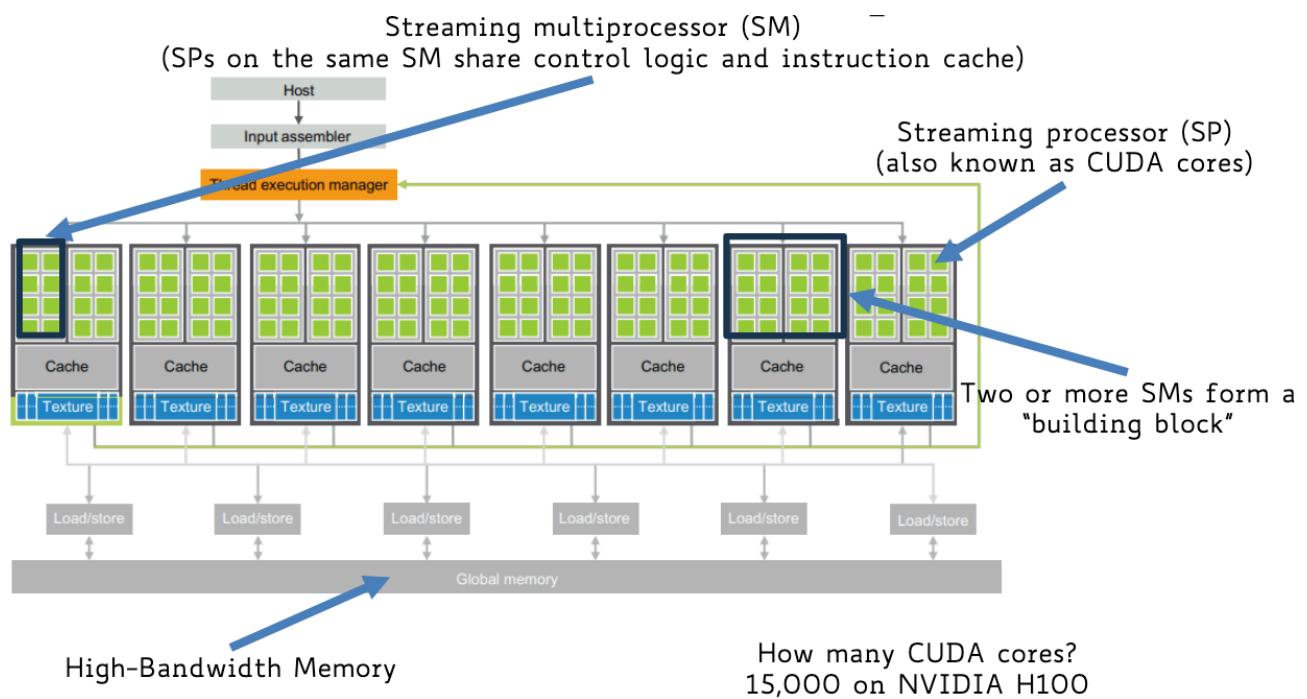
Memory

Throughput

Latency



| ARCHITETTURA di una CUDA-capable GPU



| Le applicazioni traggono vantaggio da entrambe le CPU e GPU

- CPU per parti sequenziali dove la latenza è importante
 - Le CPU possono essere 10+X più veloci delle GPU per il codice sequenziale
- GPU per parti parallele dove il throughput è importante
 - Le GPU possono essere 10+X più veloci delle CPU per il codice parallelo

| CPU-GPU Architecture

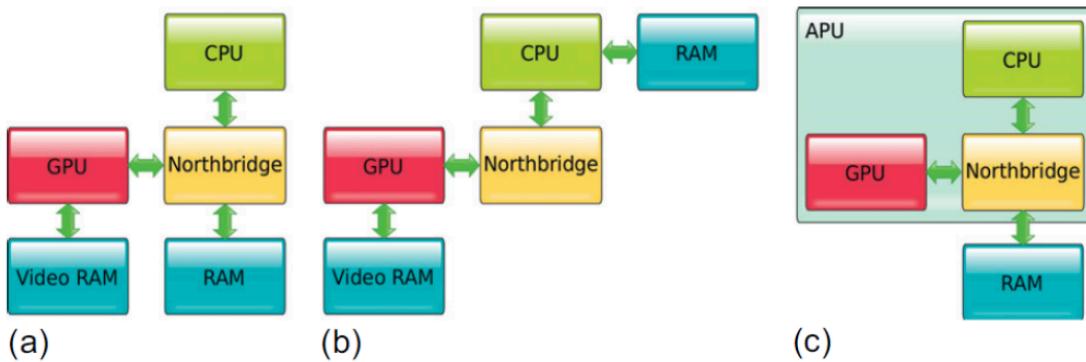


FIGURE 6.1

Existing architectures for CPU-GPU systems: (a) and (b) represent discrete GPU solutions, with a CPU-integrated memory controller in (b). Diagram (c) corresponds to integrated CPU-GPU solutions, such as the AMD's Accelerated Processing Unit (APU) chips.

| GPU programming caveat (avvertenze)

La distribuzione dei programmi su GPU ha una caratteristica che può un ostacolo importante: le memorie della GPU e dell'host sono in genere sono tipicamente disgiunte, richiedendo il trasferimento esplicito (o implicito, a seconda della piattaforma di sviluppo) dei dati, a seconda della piattaforma di sviluppo il trasferimento di dati tra i due.

Una seconda caratteristica della computazione su GPU è che i dispositivi GPU possono non aderire agli stessi agli stessi standard di rappresentazione e accuratezza in virgola mobile in virgola mobile e agli stessi standard di precisione delle CPU tipiche.

- Modi leggermente diversi di eseguire gli arrotondamenti e i casi angolari:
<https://docs.nvidia.com/cuda/floating-point/index.html>

| GPU Software Development Platforms

- **CUDA**: Compute Unified Device Architecture. CUDA fornisce due serie di API (una di basso e una di livello basso e uno più alto) ed è disponibile gratuitamente per i sistemi operativi Windows, MacOS X e Linux. Principale svantaggio: solo per l'hardware NVidia (anche se ora esistono strumenti per eseguire codice CUDA su GPU AMD).
- **HIP**: l'equivalente di CUDA per AMD. Vengono forniti strumenti per convertire il codice CUDA in HIP.
- **OpenCL** : Open Computing Language è uno standard aperto per la scrittura di programmi che possono essere eseguiti su una vasta gamma di dispositivi eterogenei. programmi che possono essere eseguiti su una varietà di piattaforme eterogenee che includono GPU, CPU, DSP o altri processori. OpenCL è supportato sia da NVidia che da AMD. È la piattaforma principale piattaforma di sviluppo per le GPU AMD. Il modello di programmazione di OpenCL corrisponde modello di programmazione di OpenCL corrisponde a quello offerto da CUDA.
- **OpenACC**: una specifica aperta per un'API che consente l'uso di direttive del compilatore (ad esempio `#pragma acc`, in modo simile a OpenMP) per mappare

automaticamente i calcoli sulle automaticamente i calcoli alle GPU o ai chip multicore, in base alle indicazioni del programmatore.

Molti altri (OpenMP, Thrust, Data Parallel C++, C++ AMP) ...

| 16 - CUDA

| CUDA: Compute Unified Device Architecture

Consente un modello di programmazione generale sulle GPU NVIDIA:

Prima di CUDA, le GPU venivano programmate trasformando un algoritmo in una sequenza di primitive di manipolazione delle immagini. sequenza di primitive di manipolazione delle immagini

Richiede hardware aggiuntivo per gestire/interagire con l'host per fornire calcolo generico

Consente la gestione esplicita della memoria della GPU

La GPU è vista come un dispositivo di calcolo che:

- è un co-processore della CPU (o dell'host)
- Dispone di una propria DRAM (memoria globale nel linguaggio di CUDA)
- Esegue molti thread in parallelo: il costo di creazione/commutazione dei thread è di pochi cicli di clock!

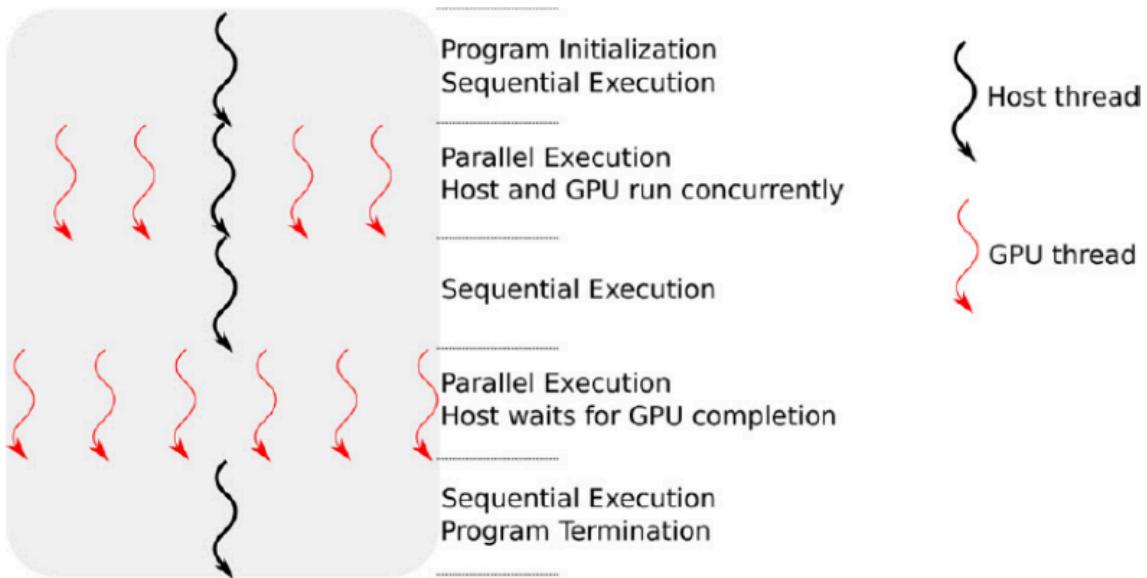
CUDA è un modello di piattaforma/programmazione, non un linguaggio di programmazione.

| CUDA Program Structure

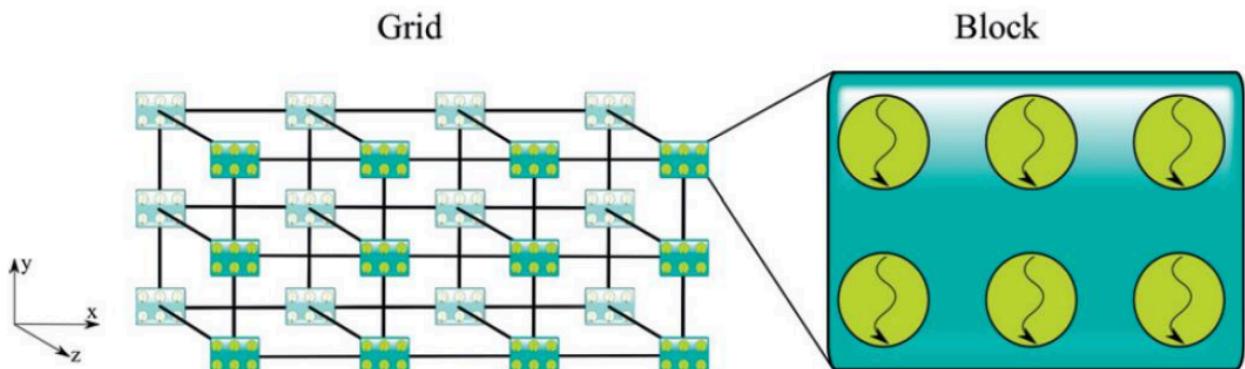
- Allocazione della memoria della GPU
- Trasferire i dati dall'host alla memoria della GPU
- Esecuzione del kernel CUDA
- Copia dei risultati dalla memoria della GPU alla memoria dell'host

| CUDA Execution Model

CUDA Model of Execution



Nella stragrande maggioranza degli scenari, l'host è responsabile delle operazioni di I/O, passare l'input e successivamente raccogliere i dati di output dallo spazio di memoria della GPU.



- CUDA organizza i thread in una struttura a 6 D (sono possibili anche dimensioni inferiori).
- Ogni thread ha una posizione in un blocco 1D, 2D, o 3D blocco
- Ogni blocco ha una posizione in una griglia 1D, 2D o 3D griglia
- Ogni thread è consapevole della sua posizione nella struttura complessiva, tramite un insieme di variabili/strutture intrinseche. Con queste informazioni, un thread può mappare la sua posizione al sottoinsieme di dati a cui è assegnato.

| CUDA compute capability

- Le dimensioni dei blocchi e delle griglie sono determinate dalla capacità, che determina ciò che ogni generazione di GPU è in grado di fare.
- La capacità di calcolo di un dispositivo è rappresentata da un numero di versione, talvolta chiamato anche “versione SM”.

- Questo numero di versione identifica le caratteristiche supportate dall'hardware della GPU e viene utilizzato dalle applicazioni in fase di esecuzione per determinare GPU e viene utilizzato dalle applicazioni in fase di esecuzione per determinare quali caratteristiche hardware e/o istruzioni sono disponibili sulla GPU in questione.

Table 6.1 Compute Capabilities and Associated Limits on Block and Grid sizes

Item	Compute Capability			
	1.x	2.x	3.x	5.x
Max. number of grid dimensions	2		3	
Grid maximum x-dimension	$2^{16} - 1$		$2^{31} - 1$	
Grid maximum y/z-dimension		$2^{16} - 1$		
Max. number of block dimensions			3	
Block max. x/y-dimension	512		1024	
Block max. z-dimension			64	
Max. threads per block	512		1024	
GPU example (GTX family chips)	8800	480	780	980

Feature Support	Compute Capability				
(Unlisted features are supported for all compute capabilities)	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x	8.x
Atomic functions operating on 32/64-bit integer values in global and shared memory	Yes				
Atomic addition operating on 32-bit floating point values in global and shared memory	Yes				
Atomic addition operating on 64-bit floating point values in global memory and shared memory	No				
Warp vote, Memory Fence, Synchronization functions, Unified Memory Programming , Dynamic Parallelism	Yes				
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No	Yes			
Bfloat16-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion		No			Yes
Tensor Cores		No		Yes	
Mixed Precision Warp-Matrix Functions		No		Yes	
Hardware-accelerated async-copy		No		Yes	
L2 Cache Residency Management		No		Yes	

CUDA: Nvidia architectures

CUDA-Enabled NVIDIA GPUs				
NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series

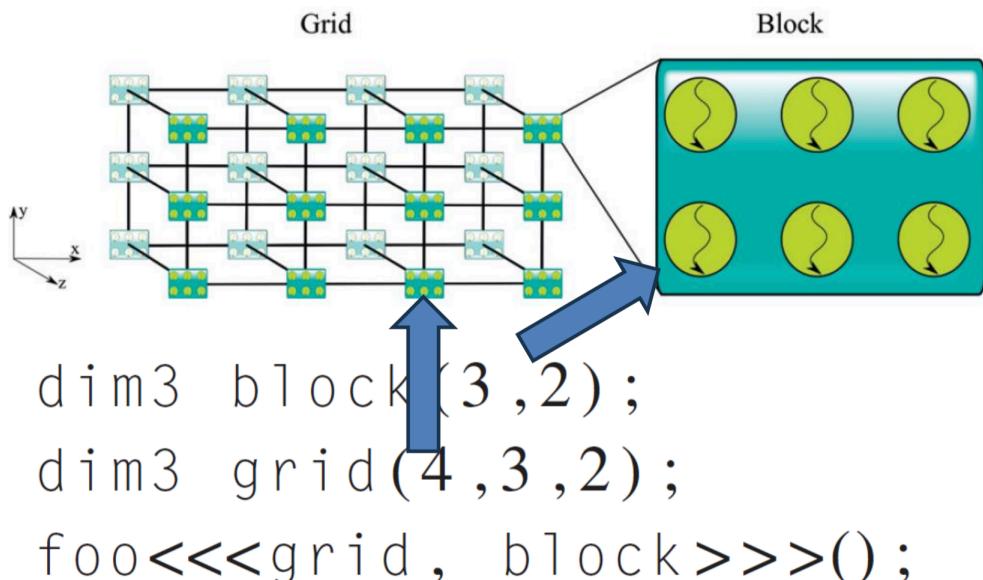
How to write a program?

È necessario specificare una funzione che verrà eseguita da tutti i thread (SIMD/SPMD/SIMT).

thread (SIMD/SPMD/SIMT)

Questa funzione è chiamata **kernel**

È necessario specificare come i thread sono disposti nella griglia/nei blocchi



dim3

```
dim3 block(3,2);  
dim3 grid(4,3,2);  
foo<<<grid, block>>>();
```

dim3 è un vettore di int

Ogni componente non specificata è impostata a 1

Ogni componente accessibile ai campi x, y, z (lo vedremo più avanti).

Devo specificare una griglia/un blocco 1D, 2D o 3D? Dipende dal problema: si lavora su un dominio 1D, 2D, ..., 6D?

| How to specify grid/block size

Hello World in CUDA

```
// File : hello.cu
#include <stdio.h>
#include <cuda.h>

__global__ void hello()
{
    printf("Hello world\n");
}

int main()
{
    hello<<<1,10>>>();
    cudaDeviceSynchronize();
    return 1;
}
```

Can be called from the host or the device
Must run on the device

A kernel is always declared as void
Computed result must be copied explicitly from GPU to host memory

printf in device code is supported by CC 2.0 and above

Blocks until the CUDA kernel terminates (like a barrier, kernel launch is asynchronous)

Compile for CC 2.0

- To compile and run:
\$ nvcc --arch=sm_20 hello.cu -o hello
\$./hello

Function decorations

- `__global__`: può essere richiamato dall'host o dalla GPU e eseguito sul dispositivo/GPU. In CC 3.5 e versioni successive, il dispositivo può anche chiamare le funzioni `__global__`.
- `__device__`: Una funzione che viene eseguita sulla GPU e che può essere chiamata solo essere chiamata solo dall'interno di un kernel (cioè dalla GPU).
- `__host__`: Una funzione che può essere eseguita solo sull'host. Il qualificatore `__host__` viene tipicamente omesso, a meno che non venga usato in combinazione con `__device__` per indicare che la funzione può essere eseguita sia sull'host che sul dispositivo. Questo scenario implica la generazione di due codici compilati per la funzione. Riuscite a indovinare il motivo?

How to get the thread position in the grid/block

Ricordate che le filettature sono disposte in uno spazio di 6D

- `blockDim`: Contains the size of each block, e.g., (B_x, B_y, B_z) .
- `gridDim`: Contains the size of the grid, in blocks, e.g., (G_x, G_y, G_z) .
- `threadIdx`: The (x, y, z) position of the thread within a block, with $x \in [0, B_x - 1]$, $y \in [0, B_y - 1]$, and $z \in [0, B_z - 1]$.
- `blockIdx`: The (b_x, b_y, b_z) position of a thread's block within the grid, with $b_x \in [0, G_x - 1]$, $b_y \in [0, G_y - 1]$, and $b_z \in [0, G_z - 1]$.

How to get a unique thread id?

Thread diversi possono avere lo stesso threadIdx ma essere su blocchi diversi
Devo combinare threadIdx e blockIdx per ottenere un identificatore univoco.

```
int myID = ( blockIdx.z * blockDim.x * blockDim.y +  
    blockIdx.y * blockDim.x +  
    blockIdx.x ) * blockDim.x * blockDim.y * blockDim.z +  
    threadIdx.z * blockDim.x * blockDim.y +  
    threadIdx.y * blockDim.x +  
    threadIdx.x;
```

IMPORTANTE

Spesso le filettature sono disposte in meno di 6 dimensioni (cioè, alcune di queste dimensioni saranno uguali a 1 e le coordinate corrispondenti a 0).

- Ad esempio, per ottenere l'ID per il caso hello world (i thread erano disposti in un blocco di 10 thread):

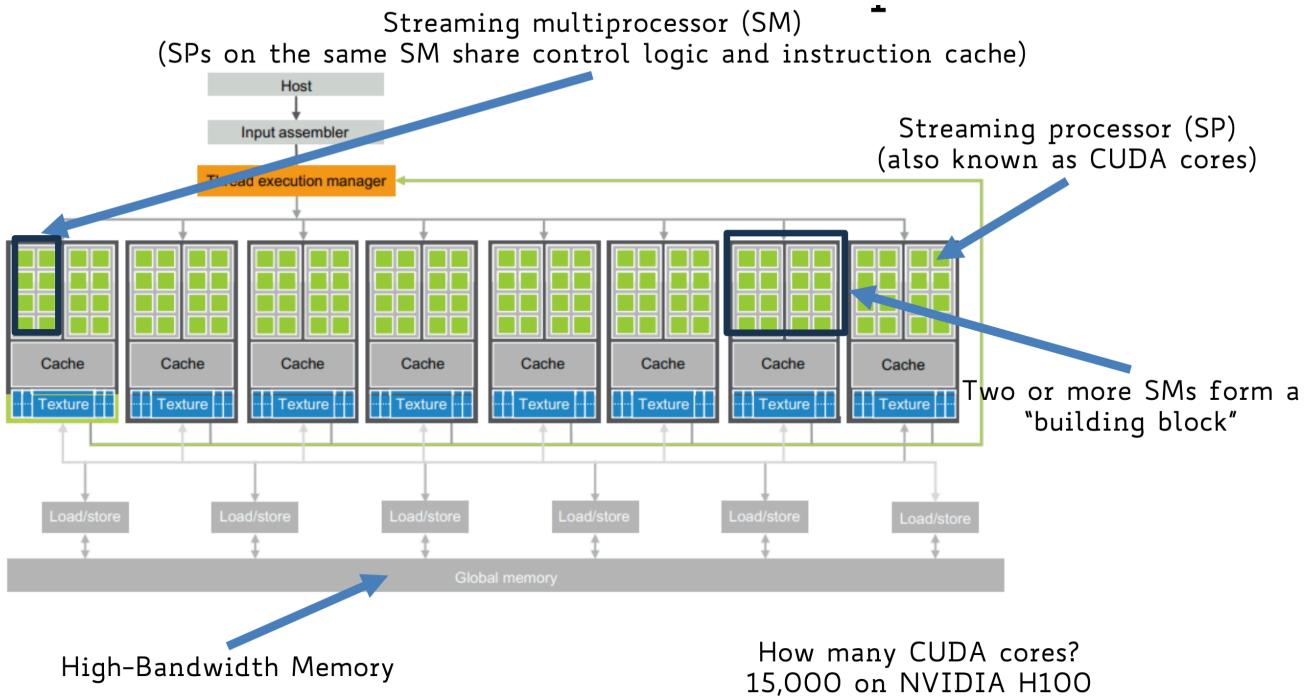
```
int myID = ( blockIdx.z * blockDim.x * blockDim.y +  
    blockIdx.y * blockDim.x +  
    blockIdx.x ) * blockDim.x +  
    threadIdx.x;
```

Hello World in CUDA

```
// File : hello.cu  
#include <stdio.h>  
#include <cuda.h>  
  
__global__ void hello()  
{  
    printf("Hello world\n");  
}  
  
int main()  
{  
    hello<<<1,10>>>();  
    cudaDeviceSynchronize();  
    return 1;  
}
```

Detrimental for performance
(GPU should not do I/O)
Only use it for debugging purposes

Architecture of a CUDA-capable GP



| Thread Scheduling

Ogni thread viene eseguito su un processore di streaming (core CUDA)

I core sullo stesso SM condividono l'unità di controllo (cioè, devono eseguire in modo sincrono le stesse istruzioni).

SM diversi possono eseguire kernel diversi

Ogni blocco viene eseguito su un SM (cioè, non posso avere un blocco che si estende su più SM, ma su più SM, ma è possibile avere più blocchi in esecuzione sullo stesso SM)

Una volta che un blocco è stato eseguito completamente, l'SM eseguirà il blocco successivo.

Non tutti i thread di un gruppo vengono eseguiti in contemporanea

| Warps

Vengono eseguiti in gruppi chiamati warp (nelle GPU attuali, la dimensione di un warp è di 32 - potrebbe cambiare in futuro, controllate in futuro, controllare la variabile warpSize)

I thread di un blocco sono suddivisi in warp in base al loro ID all'interno del blocco (cioè, i primi 32 thread di un blocco appartengono allo stesso warp, i successivi 32 appartengono allo stesso warp).

Tutti i thread in un warp sono eseguiti secondo il modello Single Instruction, Multiple Data (SIMD), ossia, in qualsiasi momento, un'istruzione viene recuperata ed eseguita per tutti i thread nel warp.

Di conseguenza, tutti i thread in un warp avranno sempre gli stessi tempi di esecuzione.

Su ogni SM possono essere presenti più warp scheduler (ad esempio, 4).

Cioè, più warp (ad esempio, 4) possono essere eseguiti contemporaneamente, ognuno seguendo contemporaneamente, ognuno dei quali può seguire un percorso di esecuzione

diverso.

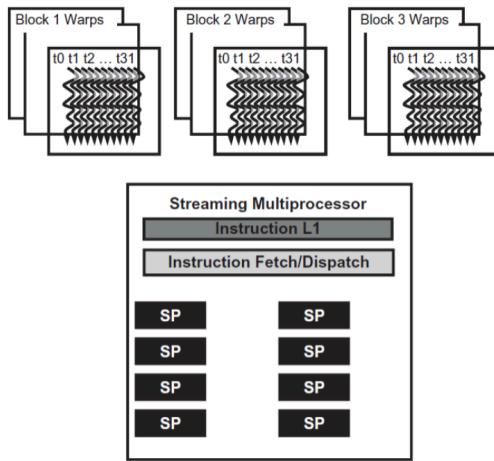


FIGURE 3.13

Blocks are partitioned into warps for thread scheduling.

| Warp Divergence

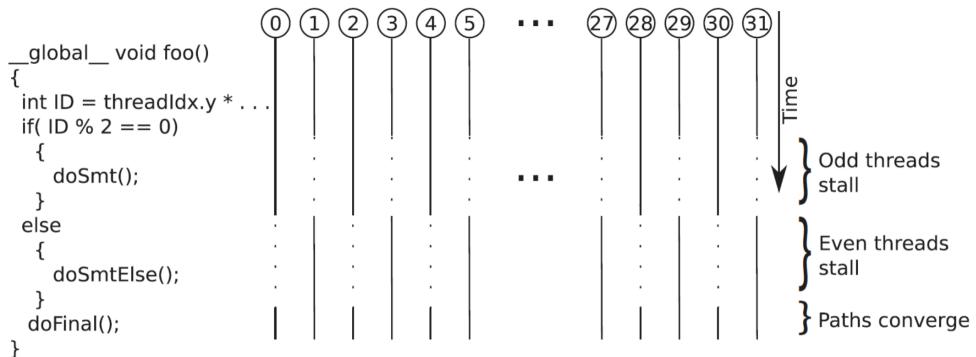
Tutti i thread in un warp vengono eseguiti secondo il modello Single Instruction, Multiple Data (SIMD), vale a dire, in ogni istante del tempo, un'istruzione viene recuperata ed eseguita per tutti i thread dell'ordito.

Di conseguenza, tutti i thread in un warp avranno sempre gli stessi tempi di esecuzione.

Cosa succede se il risultato di un'operazione condizionale li porta su percorsi diversi?

Tutti i percorsi divergenti vengono valutati (se i thread si diramano in essi) in sequenza finché i percorsi non si uniscono di nuovo.

I thread che non seguono il percorso in corso di esecuzione vengono bloccati.



| Context Switching

Di solito uno SM ha più blocchi/warps residenti rispetto a quelli che è in grado di eseguire simultaneamente

Ogni SM può passare da un warp all'altro senza soluzione di continuità.

Ogni thread ha il proprio contesto di esecuzione privato che viene mantenuto on-chip (ad es, il cambio di contesto è gratuito)

Quando un'istruzione da eseguire da un warp deve attendere il risultato di un'operazione a lunga latenza precedentemente avviata operazione a lunga latenza precedentemente avviata, il warp non viene selezionato per l'esecuzione (ad es, lettura della memoria, operazioni in virgola mobile a lunga latenza).

Al contrario, verrà selezionato per l'esecuzione un altro warp residente che non è più in attesa di risultati.

Questo meccanismo di riempimento del tempo di latenza delle operazioni con il lavoro di altri thread è spesso chiamato "tolleranza di latenza" o "toleranza di latenza". spesso chiamato "toleranza alla latenza" o "latency hiding".

Dato un numero sufficiente di warp, è probabile che l'hardware trovi un warp da eseguire in qualsiasi momento, rendendo così possibile l'esecuzione completa. in qualsiasi momento, sfruttando così appieno l'hardware di esecuzione nonostante le operazioni a lunga latenza. latenza.

Con la programmazione dei warp, il lungo tempo di attesa delle istruzioni dei warp viene "nascosto" dall'esecuzione di istruzioni di altri warp.

Questa capacità di tollerare le operazioni a lunga latenza è la ragione principale per cui le GPU

non dedicano alle memorie cache e ai meccanismi di predizione delle diramazioni la stessa superficie del chip delle CPU.

| Esempi

Example 1

- A CUDA device allows up to 8 blocks and 1024 threads per SM, and 512 threads per block
- Shall we use 8x8, 16x16, or 32x32 thread blocks?
- 8x8 blocks:
 - We would have 64 threads per block
 - To fill the 1024 threads we can have for each SM, we would need $1024/64 = 16$ blocks
 - However, we can have at most 8 blocks per SM, thus we would end with only $64 \times 8 = 512$ threads per SM
 - We are not fully utilizing the resources. Most likely, at some time, the scheduler might not find threads to schedule when some thread is waiting for long-latency operations
- 16x16
 - We would have 256 threads per block
 - To fill the 1024 threads we can have for each SM, we would need $1024/256 = 4$ blocks
 - This would allow to have 1024 threads on the SM, so there will be a lot of opportunities for latency hiding
- 32x32
 - We would have 1024 threads per block, which is higher than the 512 threads per block we can have
- In practice, it is more complicated than this, the usage of other resources as registers and shared memory must be taken into account (we will see how)

Example 2

- A CUDA device allows up to 8 blocks and 1536 threads per SM, and 1024 threads per block
- Shall we use 8x8, 16x16, or 32x32 thread blocks
- 8x8 blocks:
 - We would have 64 threads per block
 - To fill the 1536 threads we can have for each SM, we would need $1536/64 = 24$ blocks
 - Only 512 threads (8x8x8) would go into each SM
- 16x16
 - We would have 256 threads per block
 - To fill the 1536 threads we can have for each SM, we would need $1536/256 = 6$ blocks
 - We achieve full capacity (unless other resources constraints come into play)
- 32x32
 - We would have 1024 threads per block. Only one block can fit (two would bring the number of threads to 2048, which is higher than 1536)
 - Thus, we would use only 2/3 of the thread capacity of the SM (1024 out of 1536)

Example 3

- A grid of $4 \times 5 \times 3$ blocks, each made of 100 threads
- The GPU has 16 SMs
- Thus, we have $4 \times 5 \times 3 = 60$ blocks, that need to be distributed over 60 16SMs
 - Let's assume that they are distributed round-robin
 - 12 SMs will receive 4 blocks, and 6 SMs will receive 3 blocks
 - Inefficient! While the first 12 SMs process the last block, the other 6 SMs are idle
- A block contains 100 threads, which are divided into $100/32 = 4$ warps
 - The first three warps have 32 threads, and the last one have 4 threads ($32+32+32+4$)
 - Assume we can only schedule a warp at a time (e.g., because we have 32 CUDA cores per SM)
 - The last warp would only use 4 out of the 32 available cores (87.5% of the cores on each SM will be unused)
- Take home message: pay attention to how you define the grid and block sizes (we will come back to this again)

| Device properties

| Example : Listing all the GPUs in a system

```
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);
if(deviceCount == 0)
    printf("No CUDA compatible GPU exists.\n");
else
{
    cudaDeviceProp pr;
    for(int i=0;i<deviceCount;i++)
    {
        cudaGetDeviceProperties(&pr, i);
        printf("Dev #%-i is %s\n", i, pr.name);
    }
}
```

```

struct cudaDeviceProp {
    char name[256]; // A string identifying the device
    int major; // Compute capability major number
    int minor; // Compute capability minor number
    int maxGridSize [3];
    int maxThreadsDim [3];
    int maxThreadsPerBlock;
    int maxThreadsPerMultiProcessor;
    int multiProcessorCount;
    int regsPerBlock; // Number of registers per block
    size_t sharedMemPerBlock;
    size_t totalGlobalMem;
    int warpSize;
    . . .
};
```

| Memory Hierarchy

| Memory Accesses

I dati allocati nella memoria host non sono visibili dalla GPU e viceversa.

```

int *mydata = new int[N];
. . . // populating the array
foo<<<grid, block>>>(mydata, N); // NOT POSSIBLE!
```

Invece, deve essere esplicitamente copiato da/verso l'host alla GPU

| Memory Allocation and Copy

```

// Allocate memory on the device.
cudaError_t cudaMalloc ( void** devPtr, // Host pointer address,
                        // where the address of
                        // the allocated device
                        // memory will be stored
                        size_t size ) // Size in bytes of the
                        // requested memory block

// Frees memory on the device.
cudaError_t cudaFree ( void* devPtr ); // Parameter is the host
                                         // pointer address, returned
                                         // by cudaMalloc

// Copies data between host and device.
cudaError_t cudaMemcpy ( void* dst, // Destination block address
                        const void* src, // Source block address
                        size_t count, // Size in bytes
                        cudaMemcpyKind kind ) // Direction of copy.
```

cudaError_t is an enumerated type. If a CUDA function returns anything other than **cudaSuccess** (0), an error has occurred.

| Memory Copy Kind

The `cudaMemcpyKind` parameter of `cudaMemcpy` is also an enumerated type. The `kind` parameter can take one of the following values:

- `cudaMemcpyHostToHost` = 0, Host to Host
- `cudaMemcpyHostToDevice` = 1, Host to Device
- `cudaMemcpyDeviceToHost` = 2, Device to Host
- `cudaMemcpyDeviceToDevice` = 3, Device to Device (for multi-GPU configurations)
- `cudaMemcpyDefault` = 4, used when Unified Virtual Address space capability is available (see [Section 6.7](#))

| Caveat

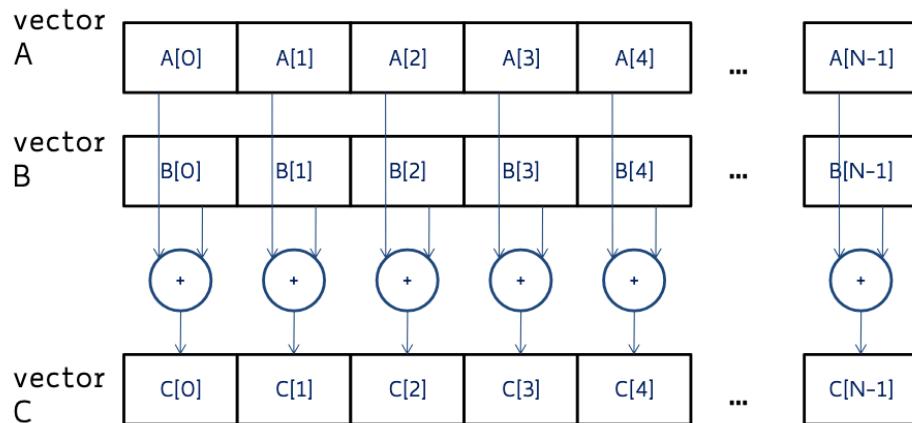
I puntatori allocati tramite `malloc()` e `cudaMalloc()` appartengono a due spazi di indirizzi diversi: un puntatore alla memoria della GPU non può essere alla memoria della GPU non può essere accessibile dall'host e viceversa.

Unified Memory (da CUDA 6) permette di accedere alla memoria della CPU e della GPU utilizzando un unico spazio

| ESEMPIO VECTOR ADDITION

Vector Addition – Conceptual View

ATTENTION: Don't make any assumption on the order in which threads are executed
(Depends on how the GPU decides to schedule the blocks)



Example: Vector Addition

h_* to indicate data allocated on host memory

d_* to indicate data allocated on device (GPU) memory

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

FIGURE 2.10

A more complete version of vecAdd().

Example: Vector Addition - Error check

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

The code shows a function `vecAdd` that performs vector addition. It uses CUDA memory management functions like `cudaMalloc` and `cudaMemcpy`. An annotation highlights the line `cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);` with a red box and a blue arrow pointing to a block of error handling code. This code checks if the allocation failed (`cudaSuccess`), prints the error message from `cudaGetErrorString`, and exits with `EXIT_FAILURE`.

```
cudaError_t err=cudaMalloc((void **) &d_A, size);
if (error !=cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

Example: Vector Addition - Better Error check

```
#define CUDA_CHECK_RETURN(value) { \
    cudaError_t _m_cudaStat = value; \
    if (_m_cudaStat!= cudaSuccess) { \
        fprintf(stderr, "Error %s at line %d in file %s\n", \
            cudaGetErrorString(_m_cudaStat), \
            __LINE__, __FILE__); \
        exit(1); \
    } }
```

```
CUDA_CHECK_RETURN (cudaMalloc ((void **) &da, sizeof (int) * N));
```

Example: Vector Addition

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

- What if n is not a multiple of 256?
- I might run more threads than number of elements in the vector
- Each thread must check if it needs to process some elements or not

- Each block has 256 threads
- We have $n/256$ blocks

FIGURE 2.15

A complete version of the host code in the vecAdd.function.

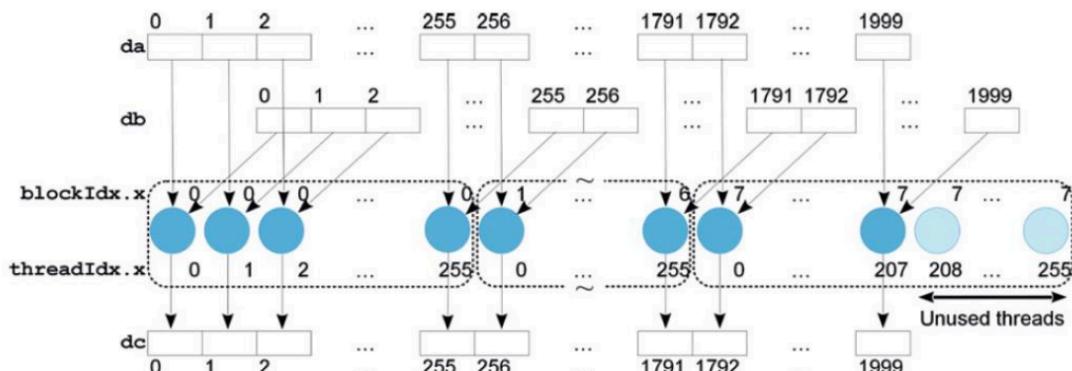
| IMPORTANTE int i

Example: Vector Addition (Kernel)

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

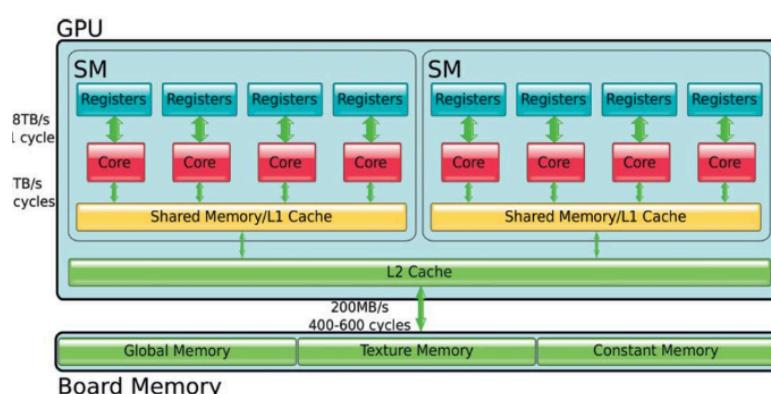
We might have more threads than elements in the array (if the number of elements is not a multiple of the block size)

Example: Vector Addition



| Memory Types

Esistono diversi tipi di memorie, alcune on-chip e altre off-chip.



- Registri: Mantengono le variabili locali
- Memoria condivisa: Memoria veloce su chip per contenere i dati utilizzati di frequente. Può essere utilizzata per scambiare dati tra dello stesso SM.
- Cache L1/L2: Trasparente al programmatore
- Memoria globale: Parte principale della memoria off-chip. Alta capacità ma relativamente lenta. L'unica parte accessibile dall'host attraverso le funzioni CUDA
- Memoria delle texture e delle superfici: Contenuto gestito da un hardware speciale che consente l'implementazione rapida di alcuni di alcuni operatori di filtraggio/interpolazione
- Memoria costante: Può memorizzare solo costanti. Viene memorizzata nella cache e consente la trasmissione di un singolo valore a tutti i hread in un warp (meno interessante sulle GPU più recenti che hanno comunque una cache).

Table 4.1 CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Global	Thread	Kernel
<code>_device_ __shared__ int SharedVar;</code>	Shared	Block	Kernel
<code>_device_ int GlobalVar;</code>	Global	Grid	Application
<code>_device_ __constant__ int ConstVar;</code>	Constant	Grid	Application

Registers

Utilizzato per memorizzare le variabili locali di un thread

I registri su un core sono suddivisi tra i thread residenti.

La capacità di calcolo determina il numero massimo di registri che si possono utilizzare per ogni thread.

Se questo numero viene superato, le variabili locali vengono allocate nella memoria globale fuori dal chip (lenta)

Le variabili versate possono essere memorizzate nella cache L1 on-chip.

Il compilatore deciderà quali variabili saranno allocate nei registri e quali saranno allocate nella memoria globale

```
$ nvcc -Xptxas -v --arch=sm_20 warpFixMultiway.cu
ptxas info    : 0 bytes gmem, 14 bytes cmem[2]
ptxas info    : Compiling entry function '_Z3foov' for 'sm_20'
ptxas info    : Function properties for _Z3foov
                 8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 9 registers, 32 bytes cmem[0], 4 bytes cmem[16]
```

Il numero di registri massimi per thread influenza il numero massimo di thread residenti che si possono avere su uno SM.

Ad esempio,

- Un kernel utilizza 48 registri
- Viene invocato a blocchi di 256 thread
- Ogni blocco richiede $48 * 256 = 12.288$ registri
- Supponiamo che la GPU di destinazione abbia 32.000 registri per SM

- Supponiamo che la GPU di destinazione possa avere fino a 1.536 thread residenti per SM
- Quindi, ogni SM potrebbe avere 2 blocchi residenti ($12.288 * 2 < 32.000$)
- Cioè, ogni SM potrebbe avere $2 * 256 = 512$ thread residenti.
- Questo è molto al di sotto del limite massimo di 1.536 thread.
- Ciò mina la possibilità di nascondere la latenza

NVIDIA definisce come occupazione il rapporto tra i warp residenti e il limite massimo di warp residenti possibili

$$\text{occupancy} = \frac{\text{resident_warps}}{\text{maximum_warps}} = \frac{2 \cdot \frac{256 \text{ threads}}{32 \text{ threads/warp}}}{48 \text{ warps}} = \frac{16}{48} = 33.3\%$$

Un'occupazione vicina a 1 è auspicabile (più è vicina a 1, maggiori sono le opportunità di scambiare tra i thread e di nascondere le latenze).

L'occupazione di un determinato kernel può essere analizzata attraverso un profiler (vedremo come).

Come aumentare l'occupazione?

- Ridurre il numero di registri richiesti dal kernel (ad esempio, evitando di avere troppe variabili temporanee, un'elevata distanza tra il riutilizzo delle stesse variabili, ecc...)
- Utilizzare una GPU con limiti di registri più elevati per ogni thread.

| Shared Memory (on-chip)

Memoria su chip

- Diversa dai registri. I dati dei registri sono privati ai thread, mentre la memoria condivisa è condivisa tra i thread
Può essere vista come una cache L1 gestita dall'utente (scratchpad)
Può essere utilizzata come:
- Un luogo dove conservare i dati usati di frequente che altrimenti richiederebbero un accesso globale alla memoria.
- Come modo per i core sullo stesso SM di condividere i dati.
Lo specificatore `__shared__` può essere usato per indicare che alcuni dati devono andare in memoria on-chip condivisa piuttosto che nella memoria globale.

Memoria condivisa vs. cache L1:

- Entrambe sono su chip. La prima è gestita dal programmatore, la seconda è gestita automaticamente
- In alcuni casi, la gestione manuale (cioè l'uso della memoria condivisa) può fornire prestazioni migliori.

- Ad esempio, non si ha alcuna garanzia che i dati di cui si ha bisogno si trovino nella cache L1, ma con la memoria condivisa gestita in modo esplicito, è possibile controllarlo

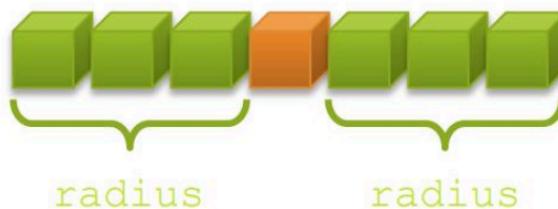
Shared Memory Example: 1D Stenci

- ▶ Consider applying a 1D stencil to a 1D array of elements
 - ▶ Each output element is the sum of input elements within a radius
- ▶ If radius is 3, then each output element is the sum of 7 input elements:

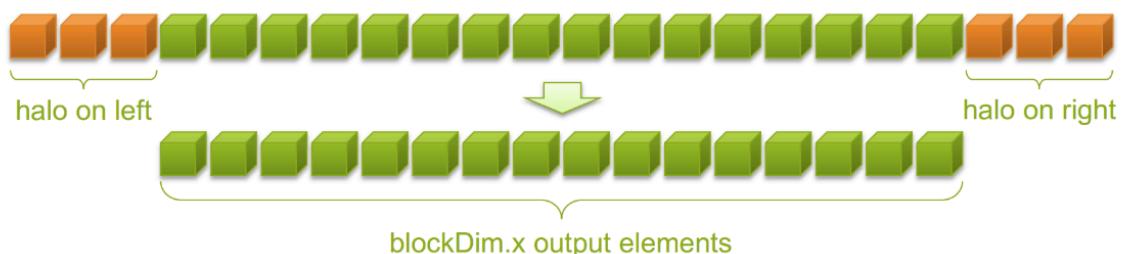


- ▶ Each thread processes one output element
 - ▶ **blockDim.x** elements per block

- ▶ Input elements are read several times
 - ▶ With radius 3, each input element is read seven times



- ▶ Cache data in shared memory
 - ▶ Read (**blockDim.x** + 2 * **radius**) input elements from global memory to shared memory
 - ▶ Compute **blockDim.x** output elements
 - ▶ Write **blockDim.x** output elements to global memory
- ▶ Each block needs a halo of **radius** elements at each boundary

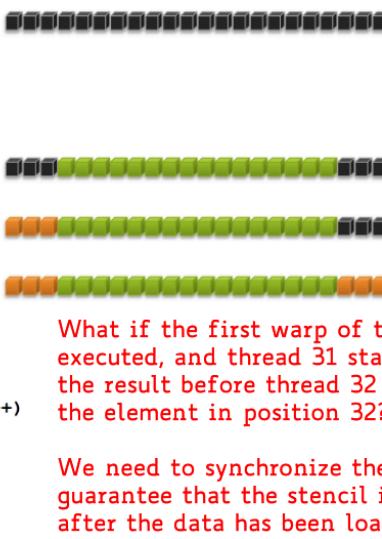


```

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
            in[gindex + BLOCK_SIZE];
    }
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
    // Store the result
    out[gindex] = result;
}

```



What if the first warp of the block gets executed, and thread 31 starts computing the result before thread 32 copies in temp the element in position 32?

We need to synchronize the threads, to guarantee that the stencil is applied only after the data has been loaded in the shared memory

- ▶ **void __syncthreads() ;**

- ▶ Synchronizes all threads within a block
 - ▶ Used to prevent RAW / WAR / WAW hazards
- ▶ All threads must reach the barrier

```

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

```

| Constant Memory

| Constant Memory (on-chip)

Memoria costante != ROM

- È solo una memoria che può contenere valori costanti (cioè, l'host può scriverla)

Due vantaggi principali

- È memorizzata nella cache
- Supporta la trasmissione di un singolo valore a tutti i thread di un warp.

Ad esempio, supponiamo di avere 10 warp su un SM, e che tutti richiedano la stessa variabile

- Se i dati sono nella memoria globale:
 - tutti i warp richiederanno lo stesso segmento dalla memoria globale
 - la prima volta il segmento viene copiato nella cache L2
 - se altri dati passano attraverso la L2, ci sono buone probabilità che vengano persi
 - ci sono buone probabilità che i dati vengano richiesti più volte

Se dati di ff sono in memoria costante:

- durante la prima richiesta di warp, i dati vengono copiati in constant-cache
- poiché c'è meno traffico nella constant-cache, ci sono buone probabilità che tutti gli altri warp troveranno i dati già nella cache

```

__constant__ type variable_name; // static
cudaMemcpyToSymbol(variable_name, &host_src, sizeof(type), cudaMemcpyHostToDevice);
// warning: cannot be dynamically allocated

```

- i dati risiedono nello spazio di indirizzi della memoria costante
- ha una durata di memorizzazione statica (persiste fino alla fine dell'applicazione)
- leggibile da tutti i thread di un kernel

| Performance Estimation

Come possiamo misurare le prestazioni per avere un'idea se stiamo saturando le capacità computazionali dell'hardware?

- FLOP/s (cioè operazioni in virgola mobile al secondo)
 - Che tipo di operazioni in virgola mobile? 64 bit, 32 bit, 16 bit, ...? (Questo deve essere specificato!)
- Oggi abbiamo sistemi capaci di 1 ExaFLOP/s (cioè 1018 FLOP/s).
- È possibile consultare il sito top500.org per avere una visione (parziale).
- FLOP/s vs FLOPs/s vs FLOPs (battaglia religiosa)
 - Ma le “operazioni in virgola mobile” sono FLOP o FLOP?
 - Utilizzeremo:
 - FLOP (“operazioni in virgola mobile”)
 - FLOP/s (“operazioni in virgola mobile per secondo”)
- Ma sappiate che libri/risorse diverse potrebbero indicarlo in modo diverso diversamente

| Esempio image blur

```
pixVal += in[curRow * w + curCol];
```

Tutti i thread accedono alla memoria globale per gli elementi della matrice di input
Supponiamo che la larghezza di banda della memoria globale sia di 200 GB/s.

- Quanti operandi possiamo caricare? $(200 \text{ GB/s}) / (4 \text{ byte}) = 50\text{G} \text{ operandi / s}$
Eseguiamo un'operazione in virgola mobile ($+=$) su ciascun operando
- Quindi, nel migliore dei casi, possiamo aspettarci una prestazione di picco di 50 GFLOP/s.

Supponiamo che la velocità di picco in virgola mobile di questa GPU sia 1.500 GFLOP/s
Questo limita la velocità di esecuzione al 3,3% (50/1500) della velocità di esecuzione in virgola mobile di picco della GPU.

Cioè, il movimento da/verso la memoria (piuttosto che la capacità di calcolo) limita la nostra capacità di esecuzione.

Diciamo che questa applicazione è legata alla memoria

È necessario ridurre drasticamente gli accessi alla memoria per avvicinarsi al valore di 1.500 GFLOP/s

Definiamo il rapporto tra calcoli e accessi alla memoria globale come il numero di calcoli in virgola mobile eseguiti per ogni accesso alla memoria.

È noto anche come intensità aritmetica/operativa (misurata in FLOP/byte)

Possiamo caricare al massimo 50 G operandi / s

Per raggiungere il picco di 1,5 TFLOP/s del processore, abbiamo bisogno di un rapporto di 30 o superiore 1,5 T / 50 G

cioè, dovremmo eseguire 30 operazioni in virgola mobile su ogni operando

La tendenza tecnologica non è incoraggiante: il throughput computazionale cresce più velocemente computazionale cresce a un ritmo più veloce della larghezza di banda della memoria.

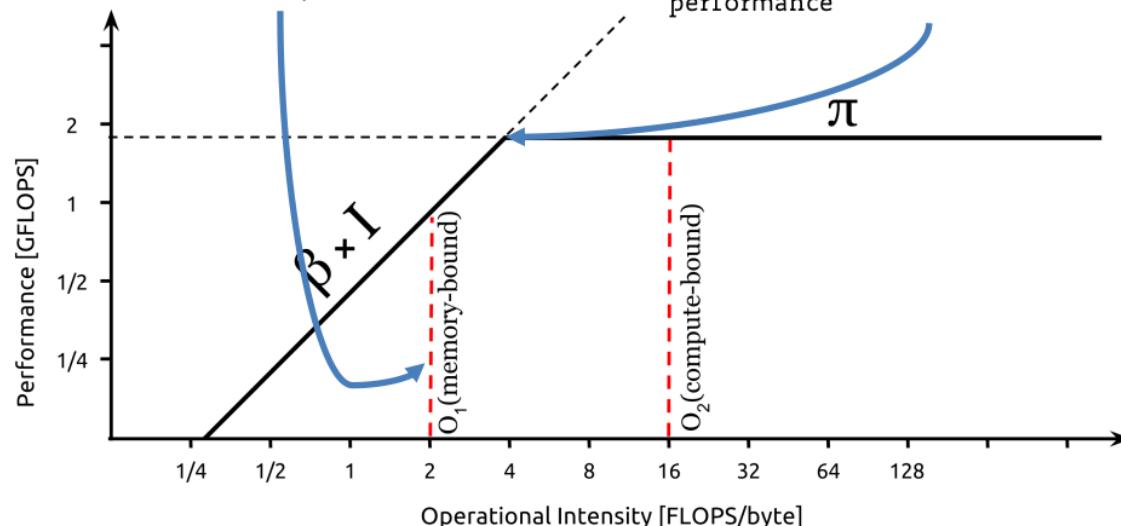
| Roofline Model

$\beta = 0.5 \text{ GB/s}$ (mem. Bandwidth)

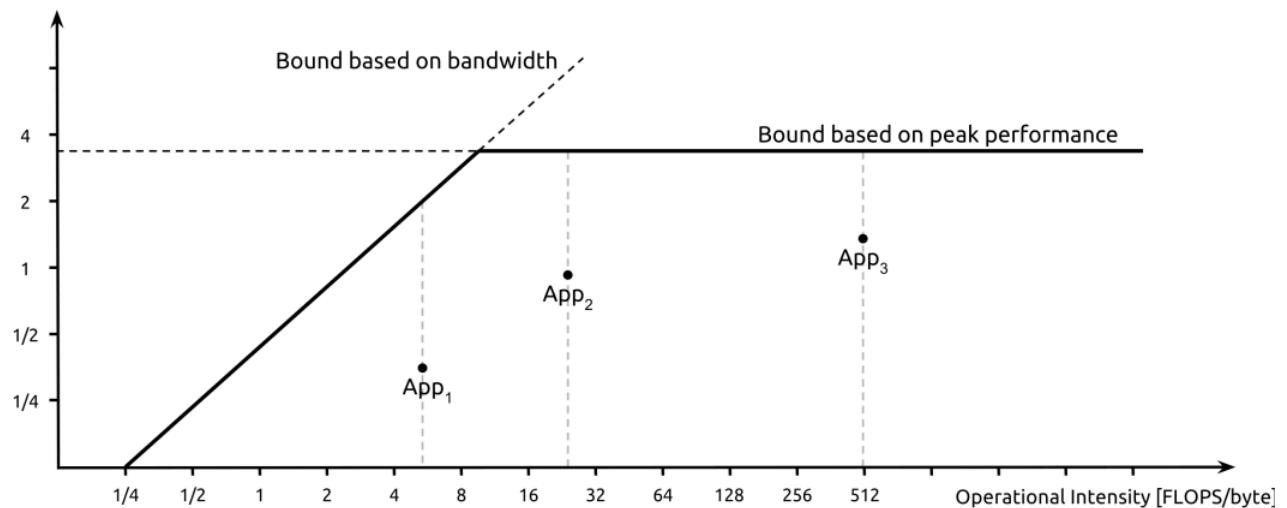
With an op. intensity of 2 flop/byte,
in the best case, I can expect a
peak performance of 1 GFLOP/s

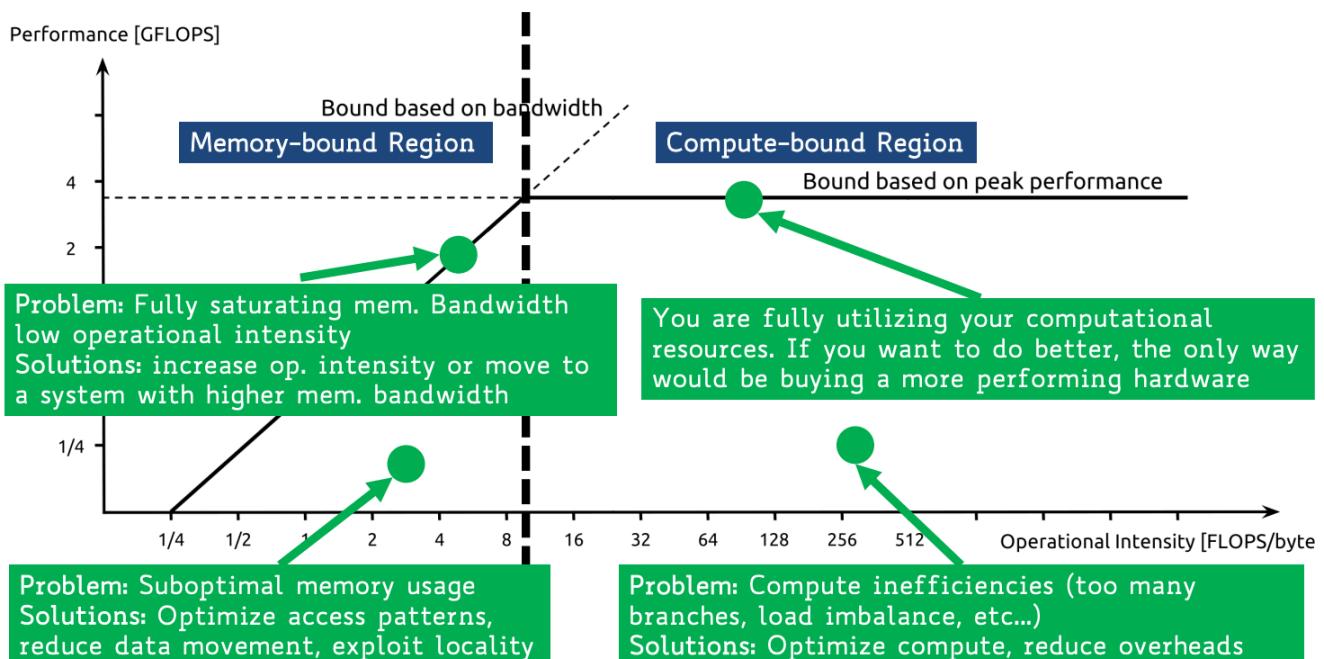
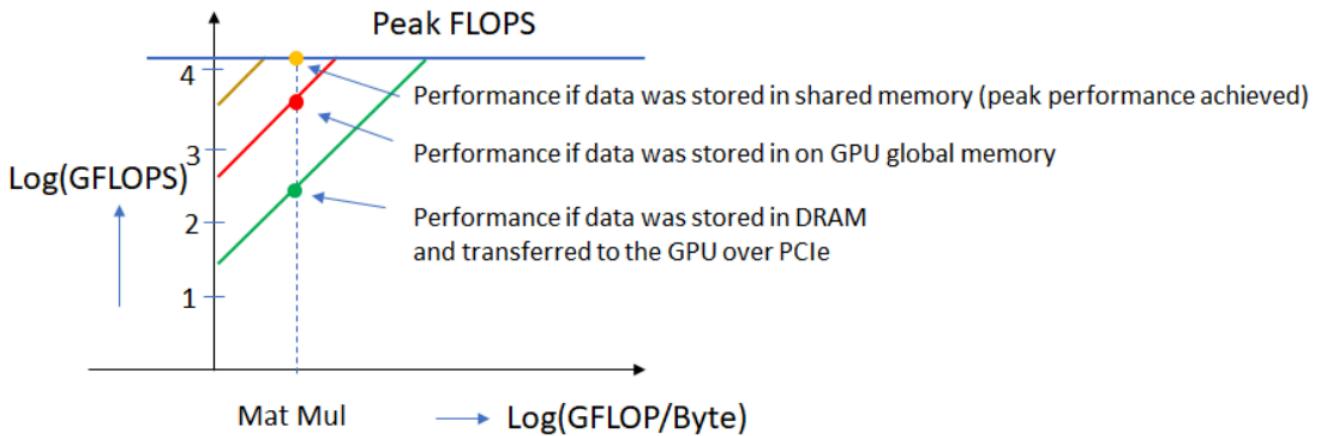
Ridge point

I need to perform at least 4 FLOP for
each byte I load in order to get peak
performance



Performance [GFLOPS]





| ESEMPIO MATRIX MULTIPLICATION WITH TILING

[Qua](#)

| 16.1 - Esempi con CUDA

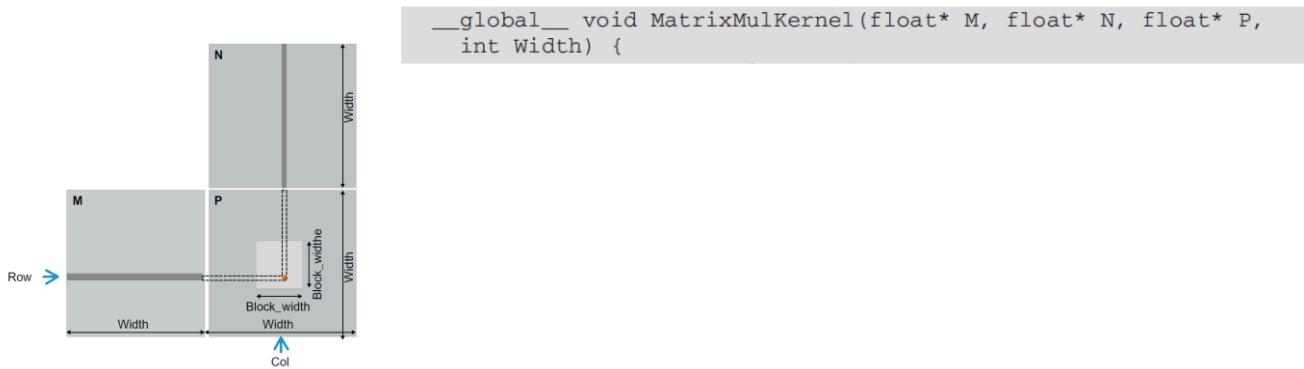
| Image Blur

Slide 18 da 29 a 46

| Matrix Multiplication with Tiling

Slide 18 da 58 alla fine

Matrix Multiplication



Matrix Multiplication

Two global memory accesses
(one element from M and
one from N)

One multiplication and
one addition

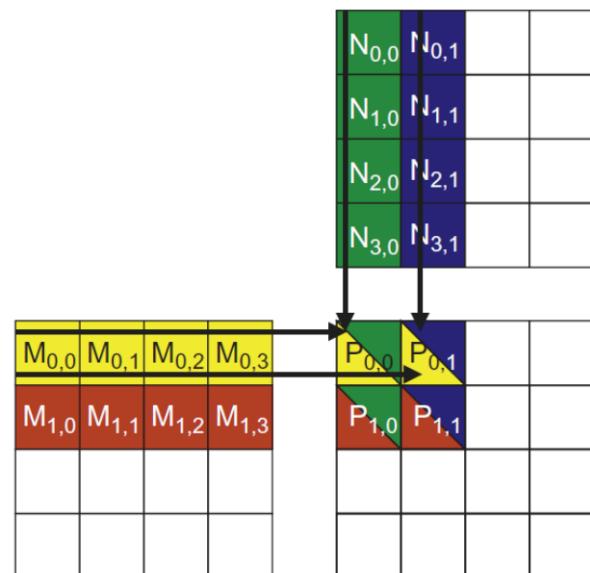
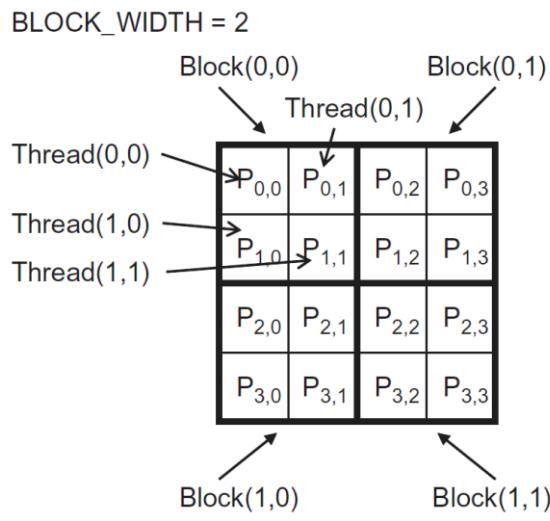
Arithmetic intensity
(FLOP/operand) = 1

Arithmetic intensity
(FLOP/byte) = 8

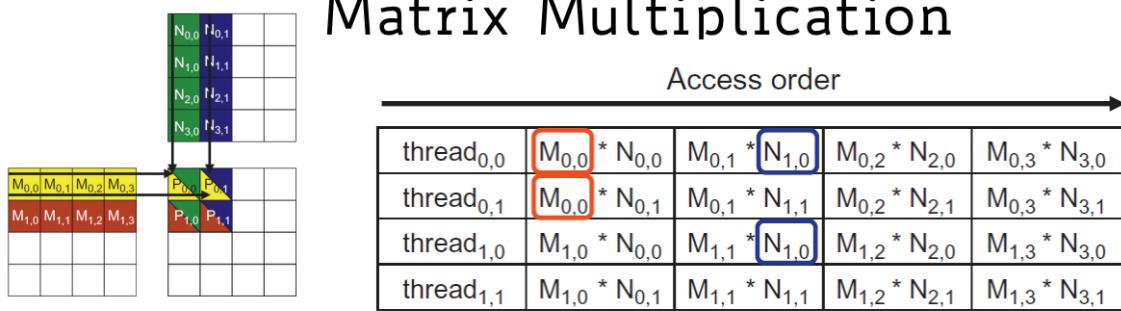
How to increase it? Let's try
to exploit shared on-chip
memory

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
                                int Width) {
    // Calculate the row index of the P element and M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column index of P and N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[Row*Width+k]*N[k*Width+Col];
        }
        P[Row*Width+Col] = Pvalue;
    }
}
```

Matrix Multiplication

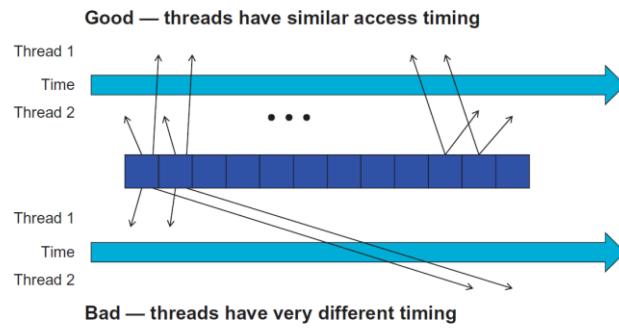


Matrix Multiplication



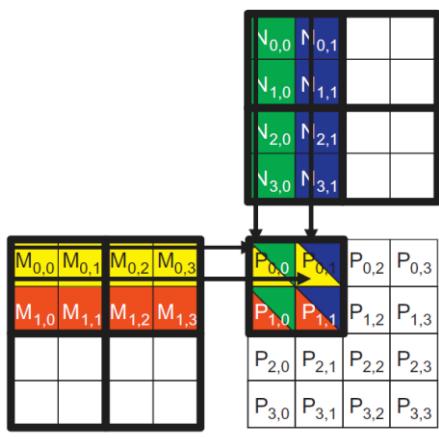
- Each element is accessed by 2 threads at the same time
- I.e., each element is loaded twice from global memory
- We could have one thread loading it from global to shared memory, and the other one reading it from shared memory, reducing traffic to global memory by 2x
- **IMPORTANT:** The potential for memory traffic reduction depends on the block size (i.e., for 16x16 blocks we could reduce it by 16x)

Matrix Multiplication



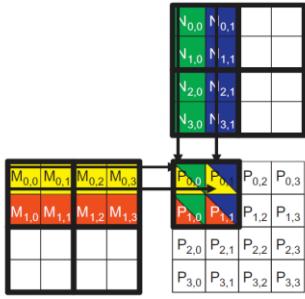
- Shared memory is small, it is important that once we load a value, that value is going to be accessed again soon
- If accesses are too distant in time, then we might have loaded something different in the meanwhile

Tiling



- We also divide the input matrices in tiles (of the same size of the tiles of the output matrix)
- For tile in (0, 2)
 - Each thread now loads one element from the first tile of M and one from the first tile of N from global to shared memory
 - E.g., P_{0,0} loads N_{0,0} and M_{0,0}
 - Sync (barrier)
 - Each thread increments the partial value of the output (using the sub-rows and sub-cols they loaded)

Tiling



	Phase 1			Phase 2		
thread _{0,0}	M _{0,0} ↓ Mds _{0,0}	N _{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M _{0,2} ↓ Mds _{0,0}	N _{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M _{0,1} ↓ Mds _{0,1}	N _{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M _{0,3} ↓ Mds _{0,1}	N _{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M _{1,0} ↓ Mds _{1,0}	N _{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M _{1,2} ↓ Mds _{1,0}	N _{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M _{1,1} ↓ Mds _{1,1}	N _{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M _{1,3} ↓ Mds _{1,1}	N _{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

time →

- Without tiling: 4 threads, each loads one row (4 loads) and one column (4 loads) = 4x(4+4) = 32 accesses to global memory
- With tiling: 4 threads: each loads 4 elements from global memory = 16 accesses to global memory (2x reduction in global accesses)

Matrix multiplication with tiling: code

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
                                int Width) {
    1.    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    2.    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    3.    int bx = blockIdx.x; int by = blockIdx.y;
    4.    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the d_P element to work on
    5.    int Row = by * TILE_WIDTH + ty;
    6.    int Col = bx * TILE_WIDTH + tx;

    7.    float Pvalue = 0;
    // Loop over the d_M and d_N tiles required to compute d_P element
    8.    for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

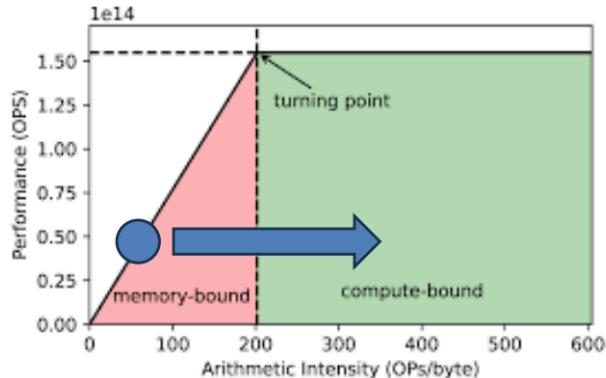
        // Collaborative loading of d_M and d_N tiles into shared memory
        9.        Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
        10.       Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
        11.       __syncthreads();

        12.      for (int k = 0; k < TILE_WIDTH; ++k) {
            13.          Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        14.      __syncthreads();
    }
    15.     d_P[Row*Width + Col] = Pvalue;
}
```

How many blocks etc?

```
for (int k = 0; k < TILE_WIDTH; ++k) {
    Pvalue += Mds[ty][k] * Nds[k][tx];
}
```

- Each block works on a tile, so `TILE_WIDTH == BLOCK_WIDTH`
- For 16x16 tiles, in each phase, the inner loop is executed 16 times. Before the loop each thread loaded two elements from global memory (one from M and one from N). In the loop we do 2 FLOP per iteration. Thus, 2 loads and 32 FLOP, we have an arithmetic intensity of 16
- For 32x32 tiles, it would be 32



- A GPU has a fixed size for its shared memory
- E.g., let's assume it has a 16KB memory size and at most 1536 threads per SM
- For $\text{ TILE_WIDTH } = 16$, each thread block uses $2*16*16*4B = 2\text{KB}$ of shared memory (i.e., we can have at most 8 thread blocks executing – i.e., $8*16*16 = 2048$ threads per SM)
 - Since we can have at most 1536 threads per SM, we would need to run 6 blocks – $6*16*16 = 1536$, **saturating the number of threads**
- For $\text{ TILE_WIDTH } = 32$, each thread block uses $2*32*32*4B = 8\text{KB}$ of shared memory (i.e., we can have at most 2 thread blocks executing – i.e., $2*1024 = 2048$ threads per SM)
 - Since we can have at most 1536 threads per SM, we would need to run 1 block – $1*32*32 = 1024$, **leaving 512 threads unused**

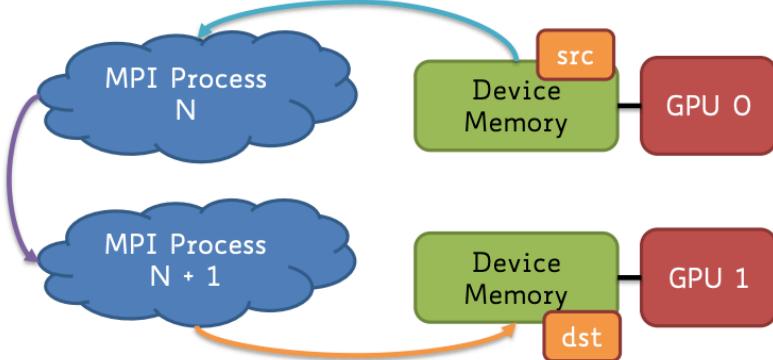
| 17 CUDA + MPI

Esistono due possibili soluzioni, in base al fatto che MPI è GPU-aware o meno:

1. MPI non è GPU-aware: i dati devono essere esplicitamente trasferiti esplicitamente dal dispositivo all'host prima di effettuare la chiamata MPI chiamata MPI desiderata. Sul lato ricevente, queste azioni devono essere ripetute all'inverso.
 2. MPI è consapevole della GPU: MPI può accedere ai buffer del dispositivo direttamente, quindi i puntatori alla memoria del dispositivo possono essere utilizzati nelle chiamate MPI.
- Al momento, sul cluster MPI non è consapevole della GPU.

| MPI not GPU-Aware

- Source MPI process:
 - – cudaMemcpy(tmp, src, cudaMemcpyDeviceToHost) – MPI_Send(tmp, ..., N+1, ...) // send tmp to N+1
- – Destination MPI process:
 - MPI_Recv(tmp, ..., N, ...) // recv tmp from N+1
 - cudaMemcpy(dst, tmp, cudaMemcpyHostToDevice)



| ATTENZIONE

Nel cluster, ogni nodo GPU ha 2 GPU

Pertanto, è necessario eseguire con 2 processi MPI per nodo (ciascuno dei quali gestisce una GPU)

È possibile specificare la GPU su cui si desidera eseguire con cudaSetDevice, ad es:

```
if(rank % 2 == 0){  
    cudaSetDevice(0); }  
else{ cudaSetDevice(1); }
```

| MPI GPU-Aware

Non è necessario copiare i dati tra GPU e memoria host memoria

- In questo modo si risparmia almeno una copia di memoria sulla GPU di memoria sul lato di invio e una copia di memoria sul lato di ricevente.
- Utile per le GPU sullo stesso nodo
- Ma anche tra nodi diversi
 - Per altri dettagli: <https://arxiv.org/abs/2409.09874>

| * CCL: Beyond MPI

Le librerie * CCL sono sempre più diffuse e sviluppate da grandi aziende (NCCL, RCCL, HCCL, ecc.).

- Forniscono alcuni collettivi (per lo più quelli necessari per la formazione in ML - allreduce, reduce-scatter, allgather e pochi altri) e point-to-point
- Non aderiscono ad alcuno standard, per cui sono più flessibili e possono innovare molto più velocemente
- MPI è uno standard enorme e qualsiasi cambiamento deve garantire la retrocompatibilità
- La progettazione di MPI è guidata dalla comunità attraverso la discussione aperta. Le modifiche richiedono anni per essere accettate e adottate dalle implementazioni
- Tuttavia, è utile avere qualcosa di standard. Che cosa se ho un sistema che utilizza sia GPU AMD che NVIDIA? (NCCL e RCCL non si parlano).

In pratica, * CCL fornisce spesso prestazioni superiori a MPI (almeno per i collettivi).

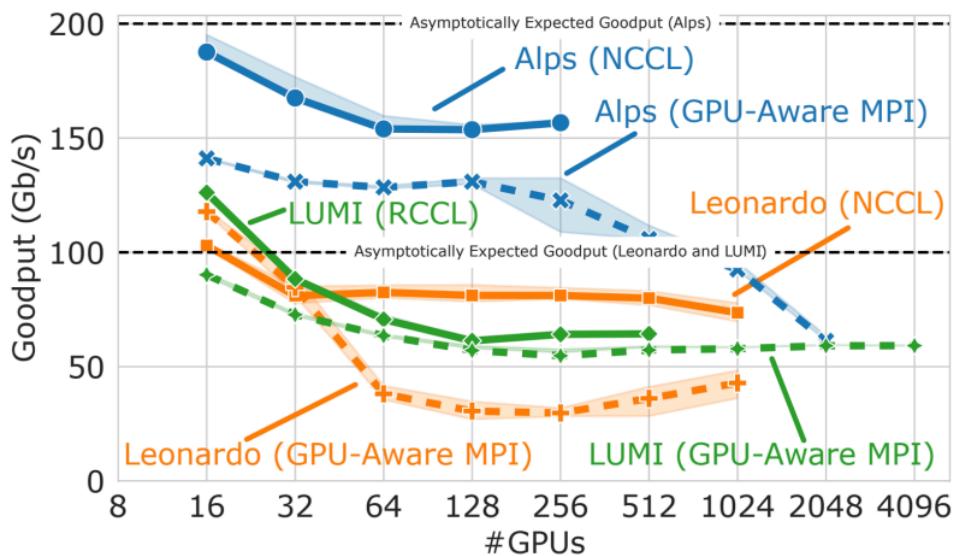


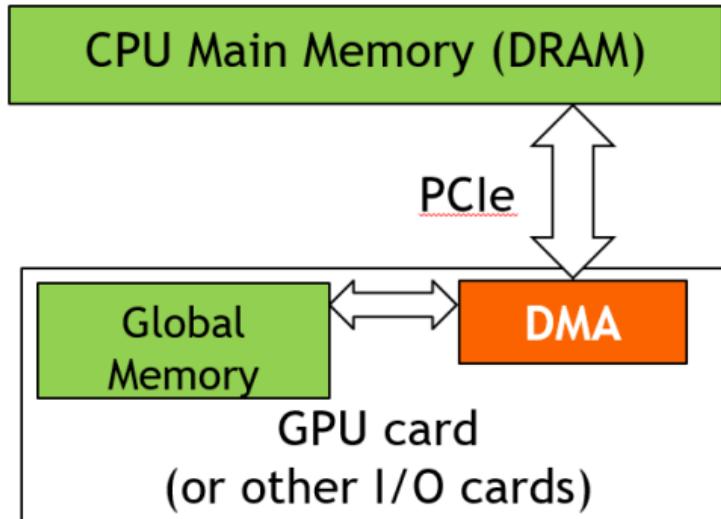
Fig. 9: 2 MiB alltoall scalability.

| Pinned Memory

| CPU-GPU Transfers using DMA

L'hardware DMA (Direct Memory Access) viene utilizzato da `cudaMemcpy()` per una migliore efficienza

- Libera la CPU per altri compiti
- Unità hardware specializzata nel trasferimento di un certo numero di byte richiesti dal sistema operativo tra regioni dello spazio di indirizzamento della memoria fisica (alcune possono essere mappate in posizioni di memoria I/O)
- Utilizza l'interconnessione di sistema (ad esempio, PCIe)



| Virtual Memory Management

I sistemi moderni utilizzano la gestione della memoria virtuale

- Molti spazi di memoria virtuale mappati in un'unica memoria fisica
- Gli indirizzi virtuali (valori dei puntatori) vengono tradotti in indirizzi fisici.

Non tutte le variabili e le strutture dati si trovano sempre nella memoria fisica.

- Ogni spazio di indirizzi virtuali è diviso in pagine che vengono mappate in entrata e in uscita dalla memoria fisica.
- Le pagine di memoria virtuale possono essere mappate fuori dalla memoria fisica (page-out) per fare spazio
- La presenza o meno di una variabile nella memoria fisica viene verificata al momento della traduzione degli indirizzi.

| Data Transfer and Virtual Memory

Il DMA utilizza indirizzi fisici

- Quando cudaMemcpy() copia un array, è implementato come uno o più trasferimenti DMA.
- L'indirizzo viene tradotto e la presenza della pagina viene controllata per l'intera regione di origine e di destinazione all'inizio di ogni trasferimento DMA.
- Nessuna traduzione dell'indirizzo per il resto dello stesso trasferimento DMA, in modo da ottenere un'elevata efficienza.

- Il sistema operativo potrebbe accidentalmente fare page-out dei dati che vengono letti o scritti da un DMA e o scritti da un DMA e impaginare un'altra pagina virtuale nella stessa posizione fisica.

La memoria appuntata è costituita da pagine di memoria virtuale appositamente contrassegnate in modo da non poter essere essere paginate fuori

Allocate con una speciale chiamata di funzione API di sistema

- anche nota come memoria bloccata dalla pagina, pagine bloccate, ecc.
La memoria della CPU che serve come sorgente o destinazione di un trasferimento DMA deve essere allocata come memoria bloccata

| CUDA Data Transfer uses pinned memory

Il DMA usato da cudaMemcpy() richiede che qualsiasi sorgente o destinazione nella memoria host sia allocata come memoria appuntata.

- Se una sorgente o una destinazione di una cudaMemcpy() nella memoria host non è allocata in una memoria appuntata, deve essere prima copiata in una memoria appuntata.
- una memoria appuntata - overhead aggiuntivo
- cudaMemcpy() è più veloce se la sorgente o la destinazione della memoria host è allocata in memoria pinnata, poiché non è necessaria una copia aggiuntiva.

| Page-locked Memory

La collocazione dei dati del programma nella memoria pinnata bloccata a pagina risparmia trasferimenti di dati aggiuntivi, ma può essere dannoso per l'efficienza della memoria virtuale dell'host.

La memoria bloccata può essere allocata con:

- malloc() seguita da una chiamata a mlock(). La disallocazione avviene nell'ordine inverso, cioè munlock() e poi free().
- Oppure chiamando la funzione cudaMallocHost(). La memoria allocata in questo modo deve essere deallocated con una chiamata a cudaFreeHost().

Il guadagno di prestazioni ottenuto con la memoria pinned dipende dalla dimensione dei dati da trasferire.

Il guadagno può variare dal 10% a ben 2,5 volte.

```
cudaError_t cudaMallocHost(
    void ** ptr, // Addr. of pointer to pinned
                 // memory (IN/OUT)
    size_t size); // Size in bytes of request (IN)

cudaError_t cudaFreeHost (void * ptr);
```

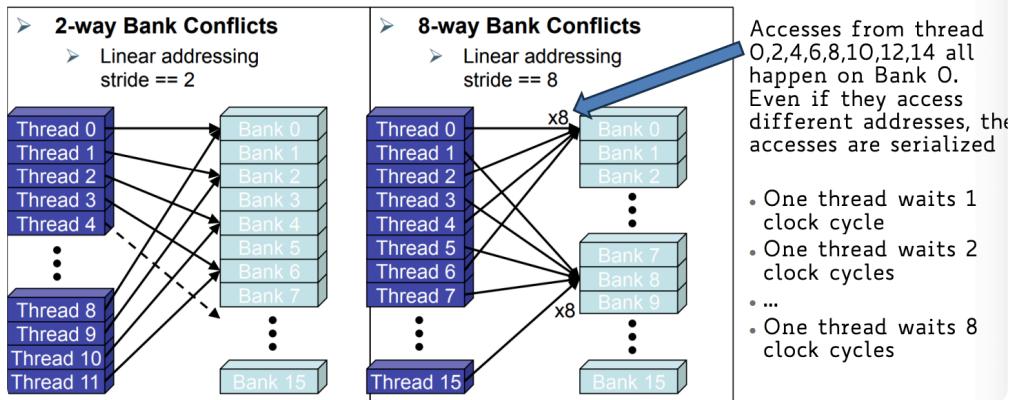
| Bank Conflicts in Shared Memory

La memoria condivisa è suddivisa in banchi come illustrato di seguito:

Address	Bank															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	
64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124	
128	132	136	140	144	148	152	156	160	164	168	172	176	180	184	188	

- Attenzione: gli indirizzi sono interfogliati
- I dispositivi CC 2.0 e successivi hanno 32 banchi. I dispositivi precedenti ne avevano 16.
- Ogni banco può servire un accesso per ciclo
- cioè, se i thread accedono a banchi diversi della memoria condivisa, l'accesso è istantaneo.
- Se i thread accedono a dati diversi ma sullo stesso banco, l'accesso è serializzato

Examples



Take-Home Message

I thread in un warp/half-warp dovrebbero evitare di accedere contemporaneamente a alle locazioni dello stesso banco di memoria condivisa

Qualche informazione utile in più:

- Se tutti i thread accedono alla stessa locazione di memoria, l'hardware effettua una lettura broadcast (non ci sono conflitti)
- A partire da CC 2.0, se sottoinsiemi di thread accedono alla stessa locazione di memoria, l'hardware esegue una lettura multicast (non ci sono conflitti).

Global Memory Coalescing

In pratica, quando un thread accede a una locazione di memoria, viene letto un di locazioni consecutive viene effettivamente letto (in modo simile a quando si carica un blocco di locazioni di memoria consecutive in una linea di linea di cache)

Quando tutti i thread in un ordito eseguono un'istruzione di caricamento, l'hardware rileva se

accedono a locazioni di memoria globale

In questo caso, l'hardware raggruppa tutti questi accessi in un unico accesso

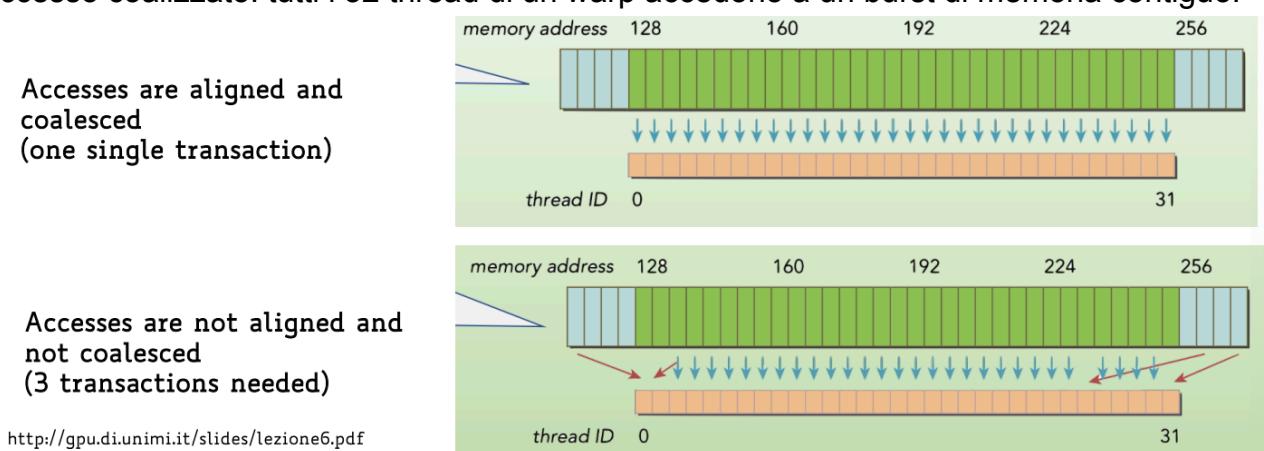
- Ad esempio, per una data istruzione di caricamento di un warp, se il thread 0 accede alla locazione di memoria globale N2, il thread 1 alla locazione N+1, thread 2 alla locazione N+2, e così via, tutti questi accessi saranno tutti questi accessi saranno coalizzati, ovvero combinati in una singola richiesta per consecutivi quando si accede alle DRAM.

I dispositivi CUDA potrebbero imporre dei requisiti sull'allineamento di N (ad esempio, deve essere un multiplo di 16)

Intuizione: se più thread in un warp accedono a posizioni vicine tra loro, viene emessa un'unica transazione di memoria, leggendo un burst di elementi.

Accesso allineato: il primo indirizzo della transazione è un multiplo della granularità della cache (di solito, 32 byte per la cache L2 e 128 byte per la L1).

Accesso coalizzato: tutti i 32 thread di un warp accedono a un burst di memoria contiguo.



Global Memory Accesses

Due tipi di carico:

- Carichi in cache:
- Utilizzato per impostazione predefinita per i dispositivi dotati di cache L1.
- Controlla L1, se non è presente, controlla L2, se non è presente, controlla la memoria globale
- Granularità del carico: linea di 128 byte

Carichi non in cache:

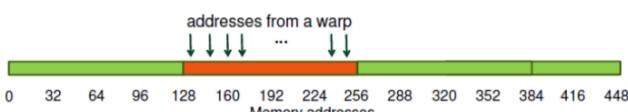
- Se il dispositivo non dispone di cache L1
- Oppure se ha la cache L1 e si compila con -Xptxas -dlcm=cg
- Controlla la L2, se non è presente, controlla la memoria globale
- Granularità del carico: riga da 32 byte

Per i negozi:

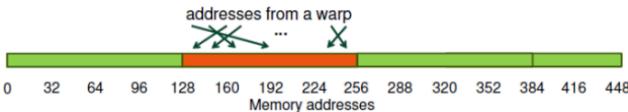
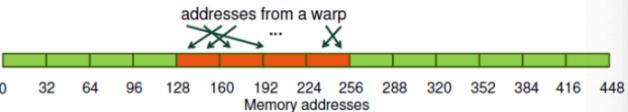
- Invalidare la L1, riscrivere nella L2
- (ecco perché non abbiamo una falsa condivisione)

Perchè dovremmo disabilitare la cache L1?

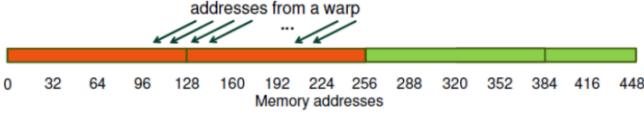
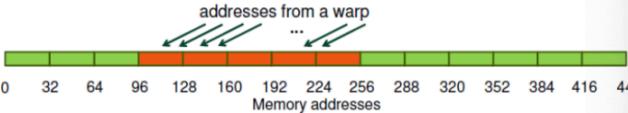
Warp requests 32 aligned, consecutive 4-byte words (128 bytes)

Caching Load	Non-caching Load
Addresses fall within 1 cache-line	Addresses fall within 4 segments
128 bytes move across the bus	128 bytes move across the bus
Bus utilization: 100%	Bus utilization: 100%
 <p>addresses from a warp ↓ ↓ ↓ ... ↓ Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448</p>	 <p>addresses from a warp ↓ ↓ ↓ ... ↓ Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448</p>

Warp requests 32 aligned, permuted 4-byte words (128 bytes)

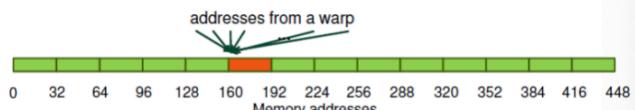
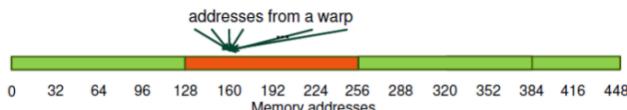
Caching Load	Non-caching Load
Addresses fall within 1 cache-line	Addresses fall within 4 segments
128 bytes move across the bus	128 bytes move across the bus
Bus utilization: 100%	Bus utilization: 100%
 <p>addresses from a warp X X ... X Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448</p>	 <p>addresses from a warp X X ... X Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448</p>

Warp requests 32 misaligned, consecutive 4-byte words (128 bytes)

Caching Load	Non-caching Load
Addresses fall within 2 cache-lines	Addresses fall within at most 5 segments
256 bytes move across the bus	160 bytes move across the bus
Bus utilization: 50%	Bus utilization: at least 80%
 <p>addresses from a warp / / ... /\ Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448</p>	 <p>addresses from a warp / / ... /\ Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448</p>

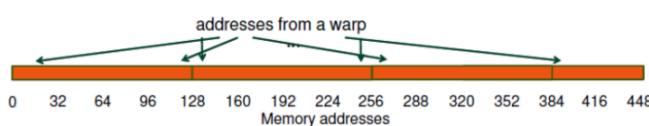
All threads in a warp request the same 4-byte word (4 bytes)

Caching Load	Non-caching Load
Addresses fall within 1 cache-line	Addresses fall within 1 segments
128 bytes move across the bus	32 bytes move across the bus
Bus utilization: 3.125%	Bus utilization: 12.5%



Warp requests 32 scattered 4-byte words (128 bytes)

Caching Load	Non-caching Load
Addresses fall within N cache-lines	Addresses fall within N segments
N*128 bytes move across the bus	N*32 bytes move across the bus
Bus utilization: 128 / (N*128)	Bus utilization: 128 / (N*32)



Riassumendo

Se si hanno accessi alla memoria non coalesced o non-aligned di memoria non allineata o non coalesceda, potrebbe essere utile considerare la disabilitazione della cache L1

Importanza della struttura dei dati organizzazione per gli accessi coalizzati

Array of Struct (AoS) vs. Struct of Array (SoA)

È meglio usare un array, in cui ogni elemento è una struct, o una struct, in cui ogni elemento è un array?

```
struct innerStruct {
    float x;
    float y;
};

struct innerArray {
    float x[N];
    float y[N];
};

struct innerStruct AoS[N];
struct innerArray SoA;
```

- Come sono organizzati i dati in memoria?
- Località della cache?
- Allineamento e accesso coalizzato?

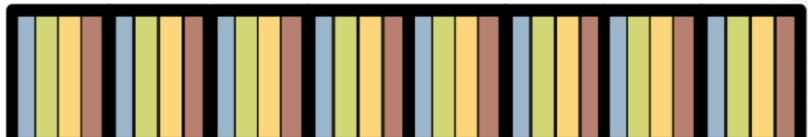
Structure of Arrays (SoA)

```
struct foo{  
    float a[8];  
    float b[8];  
    float c[8];  
    int d[8];  
} A;
```

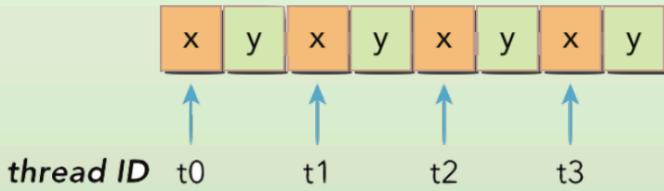


Array of Structures (AoS)

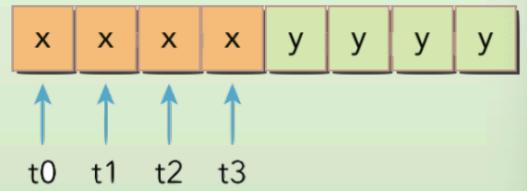
```
struct foo{  
    float a;  
    float b;  
    float c;  
    int d;  
} A[8];
```



AoS memory layout



SoA memory layout



- AoS sprecherebbe spazio nella cache a causa di valori y non necessari.
- Con SoA, inseriamo solo i burst di dati di cui abbiamo bisogno (cioè i burst contenenti solo valori "x").
- SoA consente accessi coalizzati
- SoA potrebbe anche richiedere meno spazio (AoS potrebbe avere del padding dopo ogni struttura)