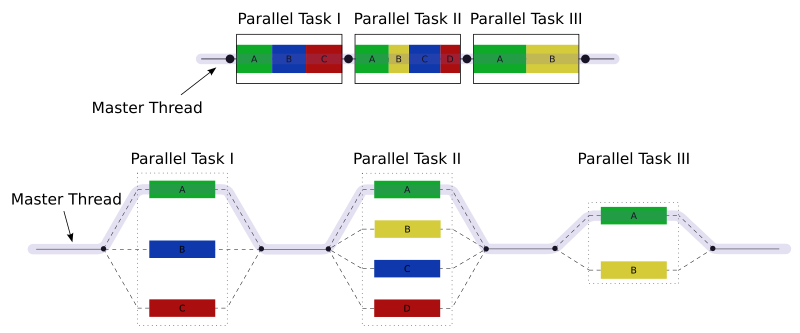
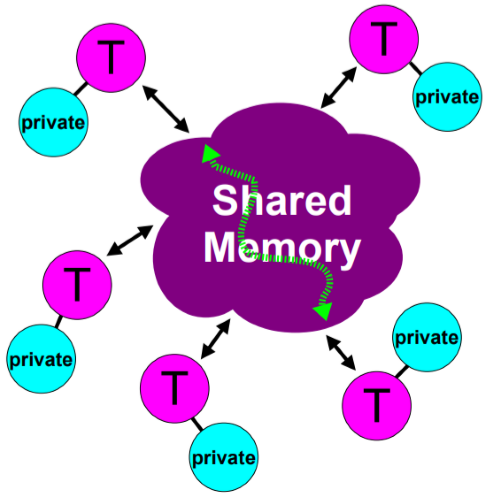
OPEN-MP

E’ **un’API** basata su direttive di compilazione(pragma) che vengono aggiunte al codice sequenziale.Una volta scritto un codice sequenziale, tramite le direttive di compilazione ,si va ad aggiungere del codice parallelo.Si basa su due importanti caratteristiche :

* Modello del parallelismo incrementale.
* Modello del parallelismo locale.

Concetto basilare che sta alla base di un programma OpenMp è che esso viene eseguito alternando una parte di codice seriale e una parte di codice parallelo.

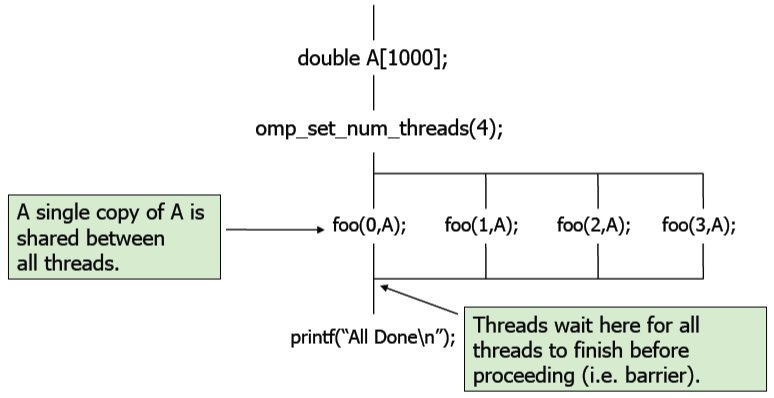
Il modello **Fork-Join** è un modello che fa eseguire porzioni di codice sequenziale in maniera parallela. Solitamente l’esecuzione sequenziale si dirama in alcune operazioni parallele (Threads) per poi ritornare sequenziale.

Il Threads che viene eseguito in maniera sequenziale si dice **Master Thead**.

Ogni thread può accedere alla memoria condivisa ma ognuno di essi contiene anche della memoria privata.

OpenMp si divide in tre parti :

* **Direttive di compilazione**
* **Clausole**
* **Funzioni**
* **Variabili D’Ambiente**

I Threads comunicano attraverso variabili condivise stando attendi alle **race condition** che avvengono quando più threads cercano di modificare una variabile.Per evitare tali problemi si possono utilizzare le diverse tecniche di sincronizzazione, infatti alla fine di ogni sezione parallela vi è una **barriera**.

I Threads vengono numerati da 0 (Master Thread) a N-1.

Ogni thread ha un proprio **contesto di esecuzione** ovvero un partizionamento di alcune risorse (es. memoria) che contiene tutte le variabili a cui un thread può avere accesso.

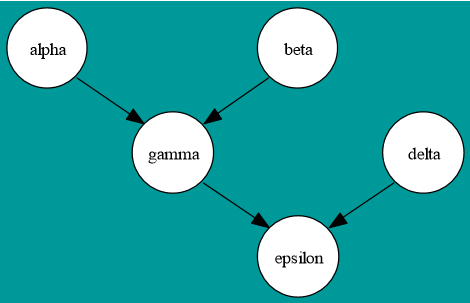
# DIRETTIVE

Sono direttive che si applicano solitamente ad un blocco sequenziale che specificano al compilatore come eseguire in parallelo il codice, ogni direttiva inizia con la dicitura **#pragma omp direttiva**.

Esistono diverse tipologie di direttive :

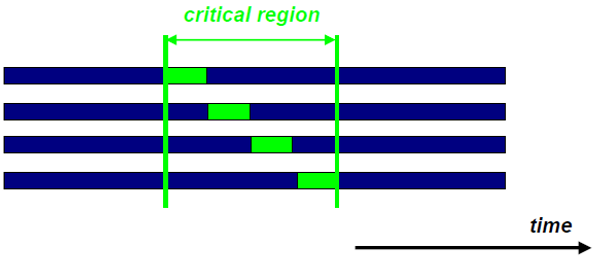
* **For parallel work-sharing**
  + Parallel
  + For
  + Sections
  + Single
* **For master and synchronization**
  + Barrier
  + Critical/Atomic
* **For data environment**

### SECTIONS

Identifica le sezioni di codice che devono essere divise tra tutti i thread. Utile nel caso in cui una porzione di codice debba essere eseguita necessariamente prima di un’altra.

Con la direttiva **#pragma omp sections** faccio partire una sezione parallela dove posso far eseguire più **#pragma omp section.**

### CRITICAL

Questa direttiva è usata per far eseguire parti di codice un thread alla volta (come i semafori) e serve principalmente per proteggere l’accesso ad un blocco di variabili condivise.Non c’è una barriera implicita alla fine della sezione critica,quindi quando un thread entra in essa non attende,alla fine,tutti gli altri.

Quando si deve gestire l’accesso ad *una sola* variabile potrebbe risultare più utile utilizzare la direttiva **atomic**.

# CLAUSOLE

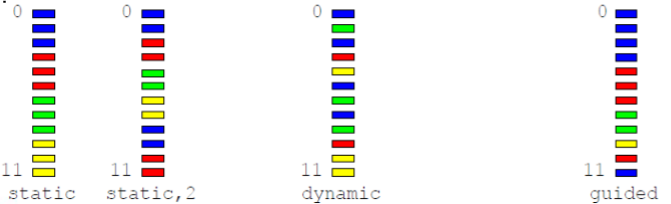
Servono a specificare delle informazioni aggiuntive per le direttive.Esistono diverse tipologie di direttive che seguono la seguente sintassi : **direttiva [clausola[ clausola]...]**

* **For general attributes**
  + If
  + num\_threads
  + shedule
    - *dynamic, guided, runtime, or static.*
  + nowait
* **For data-sharing attributes**
  + Private
  + Firstprivate
  + Lastprivate
  + Shared
  + Dafault
  + Reduction

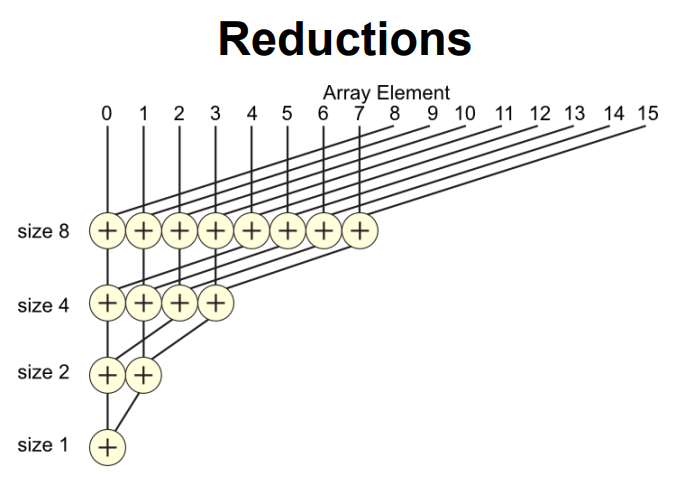
### SCHEDULING

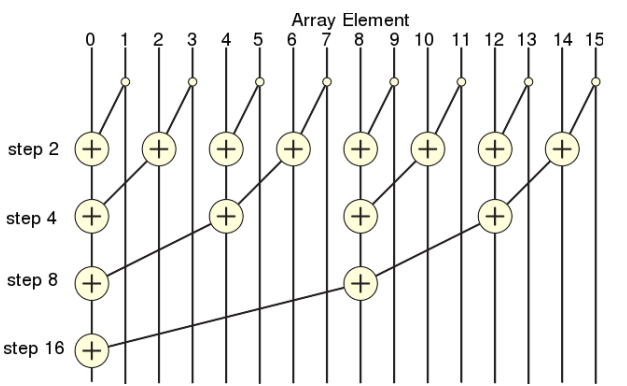
Riguarda lo scheduling su un ciclo, ovvero la gestione del lavoro di un ciclo sui vai threads. Ogni range contiguo di iterazioni di un ciclo viene chiamato **chunks** . Un chunks size più grande riesce a garantire una gestione della cache migliore ma potrebbe portare uno sbilanciamento del lavoro dei vari threads.

* **Static scheduling**
  + Basso Overhead
  + Conosco a priori che la suddivisione del lavoro deve essere equo.
  + Probabile sbilanciamento di lavoro (un thread potrebbe lavorare di più di un altro)
  + Si utilizza quando si vuole far eseguire ad ogni porzione di ciclo la stessa cosa
* **Dynamic scheduling**
  + Overhead maggiore
  + Maggior bilanciamento di carico
  + Si usa quando non si può determinare a priori quale porzione di ciclo lavorerà più degli altri.



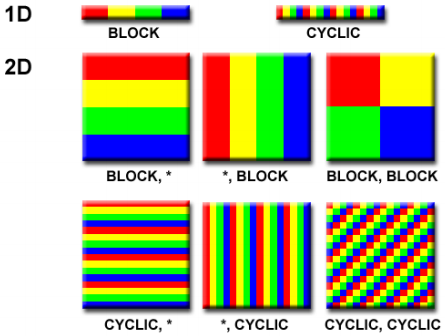
### REDUCTION

Serve per ridurre il numero di iterazioni da *n* a *logn*.



# PARTIZIONAMENTO CICLICO E A BLOCCHI

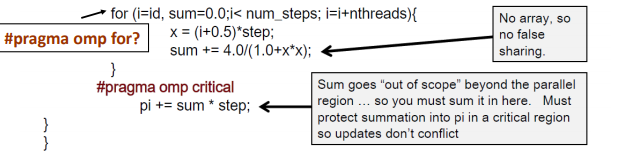
Possiamo partizionare un ciclo in due modi

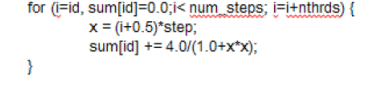


* **A Blocchi**
  + Assegniamo un singolo blocco di carico ad ogni thread.
* **Ciclico**
  + Assegniamo piccoli blocchi di carico in maniera ciclica ad ogni thread.
  + Il procedimento ciclico avviene solitamente senza la direttiva **#pragma omp for**

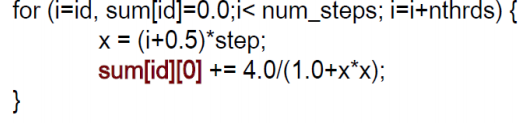
## Partizionamento ciclico

Possiamo andare ad utilizzare un partizionamento ciclico attraverso una singola variabile condivisa



Si può attuare il partizionamento ciclico anche su un vettore ma questo fa sì che il sistema soffra di **False Sharing** poiché ogni volta che un thread deve modificare il valore di memoria della propria cella , in cache viene spostato parte il vettore (comprese quindi anche le celle di memoria di altri thread).Questo può causare una **scalabilità povera**.

Per ovviare a tutti ciò si può utilizzare la tecnica del **padding** attraverso una matrice che ha come righe lo stesso numero di thread mentre il numero di colonne dipende dalla capienza della memoria cache.

Il funzionamento è semplice : *si inseriscono i valori per ogni thread uno per ogni riga ,in modo tale che il cache venga trasportato solo il vettore corrispondente ad una data riga che contiene solo il valore del thread considerato.*