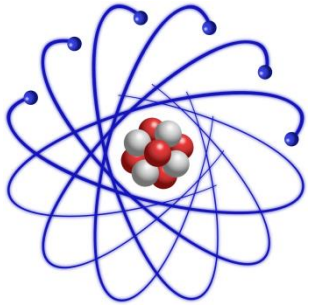
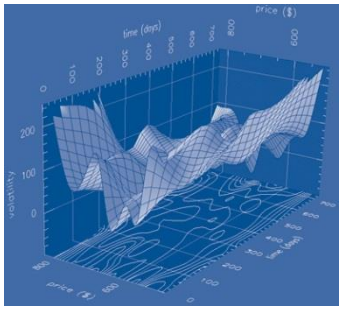
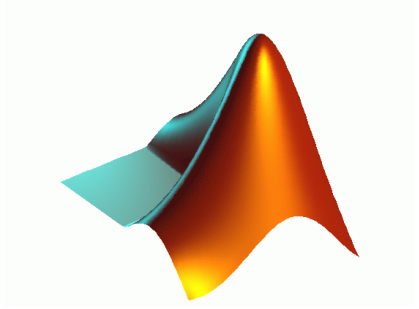




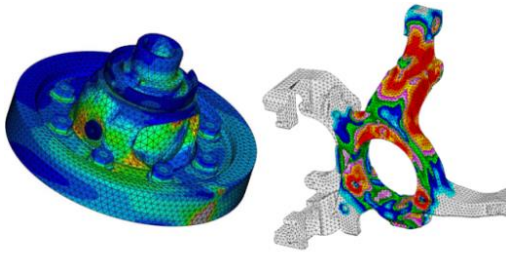


"Concetti generali della programmazione parallela: Parallel multithreading con CPU Intel x86_64 bit"

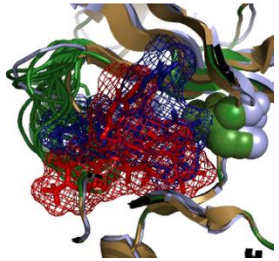
Dott. Costantino Zazza

What is a Cluster ?

- Generally speaking we have that…*”A Cluster is a group of servers/workstations interconnected through a dedicated network designed to act like a single IT processing resource/infrastructure. Clustering improves performance and the system's availability to users actually by distributing workload across all connected servers (multitasking/multiuser environment).”*

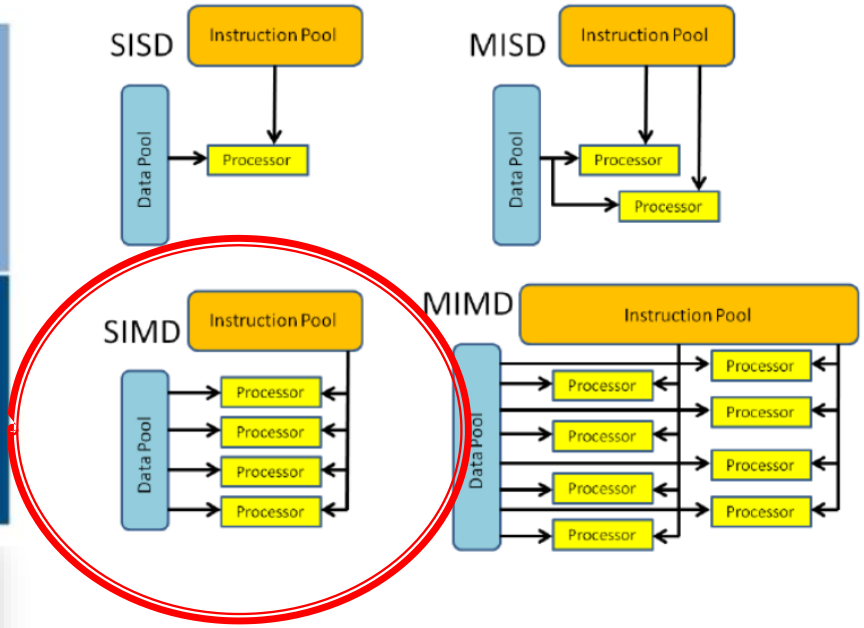


High Performance Computing



Cluster architecture based on Flynn's Taxonomy…

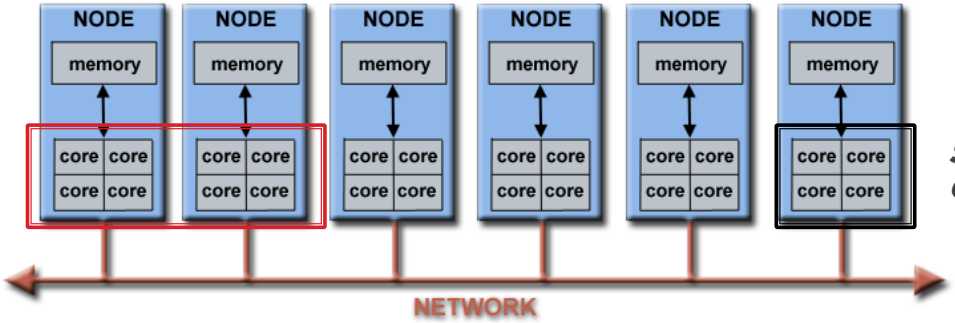
S I S D Single Instruction Single Data	S I M D Single Instruction Multiple Data
M I S D Multiple Instructions Single Data	M I M D Multiple Instructions Multiple Data



High Performance Computing

What does Parallel Computing really mean ?

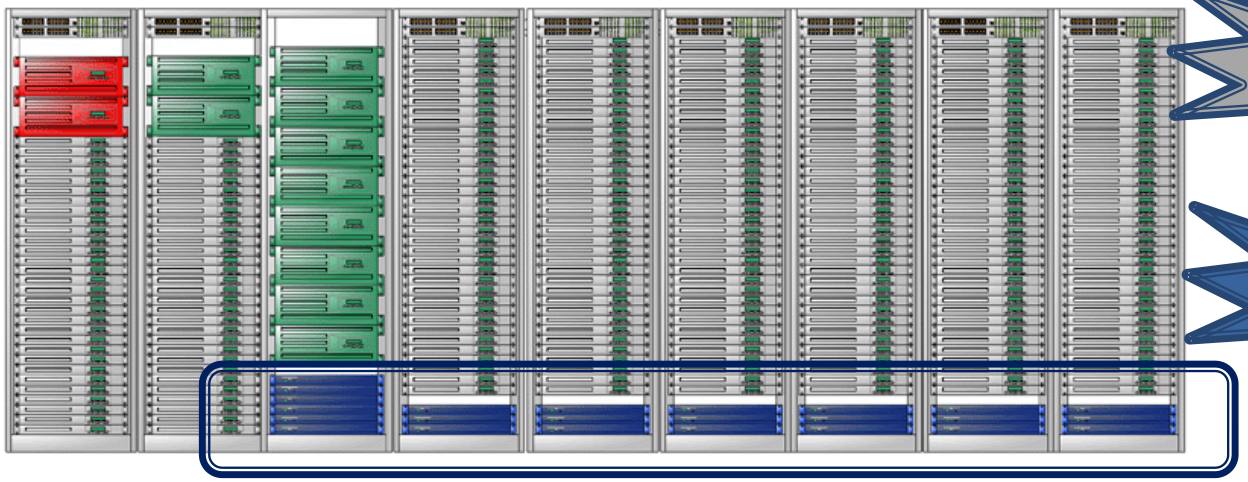
- ***Shared vs. Distributed (Memory) Parallel Computing:***



Distributed Parallel Computation

Shared Parallel Computation

CLUSTER VIEW :



Single-node simulation !

Multi-node simulation !

compute node

infiniband switch

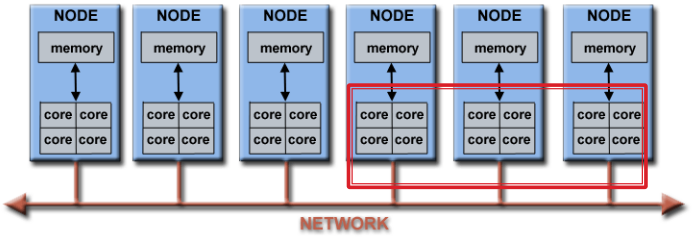
management hardware

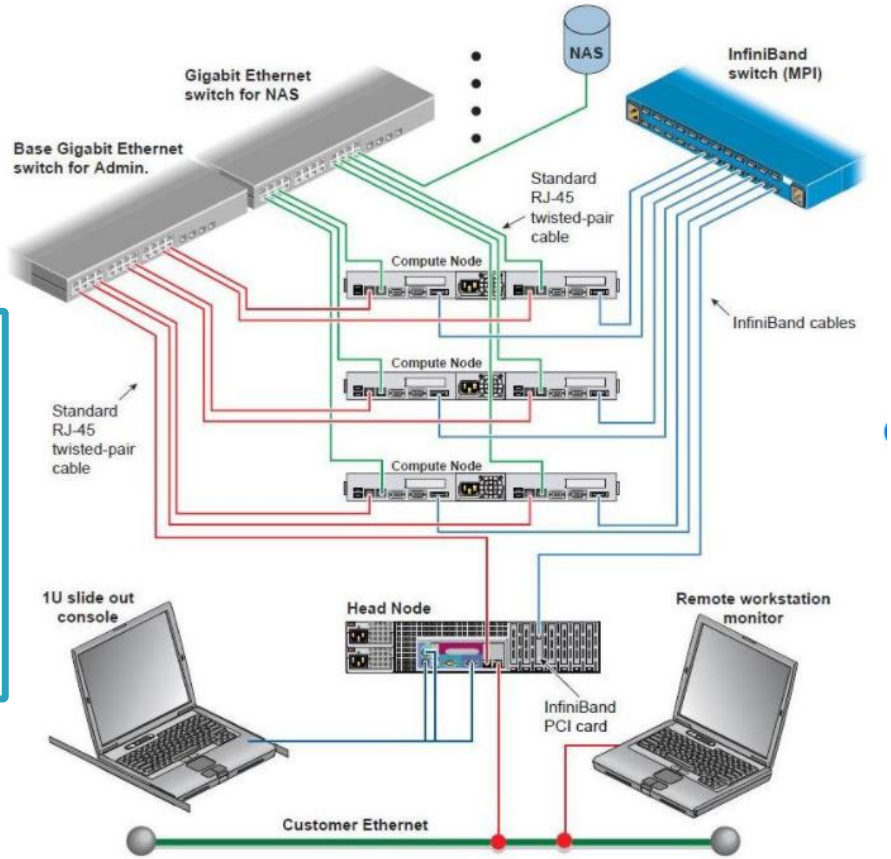
login / remote partition server node

gateway node

What does Parallel Computing really mean ?

- ***Distributed (Memory) Parallel Computing:***





Dual Ethernet and Infiniband switch cluster configuration example

3 compute nodes 1 head node 3 switches

PS. Global Array provides a friendly API for shared-memory programming on distributed-memory computers for multidimensional arrays **but performance may be strongly affected by non local data availability..**

What is OpenMP?

• De-facto standard Application Program Interface (API) to write shared memory parallel applications in C, C++ and Fortran

• Consists of compilers directives, run-time routines and environment variables

• “Open specifications for Multi Processing” maintained by the OpenMP Architecture Review Board (http://www.openmp.org)

• The "workers" who do the work in parallel (threads) "cooperate" through shared memory

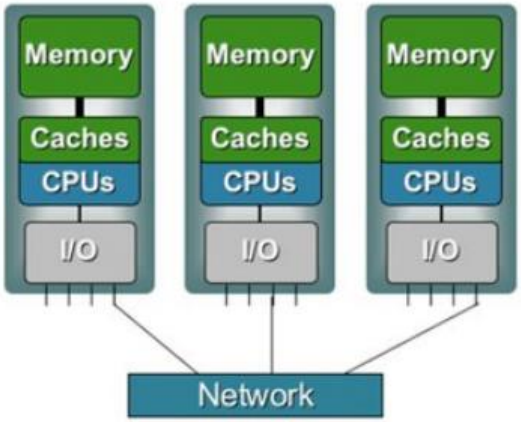
• Memory accesses instead of explicit messages

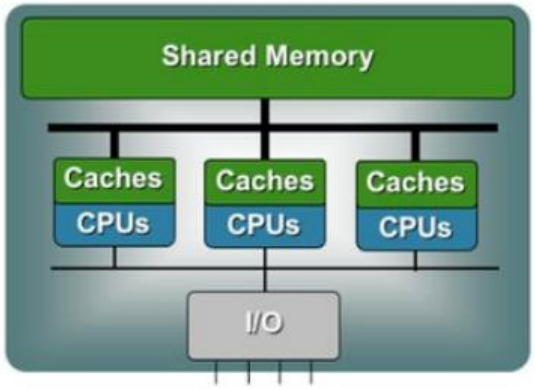
• "local" model parallelization of the serial code

• It allows an incremental parallelization

Shared memory architectures

• All processors may access the whole main – RAM – memory **at run time…**





- Non-Uniform Memory Access
  - Memory access time is non-uniform

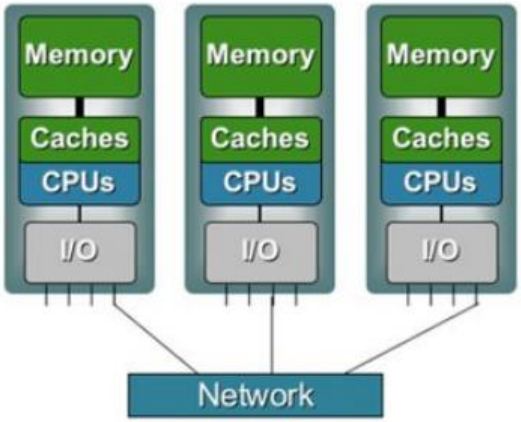
NUMA

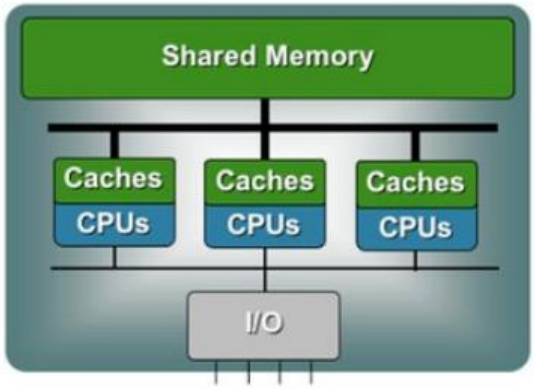
- Uniform Memory Access
  - Memory access time is uniform

UMA

NUMA vs UMA architectures

• All processors may access the whole main – RAM – memory **at run time…**





- Non-Uniform Memory Access
  - Memory access time is non-uniform

NUMA

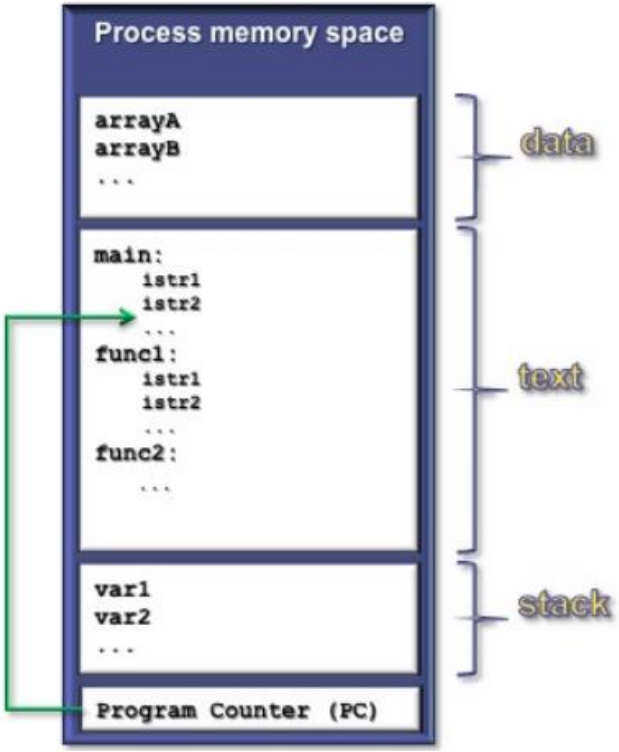
- Uniform Memory Access
  - Memory access time is uniform

UMA

Process and thread

A process is an instance of a computer program. **Information present in a process include:**

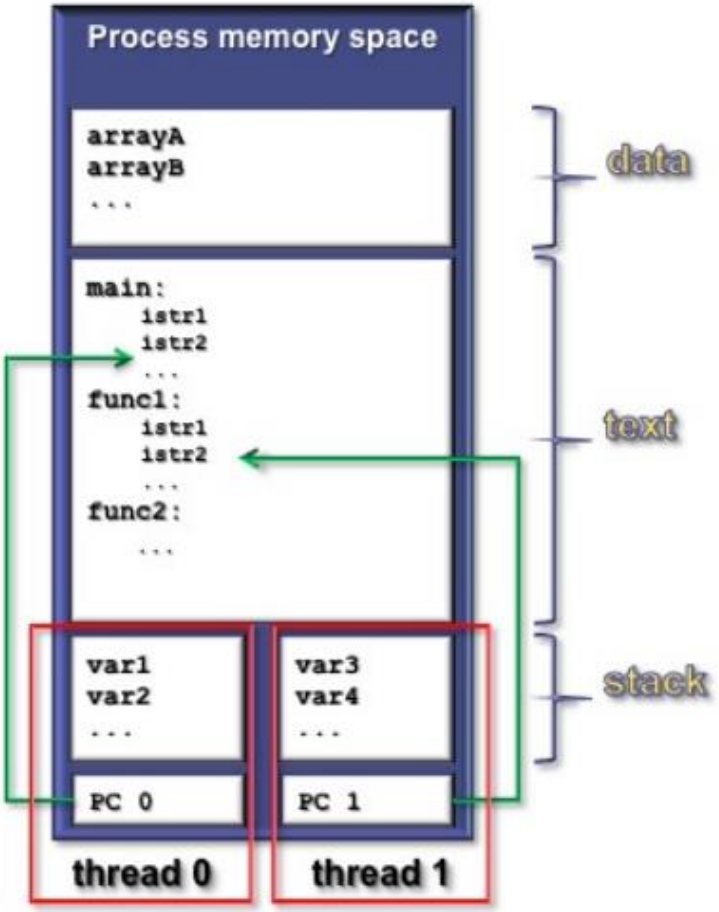
- **Text**
  - Machine code
- **Data**
  - Global variables
- **Stack**
  - Local variables
- **Program counter (PC)**
  - A pointer to the istruction to be executed



Multi threading environment

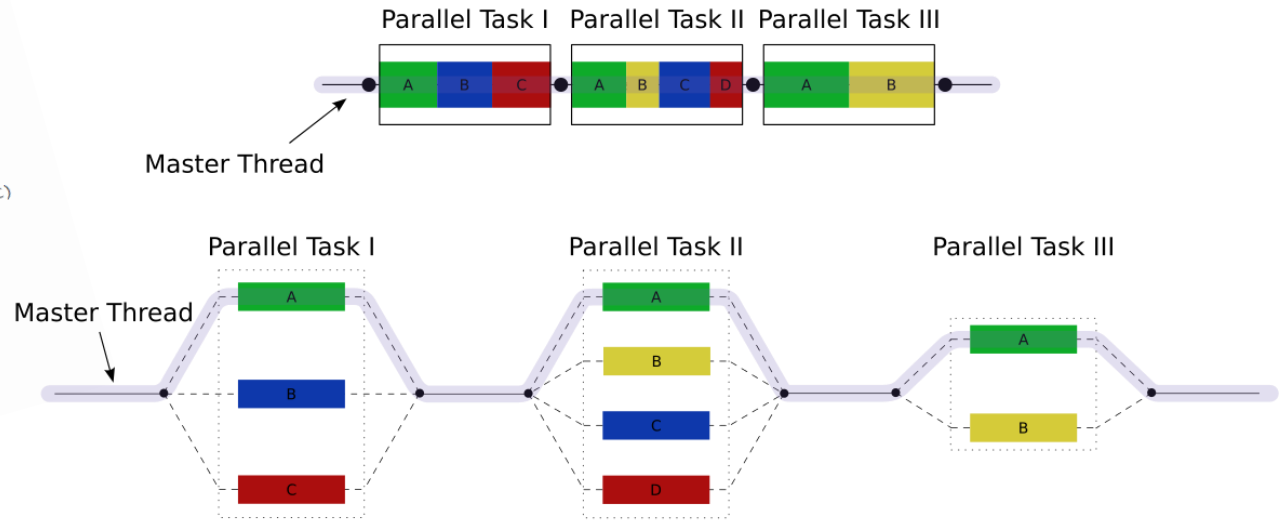
The process contains several concurrent execution flows (threads)

- Each thread has its own program counter (PC)
- Each thread has its own private stack **(variables local to the thread)**
- The instructions executed by a thread can access
  - the process global memory (data)
  - the thread local stack



OpenMP: the Fork & Join model

```
for (i= 0; i< N; i++)
  for (j= 0; j< N; j++)
  {
      A[i][j] = 2;
      B[i][j] = 2;
  }
gettimeofday(&tv1, &tz);
#pragma omp parallel for private(i,j,k) shared(A,B,C)
  for (i = 0; i < N; ++i) {
      for (j = 0; j < N; ++j) {
          for (k = 0; k < N; ++k) {
              C[i][j] += A[i][k] * B[k][j];
          }
      }
  }
```



The Fork & Join Model

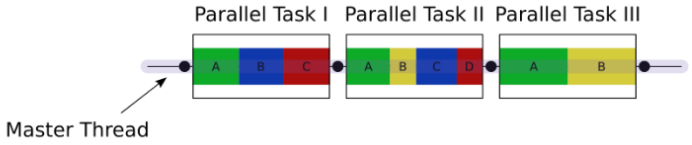
• Each OpenMP program begins to execute with a single thread (**Master thread**) that runs the program in serial mode (or sequential mode)

• At the beginning of a parallel region the master thread ***creates*** a team of threads composed of itself and a set of other threads (Thread ID, 0,1,2,3,4 etc…)

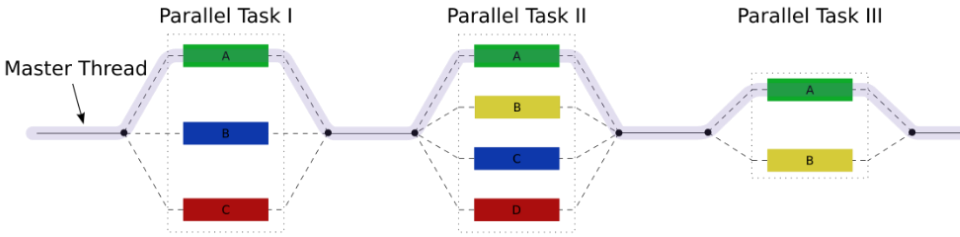
• The **thread team** runs in parallel the code contained in the parallel region (or parallel block or task) **(Single Program Multiple Data model)**

• At the end of the parallel region the thread team **ends the execution and only the master thread continues the execution of the (serial) program.**

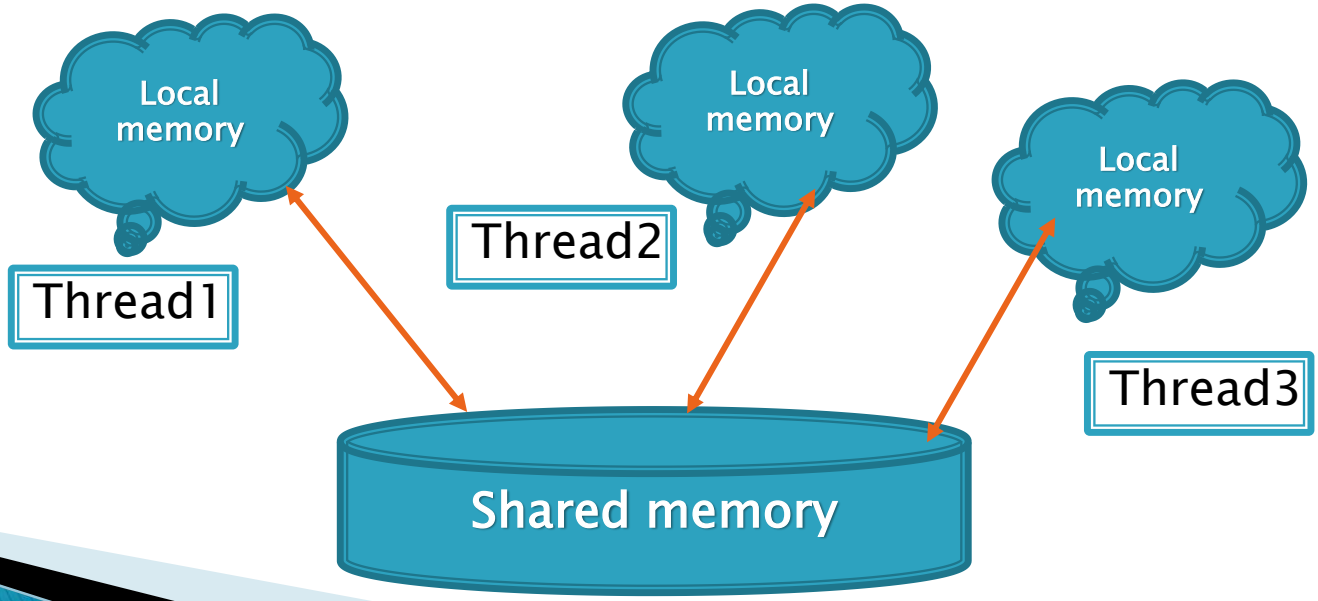
OpenMP: the MEMORY model



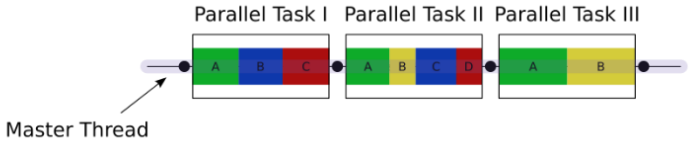
- **All threads have access to the same globally shared memory**
- **Data in private memory are only accessible by the thread owning this memory**
  - **No other thread sees the change(s)**

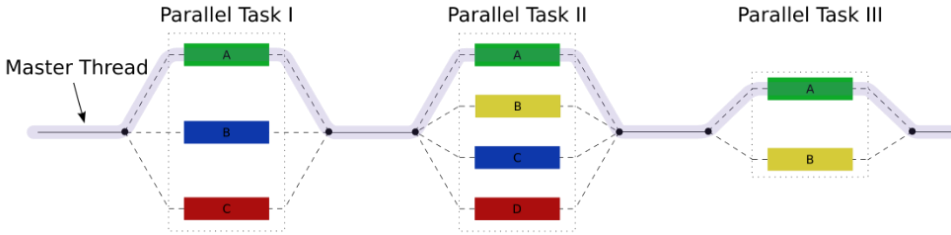


- Data transfer is through **shared memory** and is completely transparent to the application at run time !



OpenMP: the Directives





- syntax:
  - in C/C++: **#pragma omp directive**
  - in Fortran: **!$omp directive**
  - in Fortran (fixed format): **c$omp directive**

- Mark a block of a specific code;

- Specify to the compiler – ***from source code*** - how to run in parallel the code block.

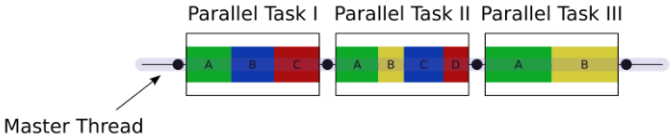
- **The serial code "coexists" with the parallel code within the same template (no different files)**

- **A serial compilation ignores the directives (conditional compilation process, precompiling)**

- A compilation with OpenMP support (**and –D_OPENMP flag**) takes them into account: **fork&join model**

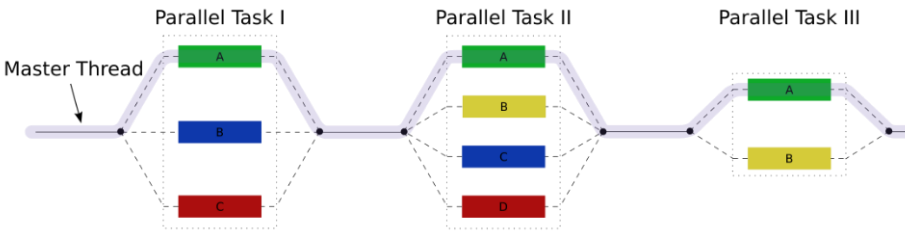
- **Variable handling**
  - What are shared among all threads (the default).
  - Which are private to each thread.
  - How to initialize the private ones.
  - What is the default.

OpenMP: the Directives



- **Execution control**
  - How many threads in the same team
  - How to distribute the work across threads

- ATTENTION: they may alter the code semantics
  - The code can be corrected in serial but not in parallel or vice versa (**race condition error**) ;



OpenMP – Flexible - VARIABLES

- **OMP_NUM_THREADS:** sets number of threads
- **OMP_STACKSIZE "size [B|K|M|G]":** size of the stack for threads
- **OMP_DYNAMIC {TRUE|FALSE}:** dynamic thread adjustment
- **OMP_SCHEDULE "schedule[,chunk]":** iteration scheduling scheme (static, guided
- **OMP_PROC_BIND {TRUE|FALSE}:** bound threads to processors
- **OMP_NESTED {TRUE|FALSE}:** nested parallelism

- To set them within a LINUX environment:
  - In csh/tcsh: **setenv OMP_NUM_THREADS 4**
  - In sh/bash: **export OMP_NUM_THREADS=4**

Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

OpenMP: run time functions

- Query/specify some specific feature or setting
  - **omp_get_thread_num():** get thread ID **(0 for master thread)**
  - **omp_get_num_threads():** get number of threads in the team
  - **omp_set_num_threads(int n):** set number of threads

- Allow you to manage fine-grained access (lock)
  - **omp_init_lock(lock_var):** initializes the OpenMP lock variable **lock_var** of type **omp_lock_t**
  - ... •

- Timing functions
  - **omp_get_wtime():** returns elapsed wallclock time
  - **omp_get_wtick():** returns timer precision

- Functions interface:
  - C/C++: #include
  - Fortran: use omp_lib (or include 'omp_lib.h')

Conditional compilation process

- To avoid dependency on OpenMP libraries you can use pre-processing directives
  - and the preprocessor macro _**OPENMP** predefined by the standard
  - C preprocessing directives can be used in Fortran too as well **!$** in free form and old style fixed form ***$** and **c$**

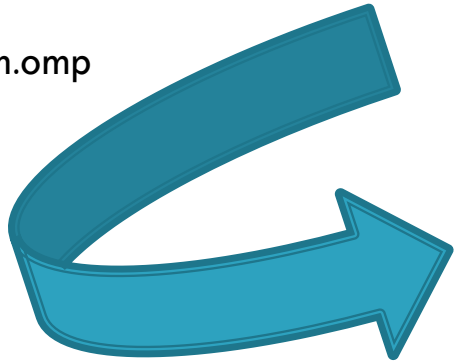
C/C++

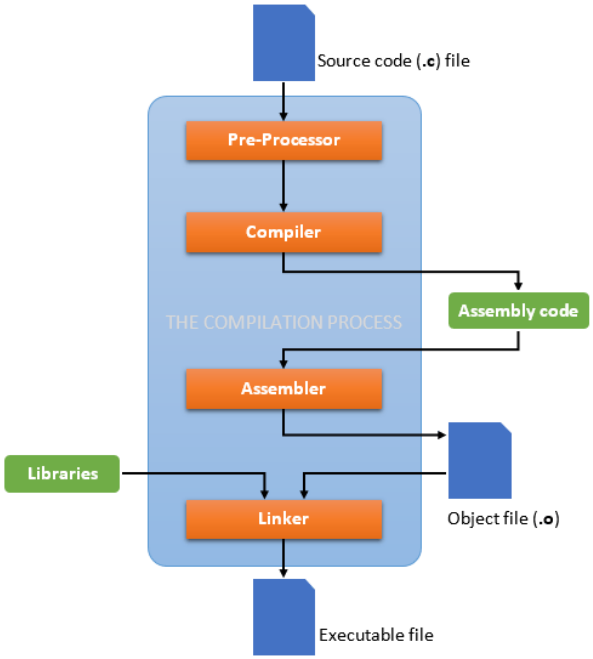
```
#ifdef _OPENMP
printf("Compiled with OpenMP support:%d",_OPENMP);
#else
printf("Compiled for serial execution.");
#endif
```

Fortran

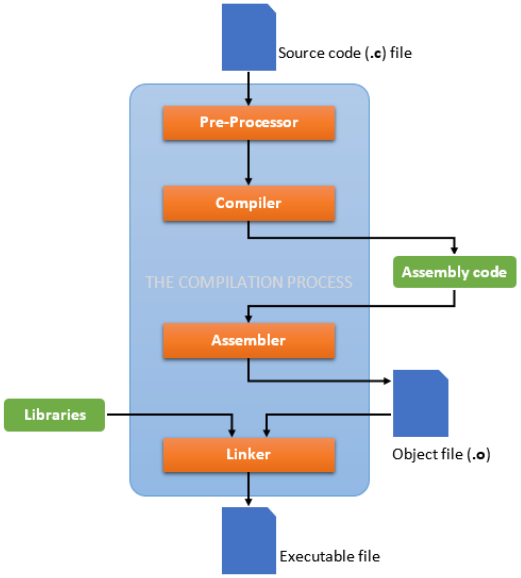
```
!$ print *,"Compiled with OpenMP support",_OPENMP
```

gcc -fopenmp -D_OPENMP -O stream.c -o stream.omp
export OMP_NUM_THREADS=${nProc}
./stream.omp (then we run the executable)





Conditional compilation



Compilazione di un programma suddiviso su più file

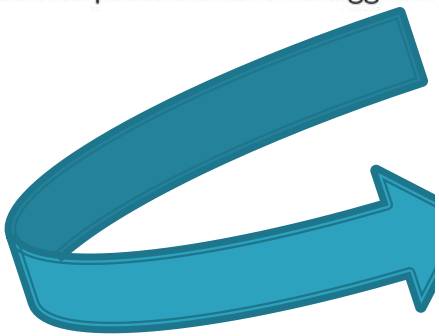
Nell'esempio precedente occorre dare i seguenti comandi di compilazione.

1. `gcc -c main.c`
   Crea il file oggetto `main.o`
2. `gcc -c calc.c`
   Crea il file oggetto `calc.o`
3. `gcc main.o calc.o`
   Chiama il linker sui fule oggetto `main.o` e `calc.o` e crea il file eseguibile.
   Il linker associa le chiamate alle funzioni `perimetro()` e `area()` contenute in `main.o` al codice oggetto delle corrispondenti funzioni in `calc.o`.

Si può fare in un unico passo:

```
gcc main.c calc.c
```

In questo caso non vengono generati esplicitamente i file oggetto `main.o` e `calc.o`.



Vantaggi

La modifica di una funzione richiede solamente la ricompilazione del file in cui la funzione è definita

Esempio 3

Se si modifica un messaggio nella funzione `main()`, occorre rigenerare solamente il file `main.o`, ma non `calc.o`

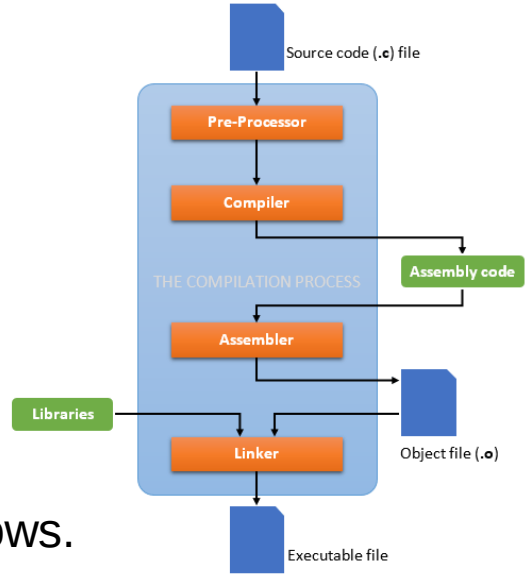
```
gcc -c main.c
gcc main.o calc.o
```

Oppure, in un unico passo:

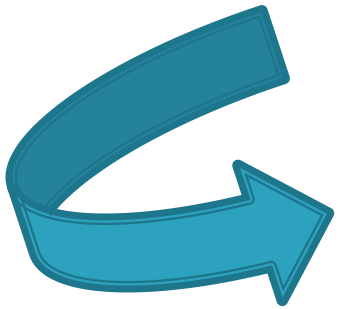
```
gcc main.c calc.o
```

In questo caso non viene generato esplicitamente `main.o`.

Conditional compilation

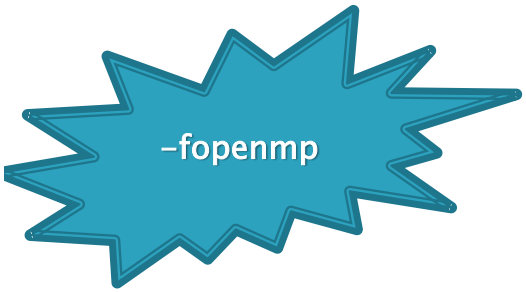


- The compilers that support OpenMP interpret the directives only if they are invoked with a compiler option (**switch**)
  - GNU: **-fopenmp** for Linux, Solaris, AIX, MacOSX, Windows.
  - IBM: **-qsmp=omp** for Windows, AIX and Linux.
  - Sun: **-xopenmp** for Solaris and Linux.
  - Intel: **-openmp** on Linux or Mac, or -Qopenmp on Windows
  - PGI (Portland Compilers): -**mp** switch
- Most compilers emit useful information enabling extra warning or report options



```
gcc -fopenmp -D_OPENMP -O stream.c -o stream.omp
export OMP_NUM_THREADS=${nProc}
./stream.omp
```

GNU



Parallel construct with OpenMP

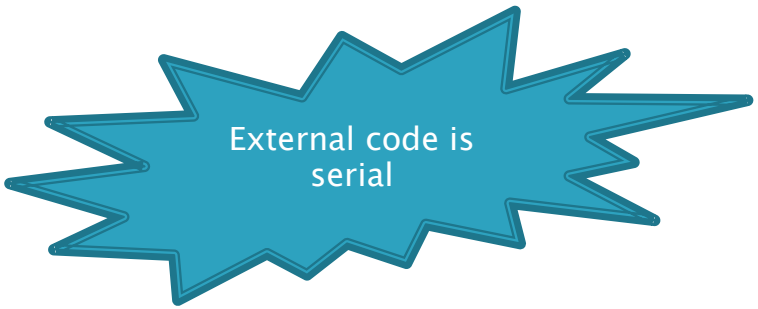
- **It creates a parallel region or section or block**
  - A parallel region is a block of code executed by all threads in the team **(in fact it creates the thread team).**

C/C++

```
#pragma omp parallel
{
// some code to execute in parallel
} // end of the parallel region (implied barrier)
```

Fortran

```
!$omp parallel
!  some code to execute in parallel
!$omp end parallel
```



omp_set_num_threads(4);

C

```
#include <stdi
int main()
{
#pragma omp parallel  num_threads(4)
   {
      printf("Hello world!\n");
   }
   return 0;
}
```

Fortran

```
Program Hello
!$omp parallel
      print *, "Hello world!"
!$omp end parallel
end program Hello
```

How is OpenMP typically used?

- **OpenMP is usually used to parallelize loops:**
  - **Find your most time consuming loops !!!!**
  - Split them up between threads with OpenMP directives

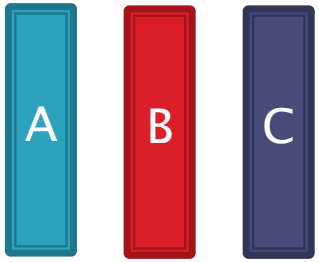
Sequential Program

```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

A()=B()+k*C()



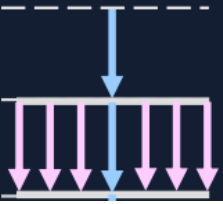
OpenMP Fork-and-Join model

```
printf("program begin\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];
```

Serial

Parallel



How is OpenMP typically used?

Single Program Multiple Data (SPMD)

```
printf("program begin\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("program done\n");
```

Serial

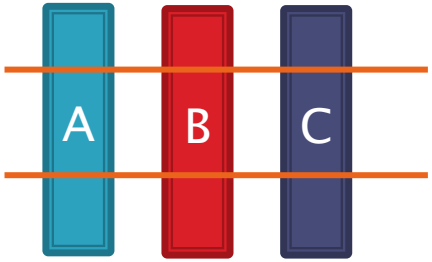
Parallel

Serial

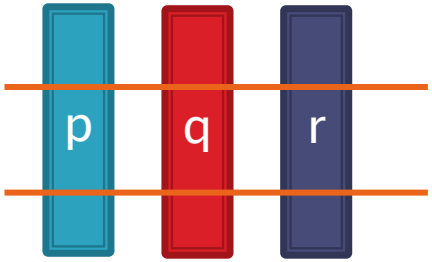
Parallel

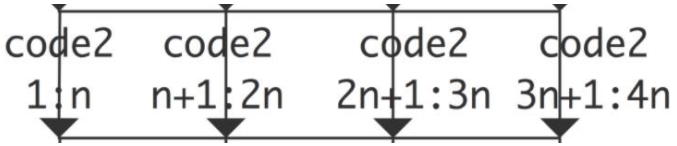
Serial

A()=B()+C()



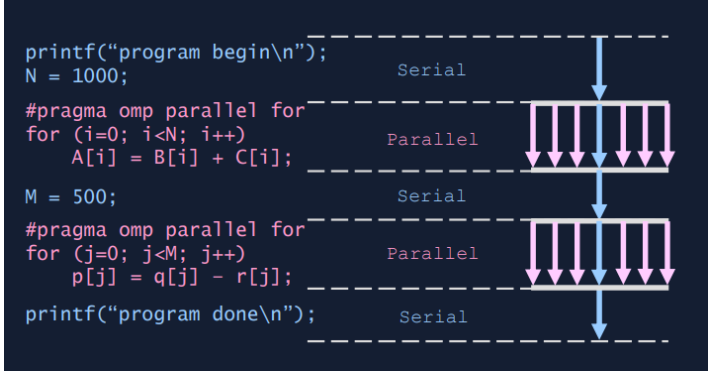
p()=q()-r()





the loop iterations are spread over four threads.

Loop Schedules



Usually you will have many more iterations in a loop than there are threads.
Thus, there are several ways you can assign your loop iterations to the threads.
OpenMP lets you specify this with the schedule **clause**.

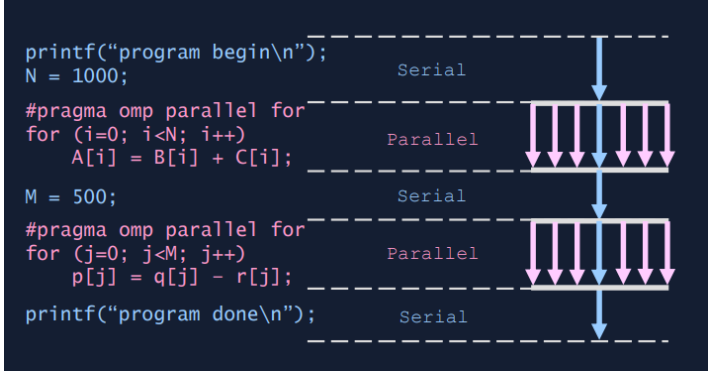
#pragma omp for schedule(....)

The first distinction we now have to make is between static and dynamic schedules.

With static schedules, the iterations are assigned purely based on the number of iterations and the number of threads (and the **chunk** parameter).

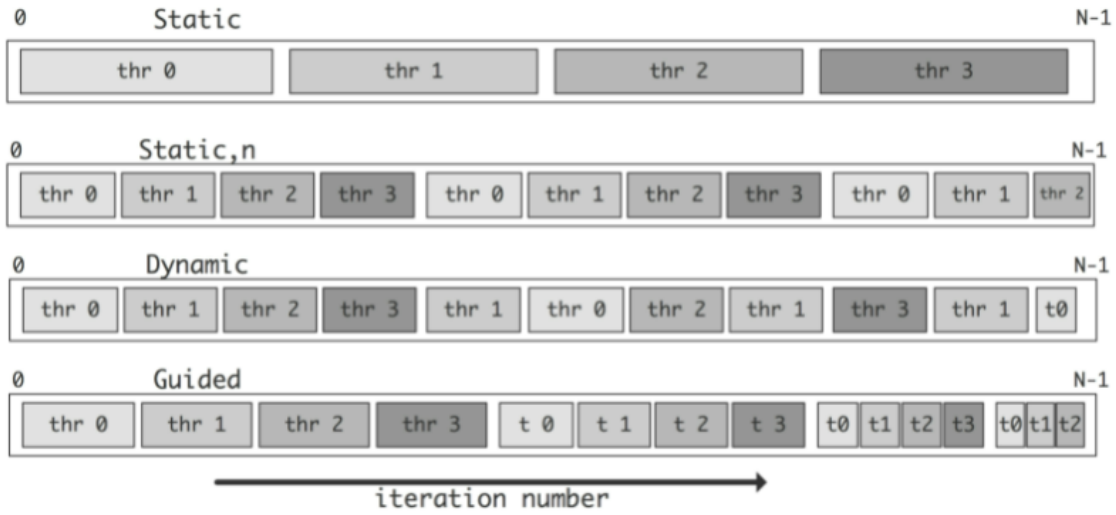
In dynamic schedules, on the other hand, iterations are assigned to threads that are unoccupied. Dynamic schedules are a good idea if iterations take an unpredictable amount of time, so that *load balancing* is needed.

Loop Schedules



The default static schedule is to assign one consecutive block of iterations to each thread. If you want different sized blocks you can define a

#pragma omp for schedule(static[,chunk])



With static scheduling, the compiler will determine the assignment of loop iterations to the threads at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.

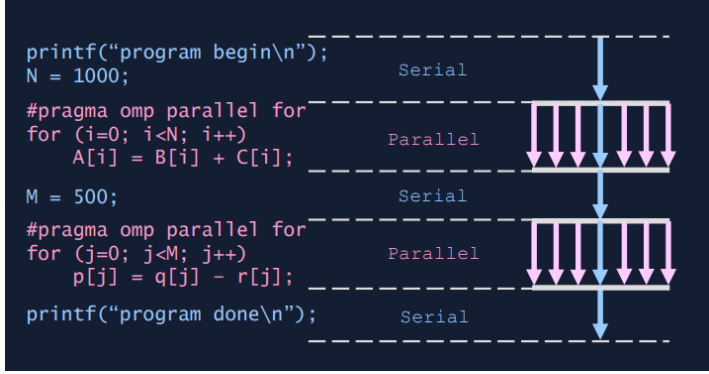
The choice of a chunk size is often a balance between the low overhead of having only a few chunks, versus the load balancing effect of having smaller chunks.

Why is a chunk size of 1 typically a bad idea?

think about cache lines…

In dynamic scheduling OpenMP will put blocks of iterations (the default chunk size is 1) in a task queue, and the threads take one of these tasks whenever they are finished with the previous.

Loop Schedules



Critical Construct

```
sum = 0;
#pragma omp parallel private (lsum)
{
   lsum = 0;
   #pragma omp for
   for (i=0; i<N; i++) {
      lsum = lsum + A[i];
   }
   #pragma omp critical
   { sum += lsum; }
}
```

Threads wait their turn; only one thread at a time executes the critical section

Calculate the sum of values in an array !

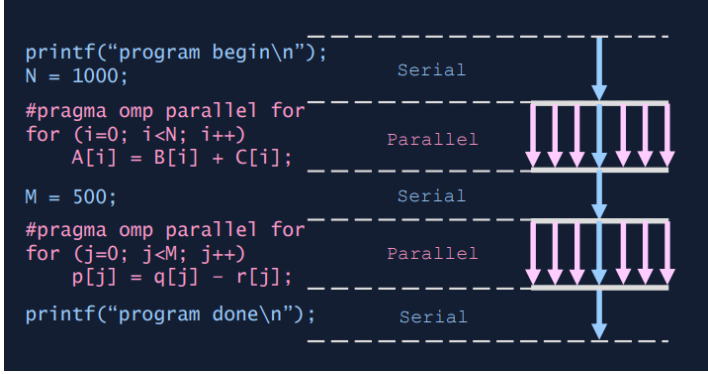
Global vs private variables

In this example we introduce **a critical section….**

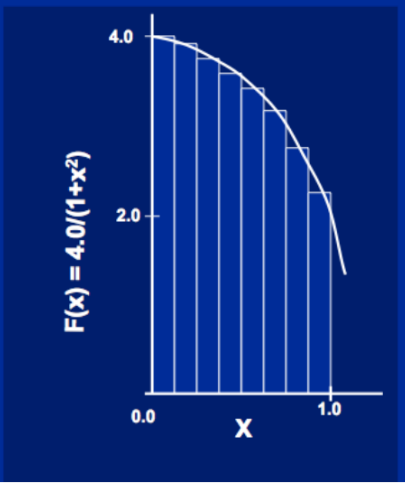
Shared variable

```
sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<N; i++)
{
   sum = sum + A[i];
}
```

Loop Schedules



Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Sequential or serial computation of pi (C programming language)

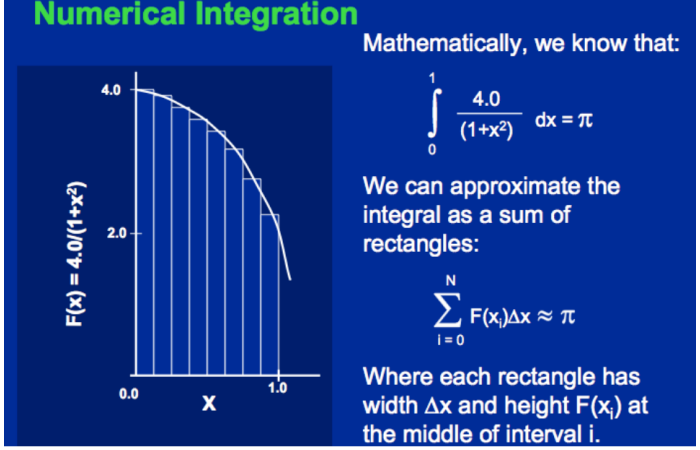
```
#include <stdio.h>       /* Serial Code */
static long num_steps = 100000;
double step;
void main () {
    int i;
    double x, pi, sum = 0.0;
   step = 1.0/(double)num_steps;
    for (i = 1; i <= num_steps; i++) {
      x = (i - 0.5) * step;
       sum = sum + 4.0 / (1.0 + x*x);
   }
   pi = sum / num_steps;
   printf ("pi = %6.12f\n", pi);
}
```

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Loop Paralellization

Parallel vs serial computation of pi (C programming language)

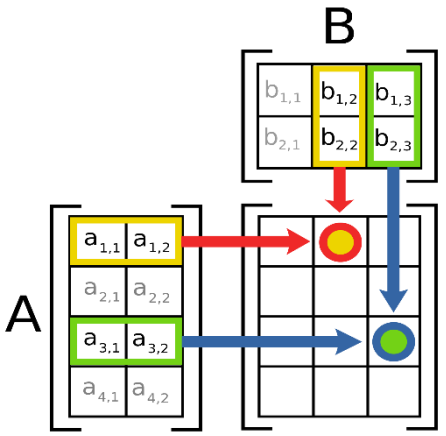



```
#include <omp.h>
#include <stdio.h>
/static long num_steps = 100000;
double step;
void main ()
{   int i;     double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
    for (i=1; i<= num_steps; i++){
         x = (i-0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
    }
    pi = sum / num_steps;
 printf ("pi = %6.8f\n", pi);
}
```

```
#include <stdio.h>       /* Serial Code
static long num_steps = 100000;
double step;
void main () {
     int i;
     double x, pi, sum = 0.0;
    step = 1.0/(double)num_steps;
     for (i = 1; i <= num_steps; i++) {
       x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0 + x*x);
    }
    pi = sum / num_steps;
    printf ("pi = %6.12f\n", pi);
}
```

Loop Paralellization

Matrix Matrix multiplication (C programming language)




```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 1000

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i,j,k;
```

```
omp_set_num_threads(omp_get_num_procs());
for (i= 0; i< N; i++)
    for (j= 0; j< N; j++)
    {
        A[i][j] = 2;
        B[i][j] = 2;
    }
gettimeofday(&tv1, &tz);
#pragma omp parallel for private(i,j,k) shared(A,B,C)
for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
        for (k = 0; k < N; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

No critical section required

Next Steps....

- OpenMP vs OpenMPI applications with code examples under GNU Suite
- Submission process in batch (live experience)
- Hybrid-Parallel computations with CUDA
- Nvidia-smi tool and Gpu Allocation with SLURM
- GpuArrays in Matlab: Benchmarks
- HPC-OpenDesk facility

Thank you for your attention !