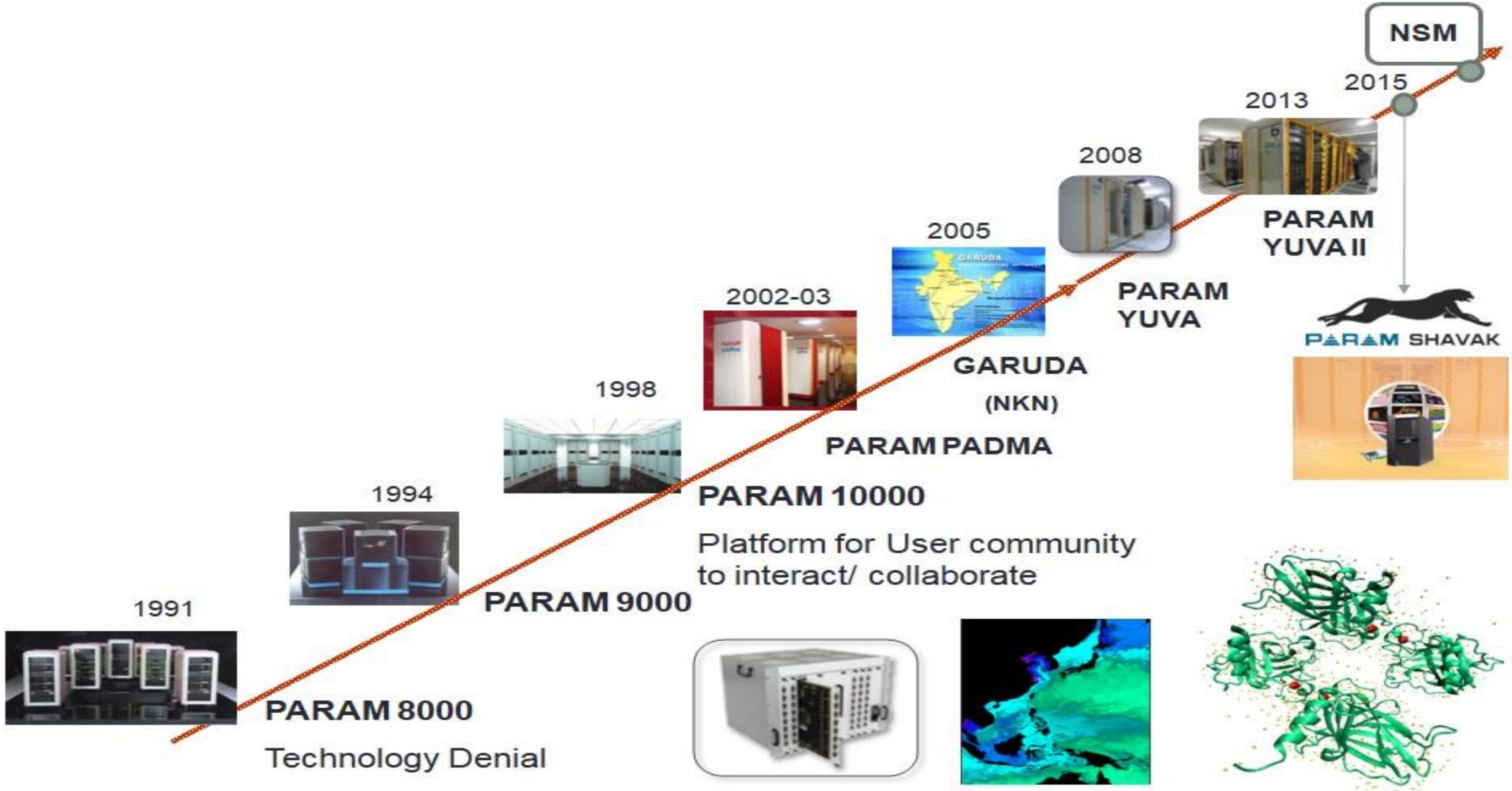
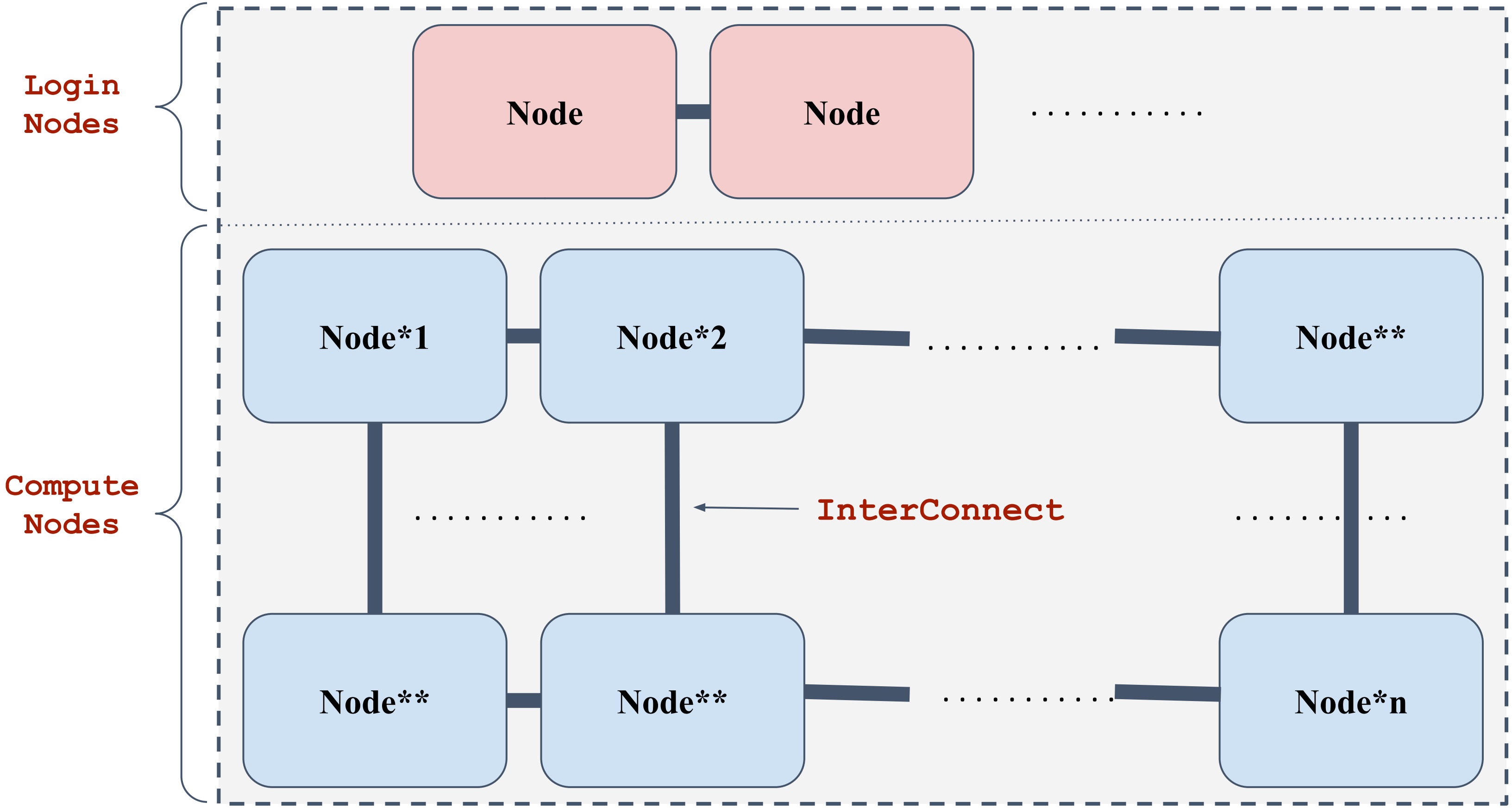
OpenMP

(Shared memory Parallel programming Model)

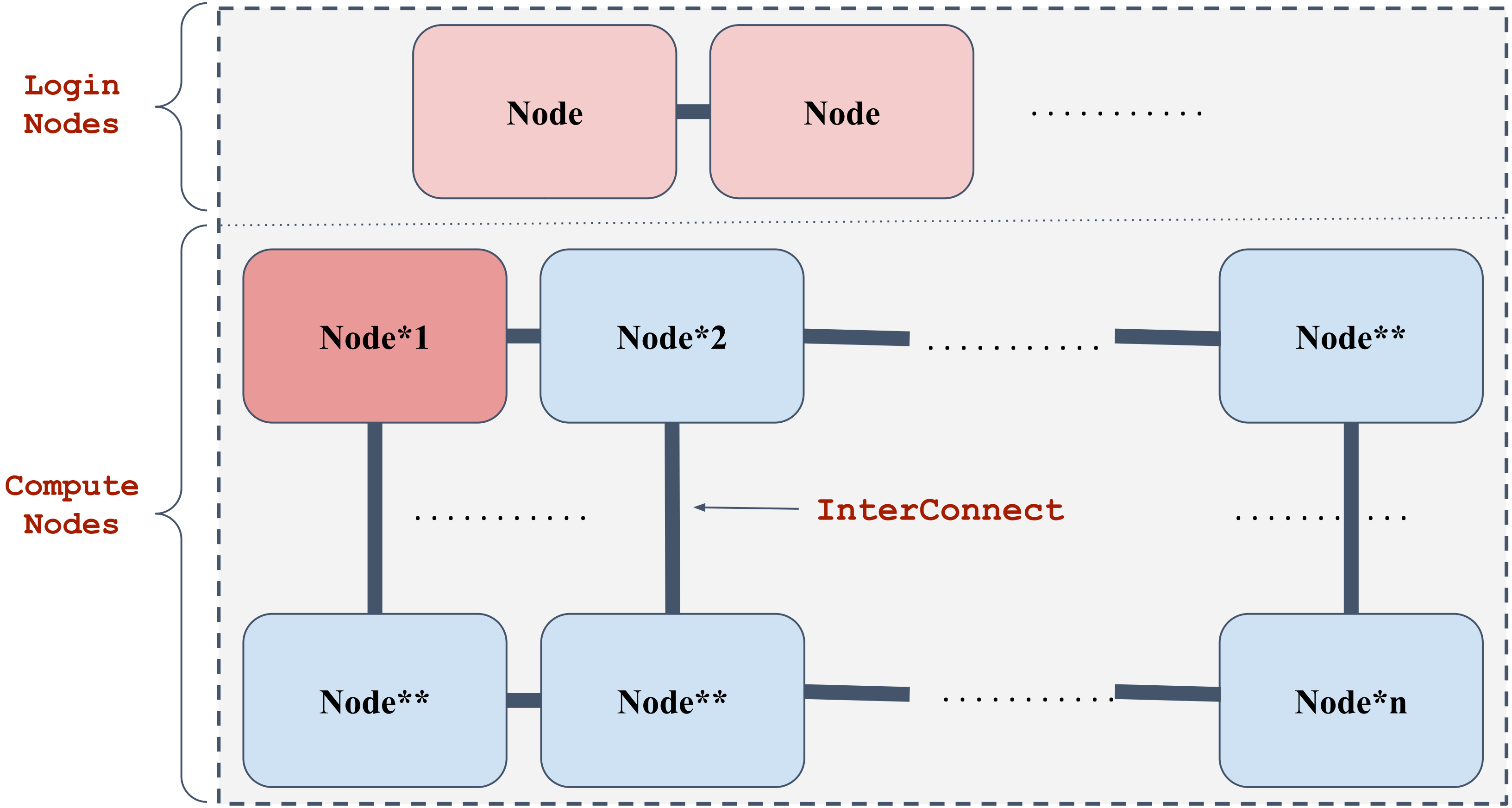
Mr. Om Jadhav
Senior Technical officer,
HPC Tech CDAC Pune


NSM
2015
2013
PARAM
YUVA II
2008
PARAM
YUVA
PARAM SHAVAK
2005
GARUDA
GARUDA
(NKN)
2002-03
PARAM PADMA
1998
PARAM 10000
Platform for User community
to interact/ collaborate
1994
PARAM 9000
1991
PARAM 8000
Technology Denial

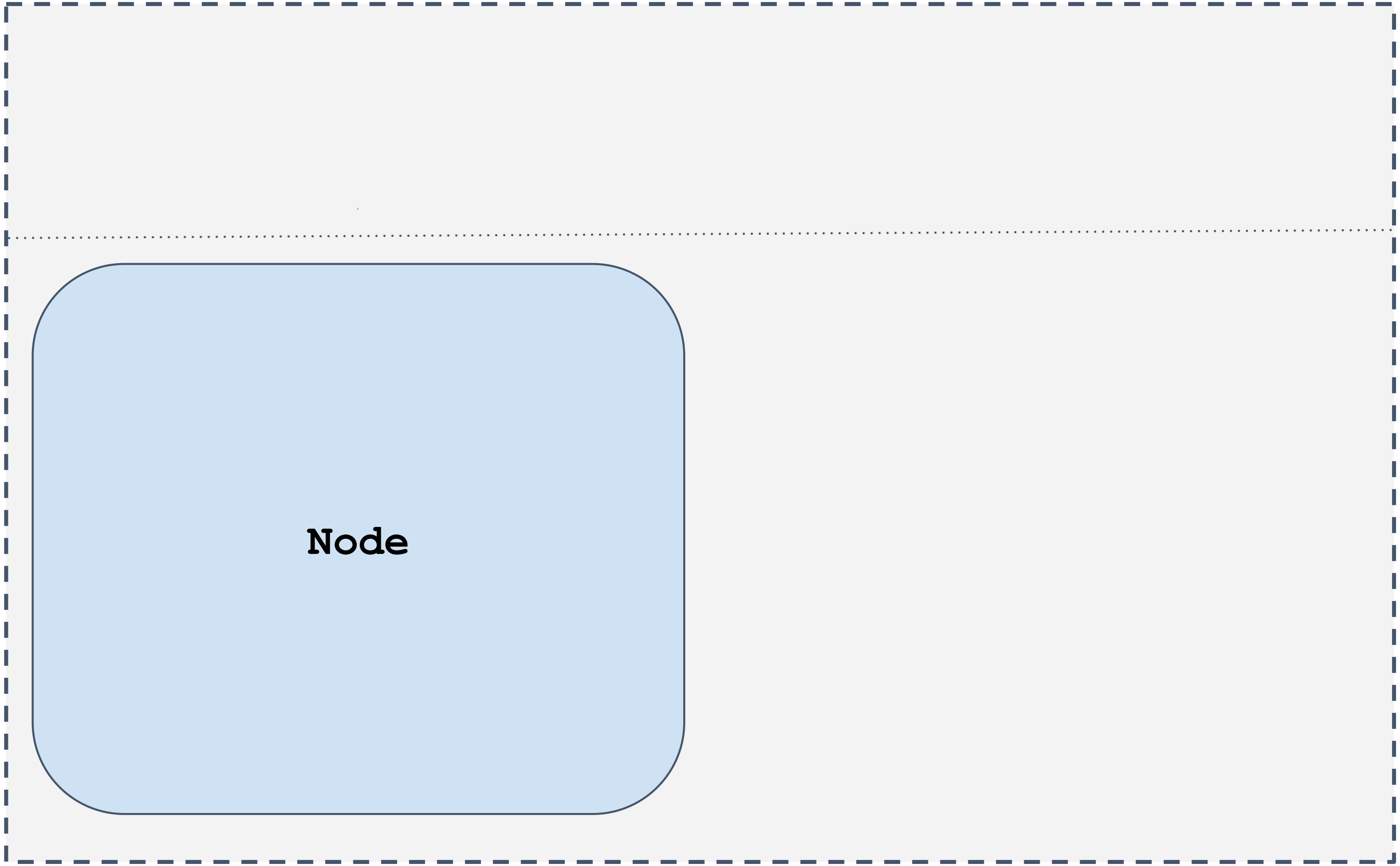
HPC Cluster


Login
Nodes
Node
Node
Compute
Nodes
Node*1
Node*2
Node**
InterConnect
Node**
Node**
Node*n

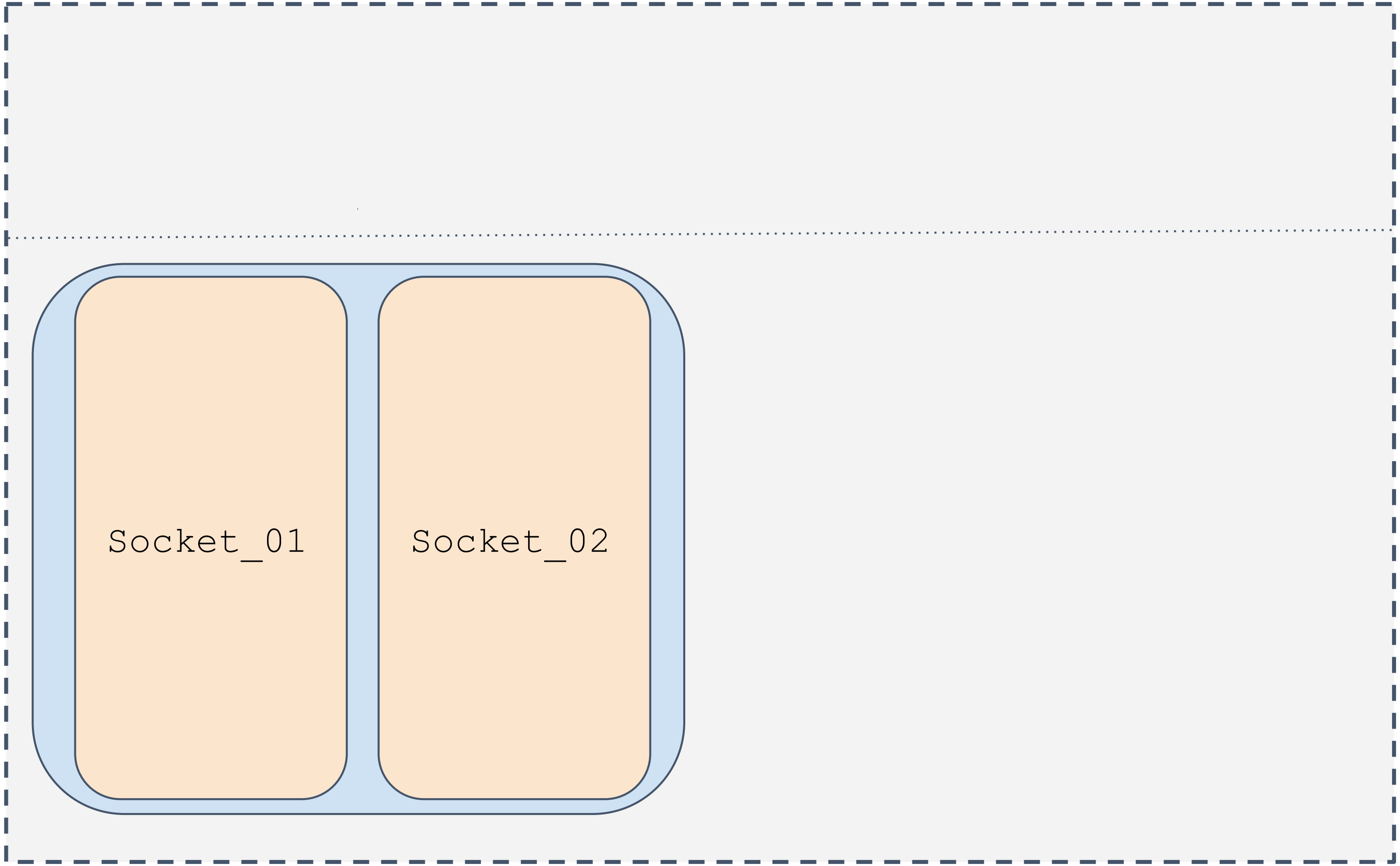
HPC Cluster


Login
Nodes
Node
Node
Compute
Nodes
Node*1
Node*2
Node**
InterConnect
Node**
Node**
Node*n

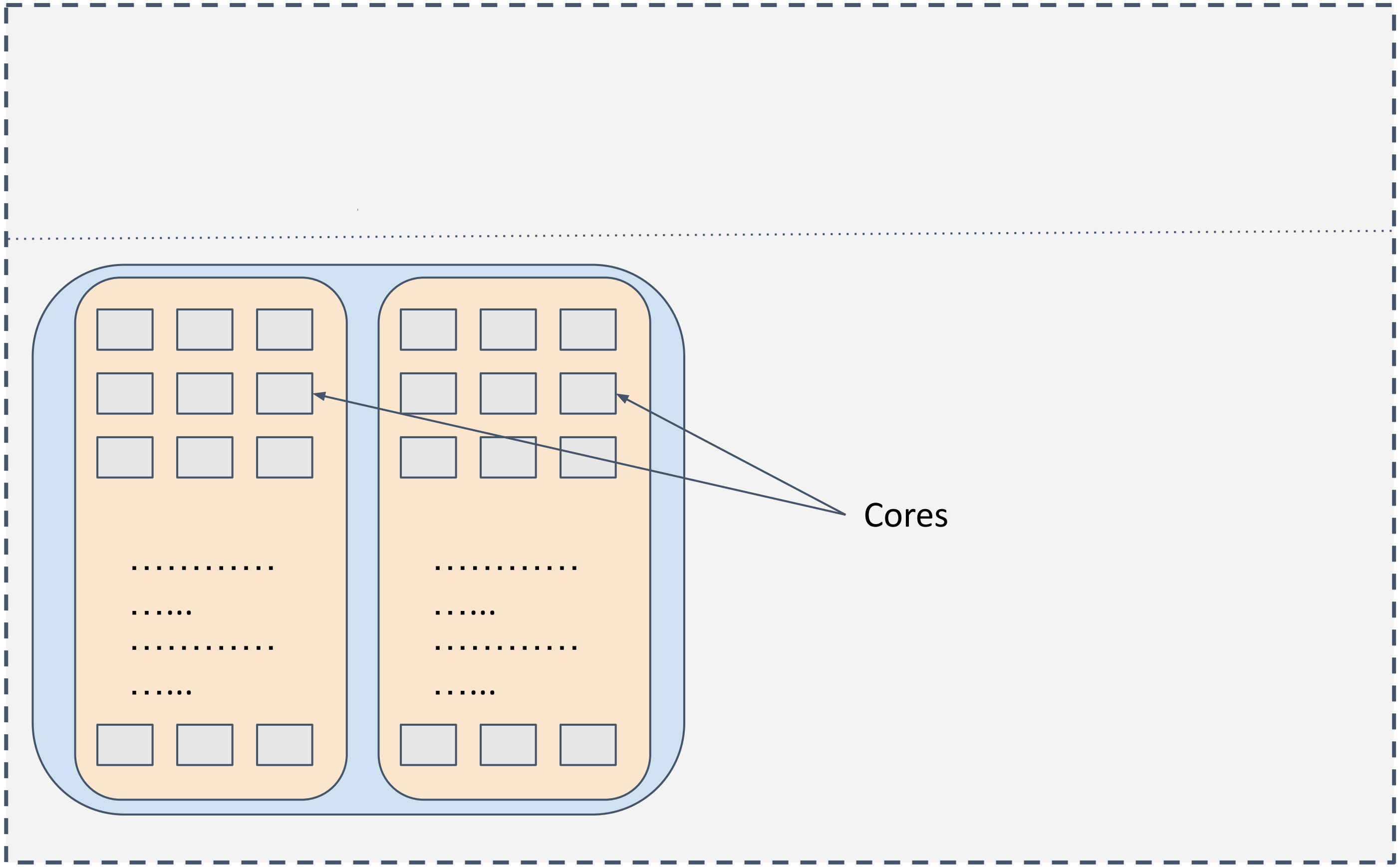
HPC Cluster



HPC Cluster



HPC Cluster

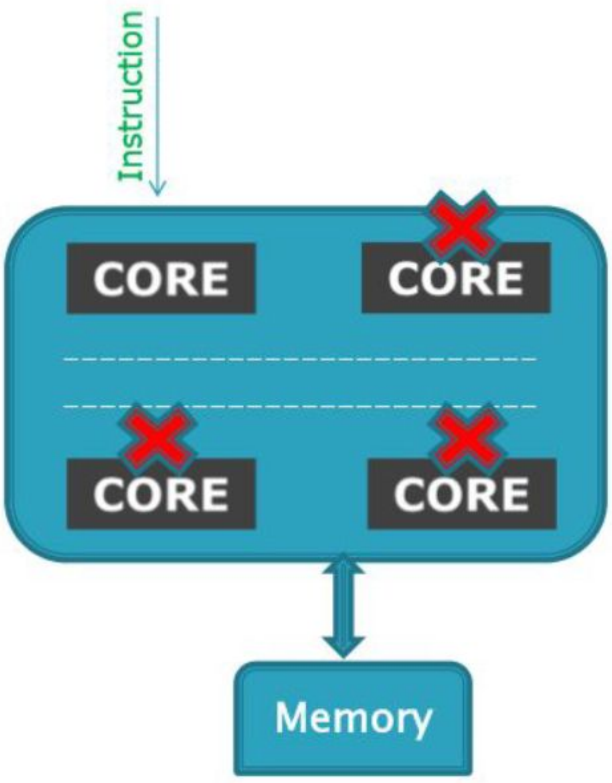


Program Execution ?

- When you run sequential program
  - Instructions executed in serial
  - Other cores are idle

- Waste of available resource...

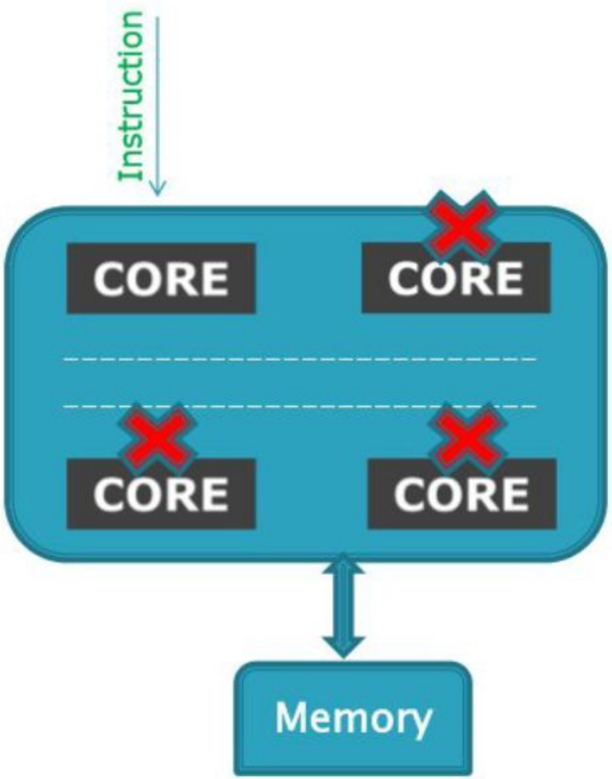
- We want all cores to be used to execute program.
  - How ?



Program Execution ?

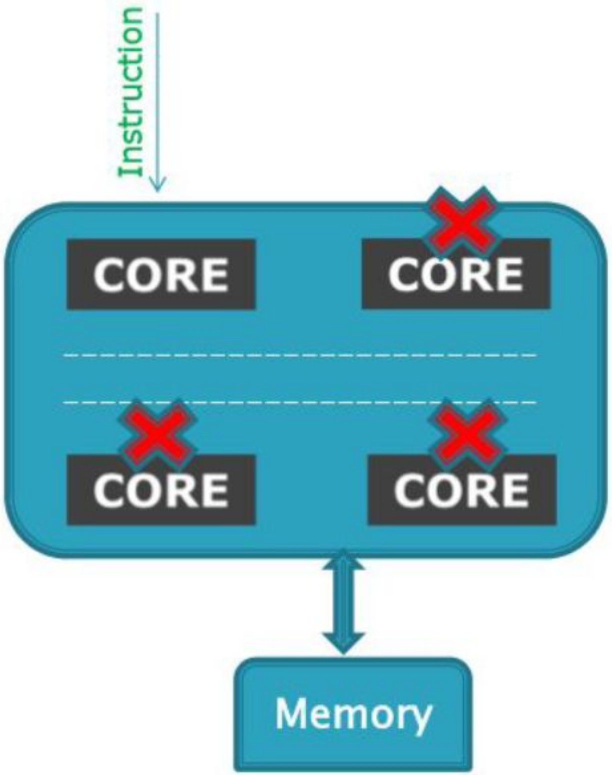
- When you run sequential program
  - Instructions executed in serial
  - Other cores are idle
- Waste of available resource...
- We want all cores to be used to execute program.
  - How ?

Parallel Programming Models



Program Execution ?

- When you run sequential program
  - Instructions executed in serial
  - Other cores are idle
- Waste of available resource...
- We want all cores to be used to execute program.
  - How ?

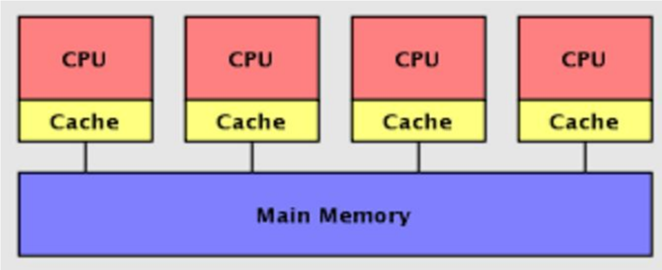


Parallel Programming Models

- **Shared Memory**
- **Distributed Memory**

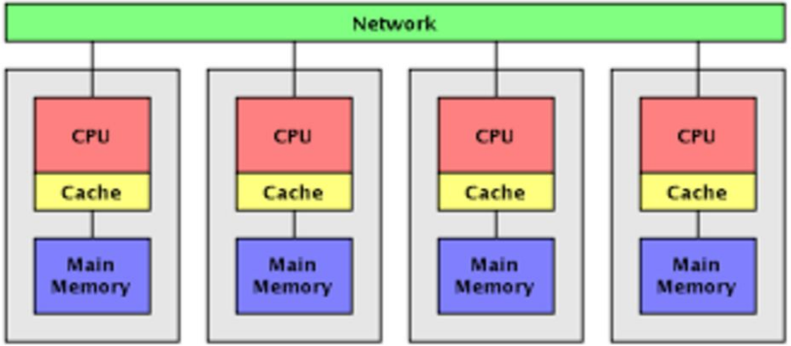
Parallel Programming Models

❑ Shared-memory Model



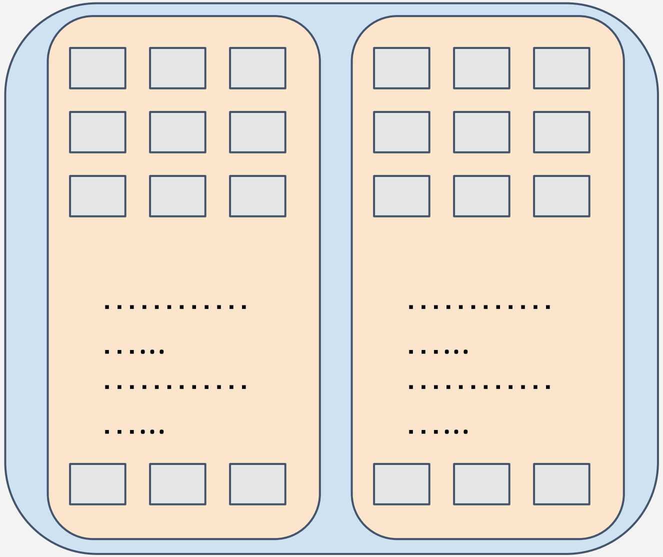
- ❖ OpenMP

❑ Distributed-memory Model



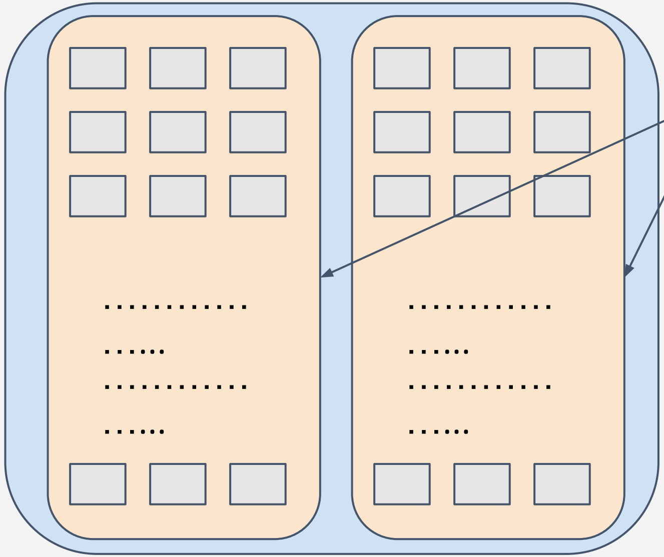
- ❖ MPI - Message Passing Interface

Shared Memory Programming Models



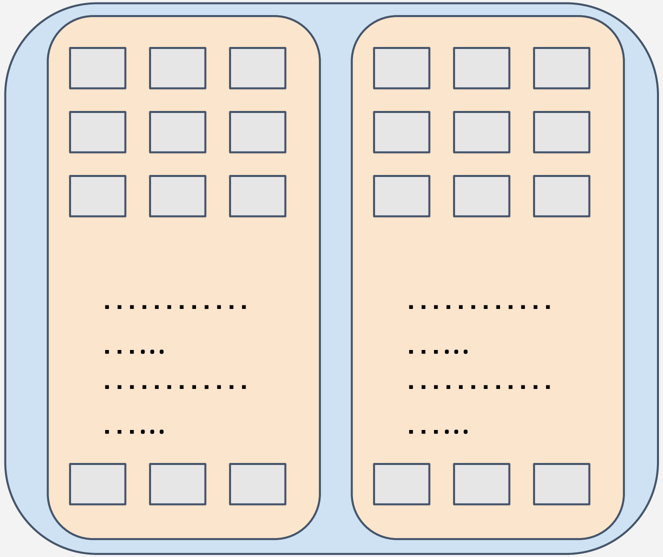
Shared Memory System

Shared Memory Programming Models

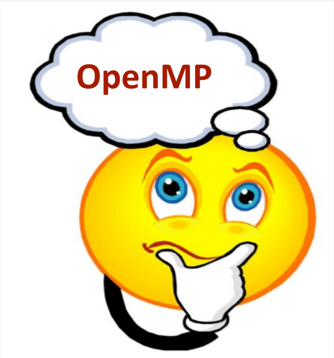


How you will utilize all the cores efficiently ?

Shared Memory Programming Models



How you will utilize all the cores efficiently ?



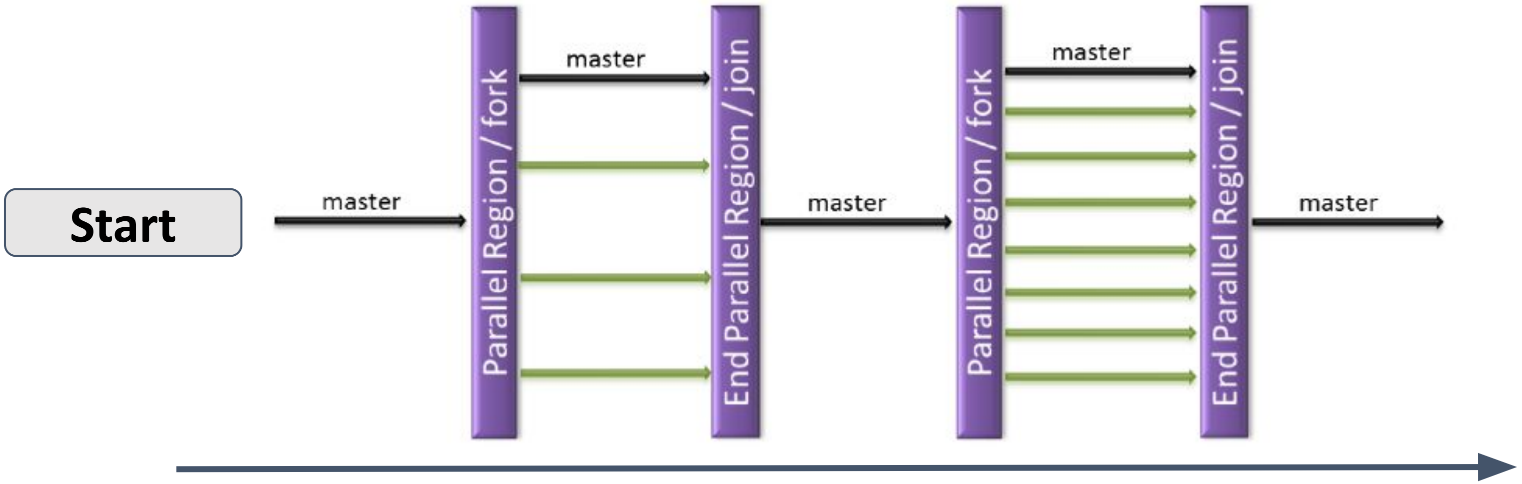
OpenMP

- Portable shared memory programming

- Easy to learn
  - OpenMP specific commands in source codes are processed by the compiler
  - OpenMP functionality is switched on by a compiler specific option

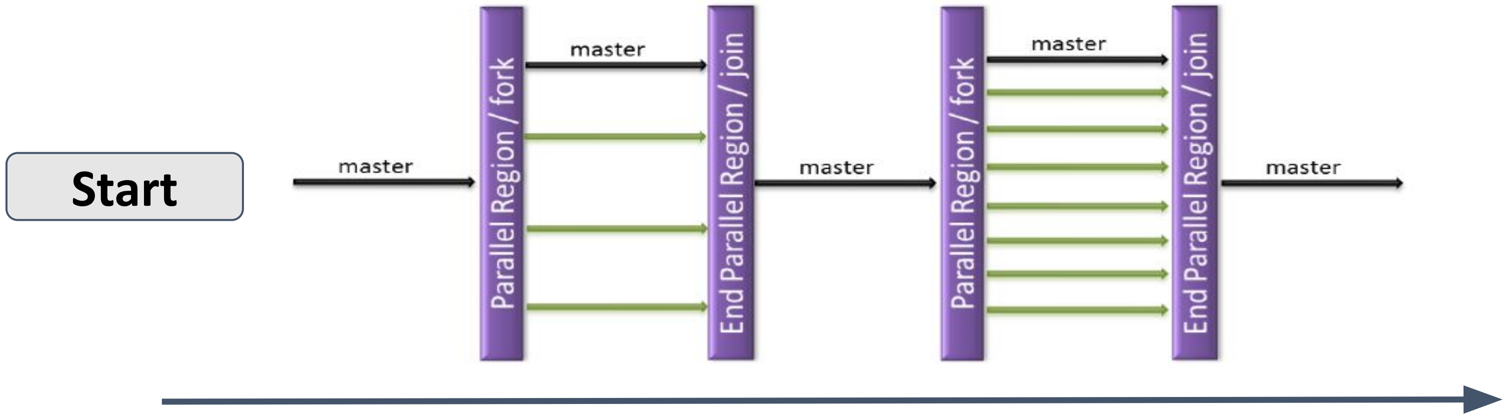
- Parallelization is fully controlled by programmer
  - Directives for Fortran 77/90 and pragmas for C/C++
  - Run-time library routines
  - Environment variables

OpenMP : Fork-Join Programming model



- ➢ Master Thread (MT) executes sequentially the program
- ➢ A team of threads is being generated when MT encounters a Parallel Region (PR)
- ➢ All but the MT are being destroyed at the end of a PR

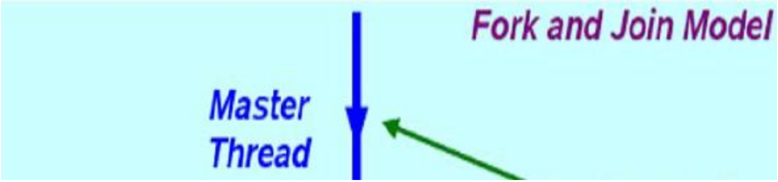
OpenMP : Fork-Join Programming model



➢ **Threads**

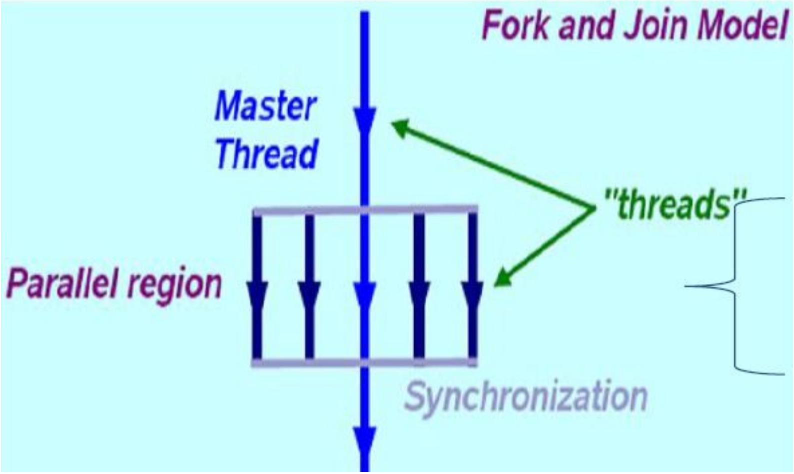
- Threads are numbered from 0 to (n - 1), n is the number of threads
- omp_get_num_threads gives the number of available threads
- omp_get_thread_num tells the thread, its number
- A single program with several threads is able to handle several tasks concurrently
- Each thread has its own **stack and registers**

OpenMP : Execution model



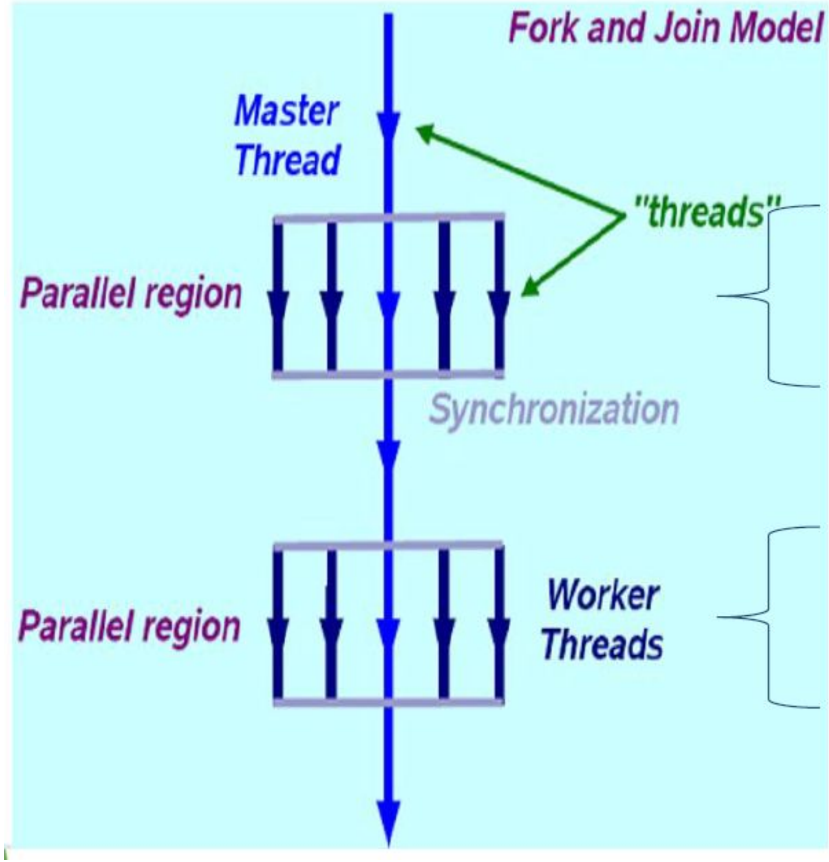

```
main (..)
{


#pragma omp parallel
{
      ......
      ......
}


#pragma omp parallel
{
      ......
      ......
}


}
```

OpenMP : Execution model




```
main (..)
{


#pragma omp parallel
{
      ......
      ......
}


#pragma omp parallel
{
      ......
      ......
}


}
```

OpenMP : Execution model




```
main (..)
{

#pragma omp parallel
{
    ......
    ......
}

#pragma omp parallel
{
    ......
    ......
}

}
```

Sample Program

```
#include <stdio.h>

int main(int argc, char* argv[])
{

    printf("Hello, Om ! \n");


}
```

Sample Program

```c
#include <stdio.h>

int main(int argc, char* argv[])
{

    printf("Hello, Om ! \n");


}
```

```
$ gcc hello_serial.c -o hello_serial
```

Sample Program

```
#include <stdio.h>

int main(int argc, char* argv[])
{

    printf("Hello, Om ! \n");


}
```

```
$ gcc hello_serial.c -o hello_serial
```

```
$ ./hello_serial

Hello, Om
```

Sample Program

```
#include <stdio.h>

int main(int argc, char* argv[])
{

    printf("Hello, Om ! \n");


}
```

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
{
     #pragma omp parallel
      printf("Hello Om, I am thread =
%d\n", omp_get_thread_num()) ;

}
```

```
$ gcc hello_serial.c -o hello_serial
```

```
$ ./hello_serial

Hello, Om
```

Sample Program

```c
#include <stdio.h>

int main(int argc, char* argv[])
{

    printf("Hello, Om ! \n");


}
```

```c
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
{
     #pragma omp parallel
      printf("Hello Om, I am thread = 
%d\n", omp_get_thread_num()) ;

}
```

$ gcc hello_serial.c -o hello_serial

$ gcc **-fopenmp** hello_parallel.c -o hello_parallel

$ ./hello_serial

Hello, Om

Sample Program

```
#include <stdio.h>

int main(int argc, char* argv[])
{

    printf("Hello, Om ! \n");


}
```

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
{
     #pragma omp parallel
      printf("Hello Om, I am thread =
%d\n", omp_get_thread_num()) ;

}
```

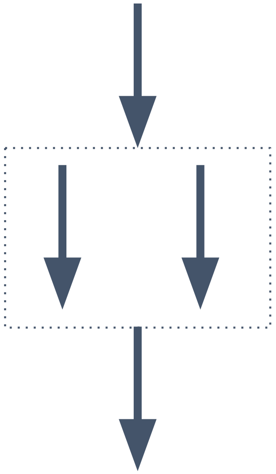
```
$ gcc hello_serial.c -o hello_serial
```

```
$ gcc -fopenmp hello_parallel.c -o hello_parallel
```

```
$ ./hello_serial

Hello, Om
```

```
$ export OMP_NUM_THREADS=2
$ ./hello_parallel

Hello Om, I am thread = 0
Hello Om, I am thread = 1
```

Sample Program

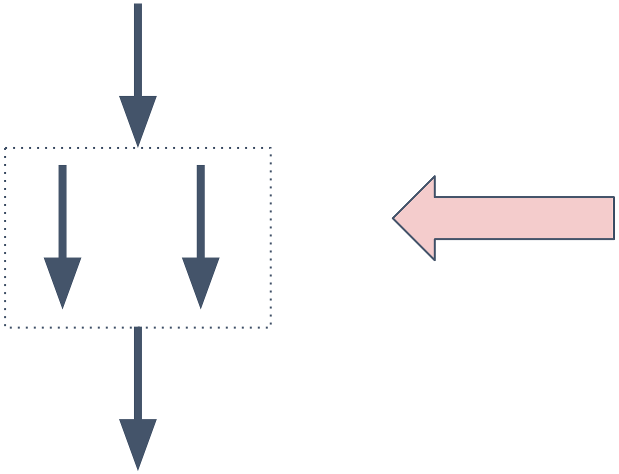


```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
{
    #pragma omp parallel
     printf("Hello Om, I am thread =
%d\n", omp_get_thread_num()) ;

}
```

```
$ gcc -fopenmp hello_parallel.c -o hello_parallel
```

```
$ export OMP_NUM_THREADS=2
$ ./hello_parallel

Hello Om, I am thread = 0
Hello Om, I am thread = 1
```

Sample Program



```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
{
    #pragma omp parallel
     printf("Hello Om, I am thread = %d\n", omp_get_thread_num()) ;

}
```

```
$ gcc -fopenmp hello_parallel.c -o hello_parallel
```

```
$ export OMP_NUM_THREADS=2
$ ./hello_parallel

Hello Om, I am thread = 0
Hello Om, I am thread = 1
```

Threads

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
{
     #pragma omp parallel
      printf("Hello Om, I am thread = %d\n", omp_get_thread_num()) ;

}
```

Method 1 : Environment Variable

OMP_NUM_THREADS

```
$ export OMP_NUM_THREADS=2
$ ./hello_parallel

Hello Om, I am thread = 0
Hello Om, I am thread = 1
```

Threads

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
{
    omp_set_num_threads(4) ;
    #pragma omp parallel
     printf("Hello Om, I am thread = %d\n", omp_get_thread_num()) ;
}
```

Method 2 : Routine

omp_set_num_threads(int num_threads) ;

```
$ ./hello_parallel

Hello Om, I am thread = 1
Hello Om, I am thread = 0
Hello Om, I am thread = 3
Hello Om, I am thread = 2
```

Threads

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
{
     omp_set_num_threads(4) ;
     #pragma omp parallel
      printf("Hello Om, I am thread = %d\n", omp_get_thread_num()) ;
}
```

Method 2 : Routine

`omp_set_num_threads(int num_threads) ;`

```
$ ./hello_parallel

Hello Om, I am thread = 1
Hello Om, I am thread = 0
Hello Om, I am thread = 3
Hello Om, I am thread = 2
```

Routines have higher priority than the environment variables !

Exercise

Write your first Parallel Program, with which you should be able to print your NAME from 4 underline cores !

Time : 3 min

Shared memory - scenario !

Shared memory - scenario !

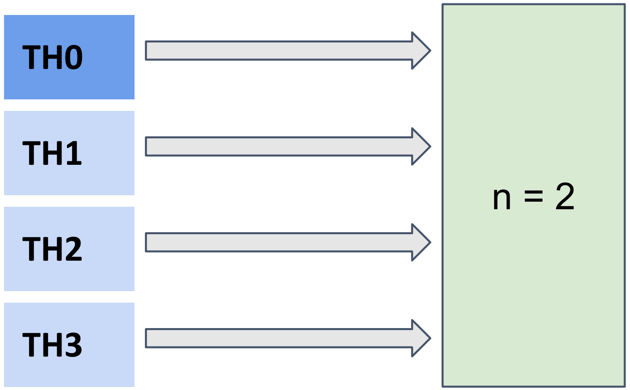
Let's understand different scenarios with different use cases !

Data variable scope

How data variables will be shared among different threads of parallel execution ?

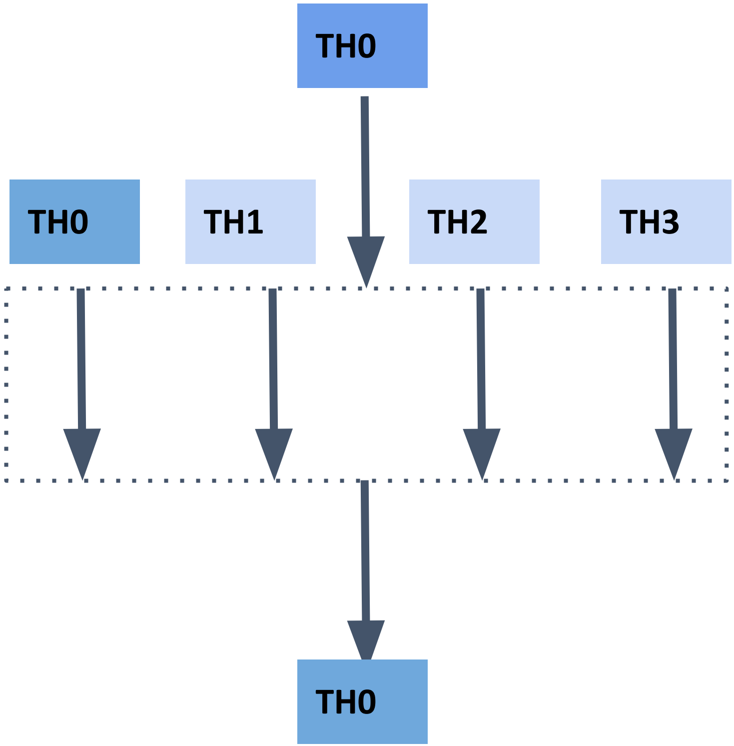
Scope of data variables : shared

- Data objects (variables) can be **shared** or **private**
- **Shared**
  - By **default** almost all variables are **shared**
  - Accessible to all threads
  - **Single instance** in shared memory



Shared Instance

Int n=2



Example -1

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_get_thread_num(),
sum) ;
}
```

Example -1

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_get_thread_num(),
sum) ;
}
```

What is expected output of the program ?

Example -1

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_get_thread_num(),
sum) ;
}
```

Every thread should add his thread-Id to a constant number and print

What is expected output of the program ?

Example -1

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_ r num(),
sum) ;
}
```

TH0	5
TH1	6
TH2	7
TH3	8

TH0	5

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 2 = 7
Value at thread 1 = 11
Value at thread 0 = 5
Value at thread 3 = 10
Value After paralle region, thread 0 = 11
[om@shrestha1 SampleCodes]$
```

Is it expected result ?

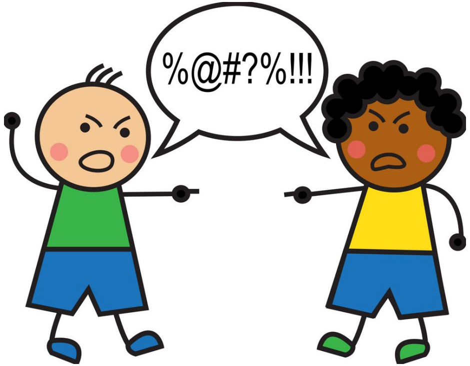
```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 2 = 7
Value at thread 1 = 11
Value at thread 0 = 5
Value at thread 3 = 10
Value After paralle region, thread 0 = 11
[om@shrestha1 SampleCodes]$
```

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 2 = 7
Value at thread 1 = 11
Value at thread 0 = 5
Value at thread 3 = 10
Value After paralle region, thread 0 = 11
[om@shrestha1 SampleCodes]$
```

So, what is the problem ?

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 2 = 7
Value at thread 1 = 11
Value at thread 0 = 5
Value at thread 3 = 10
Value After paralle region, thread 0 = 11
[om@shrestha1 SampleCodes]$
```

Race Condition



```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 2 = 7
Value at thread 1 = 11
Value at thread 0 = 5
Value at thread 3 = 10
Value After paralle region, thread 0 = 11
[om@shrestha1 SampleCodes]$
```

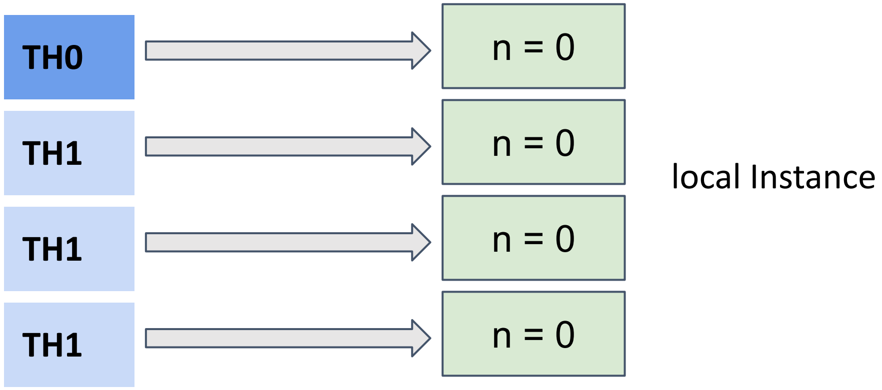
Learning : We need to be careful while declaring the scope of variables. The variables which is going to update in parallel region, should give special attention while writing a parallel program in shared memory system.

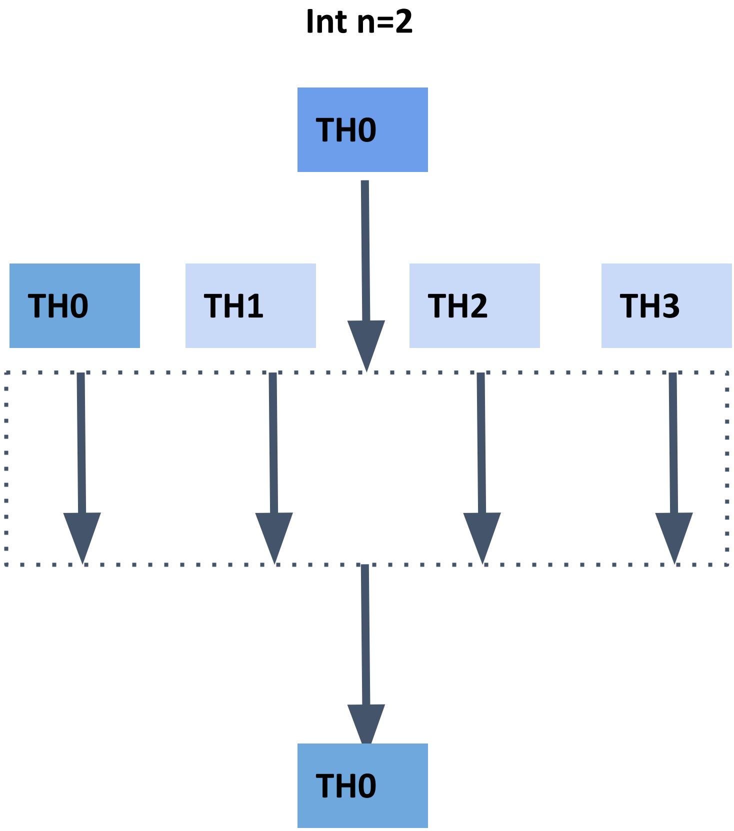
```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 2 = 7
Value at thread 1 = 11
Value at thread 0 = 5
Value at thread 3 = 10
Value After paralle region, thread 0 = 11
[om@shrestha1 SampleCodes]$
```

Solution ?

Scope of data variables : private

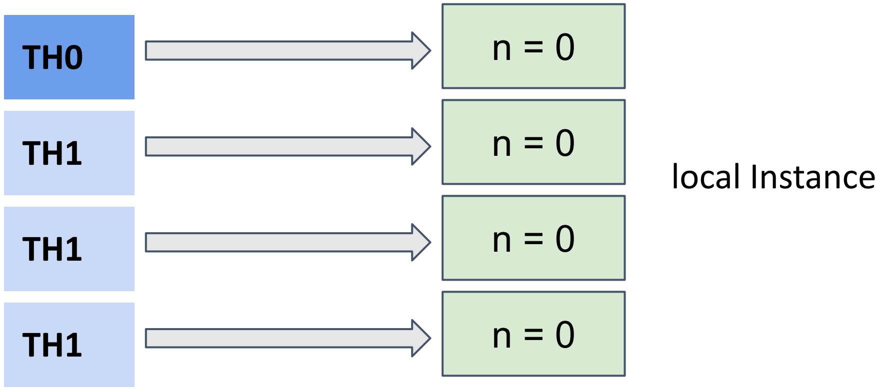
- **private**
  - Each thread allocates its **own private copy of the data**
  - Only **exists during the execution** of a parallel region!
  - Value **undefined/0** upon **entry & exit** of parallel region

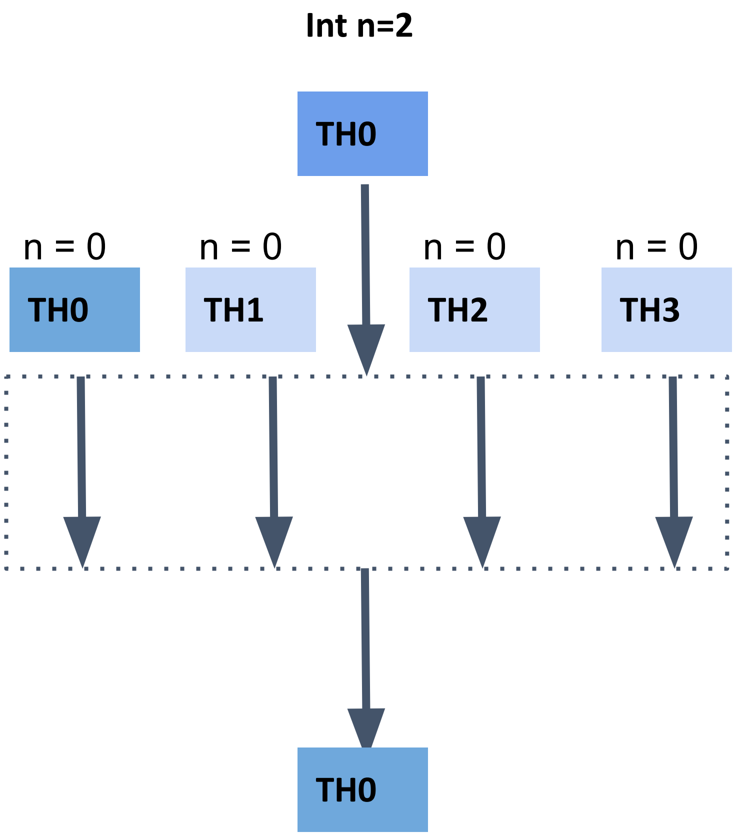




Scope of data variables : private

- **private**
  - Each thread allocates its **own private copy of the data**
  - Only **exists during the execution** of a parallel region!
  - Value **undefined/0** upon **entry & exit** of parallel region





Example - 2

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel private( ? )
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_get_thread_num(), sum) ;
}
```

Example - 2

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel private(sum)
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_get_thread_num(), sum) ;
}
```

What will be the output ?

Example - 2

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel private(sum)
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_get_thread_num(), sum) ;
}
```

What will be the output ?

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 0 = 0
Value at thread 2 = 2
Value at thread 1 = 1
Value at thread 3 = 3
Value After paralle region, thread 0 = 5
[om@shrestha1 SampleCodes]$
```

Observe the output and try to understand the difference !

shared

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 2 = 7
Value at thread 1 = 11
Value at thread 0 = 5
Value at thread 3 = 10
Value After paralle region, thread 0 = 11
[om@shrestha1 SampleCodes]$
```

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 0 = 0
Value at thread 2 = 2
Value at thread 1 = 1
Value at thread 3 = 3
Value After paralle region, thread 0 = 5
[om@shrestha1 SampleCodes]$
```

private

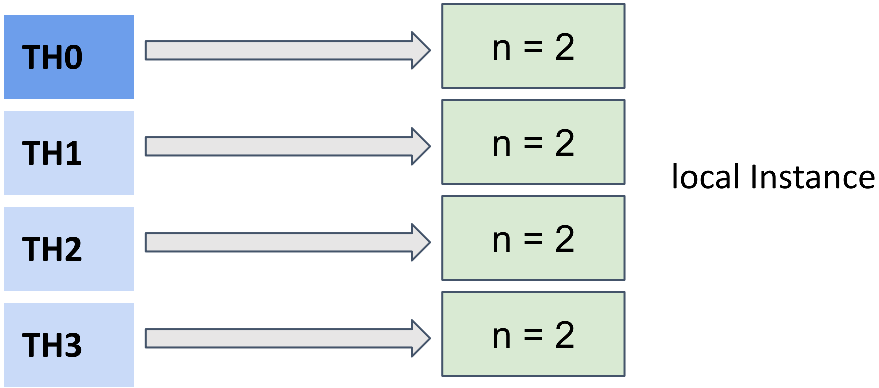
```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 0 = 0
Value at thread 2 = 2
Value at thread 1 = 1
Value at thread 3 = 3
Value After paralle region, thread 0 = 5
[om@shrestha1 SampleCodes]$
```

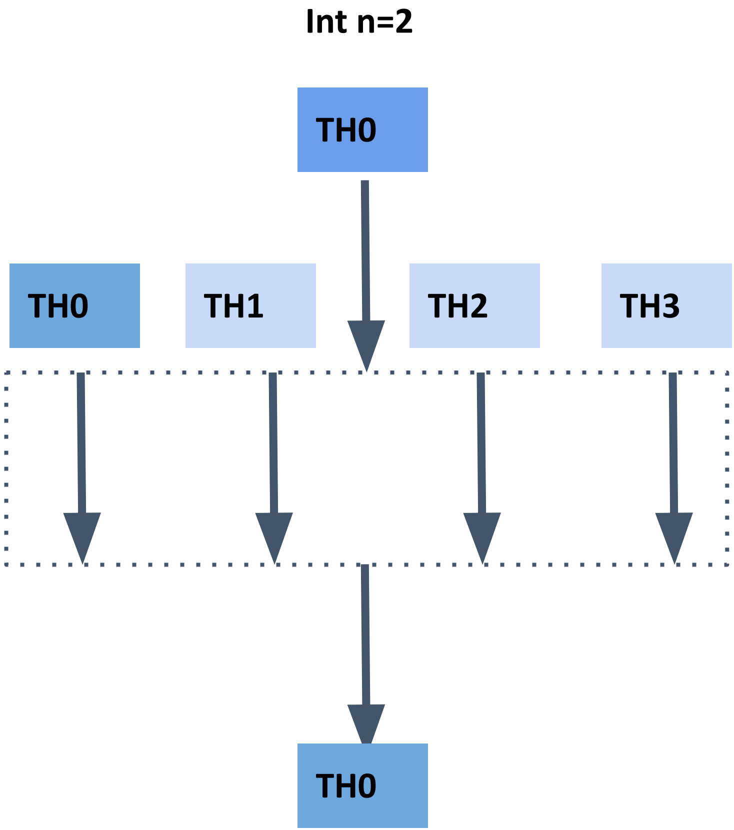
Learning :

- In case of private, the value of variable is **undefined/0** upon entry & exit of parallel region
- Only **exists during the execution** of a parallel region!

Scope of data variables : firstprivate

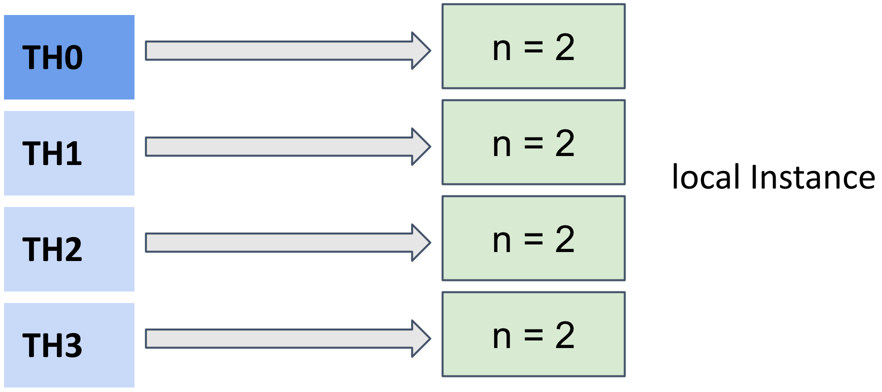
- **firstprivate**
  - Then each thread allocates its **own private copy of the data**
  - Only **exists during the execution** of a parallel region!
  - Additionally, get initialized with **value of original variable** (at entry)

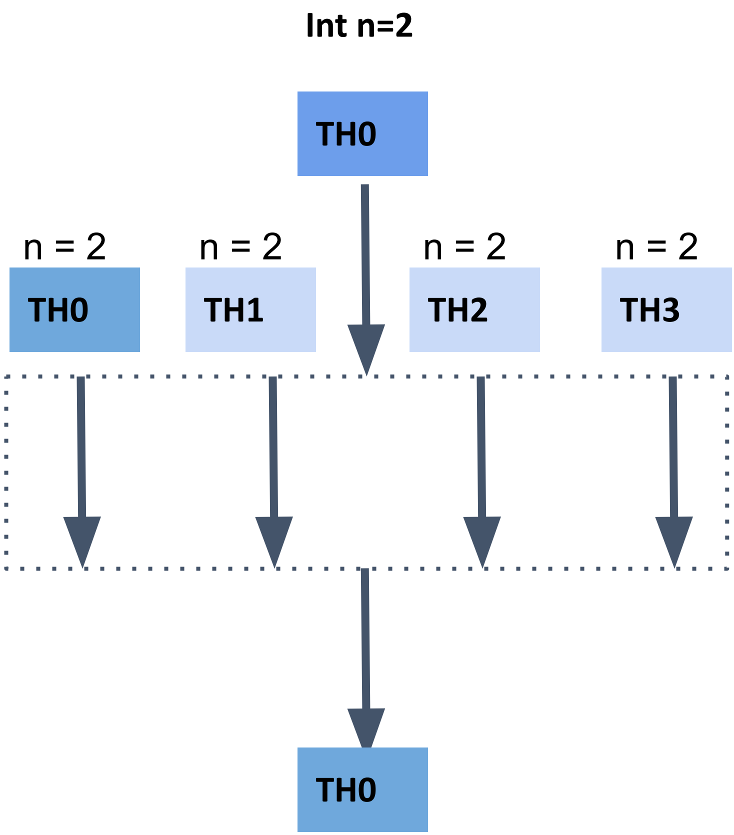




Scope of data variables : firstprivate

- **firstprivate**
  - Then each thread allocates its **own private copy of the data**
  - Only **exists during the execution** of a parallel region!
  - Additionally, get initialized with **value of original variable** (at entry)





Example - 3

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel firstprivate(sum)
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_get_thread_num(), sum) ;
}
```

What will be the output ?

Example - 3

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

int main(int argc, char* argv[])
{
int tid, sum=5 ;
#pragma omp parallel firstprivate(sum)
{
  tid = omp_get_thread_num();
  sum = sum + tid;
  printf("Value at thread %d = %d \n", tid, sum) ;
}

printf("Value After parallel region, thread %d = %d \n", omp_get_thread_num(), sum) ;
}
```

What will be the output ?

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 0 = 5
Value at thread 3 = 8
Value at thread 2 = 7
Value at thread 1 = 6
Value After paralle region, thread 0 = 5
[om@shrestha1 SampleCodes]$
```

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 0 = 5
Value at thread 3 = 8
Value at thread 2 = 7
Value at thread 1 = 6
Value After paralle region, thread 0 = 5
[om@shrestha1 SampleCodes]$
```

Learning :

- In case of firstprivate, the value of variable get initialized with **value of original variable** at the entry of parallel region
- Only **exists during the execution** of a parallel region!

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp variables.c
[om@shrestha1 SampleCodes]$ ./a.out
Value at thread 0 = 5
Value at thread 3 = 8
Value at thread 2 = 7
Value at thread 1 = 6
Value After paralle region, thread 0 = 5
[om@shrestha1 SampleCodes]$
```

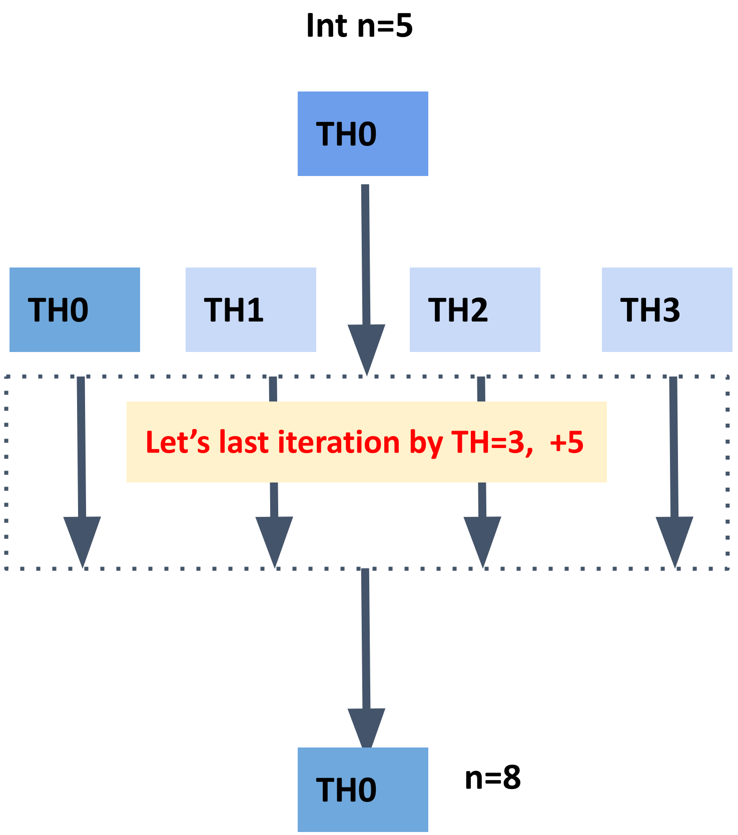
TH0	5
TH1	6
TH2	7
TH3	8

TH0	5

...achieved the initial target !

Scope of data variables : lastprivate

- **lastprivate**
  - Declares variables as private.
  - Corresponding **shared variable after parallel region gets value from that thread that finished the parallel region.**
  - It updates shared value after exit.
  - The meaning of lastprivate, is to assign **"the sequentially last iteration of the associated loops, or the lexically last section construct [...] to the original list item."**
  - Hence, there it no meaning for a pure parallel construct. It would not be a good idea to use a meaning like "the last thread to exit the parallel construct" - that would be a race condition.



Work Sharing

How the work will be distributed among threads of parallel execution ?

Work Sharing

How the work will be distributed among threads of parallel execution ?

Let's try to understand with different practical examples !

Work Sharing : Loops (parallel for loop)

Work load should be balanced, e.g. each thread should need the same period of time to handle its task

- Iterations of a parallel loop are executed in parallel by all threads of the current team of threads.
- The calculations inside an iteration must not depend on other iterations > responsibility of the programmer
- A schedule determines how iterations are divided among the threads
  - Specified by “schedule” parameter
- The form of the loop has to allow computing the number of iterations prior to entry into the loop > e.g., no WHILE loops

Work Sharing : Scheduling

- Scheduling Strategies
  - Distribution of iterations occurs in chunks
  - Chunks may have different sizes
  - Chunks are assigned either statically or dynamically
  - There are different assignment algorithms (types)
    - static
    - dynamic
    - guided
    - runtime

Work Sharing : Scheduling

1. Static

- ➢ Distribution is done at loop-entry time based on
  - Number of threads
  - Total number of iterations
- ➢ Less flexible
- ➢ Almost no scheduling overhead
- ➢ Workload is more or less the same for each iteration

Types :

- ➢ static without chunk size :
  - One chunk of iterations per thread, all chunks (nearly) equal size
- ➢ static with chunk size
  - Chunks with specified size are assigned in round-robin fashion

Example - 4

Write a Parallel C program where the iterations of a loop should scheduled statically across the team of threads. A thread should perform CHUNK iterations at a time before being scheduled for the next CHUNK of work.

Static

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE   1
#define N   4

int main (int argc, char *argv[])
{
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
```

```
#pragma omp parallel private(i,tid) //start
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n",
nthreads);
    }

  printf("Thread %d starting...\n",tid);
  #pragma omp for schedule(static,chunk)
  for (i=0; i<N; i++)
   {
   c[i] = a[i] + b[i];
   printf("Thread %d: c[%d]=%f\n",tid,i,c[i]);
   }
  }  //end
}
```

Static

Output : Number of Iterations=4, CHUNKSIZE = 1, Number of Threads=4

Static

Output : Number of Iterations=4, CHUNKSIZE = 1, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 1 starting...
Thread 1: c[1]= 2.000000
Thread 3 starting...
Thread 3: c[3]= 6.000000
Thread 2 starting...
Thread 2: c[2]= 4.000000
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
[om@shrestha1 SampleCodes]$
```

Static

Output : Number of Iterations=4, CHUNKSIZE = 1, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 1 starting...
Thread 1: c[1]= 2.000000
Thread 3 starting...
Thread 3: c[3]= 6.000000
Thread 2 starting...
Thread 2: c[2]= 4.000000
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
[om@shrestha1 SampleCodes]$
```

Output : Number of Iterations=4, **CHUNKSIZE = 2**, Number of Threads=4

Static

Output : Number of Iterations=4, CHUNKSIZE = 1, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 1 starting...
Thread 1: c[1]= 2.000000
Thread 3 starting...
Thread 3: c[3]= 6.000000
Thread 2 starting...
Thread 2: c[2]= 4.000000
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
[om@shrestha1 SampleCodes]$
```

Output : Number of Iterations=4, CHUNKSIZE = 2, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 2 starting...
Thread 3 starting...
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 1 starting...
Thread 1: c[2]= 4.000000
Thread 1: c[3]= 6.000000
[om@shrestha1 SampleCodes]$
```

Static

Output : Number of Iterations=4, **CHUNKSIZE = 4**, Number of Threads=4

Static

Output : Number of Iterations=4, CHUNKSIZE = 4, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 1 starting...
Thread 3 starting...
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 2 starting...
[om@shrestha1 SampleCodes]$
```

Static

Output : Number of Iterations=4, CHUNKSIZE = 4, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 1 starting...
Thread 3 starting...
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 2 starting...
[om@shrestha1 SampleCodes]$
```

Output : Number of Iterations=8, CHUNKSIZE = 2, Number of Threads=4

Static

Output : Number of Iterations=4, CHUNKSIZE = 4, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 1 starting...
Thread 3 starting...
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 2 starting...
[om@shrestha1 SampleCodes]$
```

Output : Number of Iterations=8, CHUNKSIZE = 2, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 1 starting...
Thread 1: c[2]= 4.000000
Thread 1: c[3]= 6.000000
Thread 2 starting...
Thread 2: c[4]= 8.000000
Thread 2: c[5]= 10.000000
Thread 3 starting...
Thread 3: c[6]= 12.000000
Thread 3: c[7]= 14.000000
Number of threads = 4
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
[om@shrestha1 SampleCodes]$
```

Work Sharing : Scheduling

2. Dynamic

- Distribution is done during execution of the loop
  - Each thread is assigned a subset of the iterations at loop entry
  - After completion each thread asks for more iterations
- More flexible - Can easily adjust to load imbalances. Workload might randomly differ from iteration to iteration
- More scheduling overhead (synchronization)
- Threads request new chunks dynamically
- Default chunk size is 1

Example - 5 : Dynamic

Write a Parallel C program where the iterations of a loop should scheduled dynamically across the team of threads. A thread should perform CHUNK iterations at a time before being scheduled for the next CHUNK of work.

Dynamic

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE   1
#define N  4

int main (int argc, char *argv[])
{
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
```

```
#pragma omp parallel private(i,tid) //start
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n",
nthreads);
    }

  printf("Thread %d starting...\n",tid);
  #pragma omp for schedule(dynamic,chunk)
  for (i=0; i<N; i++)
   {
   c[i] = a[i] + b[i];
   printf("Thread %d: c[%d]=%f\n",tid,i,c[i]);
   }
  }  //end
}
```

Dynamic

Output : Number of Iterations=4, CHUNKSIZE = 1, Number of Threads=4

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp work_share1.c
[om@shrestha1 SampleCodes]$ ./a.out
Number of threads = 4
Thread 0 starting...
Thread 3 starting...
Thread 3: c[1]= 2.000000
Thread 1 starting...
Thread 1: c[3]= 6.000000
Thread 2 starting...
Thread 3: c[2]= 4.000000
Thread 0: c[0]= 0.000000
[om@shrestha1 SampleCodes]$
```

… try to understand the difference !

<u>Note :</u> the default scheduling when no schedule clause is present is static with chunk size equal to #iterations / #threads

Work Sharing : Scheduling

3. guided

- First chunk has implementation-dependent size
- Size of each successive chunk decreases exponentially
- Chunks are assigned dynamically
- Chunks size specifies minimum size, default is 1
- Execution speed might increase/decrease with increasing iteration index
  - Chunksize is proportional to number of iterations left divided by the number of threads in the team

4. runtime

- Scheduling strategy is determined by environment variable at runtime
  - export OMP_SCHEDULE=type [, chunk]

Work Sharing : Sections

- A parallel section contains blocks of statements which can be executed in parallel
- Each block is executed once by one thread of the current team
- Scheduling of the block executions is implementation defined and cannot be controlled by the programmer
- Sections must not depend on each other
- Most frequent use case: parallel function calls

```
#pragma omp sections [clause[[,] clause] ...]
{
[#pragma omp section]
        structured block
...
}
```

Example - 6 : Sections

```
#include <stdio.h>
#include <omp.h>
int main()
{
    unsigned char a = 5, b = 9, c=4, d=7, e=14, f=15;
    omp_set_num_threads(3);
    #pragma omp parallel sections
        {
          #pragma omp section {
              printf("Thread= %d, a^b = %d, \n",omp_get_thread_num(), a^b);
          }
          #pragma omp section {
              printf("Thread= %d, c^d = %d, \n",omp_get_thread_num(), c^d);
          }
          #pragma omp section {
              printf("Thread= %d, e^f = %d, \n",omp_get_thread_num(), e^f);
          }
        }
    return 0;
}
```

Output

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp xor1.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread= 0, a^b = 12,
Thread= 1, c^d = 3,
Thread= 2, e^f = 1,
[om@shrestha1 SampleCodes]$
```

Example - 7 : Section

Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads !

Example - 7 : Section

```
#include <stdio.h>
#include <omp.h>
#define N 10
int main() {
int i, a[N], b[N];
#pragma omp parallel sections private(i)
{
#pragma omp section
{
for (i=1 ; i<=N ; i++){
        a[i] = i*2;
        printf("Thread %d : %d \n",omp_get_thread_num(), a[i]);
        }
}
#pragma omp section
{
for (i=1 ; i<=N ; i++) {
        b[i] = i*4;
        printf("Thread %d : %d \n",omp_get_thread_num(), b[i]);
        }
}
}
}
```

Output

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp section.c
[om@shrestha1 SampleCodes]$ ./a.out
Thread 0 : 2
Thread 0 : 4
Thread 0 : 6
Thread 0 : 8
Thread 0 : 10
Thread 0 : 12
Thread 0 : 14
Thread 0 : 16
Thread 0 : 18
Thread 0 : 20
Thread 1 : 4
Thread 1 : 8
Thread 1 : 12
Thread 1 : 16
Thread 1 : 20
Thread 1 : 24
Thread 1 : 28
Thread 1 : 32
Thread 1 : 36
Thread 1 : 40
[om@shrestha1 SampleCodes]$
```

Synchronization

To get accurate and proper results, synchronization among team members is must while working in team.

Race condition

<u>Race Condition :</u> When more than one thread try to update the shared data variable !

Race condition

<u>Race Condition :</u> When more than one thread try to update the shared data variable !

```c
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

What is Expected Output of this Program ?

Race condition

Race Condition : When more than one thread try to update the shared data variable !

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

What is Expected Output of this Program ?

Print sum of 0 - 9 numbers !

Race condition

Race Condition : When more than one thread try to update the shared data variable !

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i <N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Race condition

Race Condition : When more than one thread try to update the shared data variable !

```c
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i <N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp race.c
[om@shrestha1 SampleCodes]$ ./a.out
s = 29.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 21.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 43.000000
[om@shrestha1 SampleCodes]$
```

Race condition

<u>Race Condition :</u> When more than one thread try to update the shared data variable !

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i <N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i) {
     {
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Solution ?

Synchronization : critical

- A critical region restricts execution of the associated block of statements to a single thread at a time

- A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name

- Mutual exclusion : Only one thread at a time can enter the critical region.

Synchronization : critical

```c
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Synchronization : critical

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     #pragma omp critical
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Synchronization : critical

```c
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     #pragma omp critical
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp cri.c
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$
```

Synchronization : critical

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     #pragma omp critical
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

... but, wait and Think !

Synchronization : critical

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     #pragma omp critical
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

... but, wait and Think !

Code is getting executed almost Serially !

Synchronization : critical

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     #pragma omp critical
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Always use critical (& other synchronization !) regions with precautions.

Program gets extremely slow : No speedup

Synchronization : critical

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for
{
     for (i=0 ; i<N ; ++i)
     {
     #pragma omp critical
     s += a[i];
     }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Any better Solution ?

Synchronization : critical

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N],  s_local;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel private(i, s_local)
{
      #pragma omp for
      for (i=0 ; i<N ; ++i)
      {
      s_local += a[i];
      }
      #pragma omp critical
      {
      s += s_local;
      }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Critical section has moved outside of the loop

Synchronization : critical

```c
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N], s_local;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel private(i, s_local)
{
      #pragma omp for
      for (i=0 ; i<N ; ++i)
      {
      s_local += a[i];
      }
      #pragma omp critical
      {
      s += s_local;
      }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp cri1.c
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$
```

Synchronization : critical

Performance Comparison : N = 1000000

Synchronization : critical

Performance Comparison : N = 1000000

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp cri.c
[om@shrestha1 SampleCodes]$ time ./a.out
s = 499999500000.000000

real    0m3.508s
user    0m17.826s
sys     0m0.009s
[om@shrestha1 SampleCodes]$
```

```
[om@shrestha1 SampleCodes]$ export OMP_NUM_THREADS=8
[om@shrestha1 SampleCodes]$ gcc -fopenmp cri1.c
[om@shrestha1 SampleCodes]$ time ./a.out
s = 499999500000.000000

real    0m0.009s
user    0m0.001s
sys     0m0.013s
[om@shrestha1 SampleCodes]$
```

Synchronization : Atomic

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N],  s_local;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel private(i, s_local)
{
      #pragma omp for
      for (i=0 ; i<N ; ++i)
      {
      s += a[i];
      }
      #pragma omp critical
      {
      s += s_local;
      }
}
printf("s = %f \n", s);
}
```

Any other solution ?

Synchronization : Atomic

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N],  s_local;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel private(i, s_local)
{
      #pragma omp for
      for (i=0 ; i<N ; ++i)
      {
      s += a[i];
      }
      #pragma omp critical
      {
      s += s_local;
      }
}
printf("s = %f \n", s);
}
```

Any other solution ?

Synchronization : Atomic

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N], s_local;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel private(i, s_local)
{
      #pragma omp for
      for (i=0 ; i<N ; ++i)
      {
      s += a[i];
      }
      #pragma omp atomic
      s += s_local;
}
printf("s = %f \n", s);
}
```

- The ATOMIC directives ensures that a specific memory location is updated atomically. No thread interference
- Only a single variable/line gets affected and not the block.
- ATOMIC construct permits better optimization (based on hardware instructions)

Synchronization : Atomic

```c
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N],  s_local;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel private(i, s_local)
{
      #pragma omp for
      for (i=0 ; i<N ; ++i)
      {
      s += a[i];
      }
      #pragma omp atomic
      s += s_local;
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp atomic.c
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$
```

More about synchronization

Sometimes it is useful that within a parallel region just one thread is executing code, e.g., to read/write data. OpenMP provides two ways to accomplish this

- **Master construct :**
  - The master thread (thread 0) executes the enclosed code, all other threads ignore this block of statements, i.e. there is no implicit barrier

```
#pragma omp master
    structured block
```

- **Single construct :**
  - The first thread reaching the directive executes the code. All threads execute an implicit barrier

```
#pragma omp single [nowait, private(list),...]
structured block
```

More about synchronization

- **<u>Barrier :</u>**
  - The barrier directive explicitly synchronizes all the threads in a team.
  - When encountered, each thread in the team waits until all the others have reached this point
  - There are also implicit barriers at the end of parallel region, cannot be changed of work share constructs e.g SECTIONS.

```
#pragma omp barrier
```

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N],  s_local;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel private(i, s_local)
{
      #pragma omp for
      for (i=0 ; i<N ; ++i)
      {
      s_local += a[i];
      }
      #pragma omp atomic
      s += s_local;
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Any other better better way to perform such operations ?

Reduction

- Reductions often occur within parallel regions or loops
- Reduction variables have to be shared in enclosing parallel context
- Thread-local results get combined with outside variables using reduction operation
- Typical applications: Compute sum of array or find the largest element

```
reduction(operator | intrinsic : varlist )
```

- **<u>Note:</u>** order of operations unspecified → can produce slightly different results than sequential version (rounding error)

Reduction

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N], s_local;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel private(i, s_local)
{
      #pragma omp for
      for (i=0 ; i<N ; ++i)
      {
      s += a[i];
      }
      #pragma omp atomic
      s += s_local;
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Reduction

```
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N] ;
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for private(i) reduction(+:s)
{
      for (i=0 ; i<N ; ++i)
      {
      s += a[i];
      }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

Reduction

```c
#include<stdio.h>
#include <omp.h>
#define N 10

int main() {
double s = 0.0, a[N];
int i;
// initialize the array
for (i=0; i < N; i++){
  a[i] = i * 1.0;
}
#pragma omp parallel for private(i) reduction(+:s)
{
      for (i=0 ; i<N ; ++i)
      {
      s += a[i];
      }
}
printf("s = %f \n", s);
}
```

Expected Output : s = 45

```
[om@shrestha1 SampleCodes]$ gcc -fopenmp reduction.c
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$ ./a.out
s = 45.000000
[om@shrestha1 SampleCodes]$
```

Reduction

Operator	Data Type	Initial Value
+	Floating point, Integer	0
*	Floating point, Integer	1
-	Floating point, Integer	0
&	Integer	all bits on
|	Integer	0
^	Integer	0
&&	Integer	1
||	Integer	0

- The table shows: The operators and intrinsic allowed for reductions The initial values used to initialize
- Since OpenMP 3.1: min, max

Career Opportunities in HPC

- **HPC System Administrator:** Installation, configuration, and maintenance of HPC systems
- **HPC Application Developer:** Develop software applications that can take advantage of the high-performance computing resources available
- **HPC Performance Engineer:** Analyzing and optimizing the performance of HPC systems and applications
- **HPC Data Scientist:** Analyze large and complex datasets
- **HPC Researcher:** research in various areas related to HPC, such as algorithm development, performance optimization, or system architecture
- **HPC Education and Training:** develop and deliver training programs and courses on HPC technology and applications
- **HPC Consultant:** promote and sell HPC systems and solutions to potential customers etc

Any questions ?