1. What is OpenMP?

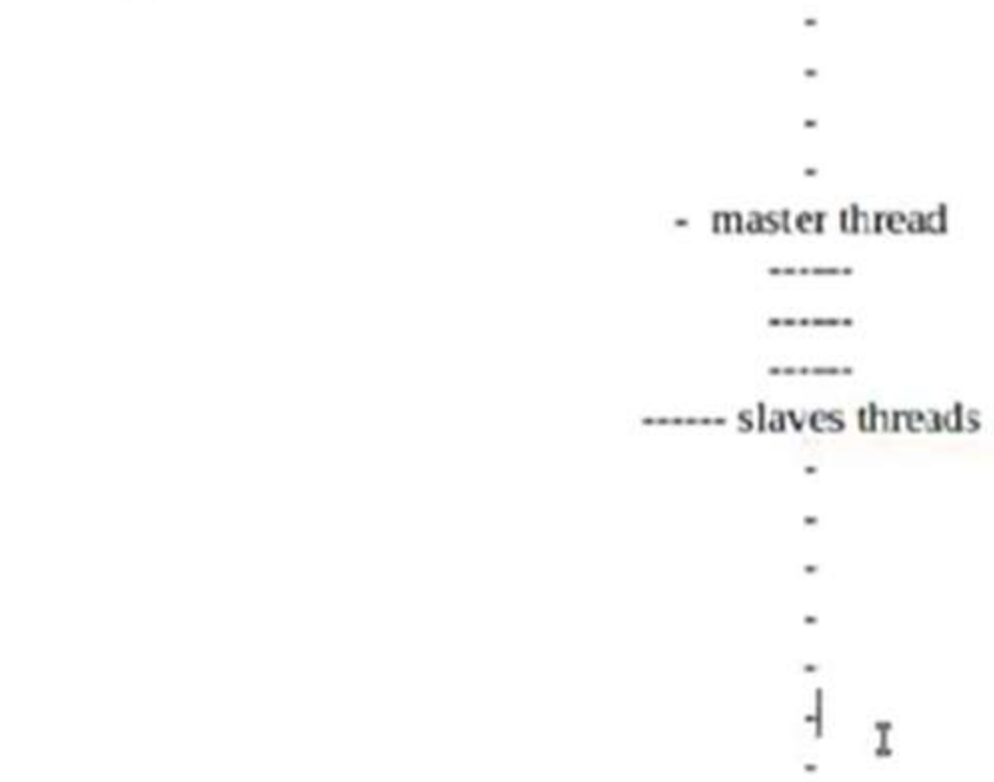
OpenMP is an API. OpenMP stands for Open Multi-Processing. 'Open' stands for Open source. Multi-Processing means Multi Threading. This means OpenMP is open source API which is used for multi-threading. It is managed by consortium OpenMP Architecture Review Board (OpenMP ARB) which is formed by several companies like AMD, Intel, IBM, HP, Nvidia, Oracle etc.

OpenMP is available for languages C, C++, Fortran. For the first time, OpenMP ARB releases OpenMP for Fortran language in 1997. In the next year, it gave OpenMP API for C/C++ languages. In this post, we will see how to use OpenMP for C/C++.

Each process starts with one main thread. This thread is call *master thread* in OpenMP. For a particular block of code, we create multiple threads alongwth this master thread. These extra threads other than master thread are called as *slave threads*.

OpenMP is called as *Shared Memory* model as OpenMP is used to create multiple threads and these multiple threads share the memory of main process.

OpenMP is called as *Fork-Join* model. It is because, all slave threads after execution get joined to the master thread. i.e. Process starts with single master thread and ends with single master thread.



Find All Match Case Page

3. A simple example to create threads:

```
#pragma omp parallel
{
printf("Hello World");

}
```

Suppose we are adding this code in a C language program. This snippet will create multiple threads. All of which are printing "Hello World". By default, number of threads created is equal to number of Processor cores.

4. How to create required number of threads?

```
#pragma omp parallel num_threads(7)
{
printf("Hello World");
}
```

If you want create specific number of threads, use num_threads() and a number indicating number of threads to be created should be passed as argument to num_threads(). In above example, seven threads will be created. Each one will be responsible for printing "Hello World".

To understand how to create multiple threads, check example of array addition given in post OpenMP Program for Array Addition.

```
#include<stdio.h>
#include<omp.h>
void main()
{
int a[5]={1,2,3,4,5};
int b[5]={6,7,8,9,10};
int c[5];
int tid;

#pragma omp parallel num_threads(5)
{
tid=omp_get_thread_num();
c[tid]=a[tid]+b[tid];
printf("c[%d]=%d\n",tid,c[tid]);
}

}
```

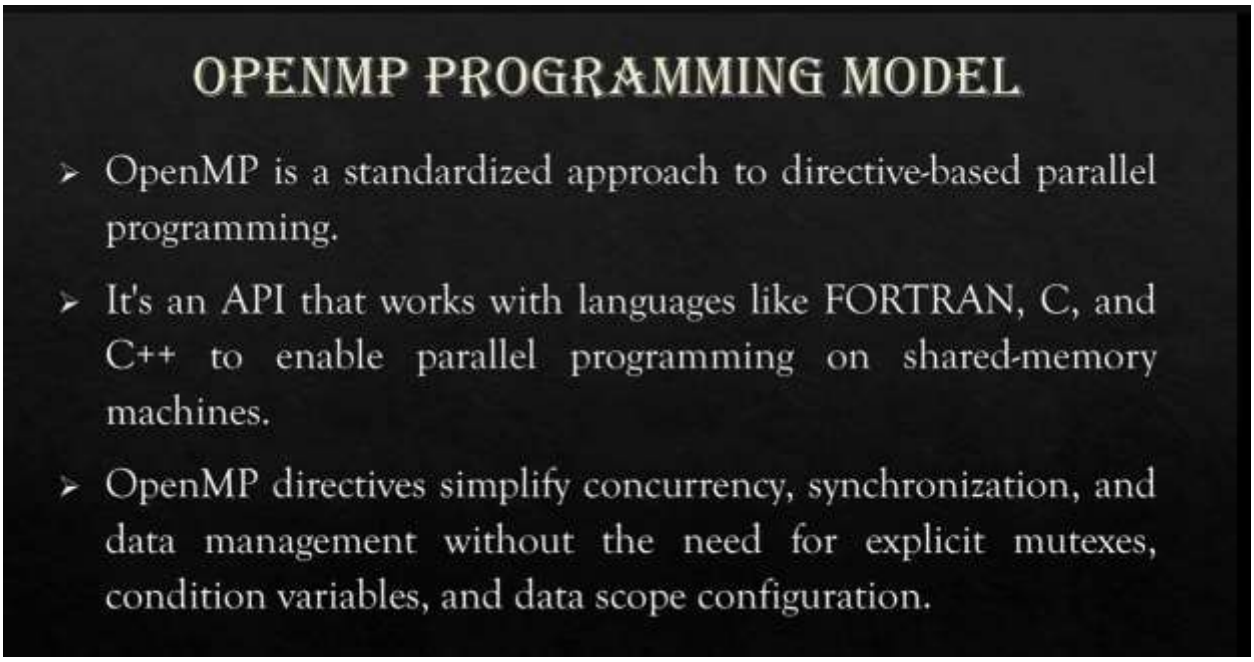
```
parag@parag-Inspiron-N4010: ~/Desktop/prog/openmp
File  Edit  View  Search  Terminal  Help
parag@parag-Inspiron-N4010:~/Desktop/prog/openmp$ gcc -fopenmp arrayaddition.c
parag@parag-Inspiron-N4010:~/Desktop/prog/openmp$ ./a.out
c[1]=9
c[2]=11
c[3]=13
c[4]=15
c[0]=7
parag@parag-Inspiron-N4010:~/Desktop/prog/openmp$ ./a.out
c[2]=11
c[0]=7
c[4]=15
c[1]=9
c[3]=13
parag@parag-Inspiron-N4010:~/Desktop/prog/openmp$
```

```c
#include<stdio.h>
#include<omp.h>
void main()
{
int a[5]={1,2,3,4,5};
int b[5]={6,7,8,9,10};
int c[5];
int tid;
int i;

#pragma omp parallel for num_threads(5)
for(i=0;i<5;i++)
{
tid=omp_get_thread_num();
c[tid]=a[tid]+b[tid];
printf("c[%d]=%d\n",tid,c[tid]);
}

}
```

6. How to allocate different work to different thread?

In OpenMP, we can allocate different work to different thread by using "sections". Check following example:

```
#pragma omp parallel sections num_threads(3)
{
```



OPENMP PROGRAMMING MODEL

- In C and C++, OpenMP directives are built using #pragma compiler directives, consisting of a directive name and optional clauses.

  #pragma omp directive [clause list]

- OpenMP programs run serially until they reach the "parallel" directive, which initiates a group of threads.

  #pragma omp parallel [clause list]

  /* structured block */

OPENMP PROGRAMMING MODEL

- The first thread to encounter this directive becomes the master and gets assigned thread ID 0 within the group.
- Clauses in the directive allow for conditional parallelization, thread count specification, and data handling.
  - Conditional parallelization: It is controlled by the "if" clause, specifying whether threads are created.
  - Degree of Concurrency: The "num_threads" clause determines the number of threads.

OPENMP PROGRAMMING MODEL

- Data handling: It clauses include "private" (variables local to each thread), "firstprivate" (similar to "private," but with initialized values), and "shared" (variables shared across all threads). The default state of a variable can be set using "default(shared)" or "default(none).

OPENMP PROGRAMMING MODEL

- A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

```
#pragma omp parallel if (is_parallel==1) num_threads(8)
private (a) shared (b) firstprivate(c)
{
/* structured block */
}
```

OPENMP PROGRAMMING MODEL

- If the value of the variable is_parallel equals one, eight threads are created.
- Each of these threads gets private copies of variables a and c, and shares a single value of variable b.
- The value of each copy of c is initialized to the value of c before the parallel directive.
- The default state of a variable is specified by the clause default (shared) or default (none).

:30 / 7:29

OPENMP PROGRAMMING MODEL

```
/* ************************************************************
An OpenMP version of a threaded program to compute PI.
************************************************************ */
#pragma omp parallel default(private) shared (npoints) \
        reduction(+: sum) num_threads(8)
{
        num_threads = omp_get_num_threads();
        sample_points_per_thread = npoints / num_threads;
        sum = 0;
        for (i = 0; i < sample_points_per_thread; i++) {
                rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
                rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
                if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
                        (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
                        sum ++;
        }
}
```

REDUCTION CLAUSE IN OPENMP

- The "reduction" clause combines multiple local copies of a variable from different threads into a single copy when threads exit.
- It is used as "reduction(operator: variable list)."
- Operators can be mathematical (+, *, -) or logical (&, |, ^, &&, ||).

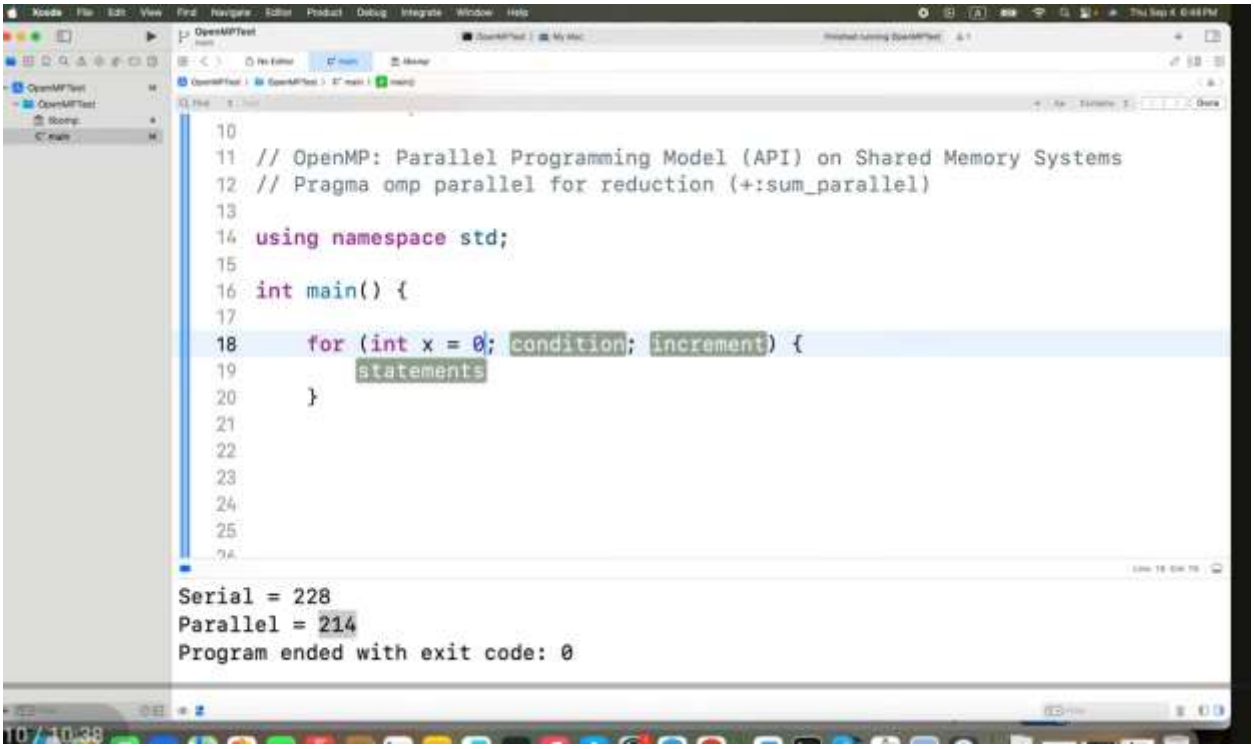
```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

5:06 / 7:29

SPECIFYING CONCURRENT TASKS IN OPENMP

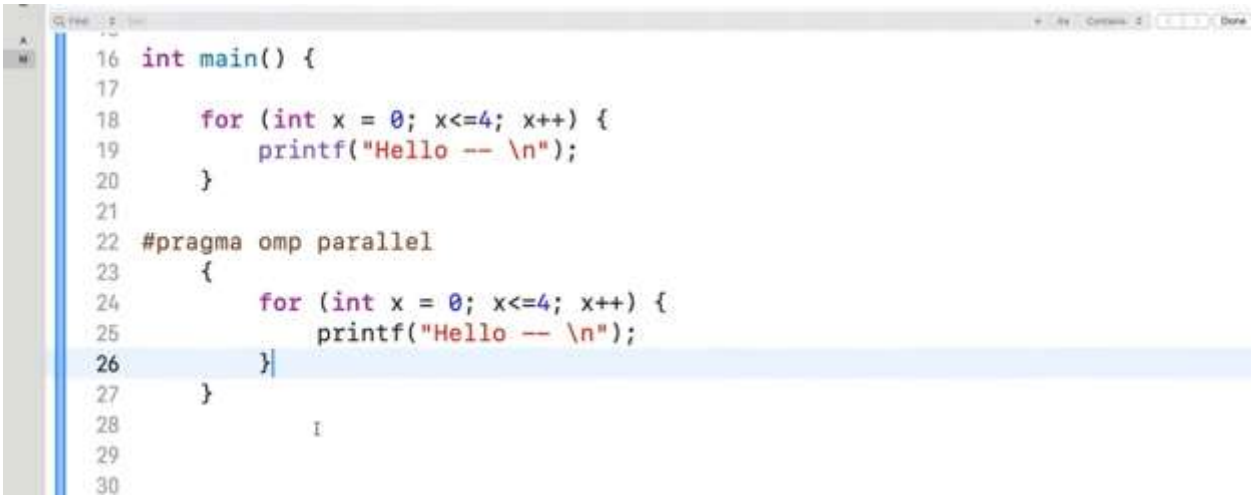
- OpenMP's "parallel" directive can be used alongside other directives to specify concurrency across iterations and tasks.
- OpenMP provides the "for" and "sections" directives for concurrent iterations and tasks.
- The "for" directive splits parallel iteration spaces across threads and supports various clauses like "private," "firstprivate," "lastprivate," "reduction," "schedule," "nowait," and "ordered."

```
#pragma omp for [clause list]
/* for loop */
```



```
// OpenMP: Parallel Programming Model (API) on Shared Memory Systems
// Pragma omp parallel for reduction (+:sum_parallel)

using namespace std;

int main() {

    for (int x = 0; condition; increment) {
        statements
    }
```

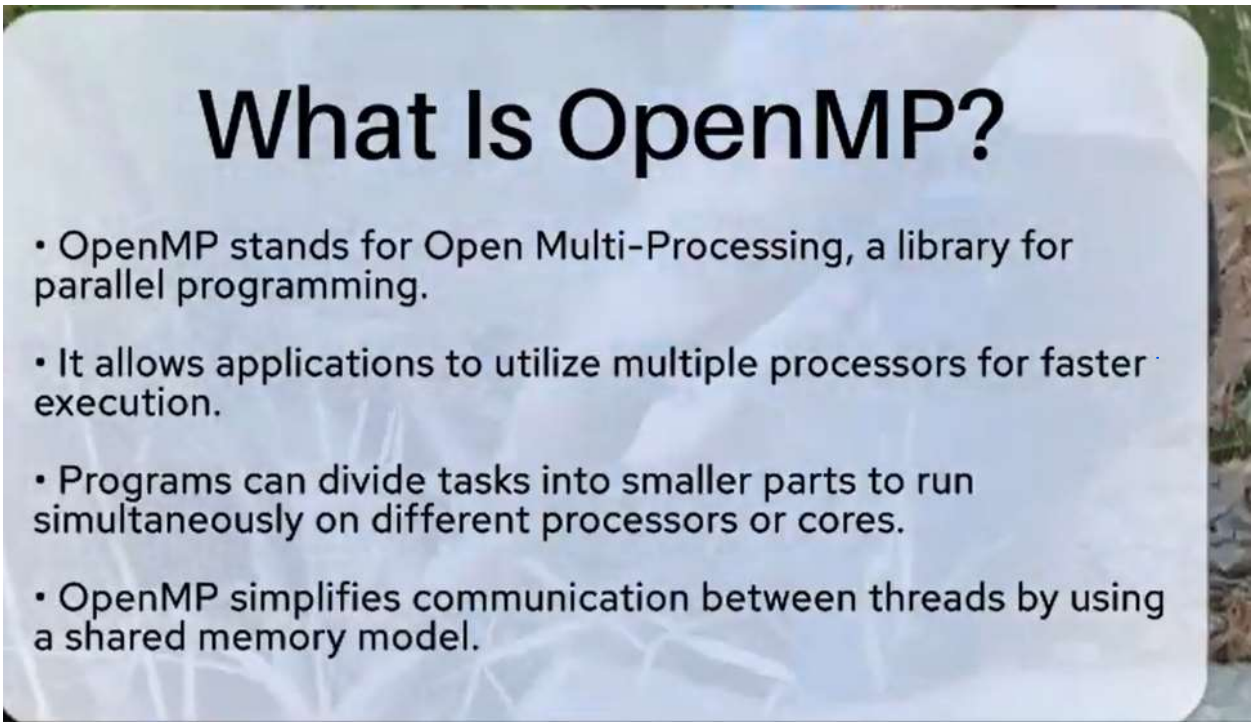
```
Serial = 228
Parallel = 214
Program ended with exit code: 0
```



```
int main() {

    for (int x = 0; x<=4; x++) {
        printf("Hello -- \n");
    }

#pragma omp parallel
    {
        for (int x = 0; x<=4; x++) {
            printf("Hello -- \n");
        }
    }
```

```c
int main() {
    for (int x = 0; x<=4; x++) {
        printf("Hello -- \n");
    }

    #pragma omp parallel for
    {
        for (int x = 0; x<=4; x++) {
            printf("Hello This Hello \n");
        }
    }
```

```
Hello This Hello
Hello This Hello
Hello This Hello
Hello This Hello
Program ended with exit code: 0
```



What Is OpenMP?

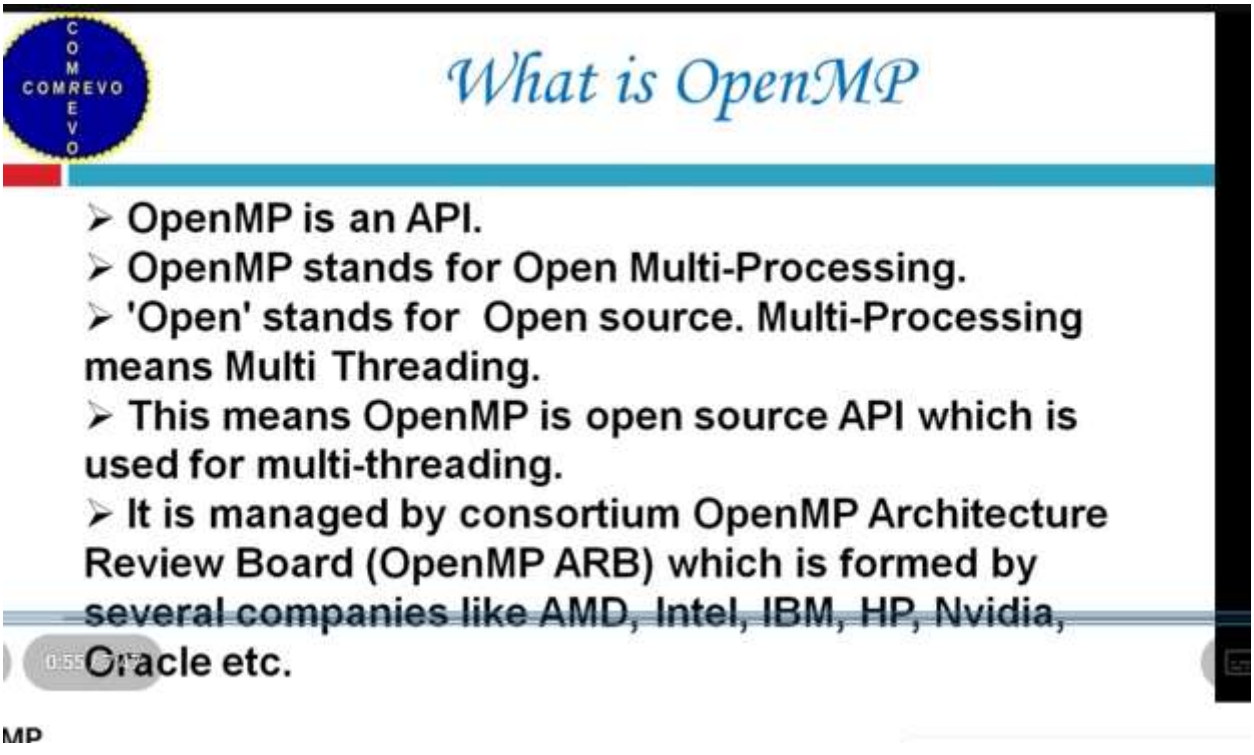
- The shared memory approach may limit scalability as more processors are added.
- An OpenMP program starts with standard execution and uses pragmas to create threads for parallel sections.
- The master thread manages sequential tasks and creates slave threads for parallel execution.

What Is OpenMP?

- Slave threads operate independently and return to the master thread upon completion.
- OpenMP includes work-sharing constructs, such as loops, for distributing workloads among threads.
- Developers can control the number of threads through environment variables or OpenMP functions.

What Is OpenMP?

- OpenMP is beneficial for high-performance computing tasks like scientific simulations and data processing.
- It is easier to use compared to other parallel programming methods.
- OpenMP is particularly relevant for C++ and C programmers in high-performance computing.
- It differs from MPI, which is suited for distributed memory systems and larger problems.



What is OpenMP

- OpenMP is available for languages C, C++, Fortran.
- For the first time, OpenMP ARB releases OpenMP for Fortran language in 1997.
- In the next year, it gave OpenMP API for C/C++ languages.

1:23 / 7:47

What is OpenMP

- Each process starts with one main thread. This thread is called as master thread in OpenMP.
- For a particular block of code, we create multiple threads alongwith this master thread. These extra threads other than master thread are called as slave threads.

1:59 / 7:47

OpenMP is called as Shared Memory model as OpenMP is used to create multiple threads and these multiple threads share the memory of main process.

OpenMP is called as Fork-Join model. It is because, all slave threads after execution get joined to the master thread. i.e. Process starts with single master thread and ends with single master thread.

For C/C++, we have to include "omp.h" header file. "omp.h" is available for gcc/g++ compilers.

A simple example to create threads using OpenMP

```
#pragma omp parallel
{
printf("Hello World");
}
```

- Suppose we are adding this code into a C language program. This snippet will create multiple threads.
- All of which are printing "Hello World".
- By default, number of threads created is equal to number of Processor cores.

How to create required number of threads?

```
#pragma omp parallel num_threads(7)
{
printf("Hello World");
}
```

- If you want create specific number of threads, use num_threads() and a number indicating number of threads to be created should be passed as argument to num_threads().
- In above example, seven threads will be created. Each one will be responsible for printing "Hello World".

How to create multiple threads using "for" loop?

```
#pragma omp parallel for
for(i=0;i<6;i++)
{
printf("Hello World");
}
```

- In above code snippet, since we are not mentioning number of threads, number of threads will be equal to number of cores.

- This "for" loop will have six iterations which are done by these many number of threads.

How to allocate different work to different thread?

In OpenMP, we can allocate different work to different thread by using "sections". E.g.

```
#pragma omp parallel sections num_threads(3)
{
  #pragma omp section
    { printf("Hello World One");}
  #pragma omp section
    { printf("Hello World Two");}
  #pragma omp section
    { printf("Hello World Three"); }
}
```

In above example, we have created three threads by mentioning num_threads(3) and each thread is printing different line.

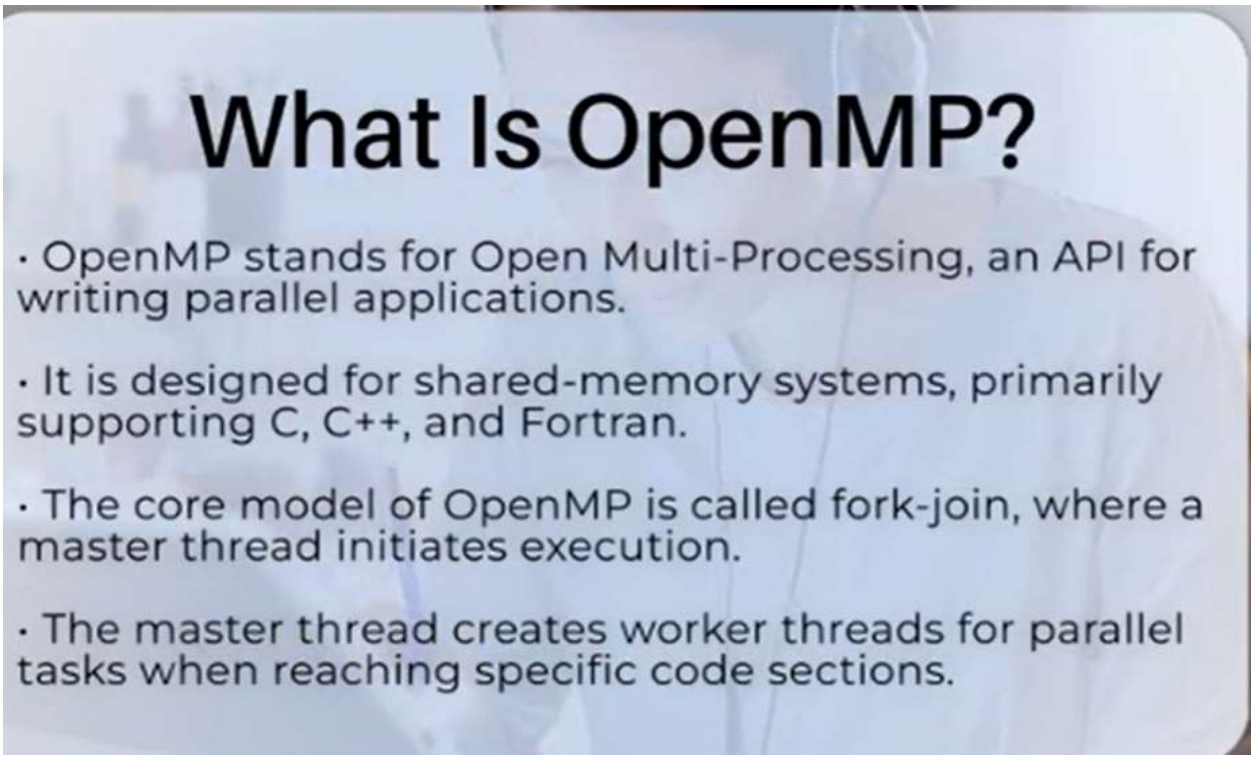
How to synchronize threads in OpenMP?

In OpenMP, We can avoid race condition among threads by using preprocessor directive "#pragma omp critical".

```
#pragma omp parallel num_threads(5)
{
  #pragma omp critical
    {
      x=x+1;
    }
}
```

In above example, we have created five threads by mentioning num_threads(5).

Here, only one thread will increment value of x at one time.



What Is OpenMP?

- Each worker thread has a unique identifier and works simultaneously on assigned tasks.
- After completing their tasks, worker threads return to the master thread to continue sequential execution.
- OpenMP allows efficient use of multi-core processors without complex thread management.

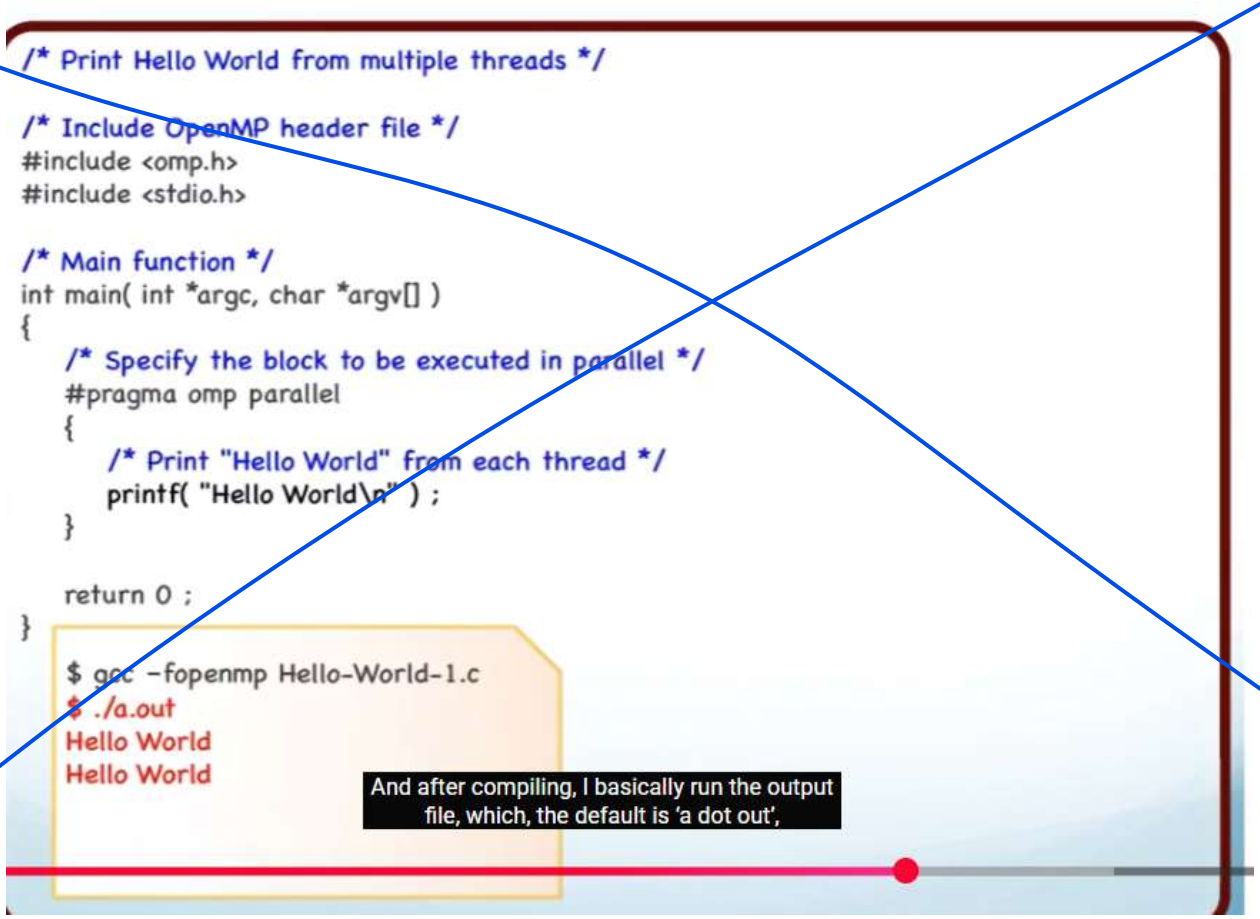
What Is OpenMP?

- It supports both task parallelism (different tasks on different threads) and data parallelism (data processing split among threads).
- Programmers use compiler directives, or pragmas, to specify parallel execution areas in the code.
- OpenMP operates on shared-memory architectures, allowing threads direct access to the same memory space.

What Is OpenMP?

- Shared memory simplifies data sharing but may limit scalability due to memory bandwidth constraints.
- In blockchain technology, OpenMP can enhance operations like cryptographic calculations and transaction validations.
- While beneficial for multi-core processors, distributed environments may require other models like the Message Passing Interface for scalability.
- OpenMP provides an accessible method for parallel programming, making it suitable for various applications, including blockchain and cryptocurrency.

OpenMP (Open Multi-processing)



1. What is Parallel & Distributed Computing?

Parallel Computing

- Multiple **processors/cores** work **simultaneously**
- Share the **same memory**
- Faster execution of programs

Example:

- Modern CPUs (4-core, 8-core)
- Multithreading programs

Distributed Computing

- Multiple **separate computers (nodes)**
- Each has **its own memory**
- Communicate over a **network**

Example:

- Clusters
- Cloud systems

---

2. What is OpenMP?

OpenMP (Open Multi-Processing) is:

- An **API** for **parallel programming**
- Works on **shared memory systems**
- Used with **C, C++, and Fortran**

Key idea:

Convert **serial code → parallel code** using **compiler directives**

3. Why OpenMP?

Without OpenMP:

- Programmer manually manages threads
- Very complex and error-prone

With OpenMP:

- Simple `#pragma` directives
- Automatic thread management
- Portable & scalable

4. OpenMP Architecture

- **Fork–Join Model**

How it works:

1. Program starts with **one thread (master)**

2. When a parallel region is encountered:
    - Master **forks** multiple threads
3. Threads execute in parallel
4. Threads **join** back to one

Serial → Parallel → Serial

5. Basic OpenMP Concepts

Concept	Meaning
Thread	Small execution unit
Parallel Region	Code executed by multiple threads
Work Sharing	Divide work among threads
Synchronization	Control access to shared data
Data Scope	Shared or private variables

6. How to Use OpenMP (Syntax)

Include header (C/C++)

c Copy code

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

Compiler command

bash Copy code

```bash
gcc file.c -fopenmp
```

7. Hello World Example

c Copy code

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

int main() {
    #pragma omp parallel
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }
    return 0;
}
```

Output:

```cpp
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

(Order may vary)

---

8. Parallel Region

```c
#pragma omp parallel
{
   // Code executed by all threads
}
```

- Creates multiple threads
- Each thread executes the same code

The End

9. Work Sharing Constructs

(a) `for` Directive

Used to divide loop iterations

```c
#pragma omp parallel for
for(int i = 0; i < 10; i++) {
    printf("Thread %d executes i=%d\n",
           omp_get_thread_num(), i);
}
```

Benefit:

- Automatic division of loop iterations

(b) `sections`

Different tasks run in parallel

```c
#pragma omp parallel sections
{
    #pragma omp section
    printf("Task 1\n");

    #pragma omp section
    printf("Task 2\n");
}
```

10. Data Sharing in OpenMP

Shared Variables

- Accessible by all threads

Private Variables

- Each thread has its own copy

Example:

```c
#pragma omp parallel private(x) shared(y)
```

11. Race Condition (Important)

Problem:

Multiple threads modify shared data simultaneously

Example (WRONG):

```c
int sum = 0;

#pragma omp parallel for
for(int i=0;i<100;i++)
    sum += i;
```

Result:

❌ Incorrect output

---

12. Solving Race Conditions

(a) `critical`

```c
#pragma omp critical
sum += i;
```

- Only one thread at a time

(b) `atomic`

```c
#pragma omp atomic
sum += i;
```

- Faster than critical

(c) `reduction` (BEST)

```c
int sum = 0;

#pragma omp parallel for reduction(+:sum)
for(int i=0;i<100;i++)
    sum += i;
```

✅ Correct
✅ Efficient

---

13. Synchronization Constructs

Directive	Purpose
barrier	All threads wait
critical	One thread at a time
atomic	Safe memory update
ordered	Ordered execution

Example:

Example:

```c
#pragma omp barrier
```

---

14. Scheduling in OpenMP

Controls how loop iterations are assigned

```c
#pragma omp parallel for schedule(static)
```

Types:

Type	Description
static	Fixed chunk
dynamic	Assigned at runtime
guided	Decreasing chunk size

15. Environment Variables

```bash
export OMP_NUM_THREADS=4
```

Or in code:

```c
omp_set_num_threads(4);
```

16. OpenMP vs MPI

Feature	OpenMP	MPI
Memory	Shared	Distributed
Programming	Easy	Complex
Performance	Medium	High
Use case	Multicore CPUs	Clusters

17. Real-World Use Cases

- Image processing

- Matrix multiplication
- Scientific simulations
- Machine learning preprocessing
- Financial modeling

18. Complete Example: Vector Addition

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

int main() {
    int n = 1000;
    int a[n], b[n], c[n];

    for(int i=0;i<n;i++) {
        a[i]=i;
        b[i]=i*2;
    }

    #pragma omp parallel for
    for(int i=0;i<n;i++)
        c[i] = a[i] + b[i];

    printf("Completed\n");
    return 0;
}
```

Copy code

19. Advantages of OpenMP

✓ Easy to learn
✓ Portable
✓ Incremental parallelism
✓ Scales with cores

---

20. Limitations of OpenMP

❌ Only shared memory
❌ Limited for large clusters
❌ Debugging can be hard

21. Exam-Ready Summary

- OpenMP is **shared-memory parallel programming API**
- Uses **compiler directives**
- Based on **fork-join model**
- Supports **parallel loops, sections, synchronization**
- Best for **multicore systems**

Problem 1: Number of Threads & Output Count

Code:

```c
#pragma omp parallel
{
    printf("Hello\n");
}
```

Given:

- `OMP_NUM_THREADS = 8`

Question:

How many times will **Hello** be printed?

Solution:

- `parallel` creates **8 threads**
- Each thread executes `printf`

✅ Answer:

```bash
Hello printed = 8 times
```

Problem 2: Fork–Join Execution Count

Code:

```c
printf("A\n");

#pragma omp parallel
{
    printf("B\n");
}

printf("C\n");
```

Given:

- 4 threads

Question:

How many times A, B, C printed?

Solution:

Statement	Execution
A	1 time (serial)
B	4 times (parallel)
C	1 time (serial)

✅ Answer:

```ini
A = 1, B = 4, C = 1
```

Problem 4: Loop Iterations Distribution (Static)

Code:

```c
#pragma omp parallel for schedule(static)
for(int i=0;i<8;i++)
    printf("T%d → i=%d\n", omp_get_thread_num(), i);
```

Given:

- 4 threads

Solution:

Static divides iterations **equally**

Thread	Iterations
T0	0,1
T1	2,3
T2	4,5
T3	6,7

Problem 5: Race Condition (Sum)

Code:

```c
int sum = 0;

#pragma omp parallel for
for(int i=1;i<=100;i++)
    sum += i;
```

Question:

Correct output?

❌ **Wrong:**

- Race condition
- Multiple threads update `sum`

Correct Solution:

```c
#pragma omp parallel for reduction(+:sum)
```

✅ **Correct Output:**

```ini
sum = 5050
```

Problem 6: Speedup Calculation (Amdahl's Law)

Given:

- 80% code is parallel
- 4 processors

Formula:

$$Speedup = \frac{1}{(1 - P) + \frac{P}{N}}$$

Solution:

$$= \frac{1}{0.2 + \frac{0.8}{4}} = \frac{1}{0.2 + 0.2} = \frac{1}{0.4}$$

Answer:

```ini
Speedup = 2.5
```

Copy code

Problem 7: Efficiency

Formula:

$$Efficiency = \frac{Speedup}{Processors}$$

Solution:

$$= \frac{2.5}{4} = 0.625$$

✅ **Answer:**

```ini
Efficiency = 62.5%
```

Copy code

Problem 8: Reduction Variable Count

Code:

```c
int x = 10;

#pragma omp parallel reduction(+:x)
{
    x++;
}
```

Copy code

Given:

- 4 threads

Explanation:

- Each thread gets private copy of `x = 10`
- Each increments once → `11`
- Final sum: `11 + 11 + 11 + 11`

✅ Answer:

```ini
x = 44
```

Copy code

Problem 9: Critical Section Count

Code:

```c
int count = 0;

#pragma omp parallel num_threads(6)
{
    #pragma omp critical
    count++;
}
```

Solution:

- Each thread increments once
- Critical avoids race

✅ Answer:

```ini
count = 6
```

Problem 10: Barrier Execution

Code:

```c
#pragma omp parallel num_threads(4)
{
    printf("A\n");
    #pragma omp barrier
    printf("B\n");
}
```

Question:

Can B print before all A?

✅ Answer:

```objectivec
NO
```

- Barrier forces all threads to finish A before B

Problem 11: Ordered Directive

Code:

Code:

```c
#pragma omp parallel for ordered
for(int i=0;i<5;i++) {
    #pragma omp ordered
    printf("%d ", i);
}
```

Output:

```
0 1 2 3 4
```

Problem 12: Scheduling (Dynamic)

Code:

```c
#pragma omp parallel for schedule(dynamic,1)
for(int i=0;i<6;i++)
    printf("T%d i=%d\n", omp_get_thread_num(), i);
```

Explanation:

- Iterations assigned **one by one**
- Output order unpredictable

🔥 Exam Tip (Very Important)

Most common exam questions:

- Count outputs
- Identify race condition
- Apply **reduction**
- Calculate speedup & efficiency
- Thread ID ranges

- Scheduling behavior