OpenMP

compiled by

Dr. N.Gopalakrishna Kini
Professor, School of Computer Engg.
Manipal Institute of Technology,
Manipal Academy of Higher Education, Manipal

OpenMP: stands for Open source software for Multi-Processing. Multi-Processing means multi-threading.

OpenMP is a directive-based Application Programming Interface (API) for developing parallel programs on shared memory architectures.

OpenMP is a API library which can be used with any programming languages such as C, C++, Fortran.

OpenMP is a shared memory model also called as Fork-Join Model, used to create multiple threads and these multiple threads share the memory of main process.

OpenMP is developed by OpenMP Architecture Review Board (OpenMP ARB). This is an organization formed by multiple companies namely Intel, IBM, HP, Nvidia, Oracle, AMD etc.

Each process start with one main thread called Master thread in OpenMP.
For a particular block of code, we create multiple threads along with this master thread. These extra threads are called to be slave threads.

Initially only the Master thread exists at the start of the process.
At some part of the program we require multithreading.
So we create slave threads. Once slave threads complete their working then they give their result back to the master thread.
Again the process continues with the single thread execution.
Note: Different processes have their own memory while different threads of single process share the memory of main process.

We create the multiple threads using FORK system call and then all these threads give their result to the master thread using JOIN system call. i.e. FORK-JOIN Model.

Use omp.h as header file in C / C++.

#pragma omp parallel //#pragma is Pre-processing directive or compiler directive

```
{
printf("Hello World");
}
```

#pragma omp parallel will create multiple threads and number of threads created is equal to number of processor cores.

Eg: i5 processor / i7 CPU then they have 4 CPU cores in it by default.

So the statement #pragma omp parallel will create 4 threads.

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

Will create 7 threads irrespective of CPU cores.

What are all the statements enclosed in { } will be executed simultaneously by 7 threads in this example.

```
1 #include<stdio.h>
2 #include<omp.h>
3 void main()
4 {
5 int a[5]={1,2,3,4,5};
6 int b[5]={6,7,8,9,10};
7 int c[5];
8 int tid;
9
10 #pragma omp parallel num_threads(5)
11 {
12 tid=omp_get_thread_num();
13 c[tid]=a[tid]+b[tid];
14 printf("c[%d]=%d\n",tid,c[tid]);
15 }
16
17 }
```

Here with parallel addition, each corresponding elements of the arrays a and b are added parallelly or simultaneously one unit of time.

omp_get_thread_num() is a function to be called to retrieve the ID of thread.
Remember the ID will be from 0 to n-1 where n is the total number of threads we create.
So in this example, first thread will have thread ID 0, second thread will have thread ID 1, third will have 2, etc…
Here this example, 0 to 4 (total 5 threads) will execute the same code by operating on different data elements of the arrays a and b.
IIIrly,
omp_get_num_procs():which returns number of CPUs in the multiprocessor system.
omp_get_num_threads():which returns number of threads active in current parallel region.
omp_get_thread_num() is a function to be called to retrieve the ID of thread.
omp_set_num_threads: which allows you to fix the number of threads executing the parallel sections of code.

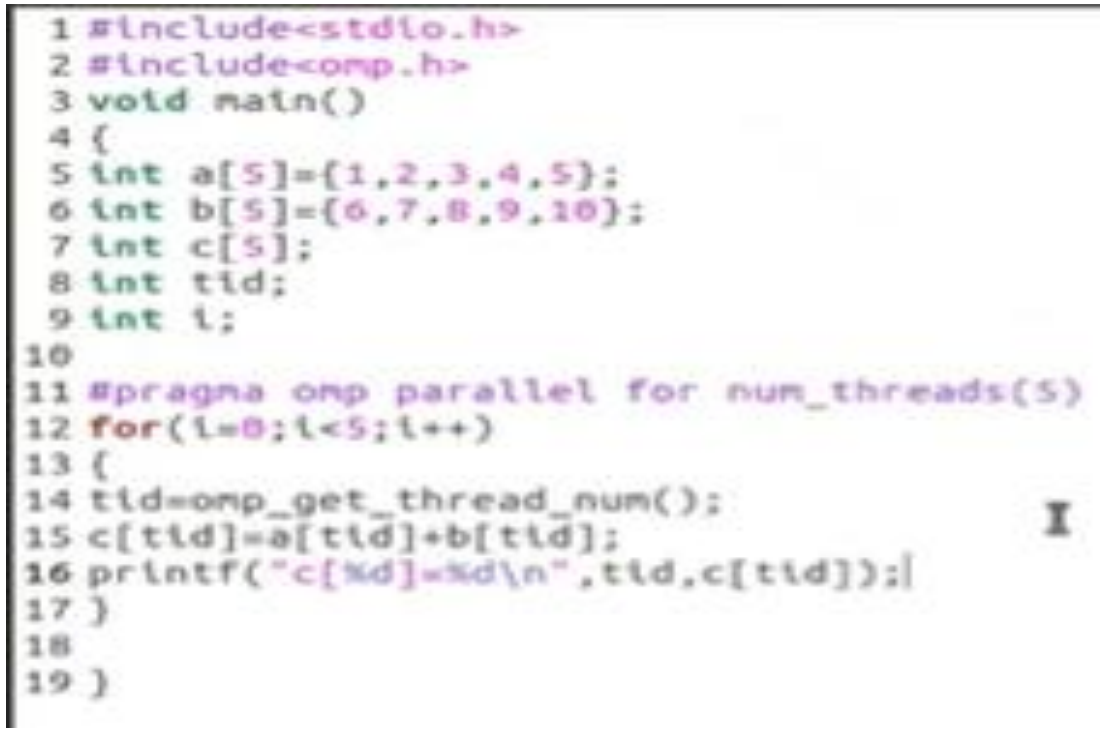
```
1 #include<stdio.h>
2 #include<omp.h>
3 void main()
4 {
5 int a[5]={1,2,3,4,5};
6 int b[5]={6,7,8,9,10};
7 int c[5];
8 int tid;
9
10 #pragma omp parallel num_threads(5)
11 {
12 tid=omp_get_thread_num();
13 c[tid]=a[tid]+b[tid];
14 printf("c[%d]=%d\n",tid,c[tid]);
15 }
16
17 }
```

Any thread can complete its execution at any time. So the order for each run can be different.

How to create multiple threads using “for” loop?

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

Note: In this for example all these 6 threads are executing parallelly or simultaneously the body of the for loop



```
#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]);
}

}
```

How to allocate different work to different threads?
Here we look at how different threads execute different tasks:

```
#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");
   }

}
```

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

void main()
{
  #pragma omp parallel sections num_threads(4)
   {
     #pragma omp section
      {
        int tid;
        tid=omp_get_thread_num();
        printf("X printed by thread with id=%d\n",tid);
      }
     #pragma omp section
      {
        int tid;
        tid=omp_get_thread_num();
        printf("Y printed by thread with id=%d\n",tid);
      }
     #pragma omp section
      {
        int tid;
        tid=omp_get_thread_num();
        printf("Z printed by thread with id=%d\n",tid);
      }
   }
}
```

Conclusion: Here we have created 3 tasks but mentioned 4 threads. Only any 3 threads out of 4 will get run.

How to synchronize multiple threads?

Critical section: is a block of code where common resources (variables) are shared by multiple threads.

There is race condition i.e. competition among threads to use these resources (variables) which may lead to the inconstancy.

To avoid such race condition, we use preprocessor directive “#pragma omp critical”

This makes only one thread to get exclusive access the common resource.

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

```
 1 #include<stdio.h>
 2 #include<omp.h>
 3
 4 void main()
 5 {
 6 int x=0;
 7
 8 #pragma omp parallel num_threads(300)
 9 {
10 x=x+1;
11 }
12
13 printf("x=%d\n",x);
14 }
```

```
1 #include<stdio.h>
2 #include<omp.h>
3
4 void main()
5 {
6 int x=0;
7
8 #pragma omp parallel num_threads(300)
9 {
10 #pragma omp critical
11 {
12 x=x+1;
13 }
14 }
15
16 printf("x=%d\n",x);
17 }
```

Now only one thread is allowed to execute the critical section. So answer will always will be 300.

How to use “Lock” in OpenMP?

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

void main()
{
int x=0;

omp_lock_t writelock;

omp_init_lock(&writelock);

#pragma omp parallel num_threads(300)
{
omp_set_lock(&writelock);
x=x+1;
omp_unset_lock(&writelock);
}

printf("x=%d\n",x);

omp_destroy_lock(&writelock);
}
```

Note:

1. omp_lock_t is data type used to declare lock variable.
2. Function omp_init_lock() is used to initialize lock. By default, lock is unset.
3. Function omp_set_lock() is used to set lock. Once any thread calls this function, lock is set and no other thread can access the critical section until calling thread calls omp_unset_lock().
4. Function omp_unset_lock() is used to unset lock. So that other threads can access critical section.
5. omp_destroy_lock() destroys the lock i.e. terminate the binding with lock variable.