PCYNLITX **Pcynlitx is an innovative open-source multi-thread computing platform**

Pcynlitx Platform has been developed by Erkam Murat Bozkurt.

A short Introduction to multi-threading with OpenMP:

Just like Pthreads, OpenMP is a multi-threading API that provides parallel execution of structured code blocks. The term structured code block will be explained in the later of this document. Although OpenMP is a multi-threading API for shared-memory programming, it have many fundamental differences with respect to other standard threading APIs such as Pthreads and std::threads. Pthreads or std::threads requires that the programmer explicitly specify the behavior of each thread. On the other hand, in OpenMP thread specific operations are much more difficult and some time impossible. However, OpenMP provides parallel execution of some particular computing task in a serially executed program. In below, a simple figure which depicts the program flows of both OpenMP and standard threading libraries is given.

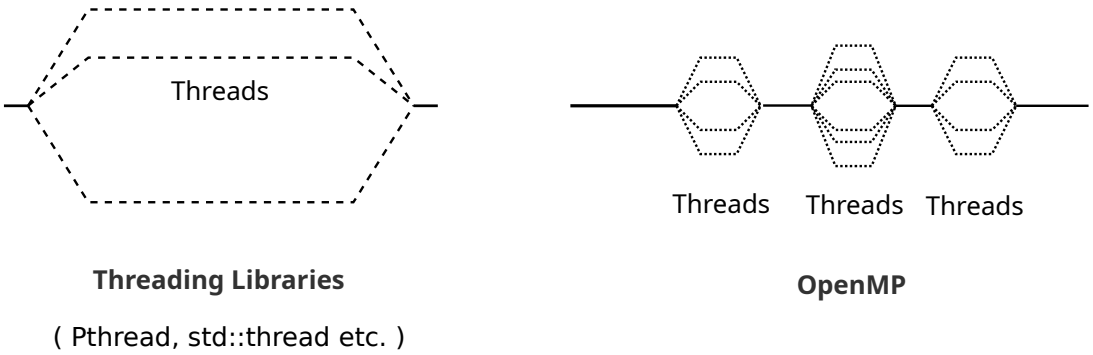

Figure 1.0 : The program flows of the multi-threading APIs.

In standard threading libraries, the programmers determine which thread will be perform which tasks and what will be the relation between the threads. Therefore, standard threading libraries are good for thread specific operations. For instance, let we assume that you want to design a text editor and you have to distribute some particular jobs between the threads. In that case, a thread may receive the commands or characters coming from the keyboard and at the same time, the other thread can record the characters into a file.

On the contrary, if you want to share some particular computing operations between the threads, OpenMP may be more practical. In that case, the programmer specifies which code block will be executed as parallel. For instance, let we assume that you want to distribute the computing operations performing in a for loop between the threads. With OpenMP, the only think which you have to do is to add a pre-processor command before this for loop. I think a simple example makes everything more clear. Compiling OpenMP applications with a GCC compiler is very easy. You can only need to add “-fopenmp” option to the standard gcc compiler commands.

```
g++ -fopenmp -o sample sample.cpp
```

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

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

    int total = 0;

    # pragma omp parallel for num_threads(4)

    for(int i=0;i<20;i++){

        total = total + 1;

        std::cout << "\n Thread Number :" << omp_get_thread_num()

        << ", total: " << total;
     }

     return 0;
}
```

Now, let we start to examine the sample program. On the headers section, “omp.h” header file is included. This heder file is the header file of the OpenMP library. The code line which starts “pragma” key word gives some message to the compiler. In C and C ++ , there are special pre-processor instructions known as pragmas. The pragmas is used in order to add some properties which are not standard C/C++ specifications. In most cases, these properties are hardware or platform depended. The pragma directives start with pound sign, # . If we use “omp” key word after the pragma key word, it indicates that this is an OpenMP directive. In the same code line, the next directive line is parallel. You might have guessed the key word parallel specifies that the **structured block of code** that follows should be executed by multiple threads.

A structured block is a C/C++ statement or a compound C/C++ statement with one point of entry and one point of exit. This definition simply prohibits code that branches into or out of the middle of the structured block. The next key word is “for” which indicates that the following code block is a for loop. Finally, in the same code line, there is also another key word “num_threads( )” determines the number of the thread to be created. In addition, “omp_get_thread_num( )” function is an OpenMP function which returns the number of the thread calling. The output of the program is shown in below.

```
 Thread Number :0, total: 3
 Thread Number :3, total: 4
 Thread Number :3, total: 5
 Thread Number :3, total: 6
 Thread Number :3, total: 7
 Thread Number :3, total: 7
 Thread Number :2, total: 8
 Thread Number :2, total: 9
 Thread Number :2, total: 10
 Thread Number :2, total: 11
 Thread Number :2, total: 11
 Thread Number :0, total: 12
 Thread Number :0, total: 13
 Thread Number :0, total: 14
 Thread Number :0, total: 15
 Thread Number :0, total: 15
 Thread Number :1, total: 16
 Thread Number :1, total: 17
 Thread Number :1, total: 18
 Thread Number :1, total: 19
```

However, the execution order of the threads is determined by the operating systems. Therefore, in each running of the program, the execution order of the threads changes.

OpenMP Directives Format for C/C++

#pragma omp directive-name [clause, ...] newline

Explanation of the directive format:

#pragma omp : Required for all OpenMP C/C++ directives.

directive-name: A valid OpenMP directive. Must appear after the pragma and before any clauses.

[clause, ...] : Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.

newline: Required. Precedes the structured block which is enclosed by this directive.

General Rules:

- Case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

What is the properties of the variables defined in parallel section ?

In a parallel section variables can be private (each thread owns a copy of the variable) or shared among all threads. Shared variables must be used with care because they cause race conditions.

Shared variables: the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.

Private variables: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private. Example is given in below.

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

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

    int Thread_ID = 0;

    # pragma omp parallel num_threads(4) private(Thread_ID)

    {
        Thread_ID = omp_get_thread_num();

        std::cout << "\n Thread Number :" << Thread_ID;
     }

     return 0;
}
```

In this code example, the braces are used in order to indicate the structured code block. In addition to this, private(Thread_ID) command specifies that each thread will have its own Thread_ID variable. The output of the program is shown in below.

Thread Number :3
Thread Number :0
Thread Number :2
Thread Number :1

Synchronization on OpenMP:

Atomic directive: Atomic directive is used in order to ensure the serialisation of a particular operation. Atomic directive can not be use for mutiple code line.

Critical sections: The usage of the critical directive prevents from the accessing of the multiple threads to the critical sections at the same time. You can serialize a whole of a code block with critical directive.

An example is given in below.

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

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

    int count = 0;

    int a[20], b[20];

    # pragma omp parallel for num_threads(4)

    for(int i=0;i<20;i++)
    {
        #pragma omp atomic
          count++;
    }

    # pragma omp parallel for num_threads(4)

    for(int i=0;i<20;i++){

        #pragma omp critical
        {
           a[i] = count + i;

           b[i] = count + i;
        }
    }

    std::cout << "\n\n count : " << count;

    for(int i=0;i<20;i++){

       std::cout << "\n a[" << i << "]:" << a[i];

       std::cout << "\n b[" << i << "]:" << b[i];
    }

    std::cout << "\n\n The end of the program ..\n\n";

    return 0;
}
```

In this code example, the command “count++” is executed separately. Only one thread can modify the “count” variable in a particular time interval. On the other hand, the operations “`a[i] = count + i;`” and “`b[i] = count + i;`” are executed serially as a code block.

Barier directive: all the threads wait for each other to reach the barrier. Simple example for the usage of barrier directive is given in below.

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

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

    int Thread_ID;

    # pragma omp parallel private(Thread_ID) num_threads(4)
    {

      Thread_ID = omp_get_thread_num();

      #pragma omp critical
      {
        std::cout << "\n The thread [" << Thread_ID << "] waits in here";
      }

      #pragma omp barrier

      #pragma omp critical
      {
         std::cout << "\n After barier, the thread [" << Thread_ID << "] prints to screen";
      }
    }

    std::cout << "\n\n The end of the program ..\n\n";

    return 0;
}
```

All threads passes from barier pragma together. The output of the program is shown in below.

```
The thread [0] waits in here
The thread [3] waits in here
The thread [1] waits in here
The thread [2] waits in here
After barier, the thread [1] prints to screen
After barier, the thread [3] prints to screen
After barier, the thread [0] prints to screen
After barier, the thread [2] prints to screen

The end of the program ..
```