Parallel Programming: OpenMP Lab 1

1 Hello, World!(Using OpenMP)

```cpp
#include <iostream>
#include <ctime>
#include <omp.h>
using namespace std;

void main(){

  clock_t t1, t2;
  double wt1, wt2;


  int threadsNo ,id;

  t1 = clock();
  wt1 = omp_get_wtime();

  #pragma omp parallel
  {
    threadsNo = omp_get_num_threads(); //returns the number of threads that has been spawned
    int id = omp_get_thread_num(); //returns the thread ID

    cout<<"Hello, World! from thread "<<id<<" (out of total "<<threadsNo<<" threads)"<<endl;
  }
  t2 = clock();
  wt2 = omp_get_wtime();


  cout << "CPU Time (in seconds) = " << (double)(t2 - t1) / CLOCKS_PER_SEC <<endl;
  cout << "wTime    (in seconds) = " << wt2 - wt1 << endl;

  system("pause");
}
```

Output

```
Hello, World! from thread 0 (out of total 4 threads)
Hello, World! from thread 3 (out of total 4 threads)
Hello, World! from thread 2 (out of total 4 threads)
Hello, World! from thread 1 (out of total 4 threads)
CPU Time (in seconds) = 0.029
wTime    (in seconds) = 0.0294827
```

- **First**, we talk about the function omp_get_wtime() which returns the time in seconds (as double), starting from some universal time (some number of years ago). You will typically see very large values (if you displayed the value of wt1 and wt2).

  wTime is the *elapsed real time*, *wall-clock time*, or *wall time* which measures the actual time taken from the start of a computer program to the end.

- **Second**, we talk about the function clock() (requires to include ctime library) which returns CPU clock time (measured in clock ticks) since the program is started (as clock_t, so we need to cast it to double).

  CPU time measures only the time during which the processor is actively working on a certain task, i.e. it does not include waiting for input/output (I/O) operations, entering low-power (idle) mode, or multitasking delays. *In contrast to the wTime.*
  CLOCKS_PER_SEC: *is the number of CPU clock ticks per second.*

- **Third**, We discuss the problems in the code above:

  1. **Problem 1 , Every thread is getting the value of the variable $ThreadsNo$, while it is enough to get its value by only one thread and share this value with the rest of threads.**

     Solution 1 , Using if statement

```cpp
void main(){

  int threadsNo ,id;

  #pragma omp parallel
  {
    id = omp_get_thread_num(); //returns the thread ID

    if(id == 0)
      threadsNo = omp_get_num_threads(); //returns the number of threads that has been spawned

        cout<<"Hello, World! from thread "<<id<<" (out of total "<<threadsNo<<" threads)"<<endl;
  }
}
```

Output

```
Hello, World! from thread Hello, World! from thread 2 (out of total -858993460 threads)
1 (out of total -858993460 threads)
Hello, World! from thread 3 (out of total -858993460 threads)
Hello, World! from thread 0 (out of total 4 threads)
CPU Time (in seconds) = 0.032
wTime    (in seconds) = 0.03279
```

Now this solution caused the following problem :
We said that only thread 0 will get the value of $threadNo$ and share it with thre rest of the threads but what happened (<u>in this run</u>) is the rest of the threads were scheduled before thread 0 so they didn't have the value of $threadNo$ so they displayed garbage. And this problem has 3 solutions :

<u>Solution 1.1</u> , Using barrier construct with the if statement

```
void main(){

  int threadsNo ,id;

  #pragma omp parallel
  {
    int id = omp_get_thread_num(); //returns the thread ID

    if(id == 0)
      threadsNo = omp_get_num_threads(); //returns the number of threads that has been spawned

    #pragma omp barrier //wait until all the threads hit this barrier and then continue
    cout<<"Hello, World! from thread "<<id<<" (out of total "<<threadsNo<<" threads)"<<endl;
  }
}
```

Output

```
Hello, World! from thread 3 (out of total 4 threads)
Hello, World! from thread 2 (out of total 4 threads)
Hello, World! from thread 1 (out of total 4 threads)
Hello, World! from thread 0 (out of total 4 threads)
CPU Time (in seconds) = 0.042
wTime    (in seconds) = 0.0431469
```

<u>Solution 1.2</u> , Making two separate parallel regions,

1) one for getting the number of threads using if statement, and
2) the second one for displaying the cout message

```
void main() {

  int threadsNo, id;
  #pragma omp parallel
  {
    id = omp_get_thread_num();

    if(id == 0)
      threadsNo = omp_get_num_threads();

  }/* end of first parallel region */

  #pragma omp parallel
  {
    cout<<"Hello, World! from thread "<<id<<" (out of total "<<threadsNo<<" threads)"<<endl;
  }/* end of second parallel region */
}
```

Output

```
Hello, World! from thread: 3 from 4
Hello, World! from thread: 3 from 4
Hello, World! from thread: 3 from 4
Hello, World! from thread: 3 from 4
```

Race Condition problem has happened. See Problem 2 for the solution in this case! [7]

Solution 1.3 , Using single construct instead of barrier and if together

```
void main(){

  int threadsNo ,id;

  #pragma omp parallel
  {
    int id = omp_get_thread_num(); //returns the thread ID

    //Any thread that is scheduled first get the number of threads and share it with the other threads
    #pragma omp single
    threadsNo = omp_get_num_threads(); //returns the number of threads that has been spawned

    cout<<"Hello, World! from thread "<<id<<" (out of total "<<threadsNo<<" threads)"<<endl;
  }
}
```

Output

```
Hello, World! from thread 2 (out of total 4 threads)
Hello, World! from thread 1 (out of total 4 threads)
Hello, World! from thread 0 (out of total 4 threads)
Hello, World! from thread 3 (out of total 4 threads)
CPU Time (in seconds) = 0.033
wTime    (in seconds) = 0.0332637
```

2. **A variable defined outside (inside) the parallel region is by default shared (private).**
Data race (Race Condition): is the condition when a (shared) variable is, *at the same time*, accessed by different threads, resulting in an undefined behavior.

Problem 2 , The variable id is shared among all the threads which could lead to a race condition.

```
#include <iostream>
#include <ctime>
#include <omp.h>
using namespace std;

void main(){

  clock_t t1, t2;
  double wt1, wt2;


  int threadsNo ,id;

  t1 = clock();
  wt1 = omp_get_wtime();

  #pragma omp parallel
  {
    id = omp_get_thread_num(); //returns the thread ID

    for(int i = 0; i < 100000; i++); //long wait so that other threads get scheduled (to make the race condition happen "almost" every time)

    #pragma omp single
    threadsNo = omp_get_num_threads(); //returns the number of threads that has been spawned

    cout<<"Hello, World! from thread "<<id<<" (out of total "<<threadsNo<<" threads)"<<endl;
  }
  t2 = clock();
  wt2 = omp_get_wtime();


  cout << "CPU Time (in seconds) = " << (double)(t2 - t1) / CLOCKS_PER_SEC <<endl;
  cout << "wTime    (in seconds) = " << wt2 - wt1 << endl;


  system("pause");
}
```

Output

```
Hello, World! from thread 3 (out of total 4 threads)
Hello, World! from thread 3 (out of total 4 threads)
Hello, World! from thread 3 (out of total 4 threads)
Hello, World! from thread 3 (out of total 4 threads)
CPU Time (in seconds) = 0.031
wTime    (in seconds) = 0.0311536
```

Solution 2 , Making the variable *id* private by either defining it inside the parallel region OR using the private (for *id*) and shared (if we want) clauses

```
void main(){

  int threadsNo ,id;

  #pragma omp parallel private(id)
  {
    id = omp_get_thread_num();

    for(int i = 0; i < 100000; i++); //long wait so that other threads get scheduled (to make the race condition happen "almost every time")

    #pragma omp single
    threadsNo = omp_get_num_threads();

    cout<<"Hello, World! from thread "<<id<<" (out of total "<<threadsNo<<" threads)"<<endl;
  }
}
```

Output

```
Hello, World! from thread 3 (out of total 4 threads)
Hello, World! from thread 0 (out of total 4 threads)
Hello, World! from thread 1 (out of total 4 threads)
Hello, World! from thread 2 (out of total 4 threads)
CPU Time (in seconds) = 0.032
wTime    (in seconds) = 0.0334435
```

In case making two Parallel Regions, Using the threadprivate clause globally (for *id*) after defining the *id* variable.

threadprivate : Specifies that variables are replicated, with each thread having its own copy.
Each copy of a threadprivate variable is initialized once prior to the first reference to that copy.

```
int id;
#pragma omp threadprivate(id)  //initialize id varaible for every thread that will be spawned
/*
    to solve repeating getting id command in both parallel regions

  it must be used globally, i.e. not inside the main function or any other function
*/
void main() {

  int threadsNo;
  #pragma omp parallel
  {
    id = omp_get_thread_num(); //first reference of the id variable copy for each thread

    if(id == 0)
      threadsNo = omp_get_num_threads();

  }/* end of first parallel region */

  #pragma omp parallel
  {
    cout<<"Hello, World! from thread: "<<id<<" from "<<threadsNo<<endl;
  }/* end of second parallel region */
}
```

Output

```
Hello, World! from thread: 0 from 4
Hello, World! from thread: 2 from 4
Hello, World! from thread: 3 from 4
Hello, World! from thread: 1 from 4
```

But making more than one parallel region is not a desirable thing because many forking and joining are occurred.

So the best code is:

```cpp
#include <iostream>
#include <ctime>
#include <omp.h>
using namespace std;

void main(){

  clock_t t1, t2;
  double wt1, wt2;

  int threadsNo ,id;

  t1 = clock();
  wt1 = omp_get_wtime();
  #pragma omp parallel num_threads(6) default(shared) private(id)
  {
    id = omp_get_thread_num();

    for(int i = 0; i < 100000; i++); //long wait so that other threads get scheduled (to make the race condition happen "almost" every time)

    #pragma omp single
    threadsNo = omp_get_num_threads();

    cout<<"Hello, World! from thread "<<id<<" (out of total "<<threadsNo<<" threads)"<<endl;
  }
  t2 = clock();
  wt2 = omp_get_wtime();


  cout << "CPU Time (in seconds) = " << (double)(t2 - t1) / CLOCKS_PER_SEC <<endl;
  cout <<"wTime     (in seconds) = " << wt2 - wt1 << endl;

  system("pause");
}
```

Output

```
Hello, World! from thread 5 (out of total 6 threads)
Hello, World! from thread 4 (out of total 6 threads)
Hello, World! from thread 2 (out of total 6 threads)
Hello, World! from thread 0 (out of total 6 threads)
Hello, World! from thread 3 (out of total 6 threads)
Hello, World! from thread 1 (out of total 6 threads)
CPU Time (in seconds) = 0.044
wTime    (in seconds) = 0.0447828
```

__Note:__

- num_threads clause is used to determine the number of threads to be spawned in the parallel region.
- shared clause is used to explicitly say that a specific variable is shared among all the threads.
- default clause is used to determine explicitly whether the default (i.e. if it is not explicitly determined) data-sharing attributes of the variables is private or shared.
- You can write as many clauses as you need and separate them by *space* or *comma*.
  For example: in the above program you can write:

  #pragma omp parallel num_threads(6) default(shared) private(id)

  #pragma omp parallel num_threads(6), default(shared), private(id)

2 Sum of an array

2.1 Sequential

```cpp
#include <iostream>
#include <iomanip>
#include <ctime>
#include <omp.h>
using namespace std;

const int n = 100000000;

void main() {
  clock_t t1,t2;
  double wt1, wt2;
  int *A, sum = 0;
  A = new int[n];

  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  t1 = clock();
  wt1 = omp_get_wtime();
  for(int i = 0; i < n; i++){
    sum += A[i];
  }
  t2 = clock();
  wt2 = omp_get_wtime();

  cout << setprecision(10);
  cout << "sum                 = " << sum << endl;
  cout << "CPU Time (in seconds) = " << (double)(t2 - t1) / CLOCKS_PER_SEC <<endl;
  cout << "wTime    (in seconds) = " << wt2 - wt1 << endl;

  system("pause");
}
```

Output

```
sum                   = 100000000
CPU Time (in seconds) = 0.335
wTime    (in seconds) = 0.3352936
```

2.2 Parallel

2.2.1 First attempt

```cpp
void main() {

  int *A, sum = 0, mySum;
  A = new int[n];
  //initializing the array with ones
  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  int threadsNo = 4;

  #pragma omp parallel num_threads(threadsNo)
  {
    #pragma omp single
    cout<<"Using "<<threadsNo<<" threads:"<<endl;

    for(int i = 0; i < n; i++){
      sum += A[i];
    }
  }
}
```

Output

```
Using 4 threads:
sum                   = 108732882
CPU Time (in seconds) = 1.52
wTime    (in seconds) = 1.5197758
```

The sum is wrong because every thread that has been spawned, *according to the use of #pragma omp parallel construct*, will execute the for loop from $i = 0$ to $i = n - 1$ (i.e. every thread did all the work that was supposed to be divided among the threads). Note that, the time is more than the sequential version. So, we need a way to divide the work among the threads.

2.2.2 Second attempt

We divide the array entries among the threads using each thread *id* and make each thread add *only* its own entries to the shared variable *sum*.

```
void main() {

  int *A, sum = 0;
  A = new int[n];
  //initializing the array with ones
  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  int threadsNo = 4;
  double len;

  #pragma omp parallel num_threads(threadsNo)
  {
    #pragma omp single
    {
        cout<<"Using "<<threadsNo<<" threads:"<<endl;
        len = (double)n / threadsNo;
    }
    int id = omp_get_thread_num(); //returns the thread ID
    int from = len * id;
    int to = from + len;

    for(int i = from; i < to; i++){
      sum += A[i];
    }
  }
}
```

Output

```
Using 4 threads:
sum                        = 28327690
CPU Time (in seconds) = 0.443
wTime      (in seconds) = 0.4435848
```

Again the sum is wrong. Why this time?
This is because the *race condition* on the shared variable *sum*.
The variable *sum* must be kept shared, so how we can solve the race condition problem on a shared variable?!

2.2.3 Third attempt

We can use one of the following two construct

1. **atomic**: *Ensures a specific storage location is accessed atomically.*
   the word *atom* has a Greek origin that means *indivisible*. So, an atomic operation is the operation that is indivisible by any other operation (i.e. no other operation can happen within an atomic operation). In concurrent programming, it is said that an atomic process runs completely independently of any other processes.

   atomic is used to solve the race condition only in the case it happens in one of the following statements:

   - **Update:** like
     $x++$, $x--$, $++x$, $--x$, x bin_op= $expr$, $x = x$ bin_op $expr$
   - **read:** v = x
   - **write:** x = expr

     where:
     bin_op is one of the following binary operators ($+, *, -, /, \&, \hat{}, |, <<, >>$), and
     expr is an expression of scalar type that does not reference the variable x.

2. **critical**: *Restricts execution of the associated structured block to a single thread at a time.*

   critical is used with all kinds of statemets.

   In our program here, both *atomic* and *critical* can be used since the *race condition* problem is in the statement

$$sum+ = A[i]$$

```
void main() {

  int *A, sum = 0, mySum;
  A = new int[n];
  //initializing the array with ones
  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  int threadsNo = 4;
  double len;

  #pragma omp parallel num_threads(threadsNo)
  {
```

```
    #pragma omp single
    {
        cout<<"Using "<<threadsNo<<" threads:"<<endl;
        len = (double)n / threadsNo;
    }
    int id = omp_get_thread_num(); //returns the thread ID
    int from = len * id;
    int to = from + len;

    for(int i = from; i < to; i++){
      #pragma omp critical
      //OR
      //#pragma omp atomic
      sum += A[i];
    }
  }
}
```

Output

```
Using 4 threads:
sum                      = 100000000
CPU Time (in seconds)    = 25.065
wTime    (in seconds)    = 25.0642369
```

Now the sum is correct. But why does it take such a long time?!
This is because when using critical or atomic constructs on the variable sum, it locks, updates, and unlocks the variable sum. This process happens as many as the entries of the array (100000000 times!) which is very time consuming.

2.2.4 Fourth attempt

We make a private variable $mySum$ for each thread to calculate the summation of the entries of this thread instead of summing directly into the variable sum. Then, we add the distinct $mySum$ variables to the shared variable sum with using critical or atomic constructs to avoid the race condition problem.
Now the **lock-update-unlock** process on the variable sum only happens as many as the number of threads (Here, 4 times!).

```
void main() {

  int *A, sum = 0, mySum;
  A = new int[n];
  //initializing the array with ones
  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  int threadsNo = 4;
  double len;

  #pragma omp parallel num_threads(threadsNo) private(mySum)
  {
    #pragma omp single
    {
        cout<<"Using "<<threadsNo<<" threads:"<<endl;
        len = (double)n / threadsNo;
    }
    int id = omp_get_thread_num(); //returns the thread ID
    int from = len * id;
    int to = from + len;

    mySum = 0;
    for(int i = from; i < to; i++){
      mySum += A[i];
    }

    #pragma omp critical
    //OR
    //#pragma omp atomic
    sum += mySum;
  }
}
```

Output

```
Using 4 threads:
sum                      = 100000000
CPU Time (in seconds)    = 0.127
wTime    (in seconds)    = 0.1267415
```

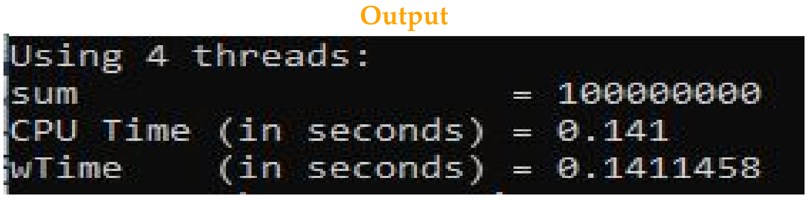
Great!
But OpenMP could do better...

2.2.5 Fifth attempt

OpenMP provides its own for loop *#pragma omp for* that is written above the for loop in your program and OpenMP does the job of dividing the data among the threads. So, there is no need for the variables $from$ and to and the for loop will go again from $i = 0$ to $i = n$

```
void main() {

  int *A, sum = 0, mySum;
  A = new int[n];
  //initializing the array with ones
  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  int threadsNo = 4;

  #pragma omp parallel num_threads(threadsNo) private(mySum)
  {
    #pragma omp single
    cout<<"Using "<<threadsNo<<" threads:"<<endl;

    mySum = 0;
    #pragma omp for
    for(int i = 0; i < n; i++){
      mySum += A[i];
    }
    #pragma omp critical
    //OR
    //#pragma omp atomic
    sum += mySum;
  }
}
```

Output



2.2.6 Sixth attempt

We introduce reduction clause.
Reduction means reducing a long expression in the right hand side of an equation into only one variable in the left hand side of it. such as:

$$sum = A[0] + A[1] + A[2] + \ldots + A[n-1] + A[n]$$

The expression $A[0] + A[1] + A[2] + \ldots + A[n-1] + A[n]$ is reduced into the variable sum.

- The reduction clause is written as follows:

  $$reduction(op : var)$$

  where:
  op is the operation of reduction and it is only one of the these operations ($+$, $-$, $*$, max, min).
  var is the variable in which the reduction will happen.
  So in our example we write:

  $$reduction(+ : sum)$$

- The reduction clause is written next to:

  #pragma omp parallel , then it will return its result (var) at the end of the parallel region, or

  #pragma omp for , then it will return its result (var) at the end of loop region.

- The reduction clause works (we are talking about summation) as follows:
  It does all the work that we have done until now:
  - it defines a private partial sum $mySum$ for every thread,
  - adds the entries of each thread to its partial sum variable,
  - adds the partial sums to the shared variable sum, and
  - prevents any race condition from happening on the shared variable.

```
void main() {

  int *A, sum = 0;
  A = new int[n];
  //initializing the array with ones
  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  int threadsNo = 4;

  #pragma omp parallel num_threads(threadsNo) reduction(+:sum)
  {
    #pragma omp single
    cout<<"Using "<<threadsNo<<" threads:"<<endl;

    #pragma omp for
    for(int i = 0; i < n; i++){
      sum += A[i];
    }
  }
}
```

Output

```
Using 4 threads:
sum                        = 100000000
CPU Time (in seconds) = 0.151
wTime      (in seconds) = 0.1520898
```

OR

```
void main() {

  int *A, sum = 0;
  A = new int[n];
  //initializing the array with ones
  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  int threadsNo = 4;

  #pragma omp parallel num_threads(threadsNo)
  {
    #pragma omp single
    cout<<"Using "<<threadsNo<<" threads:"<<endl;

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

Output

```
Using 4 threads:
sum                        = 100000000
CPU Time (in seconds) = 0.15
wTime      (in seconds) = 0.1505957
```

It seems that our program only has a for loop and nothing important more is going on in the parallel region. So, we can write #pragma omp parallel for and then write the for loop below it directly as follows:

```
void main() {

  int *A, sum = 0;
  A = new int[n];
  //initializing the array with ones
  for(int i = 0; i < n; i++){
    A[i] = 1;
  }

  int threadsNo = 4;

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

Output

```
sum                        = 100000000
CPU Time (in seconds) = 0.139
wTime      (in seconds) = 0.1385489
```