Lab 3: OpenMP

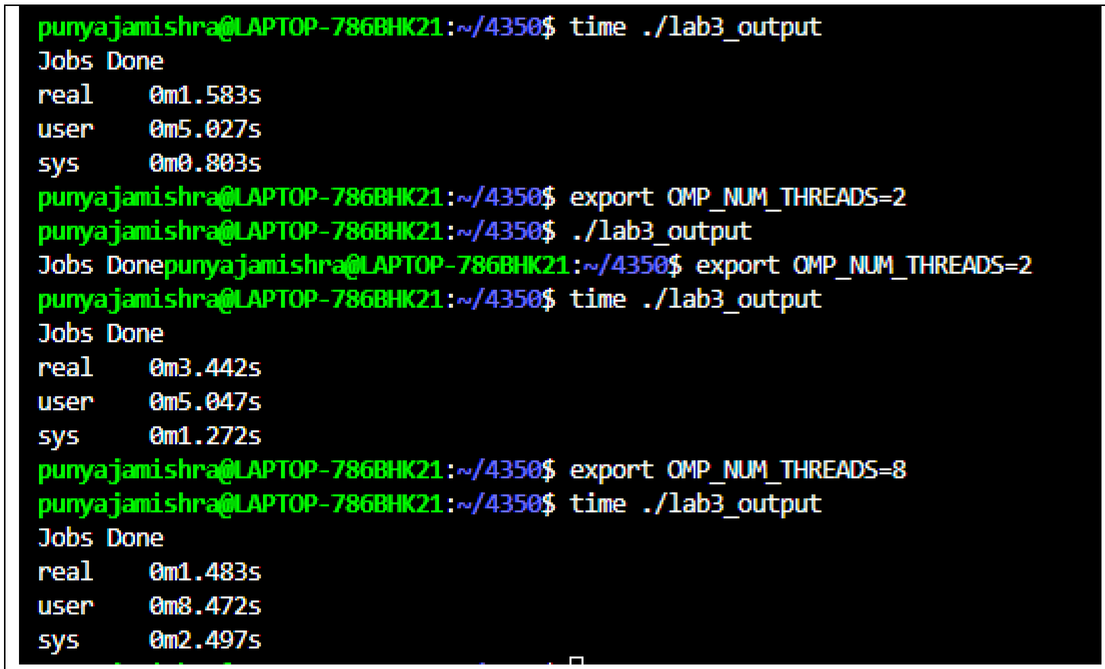
Open MP is a shared memory multithreaded programming API, it supports C/++ and Fortran.

Submission box:

Paste your single threaded/multithreaded code for the sin arrays

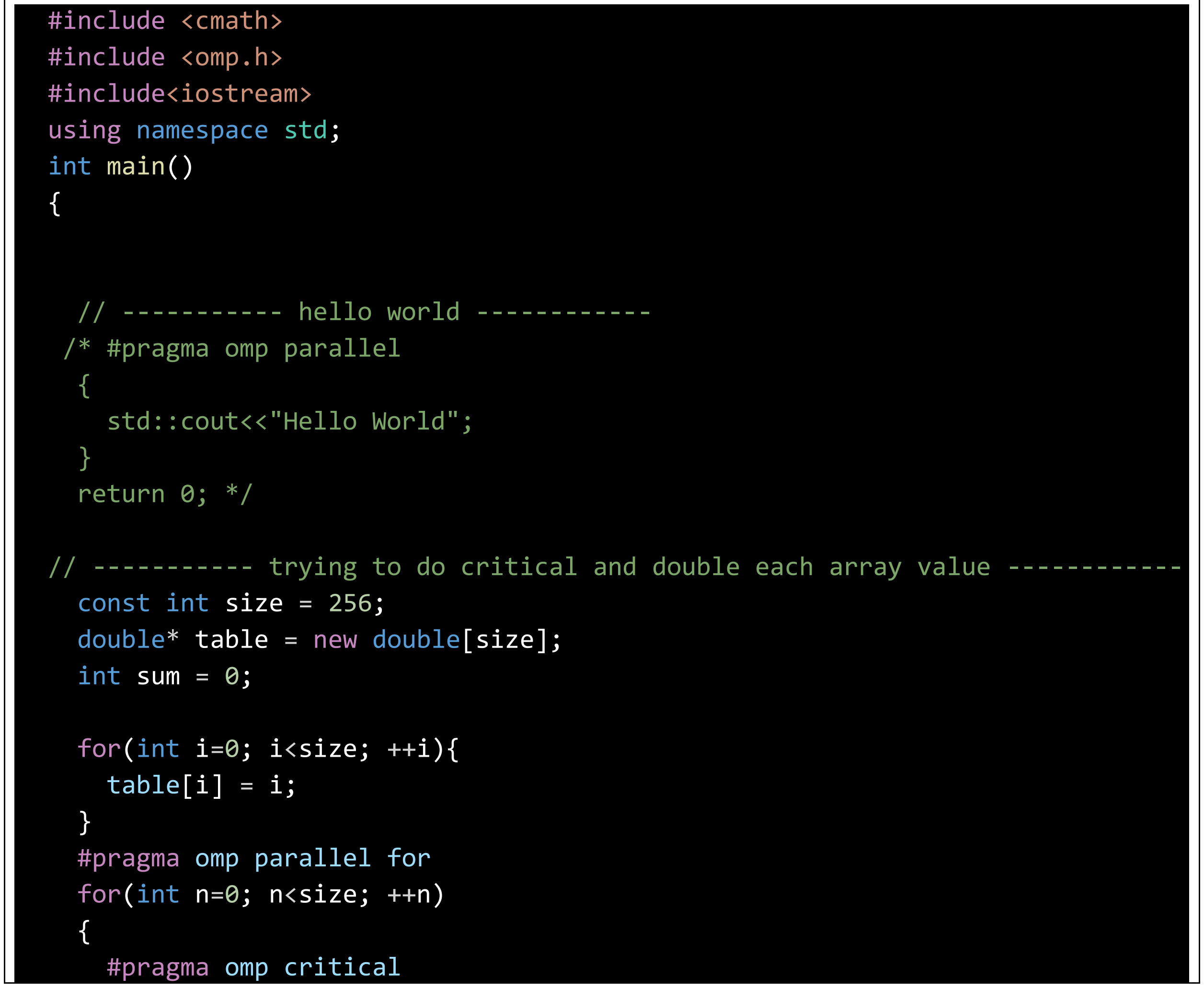
```
#include <cmath>
#include <omp.h>
#include<iostream>
using namespace std;
int main()
{


  // ----------- hello world ------------
 /* #pragma omp parallel
  {
    std::cout<<"Hello World";
  }
  return 0; */

// ----------- sin array with different thread numbers ------------
  const int size = 262144;
  double* sinTable;

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
  {
     sinTable[n] = std::sin(2*M_PI*n/size);

  }


  cout << "Jobs Done \n";


  return 0;

  // the table is now initialized

}
```

The testing of the timings - num of threads 4, 2, 8 – in this order



```
punyajamishra@LAPTOP-786BHK21:~/4350$ time ./lab3_output
Jobs Done
real    0m1.583s
user    0m5.027s
sys     0m0.803s
punyajamishra@LAPTOP-786BHK21:~/4350$ export OMP_NUM_THREADS=2
punyajamishra@LAPTOP-786BHK21:~/4350$ ./lab3_output
Jobs Donepunyajamishra@LAPTOP-786BHK21:~/4350$ export OMP_NUM_THREADS=2
punyajamishra@LAPTOP-786BHK21:~/4350$ time ./lab3_output
Jobs Done
real    0m3.442s
user    0m5.047s
sys     0m1.272s
punyajamishra@LAPTOP-786BHK21:~/4350$ export OMP_NUM_THREADS=8
punyajamishra@LAPTOP-786BHK21:~/4350$ time ./lab3_output
Jobs Done
real    0m1.483s
user    0m8.472s
sys     0m2.497s
```

Your code for the array doubling and sum



```
#include <cmath>
#include <omp.h>
#include<iostream>
using namespace std;
int main()
{


  // ----------- hello world ------------
 /* #pragma omp parallel
  {
    std::cout<<"Hello World";
  }
  return 0; */

// ----------- trying to do critical and double each array value ------------
  const int size = 256;
  double* table = new double[size];
  int sum = 0;

  for(int i=0; i<size; ++i){
    table[i] = i;
  }
  #pragma omp parallel for
  for(int n=0; n<size; ++n)
  {
    #pragma omp critical
```

```
      {
        sum = sum + 2*table[n];
      }
    }

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

    return 0;

    // the table is now initialized

  }
```

Just paste all that in this box and upload this to BB. If you are a mac user make sure it's a docx or pdf not a .pages file

Commands

Probably not required

Note: I don't know if you need this, but you might, I needed it for something unrelated to this lab, but it might do something, I'll leave it as a note to us in case something doesn't work with gcc versions.

```
sudo yum -y install gcc-c++
```

I'll leave it here as a note to self in case something breaks

I did test this all Jan 24/2023, but my WSL installation is 3 years old and works, so god knows what doesn't.

OpenMP on Mac

for Mac users, CLANG/CLANG++ DO NOT SUPPORT OPENMP.
To fix this, do "brew install gcc" in your terminal, then go to /usr/local/bin, run PATH=/usr/local/bin:$PATH, check your GCC version within /usr/local/bin using ls or similar (for me it was gcc-11), then go back to your directory you have your openmp program in and compile it by DIRECTLY CALLING YOUR VERSION OF GCC, ie: gcc-11 -fopenmp -o (outputfilename) (inputfilename).

In addition, note that while GCC CAN compile c++, it will need to be pointed to the standard c++ libraries using the command line option -lstdc++ (source: https://stackoverflow.com/questions/3178342/compiling-a-c-program-with-gcc)

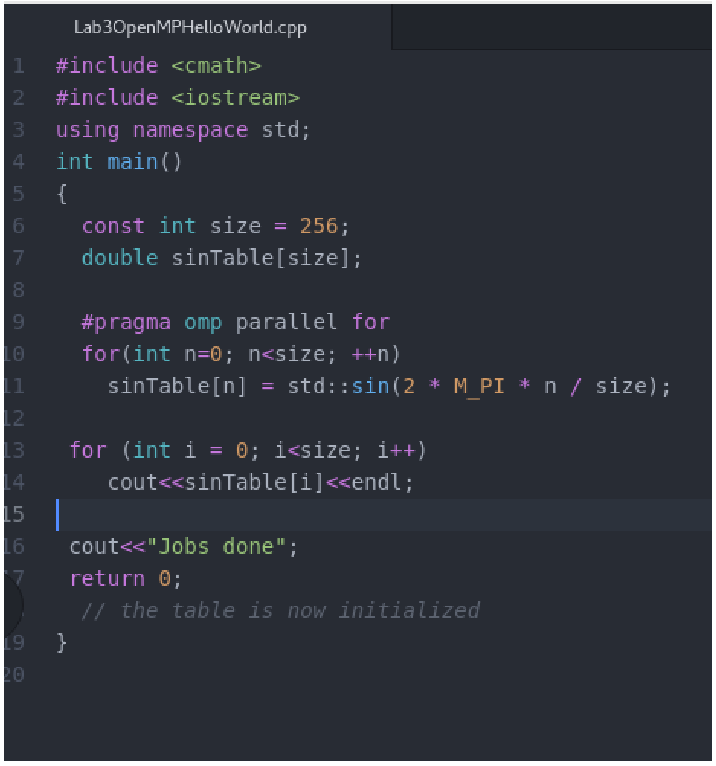
Make a Table of Numbers in OpenMP

There is a Hello World in OpenMP, we'll get there in a sec, but first we should see something simpler. The first thing to do is to parallise a trivial task.

```
#include <cmath>
#include <omp.h>
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

  // the table is now initialized
}
```

Note this program has no output or input and doesn't really do much but run. The big thing is that it does run, because if omp.h doesn't work, you're not getting anything else done.

A slightly better program that will show you what it's doing (you need to type that out, but it's mostly the same as your current program). Note that I jump to C++ but for this simple example it doesn't matter **Note: in the code below I forgot to include <omp.h>**


Lab3OpenMPHelloWorld.cpp

```
#include <cmath>
#include <iostream>
using namespace std;
int main()
{
  const int size = 256;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

 for (int i = 0; i<size; i++)
    cout<<sinTable[i]<<endl;

 cout<<"Jobs done";
 return 0;
  // the table is now initialized
}
```

(Remember to add an #include <omp.h>)

All that does is it make a 256-element array.

Build that, compile and run.

But how do you know it’s working in parallel?

Well, you don’t, at least not yet.

But let’s see an example where you do

Use a (linux) terminal to run your program you need to set an environment variable (I think to make this you use export or “set OMP…” but I’m not sure which did it):

`export OMP_NUM_THREADS=<number of threads to use>`.

In my case I set it equal to 4.

Hello World In Omp

```
#include <iostream>
#include <omp.h>
int main()
{
  #pragma omp parallel
  {
    std::cout<<"Hello World";
  }

  return 0;
}
```

Commands to compile are on the next page but it's

g+ + inputfile – o outputfile -fopenmp

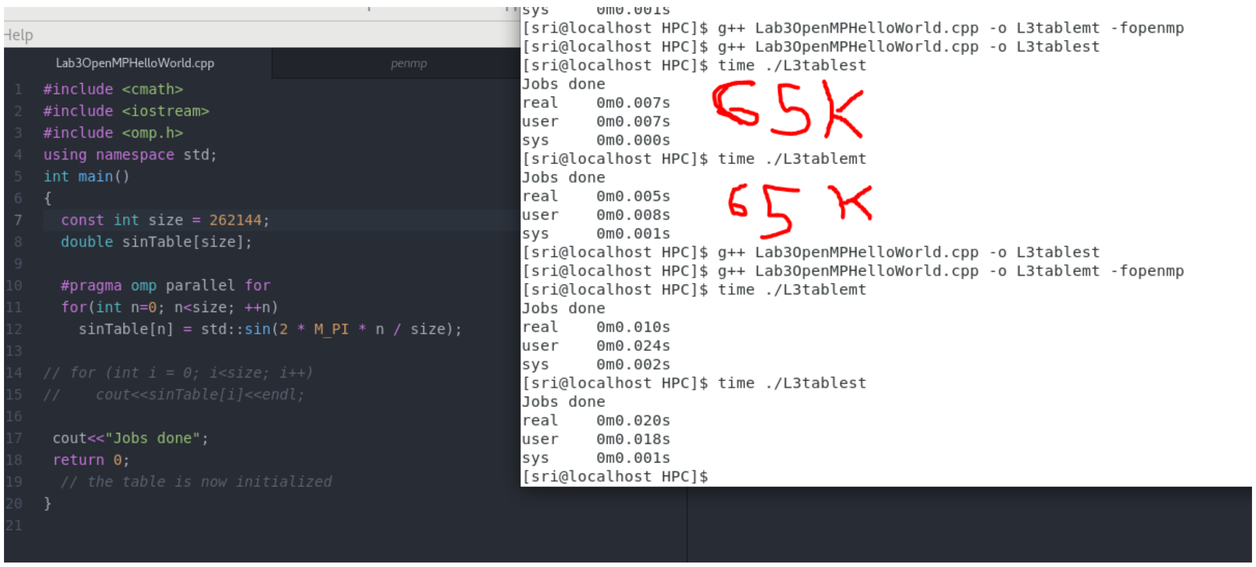
where inputfile and outputfile are what you want those to be (e.g. hello.cpp -o hello.out), and -fopenmp is what makes it compile using OpenMP.

Ok so that prints Hello World – BUT it prints Hello World OMP_NUM_THREADS times

That's the first way to know it's working.

Notice the important bit though – that code ran NUM_THREADS times – which may not have been what we wanted.

Back to our table:



```
#include <cmath>
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
  const int size = 262144;
  double sinTable[size];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

// for (int i = 0; i<size; i++)
//    cout<<sinTable[i]<<endl;

 cout<<"Jobs done";
 return 0;
  // the table is now initialized
}
```

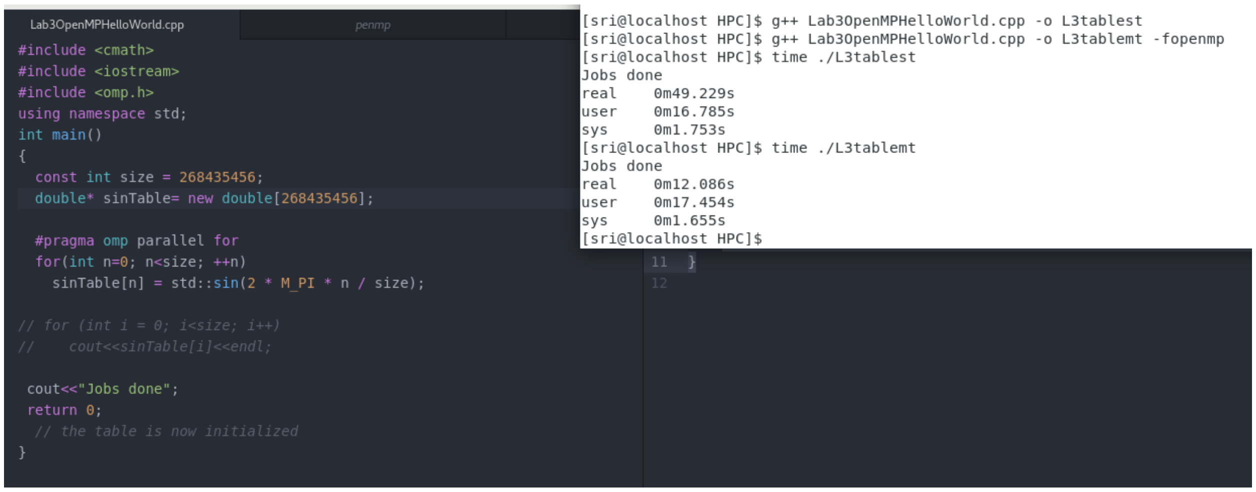
```
sys     0m0.001s
[sri@localhost HPC]$ g++ Lab3OpenMPHelloWorld.cpp -o L3tablemt -fopenmp
[sri@localhost HPC]$ g++ Lab3OpenMPHelloWorld.cpp -o L3tablest
[sri@localhost HPC]$ time ./L3tablest
Jobs done
real    0m0.007s
user    0m0.007s
sys     0m0.000s
[sri@localhost HPC]$ time ./L3tablemt
Jobs done
real    0m0.005s
user    0m0.008s
sys     0m0.001s
[sri@localhost HPC]$ g++ Lab3OpenMPHelloWorld.cpp -o L3tablest
[sri@localhost HPC]$ g++ Lab3OpenMPHelloWorld.cpp -o L3tablemt -fopenmp
[sri@localhost HPC]$ time ./L3tablemt
Jobs done
real    0m0.010s
user    0m0.024s
sys     0m0.002s
[sri@localhost HPC]$ time ./L3tablest
Jobs done
real    0m0.020s
user    0m0.018s
sys     0m0.001s
[sri@localhost HPC]$
```

The bottom two runs are at 262k (it seems like an array size > 2^20 causes a segmentation fault)

Allocating on The Heap

To go much bigger than that and we'll need to allocate the array on the heap (int * array = new int [size])

So for example:



Lab3OpenMPHelloWorld.cpp

```cpp
#include <cmath>
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
  const int size = 268435456;
  double* sinTable= new double[268435456];

  #pragma omp parallel for
  for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

// for (int i = 0; i<size; i++)
//    cout<<sinTable[i]<<endl;

 cout<<"Jobs done";
 return 0;
  // the table is now initialized
}
```

```
[sri@localhost HPC]$ g++ Lab3OpenMPHelloWorld.cpp -o L3tablest
[sri@localhost HPC]$ g++ Lab3OpenMPHelloWorld.cpp -o L3tablemt -fopenmp
[sri@localhost HPC]$ time ./L3tablest
Jobs done
real    0m49.229s
user    0m16.785s
sys     0m1.753s
[sri@localhost HPC]$ time ./L3tablemt
Jobs done
real    0m12.086s
user    0m17.454s
sys     0m1.655s
[sri@localhost HPC]$
```

Test that out on yours, try a few different numbers of threads (2, 4, 8) and see what does/doesn't work and what does/doesn't provide a speedup, that number for size isn't really important I just wanted a big number.   I used 4 because I allocated 4 cores to it, but how much speedup you get from what will likely depend heavily on your local configuration. Really, a speedup from 2 is good, making your code work with 4 is fine, anything beyond that would be for real simulations.

Ah so now we can parallelise our problems

Because this is a *shared memory* system, that 'const int size = ____' is shared across all threads. In a unshared environment you'd need to pass any values that you want to each thread, and or send them back and forth.

Some other useful stuff you can do – you can have each thread print off it's own thread ID (see textbook page 232)

Synchronization

In our previous example we had shared variable 'size' which is a mostly uninteresting constant.

The more interesting (and challenging) problem is what happens with memory that multiple threads will want to access – how to we control access to that data.

There are several types of problems that arise:

Mutual exclusion – if two threads do something which might break the completion of the other. Preventing that requires mutual exclusion (e.g. deleting the 2nd and 3rd node in a 4 node linked list with two separate threads).

Deadlock – many processes try and access one resources which is in use, so everything else stalls

Starvation – when a process is waiting to access critical data but something with higher priority keeps jumping in front, and the waiting thread never completes (or stalls at least)

Priority inversion – if a high priority task gets interrupted by a lower priority task

Busy waiting – if a resource wastes time polling to see if it has access, and this happens so often enough it may deprive processes of access to a resource (because all available time is servicing 'is this free' requests).

In OpenMP

Critical:

One option in Open MP is to make something a 'critical' (as in a critical section), only one thread can do critical tasks at a time (it will still run NUM_THREADS times) – this is a mutual exclusion.

```
#pragma omp parallel for
for (i = 0; i < N; ++i)
{
#pragma omp flush(max)
       if (arr[i] > max)
       {
#pragma omp critical
              {
                     if (arr[i] > max) max = arr[i];
              }
       }
}
```

```
Note that you could do this poorly and do:

#pragma omp parallel for
for (i = 0; i < N; ++i)
{
#pragma omp critical
       {
              if (arr[i] > max) max = arr[i];
       }
}
```

But don't.

Master

This forces only the main master thread to execute this code, other threads skip over it. Not much to say really. #pragma omp parallel

Barrier:

A point in the code where all current threads wait until all of the others are finished

You don't really need to test this it's just here for illustration

```
#pragma omp parallel
{
       int mytid = omp_get_thread_num();
       x[mytid] = some_calculation();
#pragma omp barrier
       y[mytid] = x[mytid] + x[mytid + 1];
}
```

Single

This just tells the program that only one thread can run this particular section

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
  #pragma omp single
  a++;
  #pragma omp critical
  b++;
}
printf("single: %d -- critical: %d\n", a, b);
```

```
Will print:
single:  1 – critical: 4
```

Final task for the day:

Parallelise Doubling each value in the array above, then sum it up, while trying to prevent synchronization problems

(You can check that you get it right using a single threaded version first, and then parallise).