Naima Ontiveros

Individual Task

Task 1: Study of the OpenMP API

Introduction to OpenMP API:

OpenMP is an application programming interface that can be used for “explicit direct multi-threaded, shared memory parallelism”. (openmp.org). It is an abbreviation for Open Multi-Processing. The main goal is to provide a concrete standardization of memory architecture that is simple yet efficient and portable. It is specifically specified for C/C++ and Fortran.

Compilation:

It is made up of three different components, which include compiler directives, runtime variable libraries, and environment variables. Different implementations are possible depending on the context in which they need to be applied. The compiler directives are ignored when executed unless specifically stated otherwise.
They have the following syntax: “sentinel directive-name [clause, ...]”.

The compilation statement varies depending on the platform but in Linux, it generally is g++ and the flag is -fopenmp. The header file “ #include <omp.h>” should also be included.

The general statement for the OpenMP directive should be added to sections of the code that are going to be modified:

```
“#pragma omp parallel
{
  // Code block to be executed in parallel
}”
```

Common Directives in C/C++ format:

- **Parallel**
  - It makes a team of threads execute a section of code in parallel.

```
#pragma omp parallel
{
// code to be parallelized
}
```

- **For**
  - Allows for loops to be parallelized.

```
#pragma omp parallel for
for( int i = 0, i<n; i++){
```

```
//iterations of the loop
}
```

- **Sections**
  - Allows for certain sections of code to be run in parallel.

```
#pragma omp parallel sections
{
#pragma omp section
        {
        // section 1
        }
#pragma omp section
        {
        .// section 2
        }
}
```

- **Single**
  - Allows for a portion of the code to be executed by a single thread

```
#pragma omp parallel
{
#pragma omp single
        {
        }
}
```

- **Critical**
  - States which section of code should only be executed by one thread at a time.

```
#pragma omp critical
{
}
```

- **Atomic**
  - States what variable should be accessed only atomically.

```
#pragma omp atomic
Shared_var += local_var
```

Examples:
The following is an example of an OpenMP directive “parallel”:

```
#include <omp.h>
Void main()
{
#pragma omp parallel
        {
        Int ID = 0;
        printf(“hello(%d)”, ID);
        print(“world(%d)\n”, ID):
        }
}
```

It creates a team of threads for what is inside the section of #pragma omp parallel.

Another example would be the one below for reduction:

```
int sum_parallel(int n)
{
   int sum = 0;
#pragma omp parallel for reduction(+ : sum)
   for (int i = 0; i <= n; ++i) {
      sum += i;
   }
   return sum;
}
```

Task 0.1 OpenMP pre-training:

0.1.1 What is the chunk variable used for?

The chunk variable used in OpenMP refers to the idea of scheduling loops to divide a certain amount of iterations amongst the threads in a parallel code. The chunk size determines how many iterations will be assigned to each thread. This can impact the performance of execution depending on the overhead or overall balance.

0.1.2 Fully explains the pragma :

#pragma omp parallel shared(a,b,c,chunk) private(i)

- The "#pragma omp parallel" in openMP is used to fork a team of threads to carry out tasks in parallel.

Why and what is shared(a,b,c, chunk) used for in this program?

The "shared" or "private" clauses decide how the threads will be accessed. In this case, the "shared(a,b,c, chunk)" implies that "a,b,c, chunk" will be shared amongst all the threads available.

Why is the variable i labeled as private in the pragma?

The variable "i" is labeled as private in the pragma, so that each thread can have its own copy of the variable and can modify it respectively rather than share it.

0.1.3 What is the schedule for? What other possibilities are there?

The schedule in OpenMP is used to control the loop scheduling policies, which decides how many iterations are used for each thread. There are different types of scheduling in OpenMP: static, dynamic, guided, auto, and runtime. Static works by distributing the iterations equally amongst the threads. Dynamic functions by assigning the threads dynamically and guided are similar, however, the chunks decrease in size as time goes on. Another use of scheduling in OpenMP is load balancing using scheduling such as dynamic and guided to improve the workload balance.

0.1.4 What times and other performance metrics can we measure in parallelized code sections with OpenMP?

The execution time can be measured in the parallelized code sections with OpenMP, which is the main method of determining the performance of the parallelized code. One other factor that can be measured is speedup, which compares the execution time of the original code to the parallized version. Efficiency can also be measured which consists of the speedup over the number of total threads. Load balancing and overhead can also be measured with OpenMP.

Task 0.2: Pre-training std::async

0.2.1 What is the std::launch::async parameter used for?

The std::launch::async parameter is used to create a new thread for execution. The “async” portion ensures that each task will be on a different thread.

0.2.2 Calculate the time the program takes with std::launch::async and std::launch::deferred. What is the reason for the time difference?
Std::launch::deferred should have a higher time because the tasks are not executed immediately and the program has to wait for “future.get()” and “future.wait()”, so the total time will be longer. Also, std::launch::async offers parallel execution as the other one is sequential, so it is going to take more time if only one core can be utilized.

0.2.3 What is the difference between the wait and get methods of std::future?
The “get” method works by waiting for the task to be completed and provides the result of the task. It can only be utilized once per task and provides an error message if found. The “wait” method also waits for the task to be completed but does not provide the result. Another difference is that “wait” can be used more than once in the same task. Essentially, the “wait” method is used to block the thread until the asynchronous operation with “std::future” is completed, and “get” works in a similar fashion but it returns the results of the operation.

0.2.4 What advantages does std::async offer over std::thread?
Some advantages of “std:async” over “std::thread” are that in “std:async” it is not necessary to explicitly join or detach any thread. “Std::future” automatically handles it and prevents certain issues from arising. Another advantage would be that “std::async” facilitates returning values from threads because “std::future” allows for easy value retrieval. There is also more flexibility in “std::async” because you can choose a policy that allows for more control. Resource management is also more efficient with “std::async” since creating or deleting threads is an automatic process.

Task 0.3: Pre-training std::vector
0.3.1: Which of the two ways of initializing the vector and filling it is more efficient? Why?
Direct initialization is usually more efficient because when the size is given for the vector it can be initialized at the start. The dynamic filling is more flexible, however, it could use up more computational resources because it is not initialized from the start. Dynamic initialization has multiple memory allocations, which also can cause the performance to go down.

0.3.2: Could a problem occur when parallelizing the two for loops? Why?
There can be certain problems when parallelizing two loops because a data race can occur in which the synchronization is not exact and could give inaccurate results. This happens when several threads try to access the same memory. Another possible issue is the safety of threads

because directives such as “std::vector” are not safe to perform tasks concurrently and can lead to failures.

References

[1] https://hpc-tutorials.llnl.gov/openmp/
[2] https://lsi2.ugr.es/jmantas/ppr/ayuda/omp_ayuda.php
[3]https://www.geeksforgeeks.org/introduction-to-parallel-programming-with-openmp-in-cpp/