MONASH University

[Template for Lab Session submission]

Lab/Applied Session Week {Week 3}

Student ID	**Student Name**	**Student Email**
32837933	Chua Sheng Xin	schu0077@student.monash.edu

Tasks

{Answers
Task 2

In task 2, I have implemented a multi thread processing function to output the prime numbers. There are two available cores in my machine even though the function can use up to 8 threads. The results show that the time used to process the function was around the same as in task 1. The speed-up is approximately 1.107. This means the parallel implementation is about 10.7% faster than the serial implementation.This may be due to several reasons, mainly limited number of physical cores and context switching overhead. My system has only 2 physical cores and when the function create multiple threads, they may not be able to run in true parallel beyond the number of physical cores available. With more threads than physical cores, the operating system also needs to perform context switching to give each thread CPU time. Context switching involves saving and restoring the state of a thread can incur overhead especially when the number of threads is high relative to the number of cores.

Additionally, creating and joining threads also introduces overhead. If the tasks are relatively small, the time spent in creating and managing threads might offset the benefits of parallelization. It is also important to keep in mind that for small problem sizes, the time saved by parallel computation may be negligible compared to the overhead of managing multiple threads. Lastly, console I/O operations like printing prime numbers to the console can be relatively slow and may become a bottleneck, overshadowing the benefits of parallel computation.

Task 3

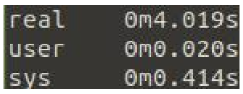
The results for task 3 show that the time used to process the function was faster than the time needed in task 1 and 2. The speed-up when compared to task 2 is approximately 1.050 and there are two available cores in my machine. The reason for this may be core utilization and actual utilization.

OpenMP automatically handles the creation and management of threads. By default, OpenMP tries to utilize all available cores. In task 2 however the number of threads exceeds the number of cores, there will be overhead due to context switching and thread management. When the machine only has 2 cores, the operating system will time-share these cores among the threads. This means that although the function created 8 threads, only 2 threads could run concurrently at any given time. The remaining threads would wait for their turn to be scheduled, which can lead to overhead and context switching. This can lead to performance degradation if the system is unable to efficiently handle all the threads. OpenMP on the other hand might handle core allocation and thread scheduling more efficiently. It typically creates as many threads as there are cores, it can also better manage thread creation and destruction compared to manual thread management.

As to actual utilization, given that OpenMP is more integrated with the runtime environment, it might better utilize available cores compared to manually managing threads with POSIX Threads when it dynamically adjusts the number of threads based on the number of cores.
}

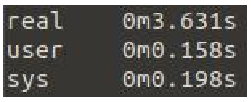
{Screenshots

Task 1



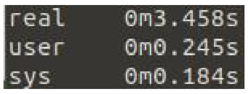
```
real    0m4.019s
user    0m0.020s
sys     0m0.414s
```

Task 2



```
real    0m3.631s
user    0m0.158s
sys     0m0.198s
```

Task 3



```
real    0m3.458s
user    0m0.245s
sys     0m0.184s
```

}

{References

ChatGPT. I have used ChatGPT in generating answers and codes for doubts I have had throughout task 1,2,3 and especially task 2 and 3.}