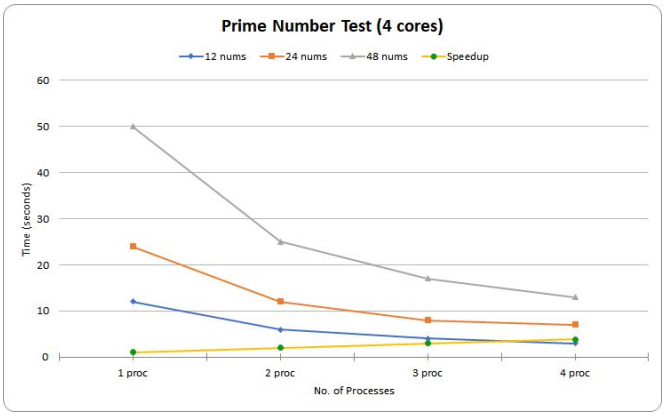
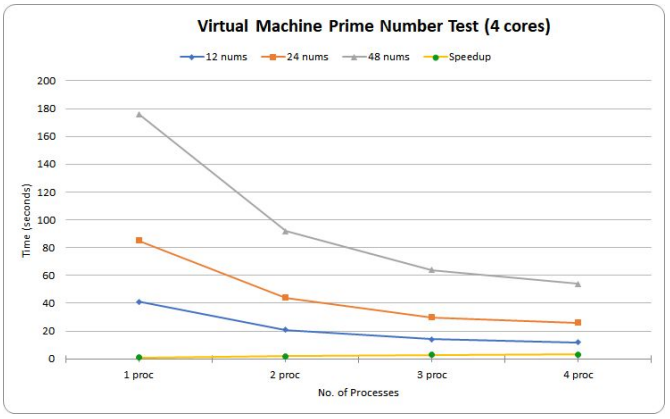
Native OS	1 core	2 cores	3 cores	4 cores
12 Nums	12	6	4	3
24 Nums	24	12	8	7
48 Nums	50	25	17	13
Speedup	1.0	2.0	2.9	3.8
Virtual Machine	1 core	2 cores	3 cores	4 cores
12 Nums	41	21	14	12
24 Nums	85	44	30	26
48 Nums	176	92	64	54
Speedup	1.0	1.9	2.8	3.1
Slowdown Rate VM vs Native OS 48 Nums	3.52	3.68	3.76	4.15

In this experiment, I chose to find the speedup of running a “find prime number” function on a single core, 2 cores, 3 cores, and 4 cores. I ran the test on my native operating system, Windows 10, and on a virtual machine, Ubuntu through VirtualBox. The aim of the VirtualBox test was to see the speedup of multiple cores being assigned to the prime number task, just like the test on the native OS, but also to see the difference in completion time between the native OS and the virtual machine.

Prime numbers with a length of 8 digits were chosen so as to ensure that the task wouldn't be too easy for the cores and thus be completed too quickly, making comparisons difficult. Three sets of numbers were used, with the first set consisting of 12 numbers, the second 24 numbers, and the third 48 numbers. I chose to double the input size, as a difference of one or two numbers wouldn't give a great differentiation between the input sets, and also because the prime number function is Big-O O(n), which is linear, a doubling of input should show a roughly doubling time to complete. All time is in seconds. The results are visualised below:





Looking at the test on the native machine, a few interesting points can be noted. Firstly, we can see that doubling the input size leads to a near exact doubling in runtime. This matches with an analysis of the code:

```
def check_prime_verbose(num):
    t1 = time.time()
    res = False
    if num > 0:
        # check for factors
        for i in range(2,num):
            if (num % i) == 0:
                print(num,"is not a prime number")
                print(i,"times",num//i,"is",num)
                print("Time:", time.time()-t1)
                break
        else:
            print(num,"is a prime number")
            print("Time:", time.time()-t1)
            res = True
    return res
```

The largest factor at play here is n, the number of inputs, and thus the Big-O is O(n), signifying that as the size of the input grows, the runtime increases linearly. Take the prime number function on the native OS on one core for example. A doubling of the input from 12 numbers to 24 leads to a doubling in runtime: 12 seconds to 24 seconds.

The second thing to note is that as more cores are applied to the task, there is a speedup. This is multiprocessing in action. Multiprocessing refers to a computer's ability to run more than one process simultaneously. Through using a Python module named “multiprocessing”, it is possible to run multiple processes independently on separate cores (called subprocesses), thus better utilizing the cores. In this experiment, I decided to test the three input sets across the 4 cores of my CPU. The results for the native OS shows that when 2 cores are used, the runtime is halved, when 3 cores are used, the runtime is nearly exactly a third less, and for 4 cores, the runtime is almost exactly a quarter less.

When a task can be run very well synchronously across processors, it is said to be highly parallelizable. However, all processes have a part of their code which is sequential. Amdahl's Law basically states that the level of parallelization possible is determined by the amount of the program that is sequential. The amount of the task that is parallelizable can be determined using the following formula, where N is the number of processors/cores.

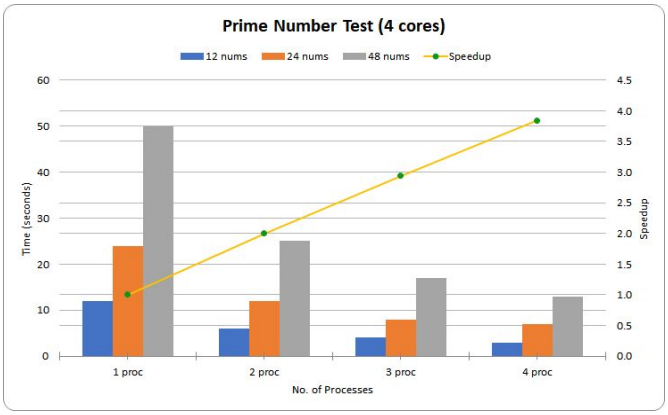
$$\text{Speedup}(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

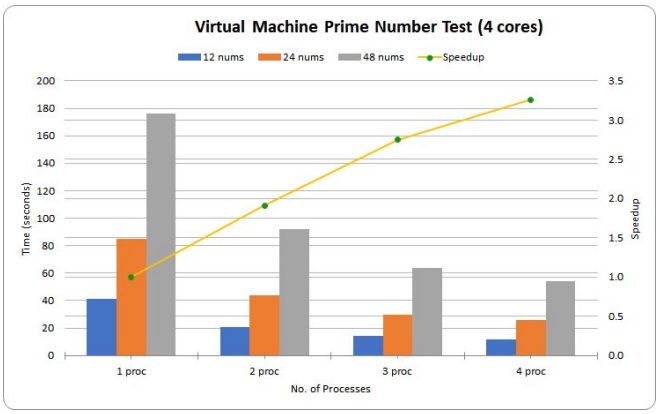
Serial part of job = 1 (100%) - Parallel part

Parallel part is divided up by N workers

http://computerscience.chemeketa.edu/cs160Reader/ParallelProcessing/AmdahlsLaw.html

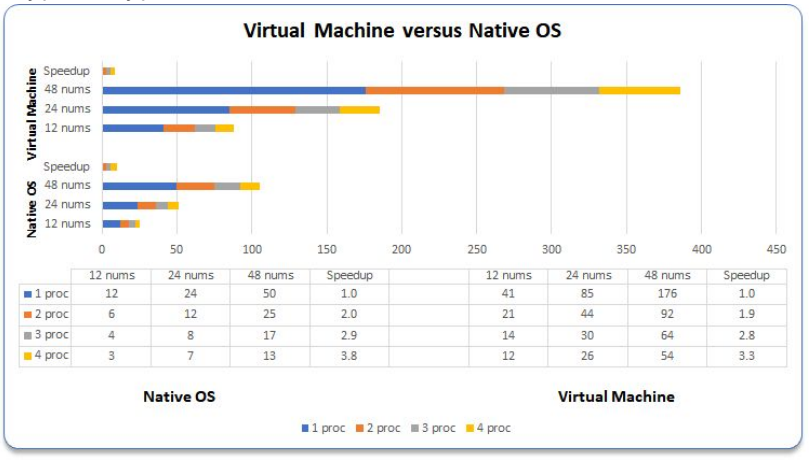
Using this formula, it can be seen that eventually the increase in performance gained from adding more processors/cores decreases as more are added. This can be seen in the experiment, as the speedup goes from 2.0, 2.9, and 3.8, with 2, 3, and 4 cores respectively, instead of an exact speedup of 2, 3, and 4.





With the virtual machine, there is an added CPU overhead as a hypervisor needs to run. A hypervisor is software which allows multiple operating systems to share a computer's hardware. A type 1 hypervisor is faster as it allows "bare-metal" access to a computer's hardware (basically the OS runs directly on the computer's hardware), whereas a type 2 hypervisor is installed on top of an existing OS and uses that OS to manage its hardware calls. Thus, type 2 is inherently slower. VirtualBox uses a type 2 hypervisor, and thus is installed on top of Windows 10 here. From the testing results it can be seen that the performance is roughly 3 to 4 times slower than the native OS. For example 24 prime numbers with 2 cores takes 44 seconds vs 12 seconds. The results also point to a decrease in speedup when extra processors are added. This is unexpected, and possibly points to the extra overhead negatively affecting the runtime. This is perhaps due to the slower I/O time and memory access on the VirtualBox when compared to the native OS, but as there is an extra layer for type 2 hypervisors to pass through, there is so much going on in the background that it is difficult to pin down exactly what is causing this behaviour. Whatever the exact explanation is, the main takeaway here is that if you need performance, avoid type 2 hypervisors.

The main lessons that can be learned from this experiment are that more processors do lead to a runtime speedup, but the level of the speedup drops off as more and more processors are added, due to Amdahl's Law. Eventually, a stage will be reached where adding processors will not result in any speedup gains, as the sequential part of the code will remain a constant factor. This would have become more apparent in this test if my processor had more cores and/or hyperthreading. We can also see that while type 2 hypervisors are useful for virtualisation, there is a hefty performance hit which goes hand in hand with its functionality. If performance is essential, a type 2 hypervisor should be avoided.



	Native OS 12 nums	Native OS 24 nums	Native OS 48 nums	Native OS Speedup	Virtual Machine 12 nums	Virtual Machine 24 nums	Virtual Machine 48 nums	Virtual Machine Speedup
1 proc	12	24	50	1.0	41	85	176	1.0
2 proc	6	12	25	2.0	21	44	92	1.9
3 proc	4	8	17	2.9	14	30	64	2.8
4 proc	3	7	13	3.8	12	26	54	3.3