OS HW1

310551055 鄭伯俞

Introduction :

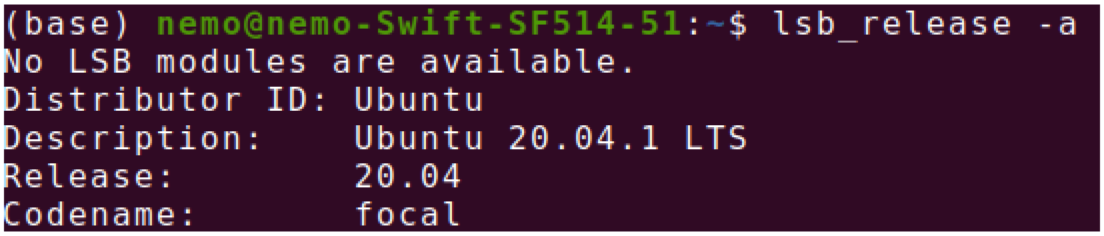
In this project, I use 2 different parallelizing methods to count specific numbers in a large array. My multi-thread program uses **mutex** to safely sum the result of each thread, while my multi-process program uses its **return code** to send its result to the parent process.

Source Code:

https://github.com/Nemo1999/NYCU_OS

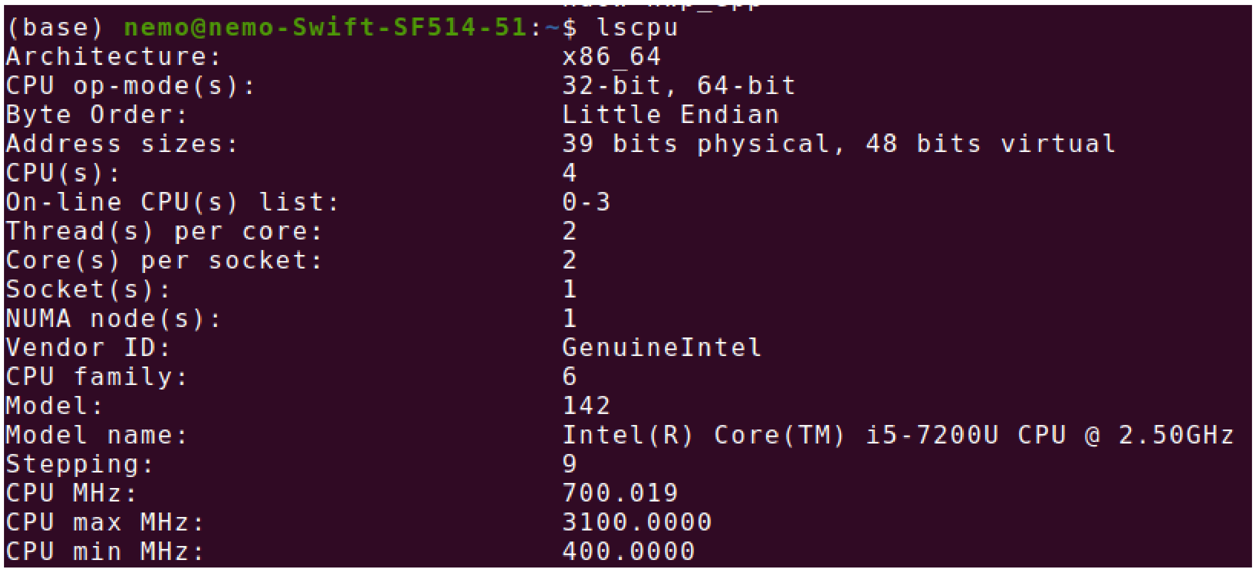
Environment Setup:

I run my code on native Ubuntu 20.04



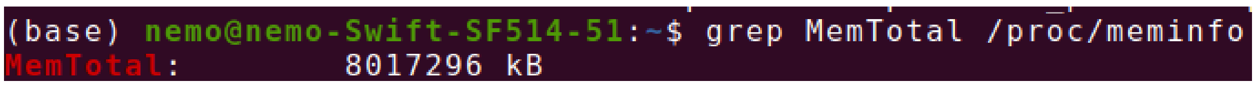
```
(base) nemo@nemo-Swift-SF514-51:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.1 LTS
Release:        20.04
Codename:       focal
```

I use a Intel-i5 cpu with 4 cores and 2 threads per core.



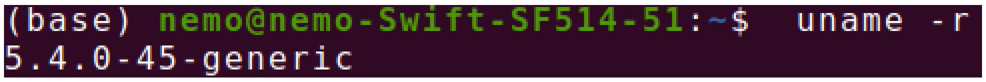
```
(base) nemo@nemo-Swift-SF514-51:~$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) per core:              2
Core(s) per socket:              2
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Stepping:                        9
CPU MHz:                         700.019
CPU max MHz:                     3100.0000
CPU min MHz:                     400.0000
```

Memory size is 8G



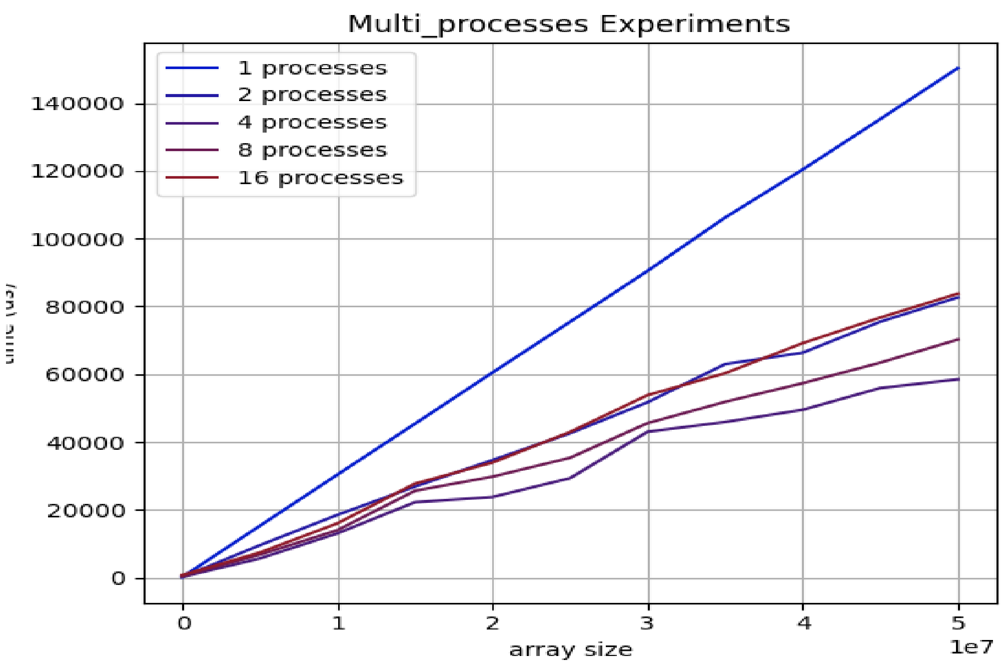
```
(base) nemo@nemo-Swift-SF514-51:~$ grep MemTotal /proc/meminfo
MemTotal:          8017296 kB
```

Kernel Version:



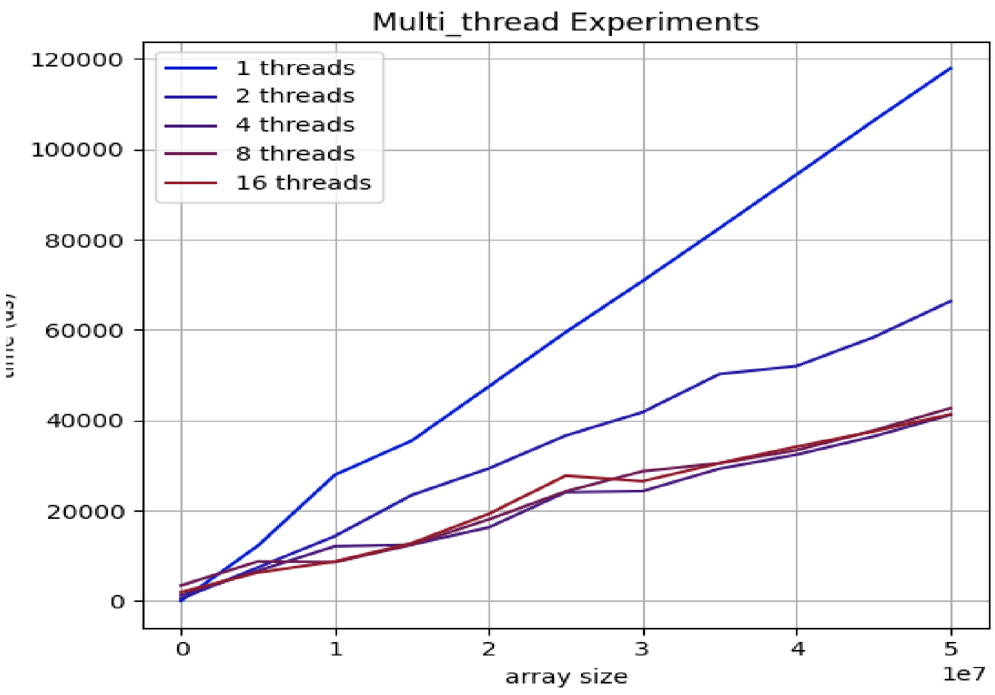
```
(base) nemo@nemo-Swift-SF514-51:~$  uname -r
5.4.0-45-generic
```

Results:



Here the y-axis shows the execution time (in u_sec) , x-axis shows the size of the input array. Each curve is drawn with a different process setting, and for each data point on the curve, 5 identical experiments are repeated, and the resulting execution times are averaged.

We can see that with more processes, the execution time goes down for each array size, but when the number of processes exceeds 4, the execution time starts to rise. 4 processes gives us best performance.



Here the y-axis shows the execution time (in u_sec) , x-axis shows the size of the input array. Each curve is drawn with a different thread setting, and for each data point on the curve, 5 identical experiments are repeated, and the resulting execution times are averaged.

We can see that with more thread, the execution time goes down for each array size, but when the number of threads exceeds 4, the execution time stops to improve . 4 processes gives us the best performance. More than 4 processes gives similar performance

Comparison

Under similar setting (same # of process and threads) Multi-thread version runs faster , this may because that threads didn’t need to copy the large input array. And the execution time is linear with respect to array size in both methods, because the time complexity is exactly Theta(n).