實驗二　application profiling & optimization

9617145 許晏峻 資工 4C

◎ **實驗目的：**

利用不同的方法找出程式效能的瓶頸。

◎ **步驟過程：**

1. *time*
   將取得 time 的 function 放到 is_prime() 的開頭和結尾，並且設置一個變數儲存秒 (sec)，來累加 (結尾 - 開頭) 的值。
2. *clock_gettime*
   將取得 time 的 function 放到 is_prime() 的開頭和結尾，並且設置兩個變數分別儲存秒 (sec) 和 奈秒 (nsec)，來累加 (結尾 - 開頭) 的值。
   compile 時須加上 -lrt 參數。
3. *getrusage*
   將取得 time 的 function 放到 is_prime() 的開頭和結尾，並且設置兩個變數分別儲存秒 (sec) 和 微秒 (usec)，來累加 (結尾 - 開頭) 的值。
   compile 時須加上 -lrt 參數。
4. *rdtsc*
   將取得 cpu clock 的 function 放到 is_prime() 的開頭和結尾，並且設置一個變數儲存 cpu clock，來累加 (結尾 - 開頭) 的值。
5. *gprof*
   compile 時須加上 -pg 參數。
   執行後再執行指令 gprof prime > result 將分析結果導入 result。
6. *pprof*
   先設置環境變數 PATH=[google bin path]、LD_LIBRARY_PATH=[google lib path]、CPUPROFILE=[profile path]、CPUPROFILE_FREQUENCY=[sampling frequency]。
   compile 時須加上 -lprofiler 參數和 -I[include path] -L[lib ptah]。
   執行後再執行指令 pprof prime /tmp/profile，再輸入指令 top 印出分析結果。

◎ **數據結果：**

code / method	primeBasic	primeOpt1	primeOpt2
time	1 sec	1 sec	1 sec
clock_gettime	941399296 nsec (0.941399296 sec)	902992269 nsec (0.902992269 sec)	1766511906 nsec (1.766511906 sec)
getrusage	740047 usec (0.740047 sec)	708043 usec (0.708043 sec)	612038 usec (0.612038 sec)
rdtsc	1572062589 cpu clock	1461790737 cpu clock	1066447953 cpu clock
gprof	0.26 sec	0.23 sec	0.12 sec
pprof	84.2%	93.5%	87.5%

◎ **討論 & 結論心得：**

從數據結果可看出幾乎是偏向 primeOpt2 的效能較好，個人認為是因為 is_prime 這個 function 中的演算法。

1. 首先 primeBasic 用了最直覺的方式去判斷一個數是否為質數：他拿所有的奇數來和代測數做 module 運算，所以每個數需要測過所有的奇數。
2. 在 pimeOpt1：是用一個陣列存所有的奇數，若某個奇數是由其他奇數組成，就不將這個奇數列入和代測數 module 運算的選單中，這樣的方法可讓代測數只和質數做運算，減少 module 時間。
3. 而 primeOpt2：則是直接從目前已算出的質數陣列取出質數和讓代測數做運算，減少 module 時間；這樣和 primeOpt1 相比減少了存取奇數陣列的時間。

以上是個人簡單的分析。

有幾筆數據似乎和其他數據相反，我覺得只是當時機器的狀態所致，有重複測過幾次發現實際上是差不多的。