實驗四 Multicore architecture and multithreading

1. 實驗目的

使用 multithreading programming (pthread)來加速程式效能，並且利用 Debugger 針對 multhreading program 除錯。

2. 步驟過程

- **Project1：matrixMultip.c**

首先，可在 mm()函式中發現，for 迴圈的 j 起始值應為 0 而不是 1，之後在對 i 做累加(i += noproc+1;)，不應該多加 1。以上為程式 bug。

計算時間的方式，使用 lab2 時教過的 getclocktime()來實作。

- **Project2：pi.c**

我發現在迴圈中，pi 是每次去累加每一次算出來的值(count*2-1)，因此可以知道 pi 值的計算是可以拆開的，所以在這裡我的作法是先把計算 pi 的地方分開，分成 n 個 thread 來各自計算(count*2-1)，最後在使用 pthread_join 的時候再將每個 thread 算出的值加總起來成真的 pi 值。

3. 數據結果

- **Project1：matrixMultip.c**

由實驗課堂上發現，多條 thread 的效能並不會比一條來的高，原因是運算量太小，使用多條 thread 反而只會增加那些建立 thread 的時間；若將運算量加大(矩陣放大)，多條 thread 的效能才容易顯現出來。

- **Project2：pi.c**

以下為實驗數據截圖，計算執行時間的方式為 time()和 Linux 下指令 time。

```
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.init
The approximate value of pi = 3.141593
wall clock time = -1296965521
17.086u 0.000s 0:17.08 100.0%   5+1533k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.ver1
The approximate value of pi = 3.141593
wall clock time = -1296965544
12.814u 0.000s 0:12.81 100.0%   5+1532k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 1
The approximate value of pi = 3.141593
wall clock time = 12
12.831u 0.000s 0:12.83 100.0%   5+1531k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 2
The approximate value of pi = 3.141593
wall clock time = 7
12.822u 0.000s 0:06.41 200.0%   5+1532k 0+0io 0pf+0w
```

圖 1. pi.c 測試結果 (part1)

圖 1 中的 pi.out.init 是一開始的程式，執行時間為 17 秒左右；而 pi.out.ver1 是我個人自己修改過的版本(尚未加入 pthread)，執行時間約為 13 秒；最後 pi.out.pthread 則是加入 pthread 的版本，第一次執行一條 thread，因此結果約為 13 秒，和 pi.out.ver1 差不多，第二次執行兩條 thread 就可看到明顯差異，執行時間縮短為 6 秒，也證明了程式整個執行的瓶頸就在計算 pi 的迴圈上。

```
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 8
The approximate value of pi = 3.141593
wall clock time = 3
14.823u 0.000s 0:02.60 570.0%   5+1534k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 16
The approximate value of pi = 3.141593
wall clock time = 1
20.298u 0.007s 0:01.97 1029.9%  5+1531k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 24
The approximate value of pi = 3.141593
wall clock time = 1
20.438u 0.000s 0:01.74 1174.1%  5+1531k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 32
The approximate value of pi = 3.141593
wall clock time = 1
20.775u 0.000s 0:01.41 1473.0%  5+1534k 0+0io 0pf+0w
```

圖 2. pi.c 測試結果 (part2)

而圖 2 則是繼續測試多條 thread 的結果，可看到執行時間還可以持續下降，執行到 32 條 thread 時，執行時間縮短為 1.41 秒。

```
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 64
The approximate value of pi = 3.141593
wall clock time = 1
20.792u 0.000s 0:01.43 1453.8%  5+1533k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 128
The approximate value of pi = 3.141593
wall clock time = 2
20.780u 0.000s 0:01.39 1494.9%  5+1529k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 256
The approximate value of pi = 3.141593
wall clock time = 1
20.832u 0.000s 0:01.32 1578.0%  5+1529k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 512
The approximate value of pi = 3.141593
wall clock time = 1
20.836u 0.007s 0:01.34 1554.4%  5+1531k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 768
The approximate value of pi = 3.141593
wall clock time = 1
20.818u 0.030s 0:01.34 1555.2%  5+1532k 0+0io 0pf+0w
bsd1 [/u/cs/96/9617145/MP/lab4/pi] -ychsu- % time ./pi.out.pthread 1024
The approximate value of pi = 3.141593
wall clock time = 2
20.803u 0.015s 0:01.34 1552.9%  5+1532k 0+0io 0pf+0w
```

圖 3. pi.c 測試結果 (part3)

而圖 3 是繼圖 2 之後繼續測試下去的結果，但是在圖 3 中，可以發現直到執行 1024 條 thread，執行時間都沒有明顯成長，一直維持在 1.3 秒至 1.4 秒之間，因此可了解到，在不更改

程式碼的狀態下，multi-thread 最多能增加的效率只能到這裡，如果希望再加快執行速度，可能要更改演算法，或對某些部位再做最佳化…等，又或者可能要分析程式新的瓶頸在哪，再對這些地方做效能的加強。

4. 結論心得

這次實驗學到了如何使用圖形化介面的 gdb—ddd 來做 debug 的動作，也學到了如何使用 pthread 來寫一個 multi-thread 的程式。

但是，要注意的是，並不是所有程式使用 multi-thread 就可以順利地加快速度，甚至會拖慢整個程式的執行時間，因為建立 thread 的時間也許遠比程式的執行時間來的久；又或者是因為程式中有許多 critical section 可能發生 race condition，因此必須使用 lock 來確保程式正確執行，這樣一來多執行緒就不一定能發揮預期的效能，甚至讓程式更複雜。所以寫 multi-thread 的程式還有很多地方值得注意學習。