高效能巨量資料與人工智慧系統 HW2

B05611047 化學四 鍾宜樺

1. Write an MPI code to perform the same matrix multiply task as Exercise 1. You can get the code from anywhere, but make sure that you understand how it works.

2. Install the MPI environment, compile and run the MPI code on a server in the CSIE Workstation Lab.with 1, 2, 4, 8, … processes… Report the execution time. Make sure it runs correctly.

＊ 作法：

將一般的矩陣乘法做分割，分別將 B 矩陣 columns 盡量以平均分配的方式分給各個 processes。即 (# of B's column) / (# of total processes)為每一個 Process 所分得的 columns 數量。並將(# of B's column) mod (# of total processes)個 column 的矩陣乘法，留給 master 做。

此作法相同於在 MPI_tutorial 此份講義中 p.72 的 Matrix-vector multiplication 的做法相同。

＊ 運行結果如下：

此結果是跑在 linux14 上面。

report the execution time 根據上圖圖(一) :

# of processes	Real time
1	0.328(s)
2	0.176(s)
4	0.179(s)
8	0.183(s)
16	Not enough available slots
32	Not enough available slots

＊ 補充：

在此處我們發現到，在不同的 server 上跑相同的程式，會發現每個 server 所能使用的 processes number 不同，因此我們將 linux1~linux8 都跑了一次，將每個 server 能使用的 processes 數試出來，結果如下表。

# of processes	1	2	4	8	16
Linux1	0.404(s)	0.379(s)	0.392(s)	0.385(s)	0.693(s)
Linux2	0.186(s)	0.173(s)	0.169(s)	0.178(s)	0.208(s)
Linux3	0.117(s)	0.077(s)	0.078(s)	0.086(s)	0.113(s)
Linux4	0.221(s)	0.142(s)	0.216(s)	0.272(s)	0.407(s)
Linux5	0.292(s)	0.224(s)	0.304(s)	0.294(s)	X
Linux6	0.597(s)	0.336(s)	0.356(s)	0.332(s)	X
Linux7	0.297(s)	0.276(s)	0.277(s)	0.318(s)	X
Linux8	0.286(s)	0.219(s)	0.221(s)	0.298(s)	X

3. Run it on multiple servers and report the execution time.

＊實驗方法：

1. 總共跑在 csie 的 8 個工作站上(linux1~linux8)。一開始需先設定 rsa key(參考作法如：https://blog.gtwang.org/linux/linux-ssh-public-key-authentication/)。設定完 rsa key 後要特別注意的是，要確認等等會用到的 server 資料都已存在 known host 中，否則就會跑不出東西，等很久也不知道錯在哪，悲劇 QQ
2. 設定完後即可開始實驗。根據改變 # of processes / server (variable_1)，去觀察 execution time，最後討論其結果。
3. 矩陣大小延續**作業一**所得之最大 multiplier = 16 下去跑。
   C[992*112]=A[992*240]，B[240*112]。

＊指令：

mpiexec -mca btl_tcp_if_include net0 --npernode variable_1 --hostfile hostfile -np variable_2 ./a.out

/*

variable_1 : # of processes / server

variable_2 : 總共在 8 台 servers 上的# of processes。

*/

＊公式：

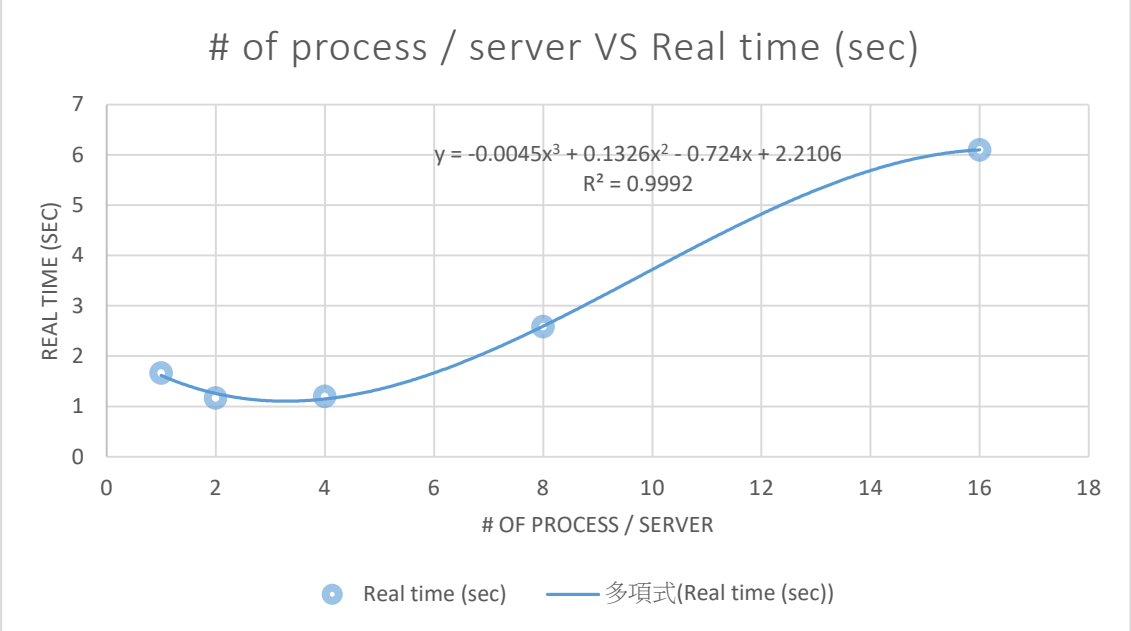
variable_2 = variable_1 * (8)

＊實驗過程及結果如下：

```
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -mca btl_tcp_if_include net0 --npe
rnode 1 --hostfile hostfile -np 8 ./a.out

real    0m1.662s
user    0m0.140s
sys     0m0.127s
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -mca btl_tcp_if_include net0 --npe
rnode 2 --hostfile hostfile -np 16 ./a.out

real    0m1.168s
user    0m0.159s
sys     0m0.190s
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -mca btl_tcp_if_include net0 --npe
rnode 4 --hostfile hostfile -np 32 ./a.out

real    0m1.199s
user    0m0.252s
sys     0m0.306s
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -mca btl_tcp_if_include net0 --npe
rnode 8 --hostfile hostfile -np 64 ./a.out

real    0m2.584s
user    0m0.392s
sys     0m0.745s
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -mca btl_tcp_if_include net0 --npe
rnode 16 --hostfile hostfile -np 128 ./a.out

real    0m6.100s
user    0m1.262s
sys     0m1.589s
b05611047@linux1 [~/HPBDAIS/HW2]
```

根據上截圖得到表格如下：

variable_1	variable_2	Real time (sec)
1	8	1.662
2	16	1.168
4	32	1.199
8	64	2.584
16	128	6.100



＊實驗數據討論：

從上表中可看到，在# of processes / server 由 1 上升為 2 時，可發現所使用的時間下降的，的確有達到加速的效果。然而在# of processes / server 繼續上升後，卻發現 cost time 以 polynomial 的速率上升了。檢討後，覺得應該是實驗中所使用的矩陣 size 不夠大(C[992*112]=A[992*240]，B[240*112])，即本實驗中的 bottleneck 出現在 communication overhead。因此造成隨著# of processes / server 卻使 cost time 更大的結果。

另外我們發現在# of processes / server = 16 的時候，cost time 出現了 6 秒，然而對照爾後的實驗會發現，這個 6 秒應該是因為當時網路不穩定的原因，才會導致速率過慢，才出現了這個很突兀的 6 秒的結果。。

＊實驗數據檢討：

根據上述討論，可將矩陣的 size 放的很大，去看在 multiple servers 上跑會不會達到加速的效果。

4. Double the values of NRA, NCA, and NCB to observe the execution time.

＊ 作法

同於作業一一樣，令 double 的倍率= multiplier，改變 multiplier 去 run code。此為結果是跑在 CSIE 的 server : linux1 所得到的。為求公平，固定# of processes = 4。

＊ 指令：

```
＊ b05611047@linux1 [~/HPBDAIS/HW2] vim mpi_mm_notopt.c
```

* b05611047@linux1 [~/HPBDAIS/HW2] mpicc mpi_mm_notopt.c
* b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -n 4 ./a.out

＊ 結果如下：

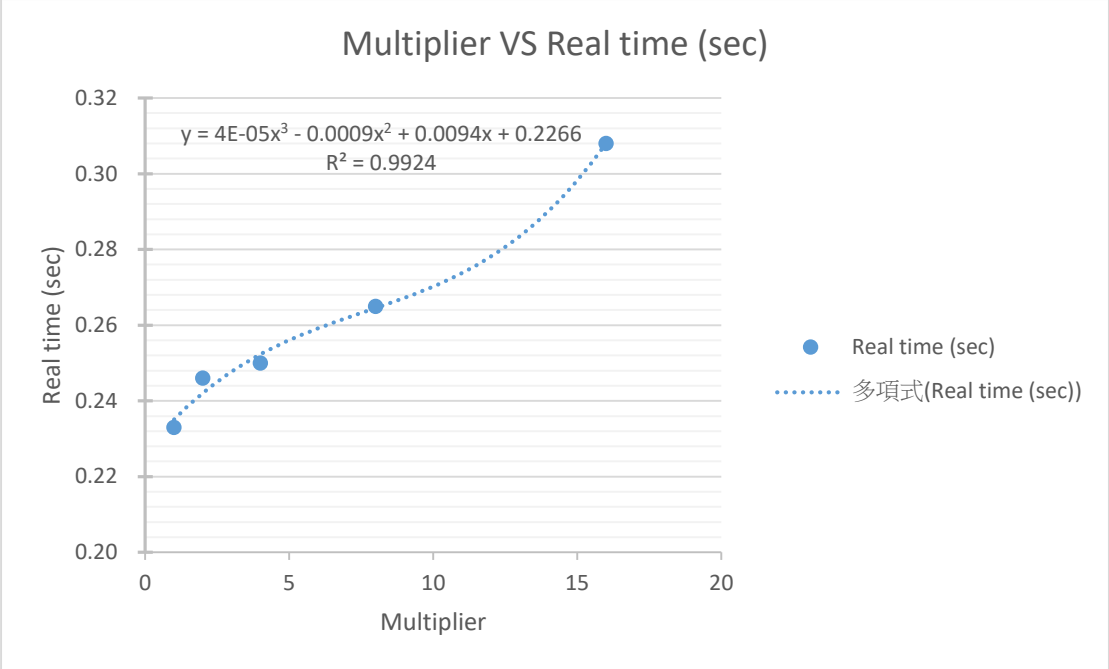
```
b05611047@linux1 [~/HPBDAIS/HW2] vim mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] mpicc mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -n 4 ./a.out
multiplier = 1

real    0m0.233s
user    0m0.110s
sys     0m0.104s
b05611047@linux1 [~/HPBDAIS/HW2] vim mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] mpicc mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -n 4 ./a.out
multiplier = 2

real    0m0.246s
user    0m0.112s
sys     0m0.099s
b05611047@linux1 [~/HPBDAIS/HW2] vim mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] mpicc mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -n 4 ./a.out
multiplier = 4

real    0m0.250s
user    0m0.098s
sys     0m0.114s
b05611047@linux1 [~/HPBDAIS/HW2] vim mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] mpicc mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -n 4 ./a.out
multiplier = 8

real    0m0.265s
user    0m0.147s
sys     0m0.096s
b05611047@linux1 [~/HPBDAIS/HW2] vim mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] mpicc mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -n 4 ./a.out
multiplier = 16

real    0m0.308s
user    0m0.383s
sys     0m0.104s
b05611047@linux1 [~/HPBDAIS/HW2]
```

```
b05611047@linux1 [~/HPBDAIS/HW2] vim mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] mpicc mpi_mm_notopt.c
b05611047@linux1 [~/HPBDAIS/HW2] time mpiexec -n 4 ./a.out
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
mpiexec noticed that process rank 1 with PID 0 on node linux1 exited on signal 11 (Segmentation fault).
--------------------------------------------------------------------------

real    0m1.189s
user    0m0.054s
sys     0m0.041s
b05611047@linux1 [~/HPBDAIS/HW2]
```

(*)

整理成表格如下：

Multiplier	Real time (sec)
1	0.233
2	0.246
4	0.250
8	0.265
16	0.308
32	X(如上*圖所示) segmentation fault



＊實驗結果觀察：

我們可以發現，從此結果之趨勢線 $y = 4E\text{-}05x^3 - 0.0009x^2 + 0.0094x + 0.2266$ 可發現，矩陣乘法還是符合 O(N^3)。並可觀察到，隨著 Multiplier 越變越大，所跑出結果的點會越精準的對在此趨勢線上。即此平行化作業後的 MPI_time $\propto$ O(N^3)。依舊符合矩陣乘法的時間複雜度。

5. (10% Bonus) Try to optimize the code. Report your results.

＊ 前情提要：

Not optimized 的 code，在建立出 A，B 矩陣之後，我們使用最簡單的方法。直接讓 master 去對每個 process 都去 broadcast B 矩陣。接著在進行運算，運算後再將所得之部分的 C 矩陣做回傳並進行重組，得到最後的矩陣 C。

未被 optimized 的 code 在下面截圖中的執行檔案命名為：mpi_mm_notopt。

＊優化方法　blocking：

在第三題的時候我們在實驗數據討論的時候提及一件事情，就是未被 optimized 的 code 最大的問題在於 communication overhead。

因此在此優化的部分，我們主要方法是利用 blocking。我們做了兩件事情：

1. 除了將矩陣切割後在進行計算。每一個 process 只會拿到他們所會使用到的 data。以達到不傳送冗餘的資料。

2. 在優化後不再使用 broadcast 去傳一個完整的矩陣 B 給各個 server。改變成每個 process 各自去 generate 所需要之 B 矩陣之部分。

由 1&2，去減少 communication overhead，達到優化的效果。

被 optimized 的 code 在下面截圖中的執行檔案命名為：mpi_mm1。

＊ 實驗中的討論：

然而，在此實驗中，我們發現兩件事情：

第一件是，因為網路的速度很不穩定，我們將欲跑來比較的兩份 exe 檔案，另外寫了 test.sh。讓未優化與優化的程式盡量相同條件下同時跑，使其不被網速影響。

第二件則是，第一筆被 run 的執行檔的時間會特別長。原因則是因為，一開始抓資料的時候產生的 cache miss。為解決此問題，因此我們又修了一下 test.sh，讓(# of processes)/(server) = =1 時的執行檔執行兩次。如同下圖截圖中的紅色框框所示。並且具有 cache miss 的那次執行結果不採納。

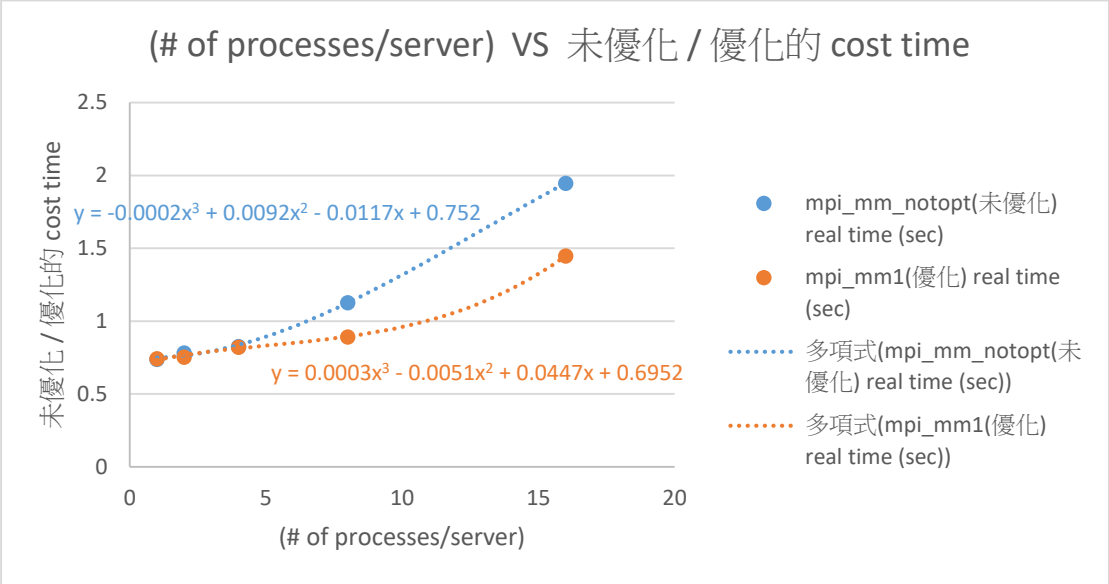
＊ 實驗結果如下圖：

```
b05611047@linux1 [~/HPBDAIS/HW2] ./test.sh
time mpiexec -mca btl_tcp_if_include net0 --npernode 1 --hostfile hostfile -np 8 ./mpi_mm1

real    0m1.025s
user    0m0.126s
sys     0m0.108s
time mpiexec -mca btl_tcp_if_include net0 --npernode 1 --hostfile hostfile -np 8 ./mpi_mm_notopt

real    0m0.678s
user    0m0.122s
sys     0m0.101s
time mpiexec -mca btl_tcp_if_include net0 --npernode 1 --hostfile hostfile -np 8 ./mpi_mm1

real    0m0.742s
user    0m0.132s
sys     0m0.098s
time mpiexec -mca btl_tcp_if_include net0 --npernode 1 --hostfile hostfile -np 8 ./mpi_mm_notopt

real    0m0.739s
user    0m0.119s
sys     0m0.102s
time mpiexec -mca btl_tcp_if_include net0 --npernode 2 --hostfile hostfile -np 16 ./mpi_mm1

real    0m0.754s
user    0m0.172s
sys     0m0.137s
time mpiexec -mca btl_tcp_if_include net0 --npernode 2 --hostfile hostfile -np 16 ./mpi_mm_notopt

real    0m0.783s
user    0m0.177s
sys     0m0.145s
time mpiexec -mca btl_tcp_if_include net0 --npernode 4 --hostfile hostfile -np 32 ./mpi_mm1

real    0m0.821s
user    0m0.248s
sys     0m0.216s
time mpiexec -mca btl_tcp_if_include net0 --npernode 4 --hostfile hostfile -np 32 ./mpi_mm_notopt

real    0m0.826s
user    0m0.209s
sys     0m0.300s
time mpiexec -mca btl_tcp_if_include net0 --npernode 8 --hostfile hostfile -np 64 ./mpi_mm1

real    0m0.893s
user    0m0.473s
sys     0m0.442s
time mpiexec -mca btl_tcp_if_include net0 --npernode 8 --hostfile hostfile -np 64 ./mpi_mm_notopt

real    0m1.128s
user    0m0.351s
sys     0m0.563s
time mpiexec -mca btl_tcp_if_include net0 --npernode 16 --hostfile hostfile -np 128 ./mpi_mm1

real    0m1.477s
user    0m1.093s
sys     0m1.016s
time mpiexec -mca btl_tcp_if_include net0 --npernode 16 --hostfile hostfile -np 128 ./mpi_mm_notopt

real    0m1.946s
user    0m0.740s
sys     0m1.329s
b05611047@linux1 [~/HPBDAIS/HW2]
```

Cache miss!

將截圖整理成表格以即圖表如下所示：

(#of processes) / server	mpi_mm_notopt(未優化) real time (sec)	mpi_mm1(優化) real time (sec)
1	0.739	0.742
2	0.783	0.754
4	0.826	0.821
8	1.128	0.893
16	1.946	1.447



從上圖表結果就可明顯的看出，即使隨著(# of processes/server)上升仍會有 overhead 的產生，但 optimized 過的 code 所 run 出來的 cost time 也會隨著 (# of processes/server) 的上升和未被 optimized 的 code 所 run 出來的結果時間差漸漸拉大。以此證明此次 optimize 實驗中的減少 communication overhead 的做法是有效果的。除此之外還可從趨勢線上升的速率看出，隨著(# of processes/server)後，他們之間的 delta time 越來越大，此 optimize 的效果將會更加明顯。

＊ **code 如附檔所示。**