Parallel programming HW4 Report

103062361 鄭喬中

1. Implementation

(a) How do you divide your data?

我將在 CPU 上切割好的 block 直接對應到 GPU 的每個 block 上去處理。
因為在 GPU 上的每一個 block 最多只有 1024 個 threads，所以我就把清況分為兩種。
第一種為 blocking factor 小於等於 32 時，所使用的方式為每一個距離都直接給一個 thread 去做處理。 E.g.

```
i = threadIdx.x
j = threadIdx.y
dist[i][j] = dist[i][k] + dist[k][j]
```

第二種方式是當 blocking factor 大於 32 時，因為 32*32 會超過 threads 的限制，所以就使用運算 column 或 row 方式來做運算。

e.g.

```
i = threadIdx.x
for j = 0 to B then
    dist[i][j] = dist[i][k] + dist[k][j]
```

(b) How do you implement the communication? (in multi-GPU versions)

in OpenMP:

先將 dist copy 到 mast GPU，然後再 DtoD 的給與 dist。

主要溝通方式是由兩個 GPU 傳送運算完的 data 給 cpu，然後在經由 cpu 去更新 dist，再回送給 master GPU，master GPU 再傳送給另一個 GPU 後才開始運算。（此部分可不經過 CPU，只需要 launch 一個 GPU 去做更新，然後再透過 D2D 的方式去溝通）

In MPI:

兩個 process 個別去計算，並且也各自擁有自己的 GPU，當運算完有 dependence 的 phase2 後，就分別去做 phase3 的部分（每個 GPU 只做自己的部分，不做完整的 phase3），然後再傳回給 CPU，CPU 再透過 MPI 傳送給對分去戶想自己去做更新的動作。
（phase3 可以做成 dynamic 的方式去給予 GPU 去做運算，並且傳送 MPI 時，可以透過 CUDA-aware MPI 方式直接透過 PCIE 傳送給對方的 GPU 直接去做更新的

動作）

(c) What's your configuration? (e.g. blocking factor, #blocks, #threads)

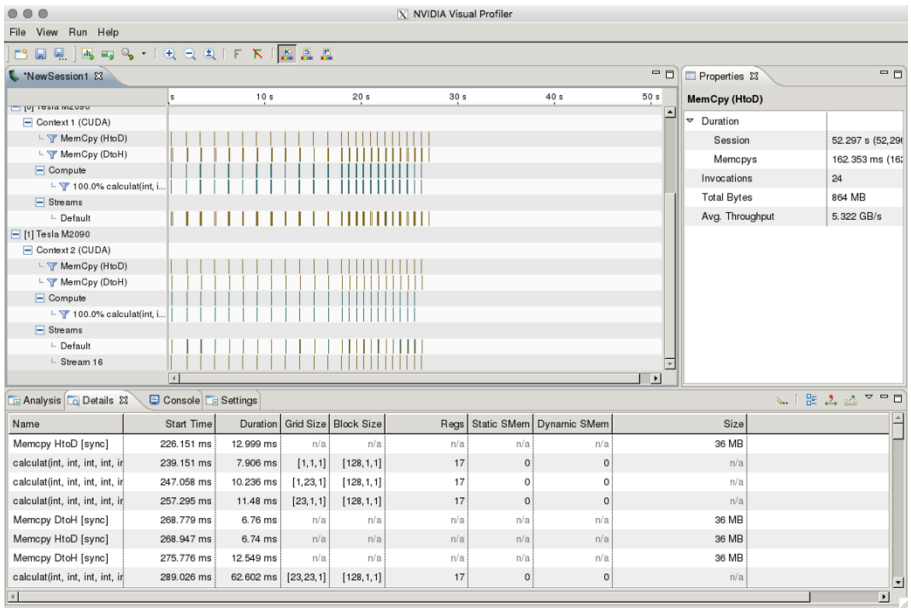
block factor 就是 B，而這便所提的 configuration，上述皆有提到。

Briefly describe your implementation in diagrams, figures or sentences.

2.Profiling Results

```
[user32@gpucluster1 hw5]$ nvprof ./HW4_cuda.exe in5 z 128
==22721== NVPROF is profiling process 22721, command: ./HW4_cuda.exe in5 z 128
#b:128 Total Cuda time = 116.700990 sec (19.300000)
computation time = 14.042231  IO = 102.318629  communication = 0.000000  memcpy = 0.192494
==22721== Profiling application: ./HW4_cuda.exe in5 z 128
==22721== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.31%  13.9498s       413  33.777ms  5.9964ms  234.79ms  calculat(int, int, int, int, int, int, int, int*)
  0.35%  49.249ms         1  49.249ms  49.249ms  49.249ms  [CUDA memcpy DtoH]
  0.34%  48.066ms         1  48.066ms  48.066ms  48.066ms  [CUDA memcpy HtoD]

==22721== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 95.89%  13.6638s       141  96.907ms  3.4380us  260.99ms  cudaDeviceSynchronize
  2.70%  384.47ms       141  2.7267ms  1.4950us  8.9561ms  cudaEventSynchronize
  0.69%  98.181ms         2  49.090ms  48.341ms  49.840ms  cudaMemcpy
  0.66%  93.720ms         1  93.720ms  93.720ms  93.720ms  cudaMalloc
  0.03%  4.7721ms       423  11.281us     457ns  52.060us  cudaLaunch
  0.01%  1.3795ms       282  4.8910us  2.4650us  45.487us  cudaEventRecord
  0.01%  1.3386ms      3384     395ns     269ns  15.508us  cudaSetupArgument
  0.00%  534.64us       166  3.2200us     288ns  112.30us  cuDeviceGetAttribute
  0.00%  292.84us         1  292.84us  292.84us  292.84us  cudaFree
  0.00%  291.72us       141  2.0680us  1.4240us  10.463us  cudaEventElapsedTime
  0.00%  257.89us       423     609ns     296ns  10.560us  cudaConfigureCall
  0.00%  63.474us         2  31.737us  31.197us  32.277us  cuDeviceTotalMem
  0.00%  53.559us         2  26.779us  24.847us  28.712us  cuDeviceGetName
  0.00%  18.656us         2  9.3280us  1.8130us  16.843us  cudaEventCreate
  0.00%  16.365us         1  16.365us  16.365us  16.365us  cudaSetDevice
  0.00%  3.1870us         1  3.1870us  3.1870us  3.1870us  cudaGetDeviceCount
  0.00%  2.6730us         4     668ns     314ns  1.5400us  cuDeviceGet
  0.00%  1.7500us         2     875ns     477ns  1.2730us  cuDeviceGetCount
[user32@gpucluster1 hw5]$ diff -b z testcase/ans5
[user32@gpucluster1 hw5]$
```



透過上面的 profiler 可以幫助我去了解哪些不分需要去做優化，上面例子可以看出在 GPU 計算時是最花時間點部分，也可看出那些 API 最需要花時間。

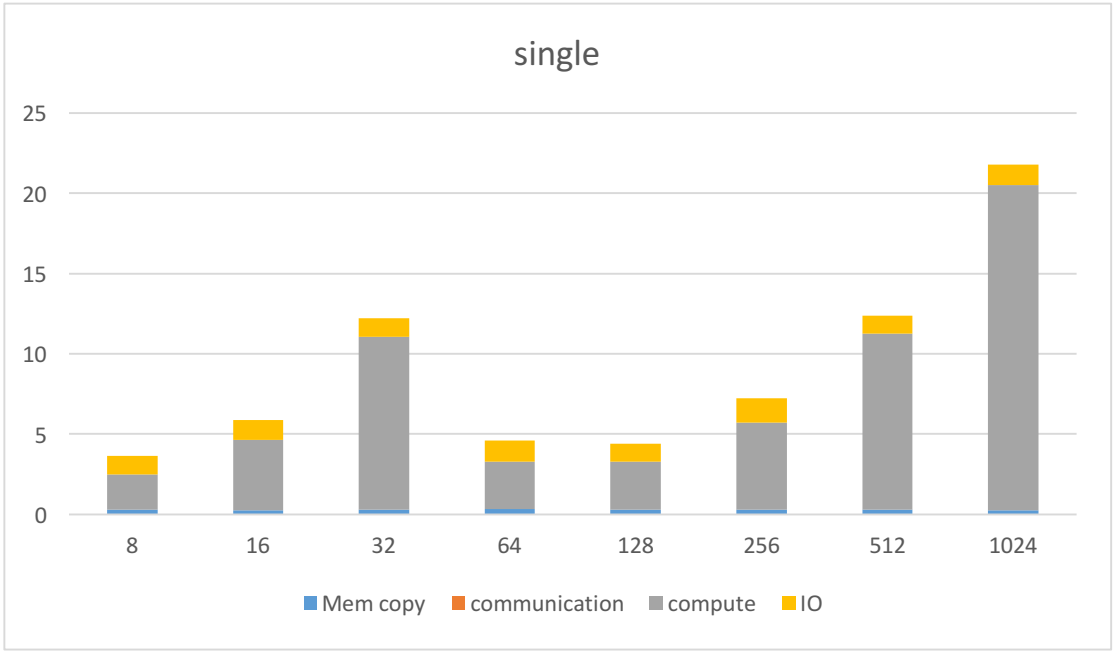
3. Experiment & Analysis

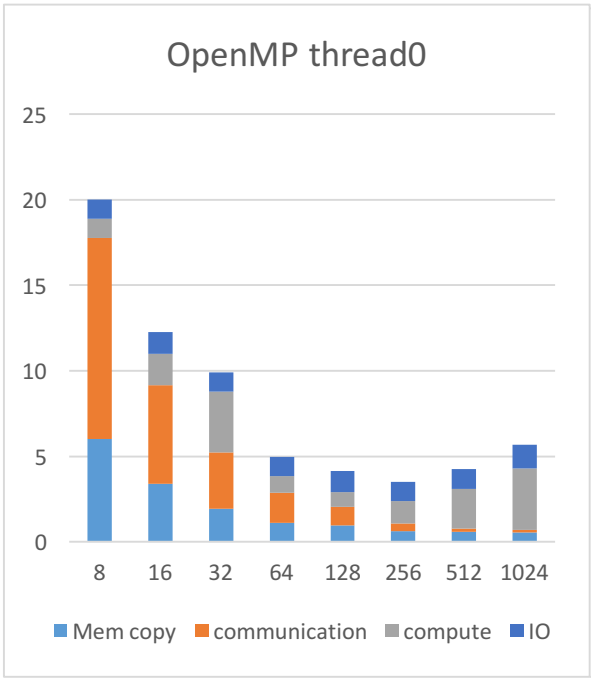
Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz

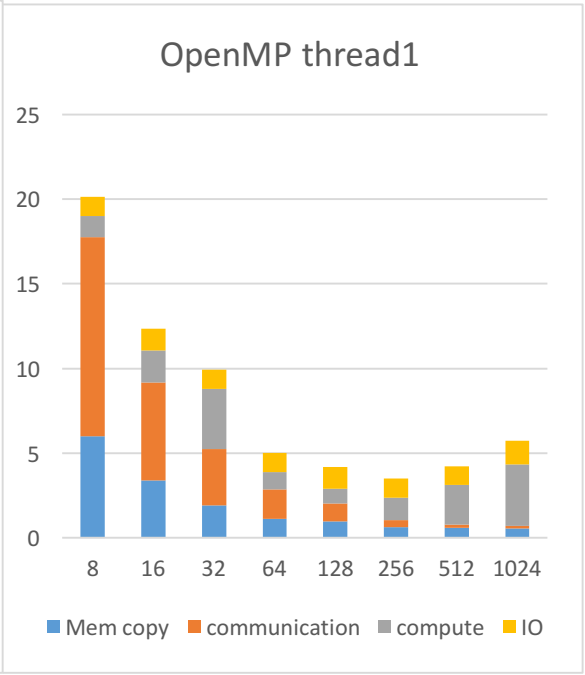
GPU: nvidia tesla K40*2

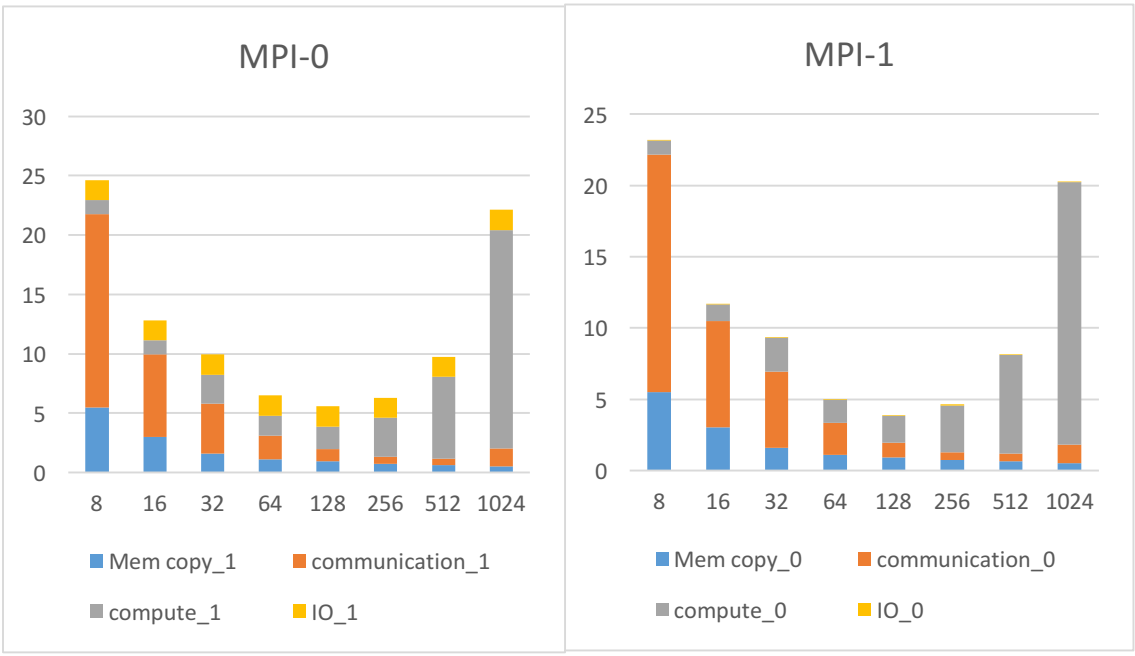
(a)

node 3000 case:in4









由以上圖可看出差異：

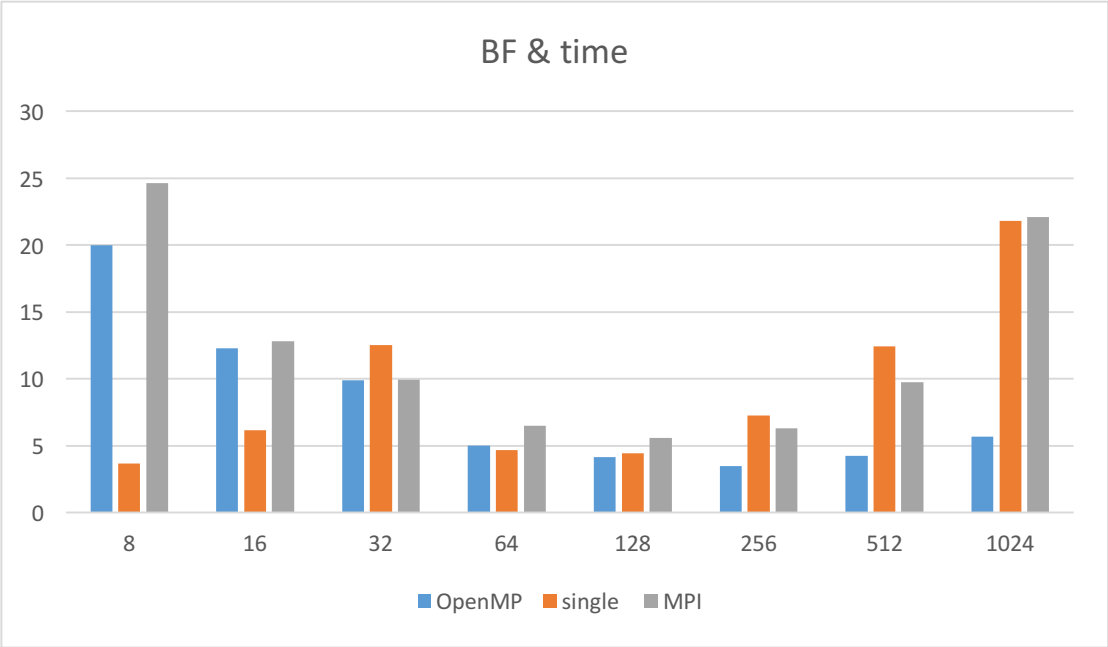
計算時間在 B=32 是分水嶺是因為在大於 32 跟小於的算法吧同，所以會有兩種不同成長方式。

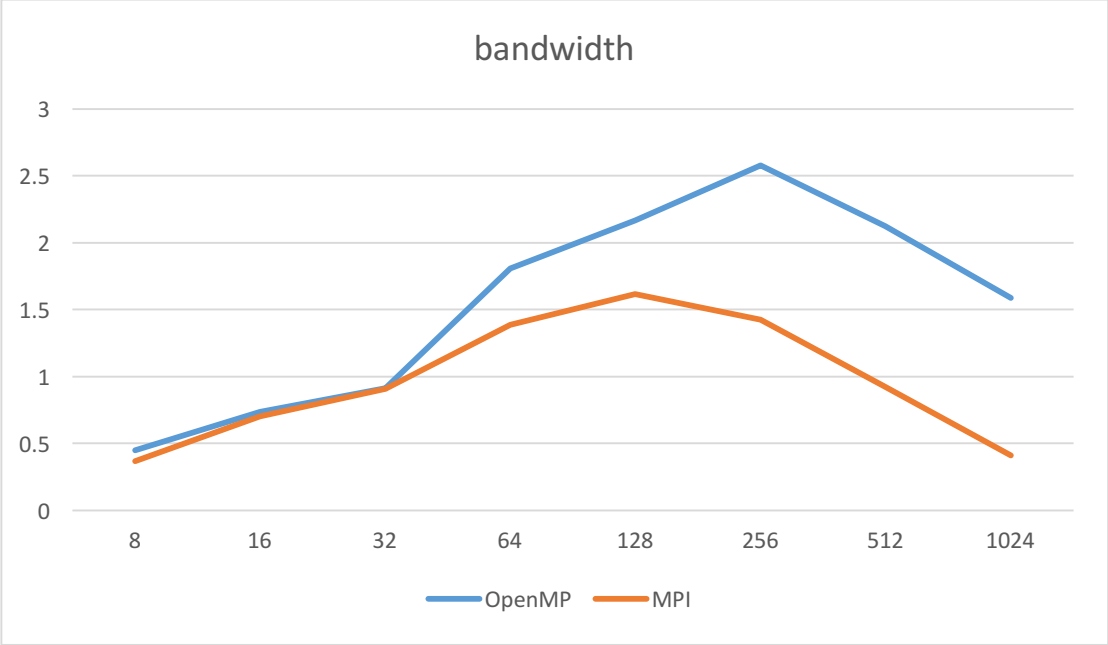
在 single GPU 中因為不需要溝通，所以當 block 越大時，就會被拖慢。

而 OpenMP 的溝通是主要重點，在(c)可看出計算最快的事 openMP，也因為我在 OpenMP 中需要將兩個 GPU 都傳送到 CPU 上才能做溝通，所以會好上很多時間，。

在 MPI 上溝通會隨著 block 月大而減少溝通的次數，進而減少溝通時間。

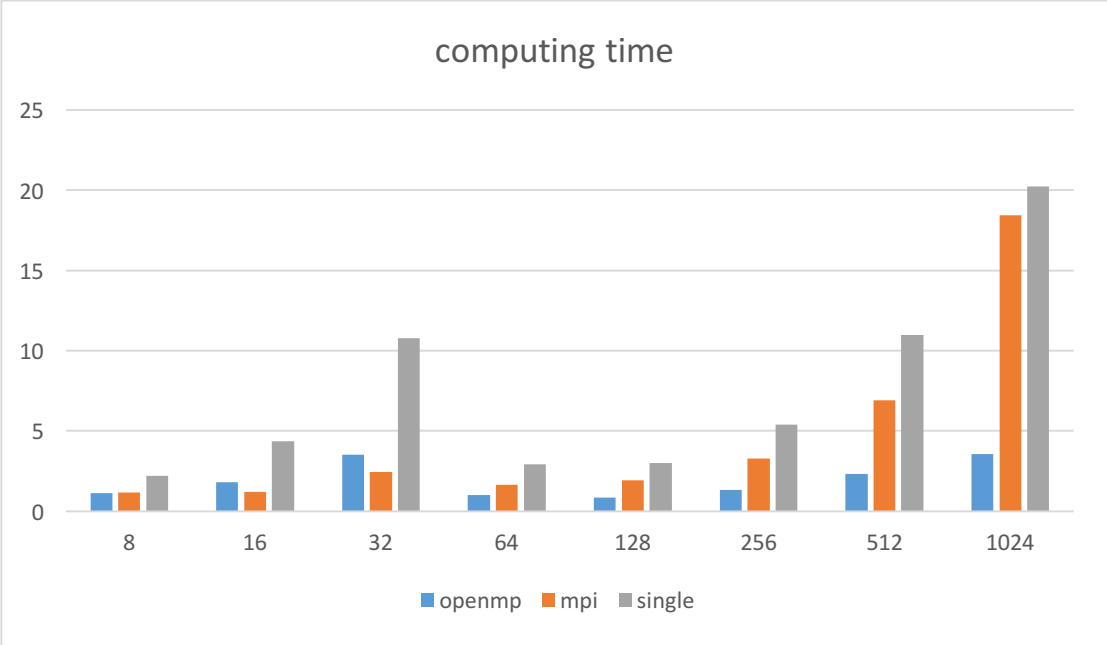
(b)





由上面兩張圖可以看出在 B=128 時，所有的效率都算是最好的，我想應該跟 GPU 的 Warp 有關。

(c)

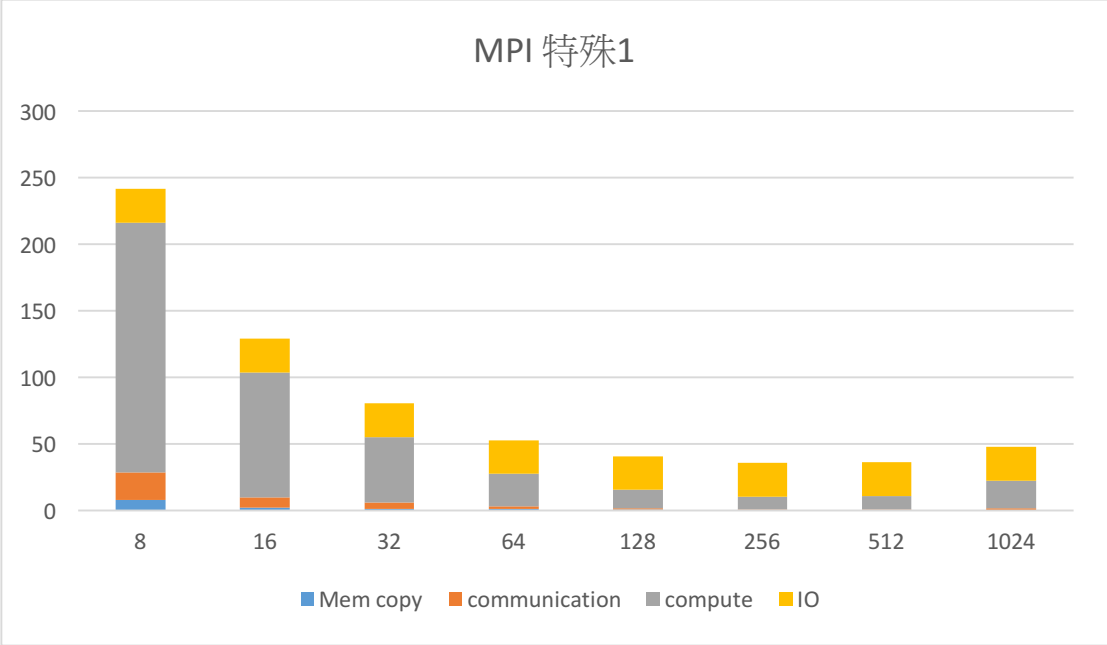


上面可以看到 multi 的計算上都比 single 還要快很多，不過最神奇的是應該要快一倍的運算數度，但由於我只有平行計算 phase3 部分，所以至少要快 1/3，又因平行的部分跟 block 又有關係，因為 phase 執行數量也跟 B 有關係。

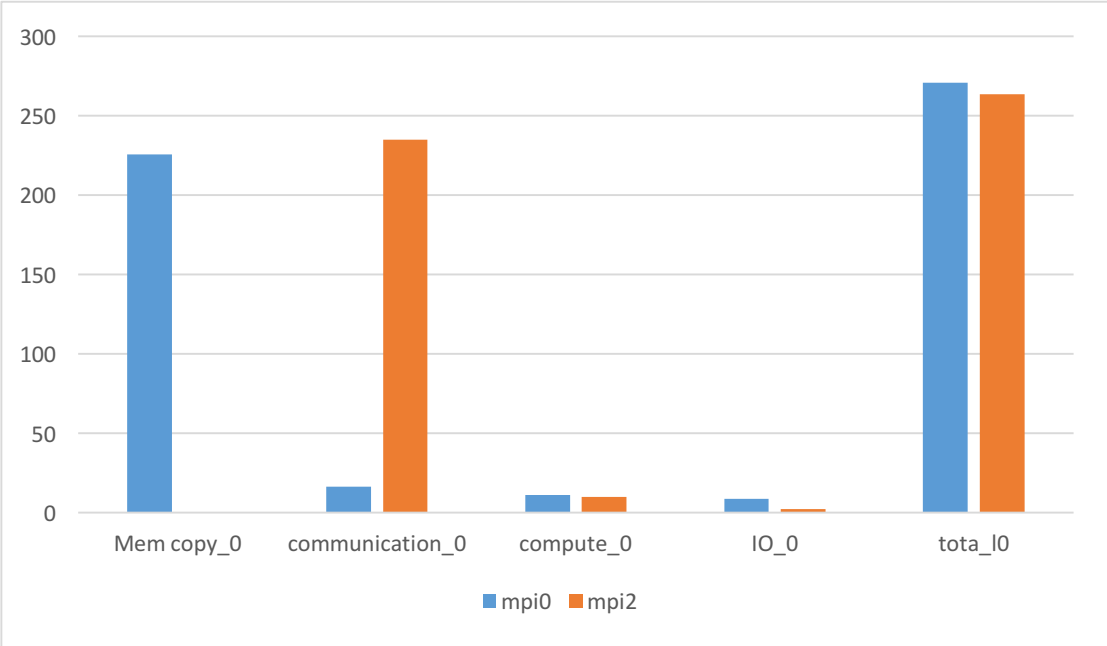
所以由上圖來看是很不合理，但因為我在一些 case 得到的是有 3/4 以上的運算時間都是 phase3，所以說 OpenMP 可以說是符合標準。

但對於 MPI 而言，運算上都跟 Openmp 一樣，但計算時間卻不同，這是最讓我訝異的地方，目前猜測這部分可能是在 run 時 server 過重的關係（跑很多次數據都不好看，可能是 deadline 今天到期有關）。

(d)



node = 6000 case:in5



由上面兩個特殊範，下面那個是 6000 個點的，因為這些資料跟我的中位數都長得很不一樣，所以就特別拿出來說。

第一個是 IO 時間都超級高，第二個是 Memcopy 跟 communication 都超高，而我的推論是認為 MPI 是最容易也最易受這個 server 的 loading 所影響的。

也因為執行 6000 都超久，所以沒有全部跑完，所以這個 report 只有放上 in4 的而已。

4.Experience / Conclusion

除了上面以外，也做過其他實驗，發現到 IO 跟 communication 最容易受到引響，有時可以發現會佔總時間到一半以上，或是大部分時間。

因此上面的主要是去掉 IO 跟 communication 比較不一樣的時候，我是取中位數跟中位數以下做平均所得到的。

而 MPI 不管做幾次效能都很差，所以會在想辦法改進。

這次學到很多，尤其是 GPU 的的架構，雖然我有想很多優化，不過都沒成功，所以會在 Demo 時問助教，優化的 code 都被我註解，希望能 Demo 能學到更多。