戴新顏

編號：28026598

交大學號：0540037

實現

OPENMP STATIC single process 共用記憶體運行，在 axis-x 上平行計算並且使用 static schedule。

各執行緒按照指定 size 反覆運算分配靜態任務。

OPENMP DYNAMIC single process 共用記憶體運行，在 axis-x 上平行計算並且使用 dynamic schedule。

各執行緒按照指定 size 反覆運算動態分配任務。

MPI STATIC mutiple node 不共用記憶體運行，在 axis-x 上平行計算並且使用 static schedule。

各節點根據 rank 根據統一規則在 axis 上分配計算的 point。

分配方式：number of points in x-axis 上連續分配，

例如：number of points in x-axis = 11， node = 3

rank0 = 4 rank1 = 4 rank2 = 3

MPI DYNAMIC mutiple node 不共用記憶體運行，在 axis-x 上平行計算並且使用 dynamic schedule。

預先確定一個中心節點（rank=0 位中心節點），其他節點空閒時向中心節點申請任務，中

心節點不計算，只負責監聽其他節點的請求並分配任務。

其他所有節點會不斷的申請任務並計算， 當申請到一個空任務時便停止計算，向中心節點發送結果。

當計算結束後，中心節點向所有要任務的節點發送一個空任務。然後等待所有節點的發送的計算結果。

HYBRID STATIC　　mutiple node 不共用記憶體運行, 在 axis-x 上平行計算並且使用 static schedule。

HYBRID DYNAMIC　　mutiple node 不共用記憶體運行, 在 axis-x 上平行計算並且使用 dynamic schedule。

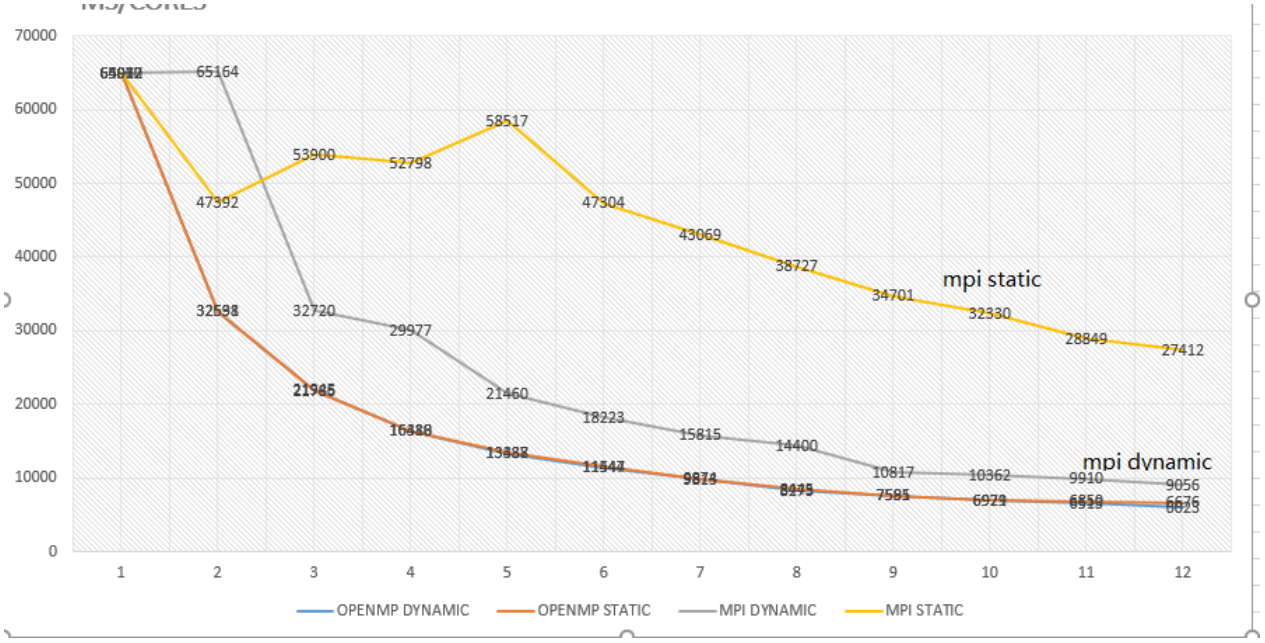
Performance analysis

(a) Scalability chart – strong & weak scalability

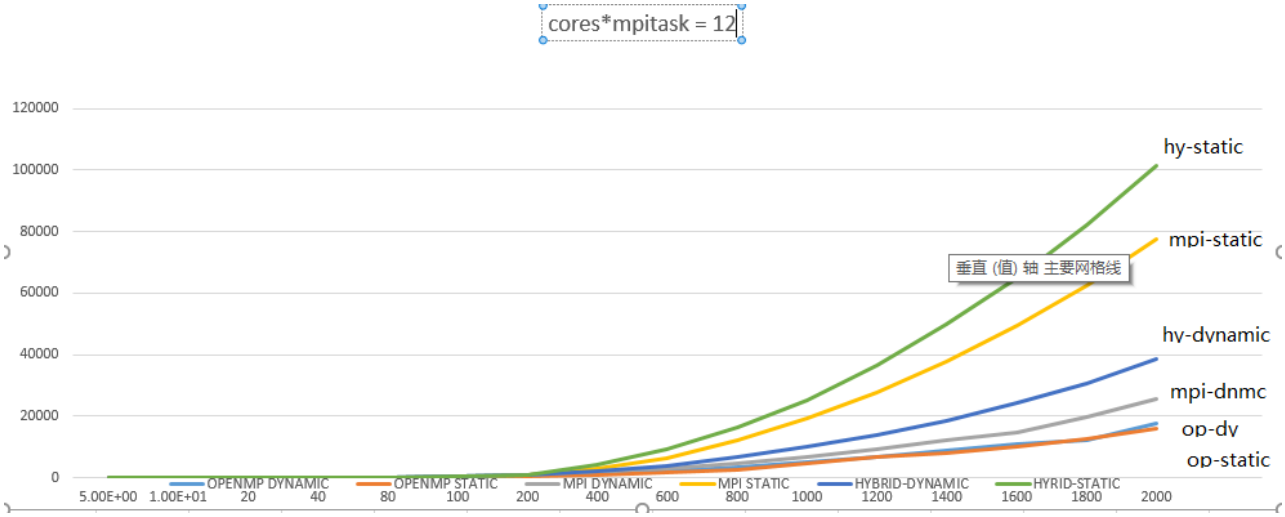
Strong scalability – scalability to number of cores (Problem size is fixed) {MPI, OpenMP}x{static, dynamic}

在 single process 12 nodes 的環境下實驗並測試資料

共用記憶體的 opemp 執行速度快於 mpi， 而預先分配任務 mpi static 執行速度最慢。



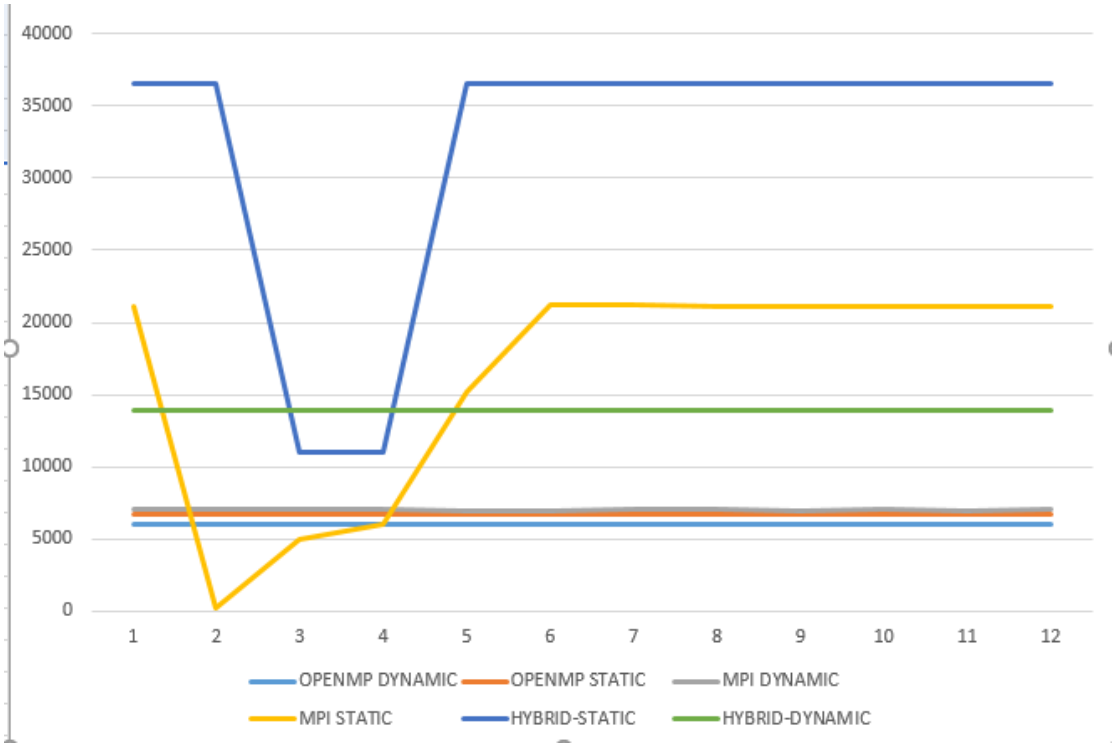
Weak scalability – scalability to problem size (# cores is fixed)
{MPI, OpenMP, Hybrid}x{static, dynamic}



(b) Load balance chart
{MPI, OpenMP, Hybrid}x{static, dynamic}

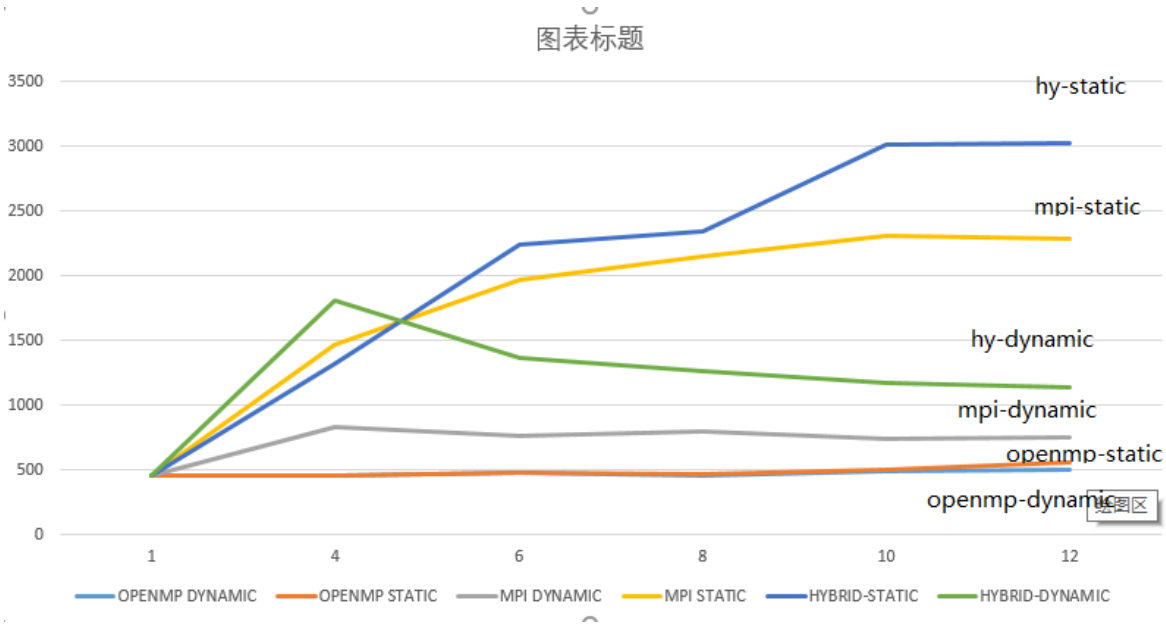
mpi 的 static 版本都出現了 load 不均衡現象，而 openmp 的 static 版本卻沒有。

關於 static 版本在後面有改進，但是實驗資料還是只用舊版本。



(c)node 效率

横坐標是核心數目，縱坐標是單個 node 計算 100*100 個 points 花費的平均時間。



靜態配置策略的優化：

用於測試的程式的任務靜態配置方式是把連續任務平均分配了，導致部分區域計算難度大，

成為瓶頸。

後來改進使用了這一種方式：

對於：number of points in x-axis 上不連續分配，

例如：number of points in x-axis = 11， node = 3

rank0 ：1 4 7 10

rank1 ：2 5 8 11

rank2 ：3 6 9

效果比對：

（以下實驗資料在 12*1cores 環境得到）

下表是執行時間比較：

時間/ms	2000*2000	1500*1500	800*800	200
連續分配	58853	33000	9450	616
不連續分配	15077	8529	2461	168

下表是各 node 執行時間的極差比較：

極差時間/ms	2000*2000	1500*1500	800*800	200
連續分配	58341	32712	9350	605
不連續分配	48	50	53	26

可以看出 不連續分配可以更好的平均分配.

並行佇列優化

如果需要並行化的繪製：

XSetForeground 與 XDrawPoint 這兩個函數式必須並行化的進行的。考慮到這可能成為瓶頸，所以使用了一個 (FIFO) 佇列來存儲計算結果，然後再開啟一個單獨的執行緒從這個佇列中拉取結果來繪製。

在 XSetForeground 與 XDrawPoint 這兩個函數執行成本很高時應當是有效的。

Experience

1. 瞭解學習了 x11
2. 在編寫並行佇列的過程中參考了 KFIFO(當只有一個讀經程和一個寫執行緒併發操作時，不需要任何額外的鎖，就可以確保 KFIFO 是執行緒安全的)，瞭解了 linux 內核以及進而無鎖程式設計技術。
3. 熟悉了 linux 平臺的使用，瞭解了更多的 linux 命令 (ln, vim 的操作)
4. 程式設計上更加熟悉了 MPI 與 OPENMP。
5. 處理不可預測的問題，簡單的按數量可能會導致計算分配不均衡，導致有的機器結束計算了，而有的機器計算還沒有完成一半。更加細分、更加隨機的分配有助於解決這樣的問題。
6. 為了保持高度的動態分配，雖然可以達到各節點負載平衡，但是可能會導致通信成本太昂貴。

7. 程式計算量太小的時候無法準確的估算效率。