HW1: ODD-Even Sort

106062588 王意達

Implementation

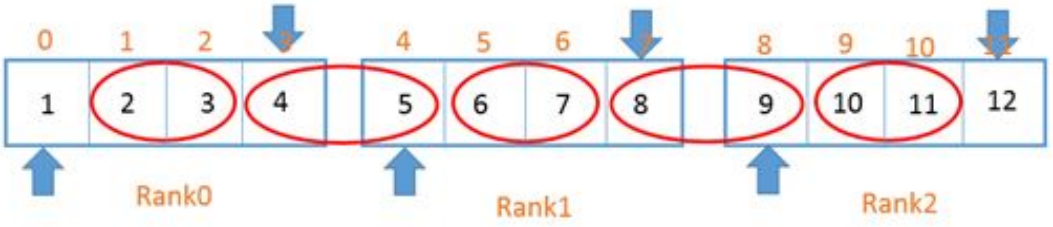
Basic-design concepts:

Odd-Even Sort 是一種會分成odd phase與even phase兩種部分交替運作的排序演算法，本次的實作basic版依然會分成兩種部分，也就是odd phase與even phase交替運作，不同點在於data分散在不同process上執行，不同的process間利用message passing的方式溝通。

Process 內的資料量是奇數或是偶數在實作上必須分開處理，在實作初始階段，我會將N個data平均分給 P個 process，如果遇上N mod P不為0的狀況，表示無法平均分配完所有的data，此時在basic版本中我會將多餘的data分給rank(process 編號)最高的process，以下圖一為例，圖中分別為Rank0-2的process，上方橘色為data的編號、圖中黑色數字為data，下圖(圖一)表示在odd phase時哪些編號的data是會彼此判定需不需要做swap的，因此我們可以清楚看到data編號5、6兩個data分別存在不同的process的記憶體中，如果他們兩者要判定是否要做swap則需要利用message passing溝通。

(圖一)

上圖為odd phase且各process資料個數為奇數的情況，可以發現rank為奇數的process中的最後一個資料必須send 到它rank後一位的process，根據data的編號也可以發現，如果process中其最後一個data的data編號如果為奇數則必須透過message passing傳到下一個rank的process，相對的如果process中第一個data的data編號為偶數則必須將data 傳給前一個rank的process做判斷，除了以上的規律分析外，參考上圖會發現rank0中第一個data的編號為偶數，但因為rank0前並沒有其他編號更小的process，所以必須從上述狀況中做排除，同理，最後一個rank的process也必須額外做限制。



(圖二)

另外，在odd phase時也可能會遇到如上圖二所顯示的狀況，也就是各個process內所分到data個數為偶數個，這邊先不討論最後一個rank分到的data個數，我們可以發現需要做message passing的情況與前一面的圖一相同的，可是如果在even phase 且資料個數為奇數的時候，情況就會如下圖三所示，變成各個process 的最後一個data如果編號為偶數，則必須將該data送到下一個process去判定，而如果該process中第一個data之編號為奇數就必須將該筆data送到比自己rank編號少一的process去做判定，當然rank 0與最後一個rank也是要額外考慮。

圖三

依此類推，在even phase中data個數為偶數時也可以利用前一段的方式歸納出各個process中需要message passing的狀況。

綜合上述討論，我在basic odd-even sort的實作會分為以下步驟。

1. 將 N個DATA平均分給P個process，若無法將data全部分配完則將多餘的data都給rank最大的process。
2. 各個process中根據第一個data的編號來決定其data array做odd-even sort 的起始位置，如圖一所示，可以明顯看出每個process中他們儲存data的陣列的第一個元素，如果它的data編號為偶數，則該process必須從第二個元素開始做兩兩是否要交換的swap判斷，也是紅色圈起來的部分，但如果data編號為奇數，則從第一個element開始做swap的判斷，同理在even phase也是同樣的分析方式。
3. 利用”MPI_Sendrecv”函數實現第二步驟提到的swap判斷，當process接收到其前一個rank的data會與自己現有array中的第一個data做比較，並留下比較大的數，更新到array中的第一個element，如果process接收到後面那個process的data則會與自身最後一個element比較，並留下比較小的那個data。
4. 利用MPI_Allreduce()確認是不是所有的process在此輪都沒有swap發生，表示已經排序完畢。
5. 各個process將各自的data寫入檔案即完成。

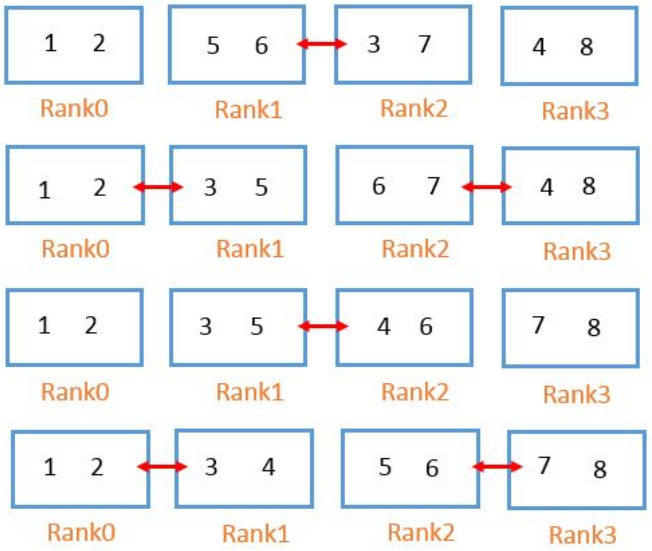
備註:MPI_Sendrecv( )是一將blocking send與blocking receive合在一起的，如此一來就不需要分開使用MPI_Send()與MPI_Recv()，我覺得很方便。

advanced-design concepts:

在advanced-design這部份因為少了element只能與相鄰的data做交換的限制，所以在advanced版本我將採用不同的方式來處理N個需要排序的資料，另外在資料分配上也為了避免如同BASIC版本中需要將多餘的data都配給rank編號最大的process，我原本認為這樣會造成最後一個process有可能因為資料量大於其他process所擁有的，而使得在worst case下拖慢排序時間，也就是最小的data在rank最大的process儲存資料的array的最右方，然而大部分的process需要花更多的時間在等待最小的data傳過來，所以在advanced版本我的資料分配方式是將資料量N與process個數P相除後無條件進位，意即除了rank最大的process外，大部分的process都被分配到⌊N/P⌋個data，而rank編號最大的process可能會拿到比較少量的data，然而這種分配方式可以

使資料的分配較為平均，但其實作後發現message passing的時間可以說與process個數正相關，稍微不平均的分配方式並不會影響performance太多。

在advanced版本中，當每個process拿到data後會先用quick sort排序，這部份我會借助C++ library提供的sort()function，然後相鄰的rank會在不同時間互相傳送彼此擁有的全部data，當拿到相鄰rank傳送過來的資料後再根據需求，利用merge sort中merge的步驟取出需要的data，merge方式我有稍微改良，如下圖四所示，圖中每一列代表某一輪process利用message passing的情形，在每一輪每個process只會對一個process進行資料傳輸，而且只會對後面或是前面那個process進行send()或是recv()的運作，例如圖四中的第一列，rank1與rank2會互相傳遞彼此的data，rank1將{5,6}傳給rank2而rank2將{3,7}傳給rank1，分別拿到資料後會執行merge的步驟，我的merge改良的方法是只需要拿到自己需要的部份資料就停止merge，像是圖四中第一列的rank1拿到{3,7}這兩個從rank2傳送過來的data，它只會分別從自己的local buffer{5,6}與接收到的{3,7}挑出兩個較小的data並存在merge完的buffer內，詳細的步驟為先比較自己buffer內與接收到的資料陣列第一個element並留下小的，像是圖四第一列rank1會宣選到rank2的"3"，而rank1的"5"會繼續跟rank2的"7"比較，然後"5"會被挑到，此時已選到兩個數字，與原本local buff的容量相同，此時停止merge，結果就如同rank1一樣，{3,5}會留在rank1中；不同於rank1，rank2 merge的方式是留下較大的data 可見下圖四第二列rank2中最後保留的data為{6,7}。



圖四

由上圖四顯示，除了rank0、rank3，所有的process皆需與前後的process做資料傳遞，而如果接收到rank-1傳來的data則需要以留下較大值的方式做merge，如果接收到rank+1來的data則是以留下較小值的方式做merge，如果圖四中每一列為一輪的情況，worst case下最多要P輪(P為process數量)才可完成排序，然而每輪的步驟則利用rank的編號做分類，判斷是要往前執行MPI_Sendrecv()還是往RANK+1的方向，Worst case會發生在最大的data在rank0而第一輪時rank0並沒有被選為與rank1做data的傳輸，或是最小值在最後一個rank而第一輪時其process並沒有與前一個rank的process做資料傳輸，所以根據worst case我在advanced版本是強制進行p次的sendrecv與merge。

綜合上述討論，我的advanced-oddeven sort會分成以下步驟。

1. 資料分配給各個process，除了最後一個rank的process，其他process會拿到$\lfloor N/P \rfloor$個data，每個process會有三個array儲存資料，分別為local_buf[ ]、recv_buf[ ]以及out[ ]，local_buff儲存自己拿到的資料，recv_buff則是儲存鄰近process傳輸過來的資料，out[ ]則是local_buf與recv_buf merge過程中資料存放的位子，因為merge方法的關係，out[ ]與local_buf[ ]陣列大小一樣，而recv_buf我則是在不同process中都設定為$\lfloor N/P \rfloor$的大小，即使

倒數第二個rank並不會從最後一個rank拿到如此多的data，但只要在merge的fuction多加一個參數紀錄每個process recv_buff[ ]中真正有用的資料量即可。
2. 各個process利用quick sort排序local buf[ ]內的資料。
3. 利用MPI_Sendrecv()傳送自己local_buff內所有的data到 "rank+1"也就是rank編號比自身大一的process內，而接收到資料的process會利用merge fuction挑出local_buf與recv_buf內較大的data儲存到out陣列中，merge結束後會將out陣列的資料依序寫回local_buf陣列內做更新。
4. 利用MPI_Sendrecv()傳送自己local_buff內所有的data到 "rank-1"也就是rank編號比自身小一的process內，而接收到資料的process會利用merge fuction挑出local_buf與recv_buf內較小的data儲存到out陣列中，merge結束後會將out陣列的資料依序寫回local_buf陣列內做更新。
5. 每個process在每一輪只會執行第三步驟或是第四步驟一次，交替執行共P次後(P為process數)即排序完成。
6.每個process分別寫回檔案。

Other Effort:

在basic版本與advanced版本時，可能會因為process數目大於data數目N的狀況，根據我參考MPI reference的發現，使用MPI_Group_range_excl() fuction call 可以去掉冗餘的process，將data數N分配完後，如果有資料數為零的process則會直接call MPI_Finalize()結束執行。

Experiment & Analysis

System Environment
我是使用平行程式課程提供的Batch Cluster.

Performance Metrics:

Time measurement:

使用MPI_Wtime( )取得的時間進行時間量測，Execution Time的部份，我是在程是碼中MPI_Init( )的後面加上MPI_Wtime( )以及MPI_Finalize( )前面加上MPI_Wtime( )，取兩者差值，由此方式可得知Execution Time ;I/O TIME的取得方式與前者相似，同樣在Read file與Write File的前後加上MPI_Wtime( )來獲取讀檔寫檔的時間，兩者加總後即可得知程式花在I/O的總時間。

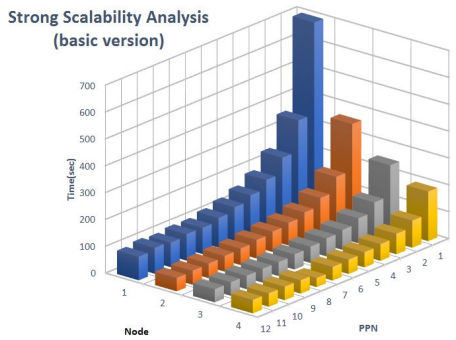
Communication Time比較特別，因為每個process的communication time差異可能很大，所以我有用MPI_Allreduce()來幫助我拿到各個process的communication time來取平均值，各個process中我記錄communication time的方式都相同，也就是在程式碼有MPI_Sendrecv()的部份前後都放MPI_Wtime( )，並取其執行MPI_Sendrecv()前後執行時的時間差值來計算每一輪的communication time，每一輪時間加總後即可得知該process花在communication的總時間。

Computing time我是以Execution time-(I/O time+Communication Time)的方式獲得。

測試資料:

測試資料的部份我是產生一個由997920遞減到1的資料，這在我的排序方式是一個worst case，"997920"這個是數字的由來是根據數字1到12的乘積所產生的48種可能組合的最小公倍數322640在放大三倍所得到的數字，取最小公倍數的倍數是為了保證在任何process總數下都可以整除我的資料總數，減少一個變因。

Strong Scalability Analysis
Basic(N=997920) Strong Scalability
(x-axis:# of process (ppn) ; y-axis: #of node ; z-axis :execution time (s))



在Basic版本中可以由上圖發現，當process數增加後，平行化的效果可以明顯展現出來!也可從上圖發現，Execution time在process增加當一定的量之後，加速的效果會漸趨微弱，由下圖fig.5與fig.6兩張time profile分別看出Node=1與Node=4時在不同process數量的時間分配情況，藍色部分為computation time、橘色部分為I/O time 、紅色部分為communication time，可以明顯看出communication time在process數目上升時會變多，而在Node=4 、ppn(process per node)數為8時可以得到更好的performance。

Basic(N=997920)Strong Scalability - Time profile

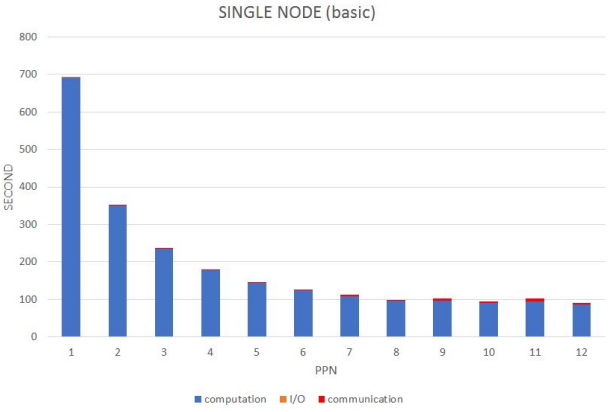


fig.5

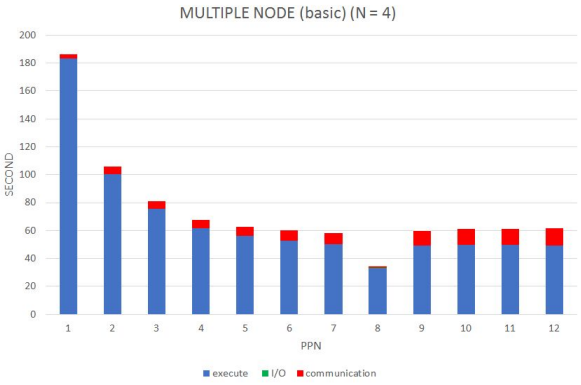
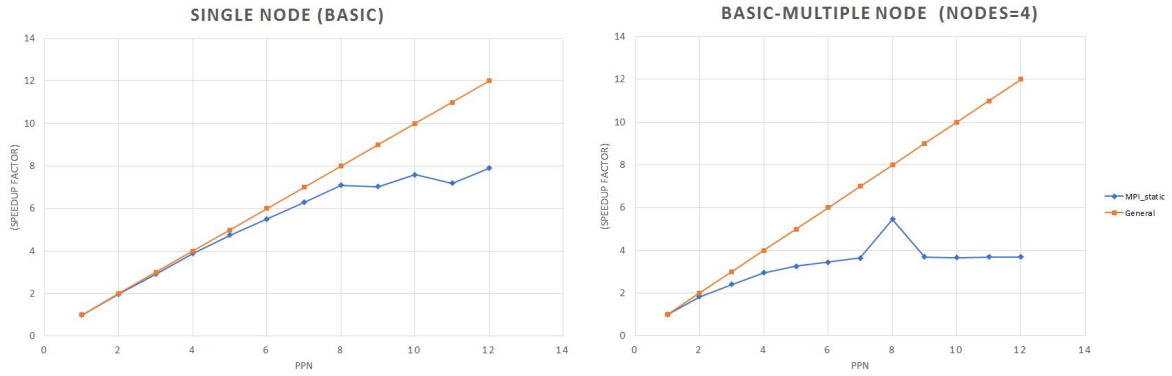


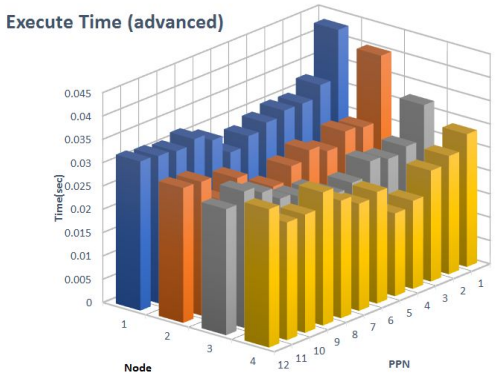
fig.6

Speedup Analysis
Basic(N=997920) (x-axis: # of process per node; y-axis:Execution time(s))



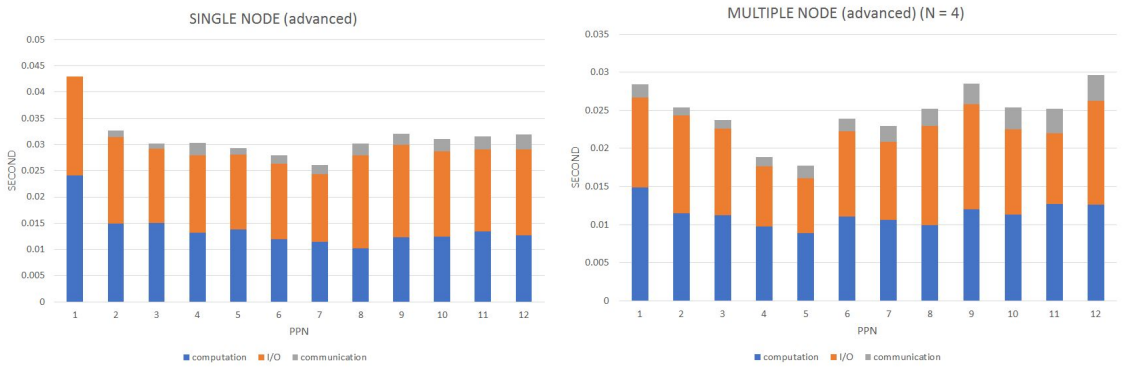
這部份為Single node(Node=1)與Multiple node(Node=4)當ppn增加時所得到的speedup分析，上圖中橘色線為理想的speedup，藍色線為實驗的結果，可以發現BAIC版本中當NODE數為一的時候ppn增加時speedup也明顯上升，但是在multiple node時卻不然，在speedup上升到一定程度時便趨緩了，這表示加速的效果遇到了bottleneck，無法靠著ppn的增加得到想要的speedup，然而同時可以發現Multiple node的圖中，在{node,ppn}={4,8}的時候會出現一個peak，這表示在node數為4的時候，ppn=8為最好的搭配，我推測在node數不同時可以找到不同的ppn來達到該node數下較好的performance。

advanced (N=997920) Strong Scalability Analysis
advanced(N=997920)
(x-axis:# of process (ppn) ; y-axis: #of node ; z-axis :execution time (s))



由上圖的advanced版本的Execution Time再各種node數目下隨著process數目提高並沒有顯著的下降，僅僅在某些node數的情況下ppn數目四以下的結果比較能看出ppn數目變多時execution time有隨之減少，但總的來說我的advanced版本的確大幅降低了整體排序的執行時間。

advanced(N=997920) Strong Scalability - Time profile
(x-axis : # of process per node ; y-axis : Execution time (s))



上圖左側為advanced版本中node=1的time profile;而右側則是node=4的情況，可以看出我的advanced版本的I/O time幾乎與computing time一樣多、剩至超過。

Speedup Analysis
advanced (N=997920) (x-axis: # of process per node; y-axis:Execution time(s))
advanced : blue line　basic : orange line

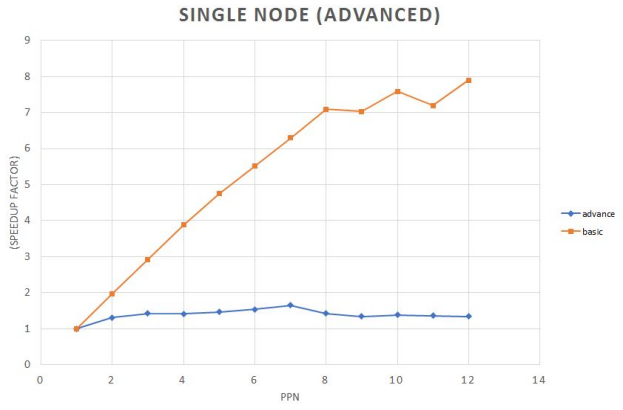


fig.6

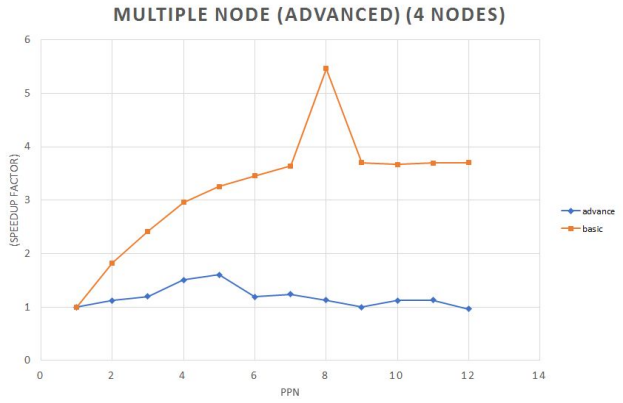


fig.7

以上兩圖是拿advanced版本與basic版本兩者在single node 與 multiple node時隨著ppn增加所得到的speedup做比較，如同前面所述，advanced的效果不如basic明顯。

Discussion:

The advanced implementation is faster than the basic:

從數據來看baic版本最快也要49秒左右，而advanced版本則可以到0.021秒，明顯快很多，若以時間複雜度分析更可以看出兩者的差異。

以下N代表data量，P代表process數量:

basic-time complexity : O(N^2/P)

sequential 版本的odd-even sorted時間複雜度是O(N^2) ，簡單來說平行化的時間複雜度與Process數目有關，更仔細的來分析，每個process之local buffer所儲存的data數量為N/P個，所以只要有任一個Process在上一輪還有做swap的動作，每個process就需要同時花O(N/P)的時間做是否swap的for loop判斷，而最多O(N)輪，因為worst case 為data的組成是N到1的遞減數列，所以basic版本的時間複雜度為O(N^2/P)。

advanced-time complexity : O(⌊N/P⌋ lg ⌊N/P⌋)

advanced版本主要時間分為幾個部分，1. local的data是以quick sort的方式進行排序，故時間複雜度為O(⌊N/P⌋ lg ⌊N/P⌋)，2.merge的部份與merge完後資料必須更新到local buffer，這部分兩者做一次的時間複雜度都是O(⌊N/P⌋)，然而最多做O(P)次，同樣因為在worst case下rank 0 中擁有最大的數字N，至少要經過P次移動才能到最後一個rank的process中；由上述分析可知advanced版本的主要時間複雜度仍然是 O(⌊N/P⌋ lg ⌊N/P⌋)。

Bottleneck:

在bassic版本中，可參考fig.6 可發現computing time為主要的瓶頸，local buffer的swap判斷花上了大量的時間，這部份可以利用其他排序演算法做改良，像是我再advanced版本就大大降低了computing time，然而在advanced版本，I/O time大都略高於computing time，兩者都為advanced版本的瓶頸，若需改善則需要更快速的演算法來解決computing time 的bottleneck，或許Bucket sort演算法可以幫助到這部份的改良，另外在I/O time的部份，因為已經使用MPI_IO進行平行的存取檔案所以比較難以改良這部份的時間，這與硬體有較大的關係。

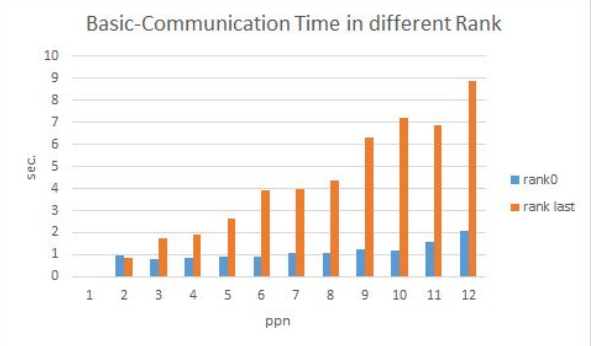
Scalability:

在basic版本我的scalability不錯，可由basic strong scalability 那部份的圖看出端倪，因為basic主要的弱點在於computing 的部份太花時間，而process增多的情況，使得平行化的效果大大的展現出來，然而我的advanced版本，scalability效果不明顯的主要原因是I/O time的bottleneck，computing time的比例可能會隨著測資變大而縮小，然而I/O time依然還會占很大一部分，畢竟I/O time是從記憶體或是其他硬體讀、寫資料這會花上大量的時間，另外從performance的成果來看，我很滿意我advanced達到的成果。

Other Analysis:

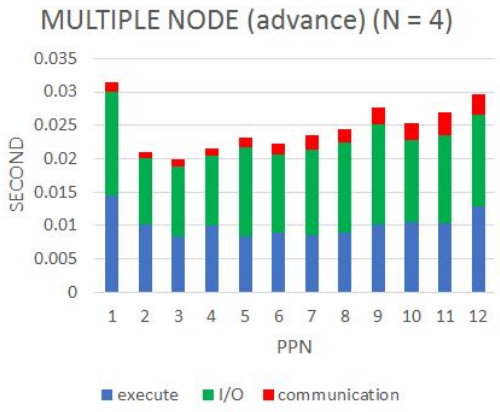
Part A : Communication Time in different rank

右圖為basic版本中，node=1時，rank0與rank last(最後一個編號的processor)兩者個communication time，可以發現差異很大，原因其一為每個process執行速度不同，所以可能有的process早早就做完send data的動作，可是鄰近的process卻還沒有完成，此時process就會等待資料的傳輸，再



者分割的不平均也會造成溝通時間的不同，所以我再取樣communication time的時候是用全部process所記錄的時間取平均。

Part B : fread() VS. MPI_read_at( )



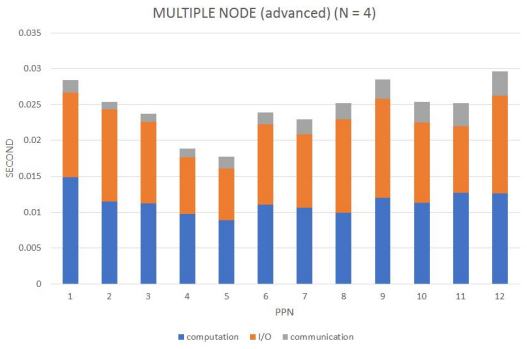
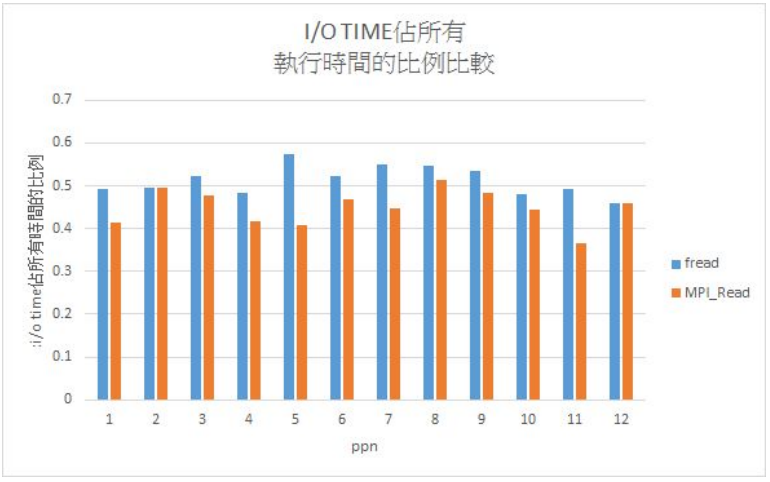


fig.fread

fig.MPI_Read_at



以上三圖是拿來比較使用fread( )與使用MPI_Read_at( )兩者的差異，三圖都是再Node=4的情況下做測試，可由fig.fread看出使用fread的I/O時間較fig.MPI_Read_at的長，另外看上圖，上圖為各個ppn下 I/O time/Execution time 的比例，可以發現在大多數情況下，fread的使用會使得I/O的瓶頸更為嚴重! fread 的壞處在於，每個process需要讀取N個data的資料後再選擇真正被分配到的資料執行，然而MPI_IO可以提供更方便的平行讀取方式，可以增加Performance。

Part C : advanced with different merging method

這部份我將介紹我的advanced版本的一個重大優化過程，也就是我改善了merge得部份，原本的merge方式為將相鄰process傳來的data大小為O(m)陣列，與自己原本儲存資料得大小為O(n) 得local buffer根據下列pseudo code實作，並且可得知這種merge方式的時間複雜度為O(m+n)。

```
//recv_buf[ ]存取收到得data，local_buf[ ]存取原本的data，out[ ]儲存merge後的data
while(recv_buf與local_buff都尚未scan完成) do
        if( local_buf[i ]< recv_buf[j] )out[outcount++]=local_buf[i++];
                else out[outcount++]=recv_buf[j++];
while(local_buf[ ]尚未scan完成)do
        out[outcount++]=local_buf[i++];
while(recv_buf[ ]尚未scan完成)do
        out[outcount++]=recv_buf[j++];
```

我後來改良的merge方式已經在前面有提到，下圖fig.9測資是為322640000遞減到1得數列，可以看出新的merge方式的確有改善performance!

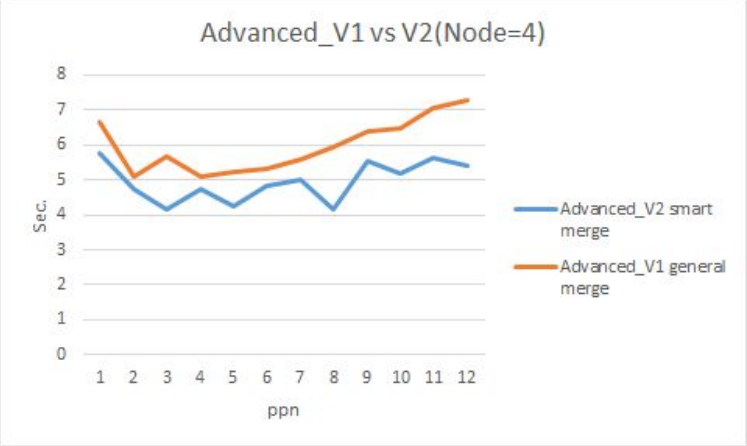


fig.9

Part C : DATA分配方式比較

我在BASIC版本與advanced版本兩者的資料分配方式不同，basic版本是將Node數除process數後取整數，advanced版本則是取ceil( )，由於advanced版本速度較快，所以我實測了兩者分配資料的方式對最後一個rank編號的process之communication time得影響，會取最後一個process討論的原因是資料如果無法分配平均，以[N/P]的方式作資料分配，最後一個Process的data量可能會多出其它process很多!以下圖fig.10為例，我們可以看出ceil(N/P)的分配方式會稍微改善了最後一個rank的communication時間，但整體來說我們可以由圖的趨勢發現communication time主要是受process數目影響。

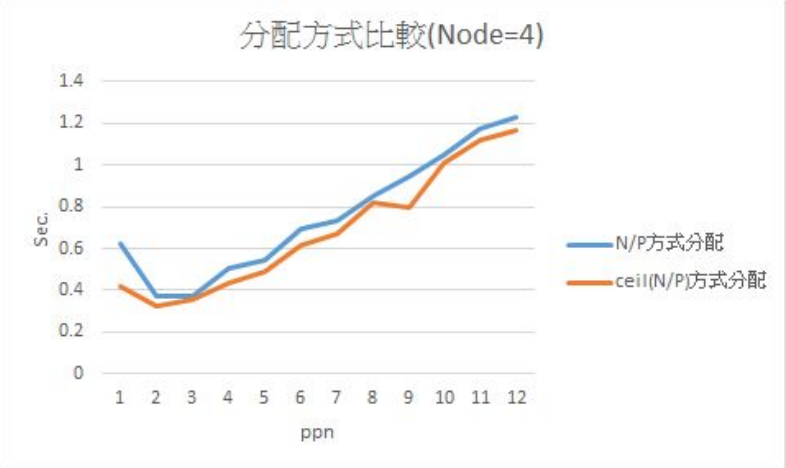
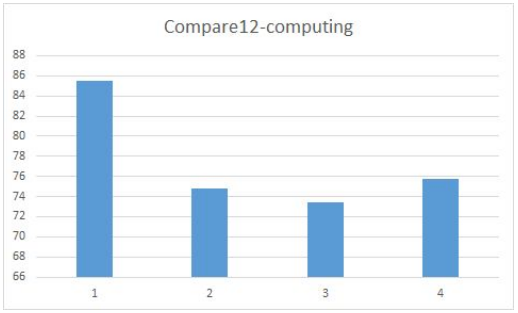
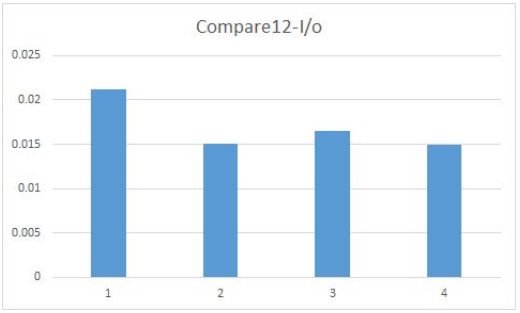


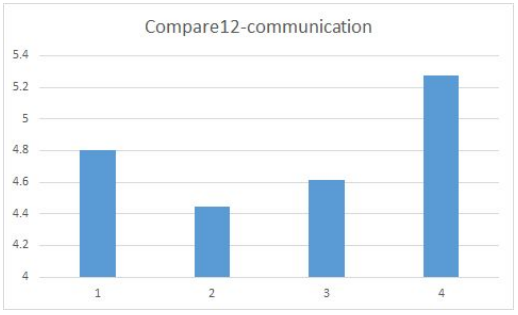
fig.10

Conclusion :

Basic版本的scalability效果顯著，隨著process數目增加往往可以得到更好的performance，然而process數目相同時，但Node數目不同時是否會得到相同的performance提升呢?我用以下三圖分別比較node數為1到4且總共process數目都是12的狀況時Computing time, I/O time 和Communication Time的差異。







可以發現除了communication time以外，node數目為一的時候I/O time與computing time都明顯比其他node數目的多，I/O的部份我猜測是與讀取I/O的BUS有關，可能是經由一個Node讀取資料後再傳到給其他process，而computing time的部份其實差十秒的差距，在量測時並不會太誇張，我時常在不同時間測試相同變因的數據，像是{node,ppn}={1,10}我就時常拿到不一樣的數據，每個個體可能會相差10秒左右，測試機台的不穩定，使得記錄數據時不需測驗多次後取平均值比較妥當，computing time的部份node=1明顯較多，我猜是同一個node中12個core被分配到的資源或許比較少，比較值得討論的是communication time，由圖可發現，node=4時，因為橫跨不同node的溝通大量增加而有明顯的增加而大於其他node數的狀況，另外，如同我在先前提到的，communication time會隨著process增加而上升。

在advanced版本中我數度嘗試了不同的方式改善basic版本的performance，我們可以發現利用高等排序的演算法可以明顯提高速度，但由於我並沒有針對quick sort演算法做平行化，而是利用不同process自己執行quicksort後在merge起來，所以scalability並沒有顯著的表現出來，但只看{node,ppn}={1,1}與{4,12}時可以發現其execution time分別為0.0429秒與0.0296秒，可看出process變多仍然有scalability的效果，最重要的是，與basic比較我得到了滿意的performance提升。

Experience:

我在這次的作業裡學到了MPI library的使用方式、make file的製作以及unix-like系統的操作方式，另外根據實驗結果發現，平行化的技巧處理大量的數據是未來必須要學會的技巧，而這次的作業也讓我學到不少經驗，算是令我獲益良多的一次實作作業。

在這次的作業中basic版本的實作方式我想了一天才想出來，可見得對於平行化的知識尚為不足，由於我的coding技巧並不強，實作過程中常常出現debug一整天的情況，尤其感謝實驗室好夥伴羅嘉諄跟我一起研究討論方法還有debug，在advanced的版本中，光是buffer的設定、offest的設定就常常出錯，導致我這次的作業實作的過程是弄到焦頭爛額的，雖然是份很辛苦的作業，但其實我很樂在其中，資料的分割、merge的方式我就實作了不同種出來，慢慢的改良我的advanced版本，這是非常有成就感的事情!

由這得作業我裡解到這門課的loading是很重的，我覺得未來的作業難度會越來越高，勢必得更早的開始準備，其實最花時間的竟然是蒐集數據這環，如果花了大把時間coding最後因為該拿得數據沒時間拿就吃虧大了。

Comment:

這次的數據是以float為主，且有限制只能與附近的process溝通，不知道以後可不可以把溝通的限制拿掉，當作是加分的項目?感覺會有更多新奇有趣的想法被大家實作出來。

參考網站:

MPI reference : https://www.mpich.org/static/docs/v3.2/www3/index.htm