Parallel Programming Homework1
106065509　林庭宇

1. Implementation

1.1. MPI_IO和process初始化

首先必須先設定好每個process拿到的float數量，這裡是用的方式是判斷process的rank是否小於num_size%proc_size，若是則list_size = num_size/proc_size + 1，否則list_size = num_size/proc_size。如此一來每個process拿到的數量便只可能為兩相異值，且兩值差為1，可以視為平均分佈。知道每個process的list_size，就可接著利用global index來做MPI_IO，做完MPI_File_read_at()，每個process的list就準備好了，list的大小為list_size。

1.2. Basic version

邏輯的架構就如同spec上的圖一樣，透過Even phase和Odd phase的交替來達到排序的效果，這邊就不贅述。值得一提的是，因為我的實作中每個process拿到的數量雖然是平均分佈的，但就會有不同的list_size出現在不同process的現象。這樣造成儘管有相同的list_size，因為是在不同的process，有不同的head_gindex和tail_gindex，所做的動作便會不一樣。因此我在實作的時候，自己列了一個truth table，方便實作。

head_gindex	tail_gindex	swap_type	Even Phase		At First		At Middle		At Last		Local Loop	
			Head要傳	Tail要傳	Head要傳	Tail要傳	Head要傳	Tail要傳	Head要傳	Tail要傳	開始index	結束index
Odd	Odd	0	T	F	N/A		T	F	T	F	1	list_size-1
Odd	Even	1	T	T	N/A		T	T	T	F	1	list_size-2
Even	Odd	2	F	F	N/A		N/A		N/A		0	list_size-1
Even	Even	3	F	T	F	T	F	T	F	F	0	list_size-2
head_gindex	tail_gindex	swap_type	Odd Phase		At First		At Middle		At Last		Local Loop	
			Head要傳	Tail要傳	Head要傳	Tail要傳	Head要傳	Tail要傳	Head要傳	Tail要傳	開始index	結束index
Odd	Odd	0	F	T	N/A		F	T	F	F	1	list_size-2
Odd	Even	1	F	F	N/A		N/A		N/A		1	list_size-1
Even	Odd	2	T	T	F	T	T	T	T	F	0	list_size-2
Even	Even	3	T	F	F	F	T	F	T	F	0	list_size-1

雖然讓人一點也不想看，但是有了truth table便可以很方便地去實作，以下大略講解一下truth table的內容。

首先是判斷頭尾global index的奇偶性，以swap_type作為分別。接下來則是分成Even phase和Odd phase來探討，頭尾需不需要跟前後的process傳值交換，但這只是general case。我們還必須考慮process是在第一個(At First)、最後一個(At Last)、中間的情形，而因為head_gindex為奇數的process不可能在第一個，所以值為N/A，若原本就皆為False那也是以N/A表示。考慮完了頭尾的情況，還必須考慮local loop做交換的開始和結束index。

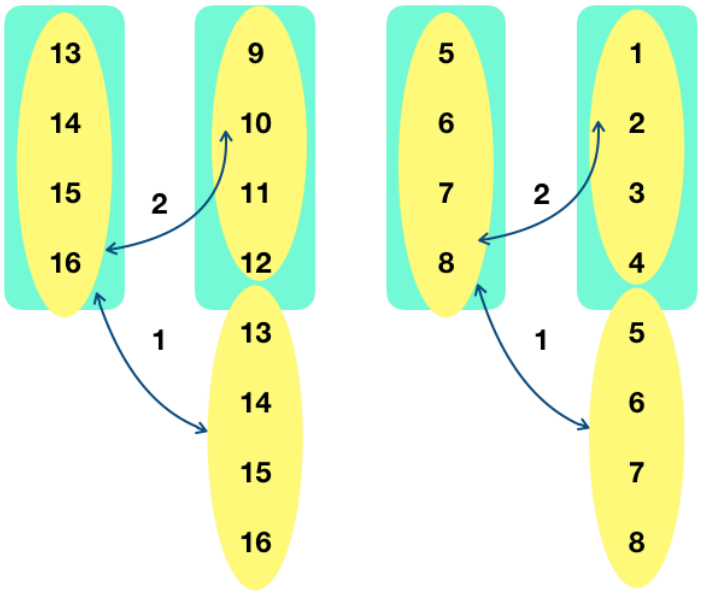
最後就以switch和if/else的架構來實作這張truth table，以期待達到最佳的效能。

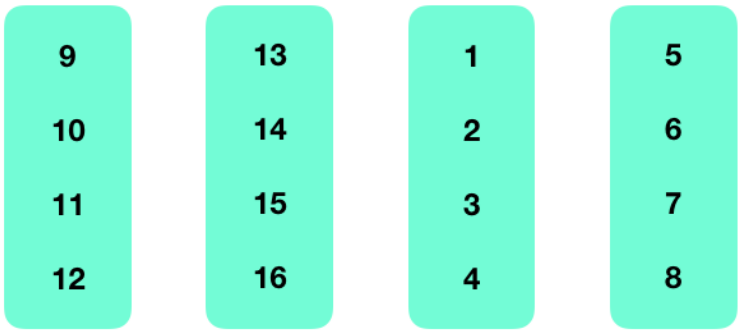
1.3. Advanced version

在advanced中因為沒有限制send/receive的數量，所以可以先對每個process內的list做local sort，再把整串list交給下個process，和該process的list merge後，再傳回。這樣的做法可以看作廣義的Odd/Even sort，當Even phase時rank為even的process向後傳遞，Odd phase則相反，rank為odd的process往後傳遞。

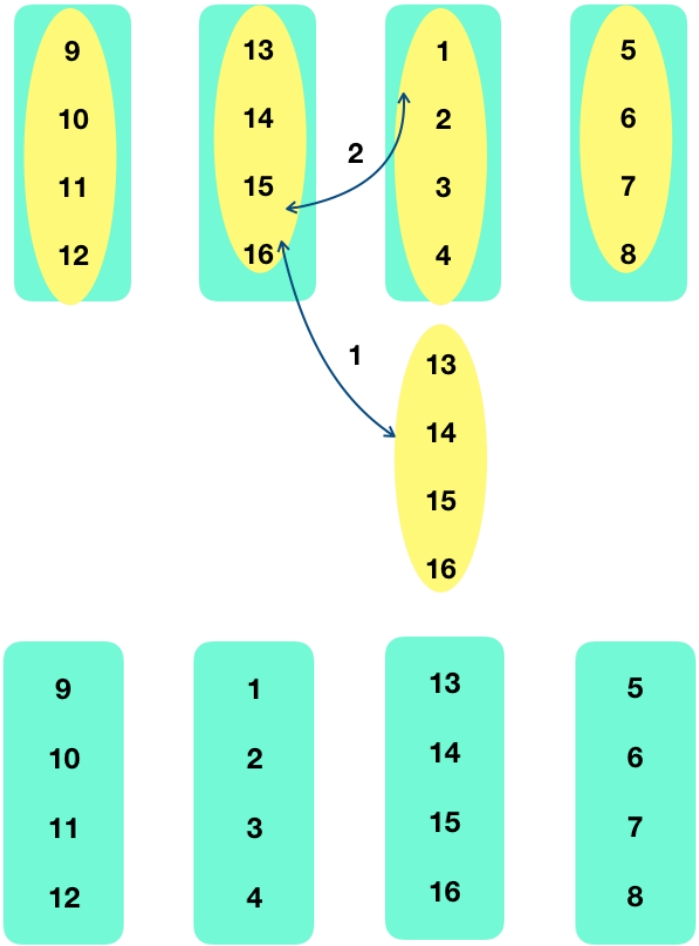
以下用簡單的圖表來呈現，首先將各個process的list做sort，接下來進入Even/Odd phase。

Even phase: rank為偶數的向後傳，rank為奇數的排序merge兩個list，把前一半往回傳給rank為偶數的process。





Odd phase: rank為奇數的向後傳，rank為偶數的排序merge兩個list，把錢一半往回傳給rank為奇數的process。



另外還會特別判斷當proc_size=1的時候，就直接做一次sort就結束。而在sort選擇上，我是使用內建的std::sort，雖然保證的time complexity也是O(nlogn)，和常被使用的merge sort一樣，但經過實測std::sort在大測資時的表現更好，將在下一章提到。

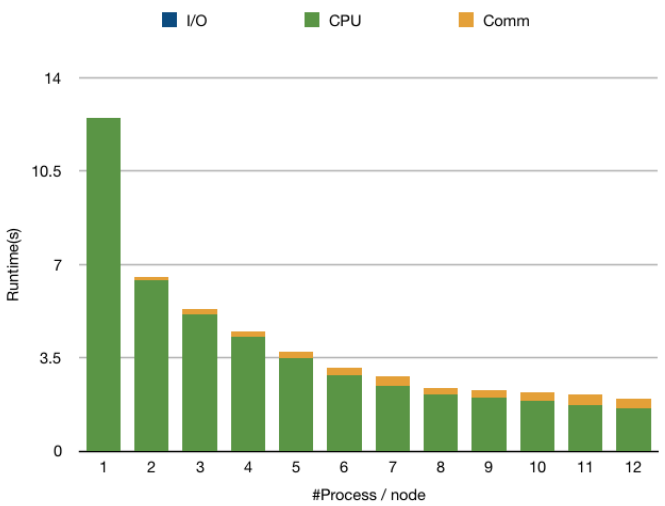
2. Experiment & Analysis

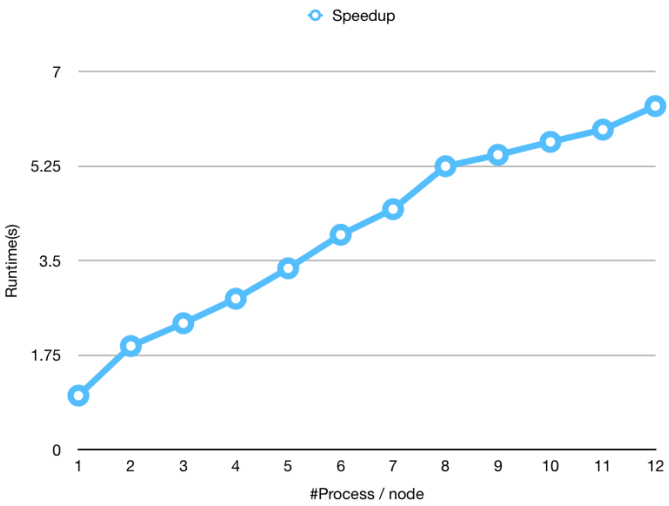
2.1. Problem metrics

我是使用clock_gettime(MONOTONIC)來抓時間，這個function的精度可以達到10^-9。我將執行的時間分為三種類：

- I/O time，指的是MPI_IO的時間，
- CPU time，指的是在local的loop時間，basic是odd-even sort，advanced則是std::sort，
- Communication time，指的是MPI_Isend跟MPI_Recv所花的時間。

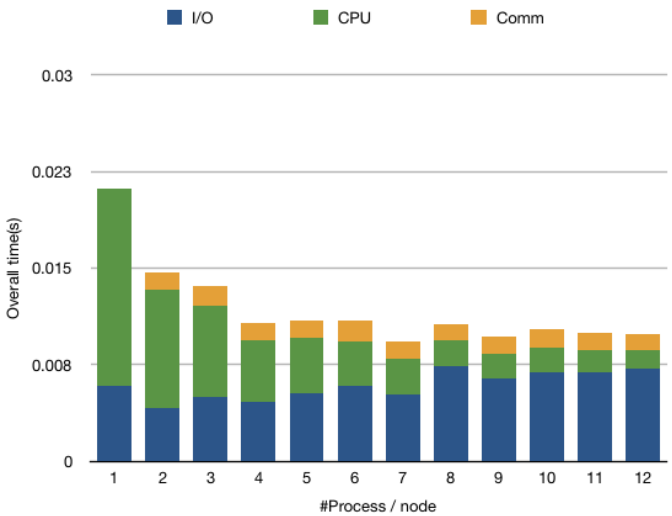
2.2. Basic on Single node (n = 10^5)

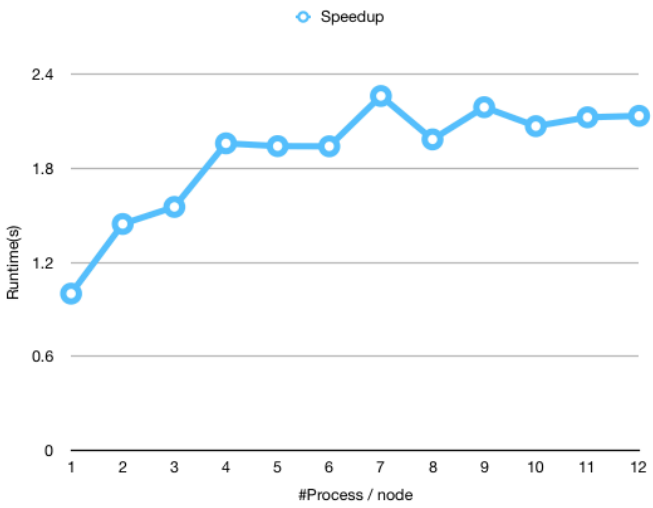




可以看到很明顯的speed up factor，CPU time是最主要的時間。

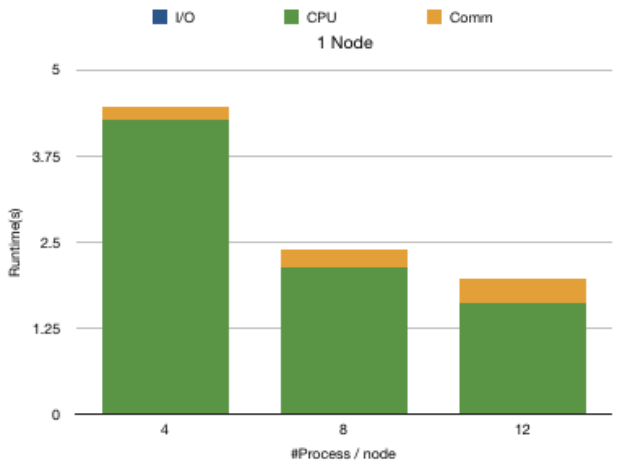
2.3. Advanced on Single node (n = 10^5)

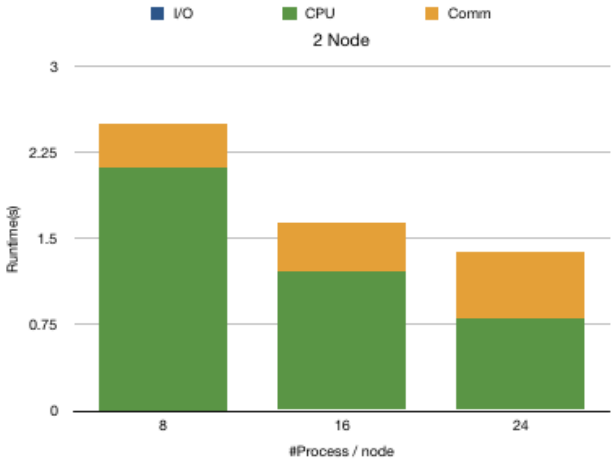


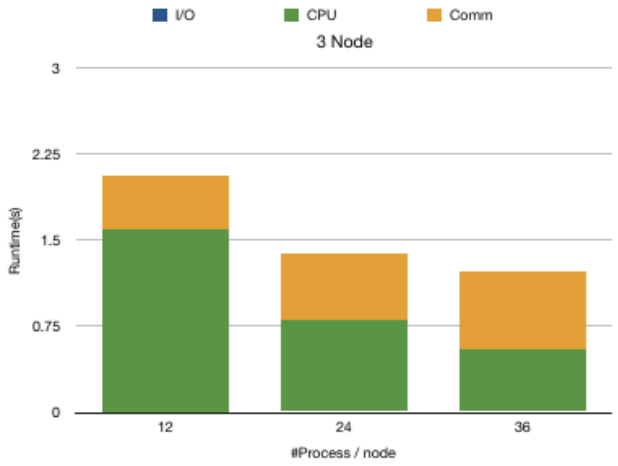


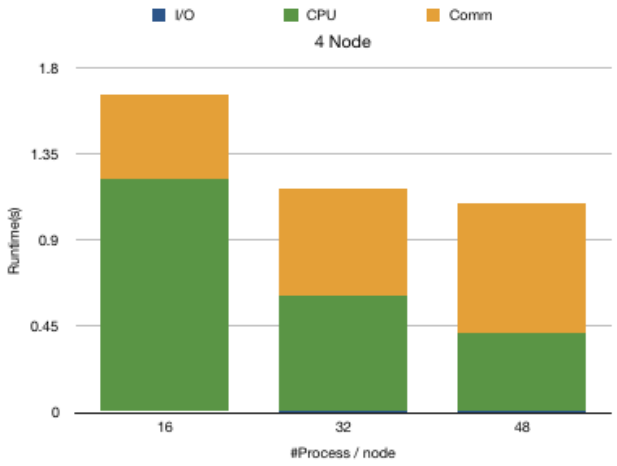
Advanced跟Basic的差異很明顯，因為CPU時間大幅縮短，整個I/O的overhead就凸顯出來了，所以speed up的成長很有限。

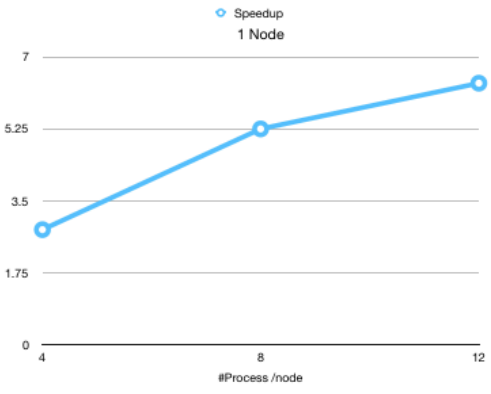
2.4. Basic - Multi node (n = 10^5)

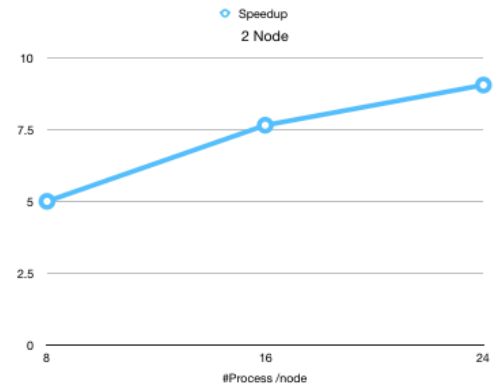


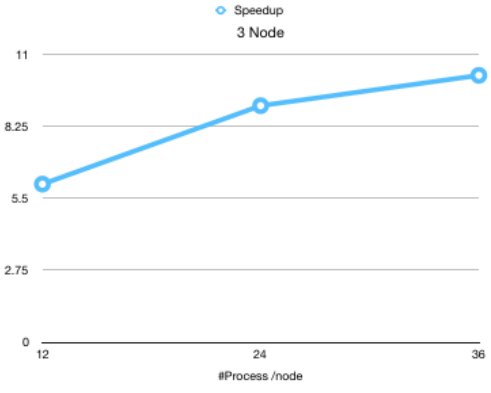


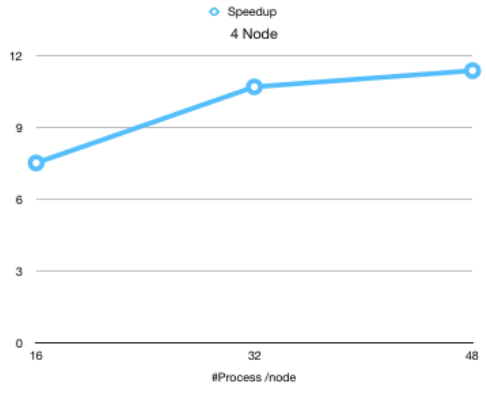












可以清楚地看到CPU time降得很多，Speed up factor也很接近linear。

2.5. Advanced - Multi node (n = 10^9)

因為知道(n = 10^5)的testcase size對advanced來說太小，因此我就自己生了一個1g的testcase，並以此比較merge sort和std::sort的差別。

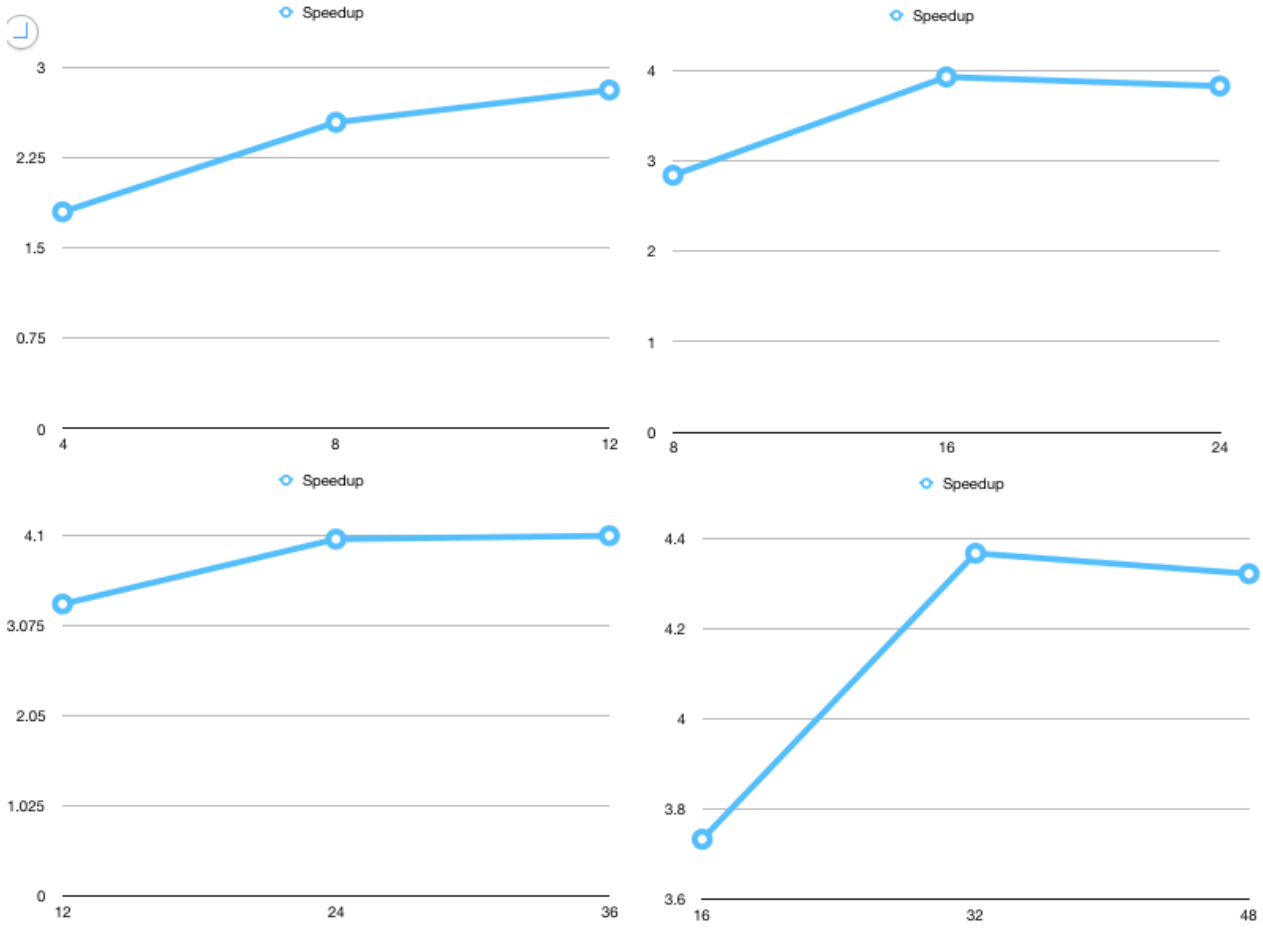
Advanced_Multi_merge_1G

#Node	#Process	I/O	CPU	Comm	Total	Slurm
1	4	10.842751	57.227245	6.621223	74.691219	28075
1	8	6.928076	31.595196	8.515867	47.039139	28125
1	12	7.385641	22.637097	10.266523	40.289261	28175
2	4	5.430254	29.687703	5.659325	40.777282	28233
2	8	6.845938	16.493340	7.209908	30.549186	28243
2	12	3.700872	12.843274	9.282096	25.826242	28295
3	4	6.722273	20.075360	5.097811	31.895444	28344
3	8	7.354158	11.660502	6.546795	25.561455	28359
3	12	5.240060	9.815104	9.007722	24.062886	28370
4	4	5.884496	15.820476	5.261343	26.966315	28421
4	8	5.257146	9.513111	6.583694	21.353951	28424
4	12	6.414176	8.194368	8.823246	23.43179	28450

Advanced_Multi_std_1G

#Node	#Process	I/O	CPU	Comm	Total	Speedup
1	4	8.510364	35.092968	6.700288	50.30362	1.80245855069675
1	8	6.459055	20.271170	8.879145	35.60937	2.54624527196072
1	12	7.303951	14.800480	10.107227	32.211658	2.81482530331099
2	4	7.232889	18.925245	5.717341	31.875475	2.84451259157707
2	8	4.610428	11.233857	7.205288	23.049573	3.93370367425028
2	12	4.054883	9.352914	10.247342	23.655139	3.8330017845171
3	4	8.723615	13.335716	5.251961	27.311292	3.31987919136158
3	8	7.297943	8.309122	6.740827	22.347892	4.05721443436365
3	12	5.776491	7.365849	8.997055	22.139395	4.09542311341389
4	4	7.954644	10.894748	5.433437	24.282829	3.73392202366536
4	8	7.023761	7.001649	6.735037	20.760447	4.36744883190617
4	12	5.652325	6.437030	8.888586	20.977941	4.32216822423135

可以看到std::sort大概會快到2~3秒。以下是std::sort的speedup圖。



3. Experience/Conclusion

學到了很多平行程式的相關知識，也透過實作更佳清楚明白。