HW1

112062706 林泳成

- HW1
  - Implementation
    - 1. Handle arbitrary number of input items and processes
    - 2. Sorting
    - 3. End detection
    - 4. Other implementation detail
  - Experiment & Analysis
    - Performance matrix
    - Profile
    - Speedup
    - Discussion
  - Conclusion

Implementation

1. Handle arbitrary number of input items and processes

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Group world_group, new_group;
MPI_Comm new_comm;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);

int arr_size = atoi(argv[1]);
char *input_filename = argv[2];
char *output_filename = argv[3];

// rank的資料量
int dataSize = rank < (arr_size%size) ? arr_size/size + 1 : arr_size/size;
// rank+1的資料量
int nxtSize = rank + 1 < (arr_size%size) ? arr_size/size + 1 : arr_size/size;
// 參與計算的process數量
int involveSize = arr_size > size ? size : arr_size;
int processIncluded[involveSize] = {0};
for(int i=0;i<involveSize;++i)
    processIncluded[i] = i;
MPI_Group_incl(world_group, involveSize, processIncluded, &new_group);
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
```

由於每個process都會需要知道自己和rank+1 process的資料量，所以需要計算自己的資料量dataSize(rank較低的process可能會比平均item數還多一個)，以及rank+1 process的資料量nxtSize(直接計算下一個process的資料量)

如果item數量比process少，那就只需要開item數量個process就好(用involveSize表示)
並且把會參與運算的process全部加到一個新的communicator裡，後面溝通就不會卡到沒參與的process

```
// 是否參與計算
bool involved = false;
// 讀檔起始點
int beginOffset;
if(arr_size % size == 0)
    beginOffset = rank*(arr_size/size);
else
    beginOffset = rank <= (arr_size%size) ? rank*ceil(arr_size, size) : rank*ceil(arr_size, size)-((rank-(arr_size%size)+size)%size);

if(dataSize > 0){
    arr = (float*)malloc(dataSize*sizeof(float));
    res = (float*)malloc(nxtSize*sizeof(float));
    involved = true;
}
else{
    beginOffset = 0;
}

// Open file
MPI_File input_file, output_file;
if(dataSize > 0){
    MPI_File_open(new_comm, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &input_file);
    MPI_File_read_at(input_file, sizeof(float) * beginOffset, arr, dataSize, MPI_FLOAT, MPI_STATUS_IGNORE);
    MPI_File_close(&input_file);
}
```

beginOffset用來記錄讀檔的起始點
若dataSize == 0，process就不需要宣告儲存item的空間，也不需要讀檔

2. Sorting

1. Sort in each process

```
if(involved)
    boost::sort::spreadsort::spreadsort(arr, arr + dataSize);
```

為了要做merge sort，每個process內部的item必須先排序好，才能開始merge
原本是用qsort，後來聽很多人都用spreadsort，所以去簡單看了一下
spreadsort
spreadsort好像是用bit運算硬體加速達到O(n)的時間複雜度
spreadsort也可以直接根據參數的type等價於integer_sort、string_sort、float_sort

2. Sort between process

```
int swapper(float* arr, float* res, int dataSize, int nxtSize){
    float tmp;
    int nonSwapped = 1;
    float* combine = (float*)malloc((dataSize + nxtSize)*sizeof(float));
    int arrPtr = 0, resPtr = 0;
    for(int i=0;i<dataSize+nxtSize;++i){
        if(arrPtr == dataSize){
            combine[i] = res[resPtr];
            resPtr++;
            continue;
        }
        else if(resPtr == nxtSize){
            combine[i] = arr[arrPtr];
            arrPtr++;
            continue;
        }
        if(arr[arrPtr] > res[resPtr]){
            combine[i] = res[resPtr];
            resPtr++;
        }
        else{
            combine[i] = arr[arrPtr];
            arrPtr++;
        }
    }
    for(int i=0;i<dataSize;++i){
        if(combine[i] != arr[i])
            nonSwapped = 0;
    }
    for(int i=0;i<dataSize;++i)
        arr[i] = combine[i];
    for(int i=0;i<nxtSize;++i)
        res[i] = combine[dataSize + i];
    return nonSwapped;
}
```

發送item array的process和接收的process會做merge
小的item會在最後複寫在接收的process裡，大的會寫回給發送的process
如果發現小的item都和接收process長一樣就會紀錄**nonSwapped = 1**，作為後續的終止判斷

```
while(involved && size > 1){
    start = MPI_Wtime();
    int nonSwapped = 1;
    // even phase
    if(!(round & 1)){
        // odd rank
        if(rank & 1){
            // MPI_Send(arr, dataSize, MPI_FLOAT, rank-1, 0, new_comm);
            // MPI_Recv(arr, dataSize, MPI_FLOAT, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
            MPI_Sendrecv(arr, dataSize, MPI_FLOAT, rank-1, 0,
             arr, dataSize, MPI_FLOAT, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
            // MPI_Sendrecv_replace(arr, dataSize, MPI_FLOAT, rank-1, 0, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
        }
        // even rank
        else{
            if(rank+1 < involveSize){
                MPI_Recv(res, nxtSize, MPI_FLOAT, rank+1, 0, new_comm, MPI_STATUS_IGNORE);
                nonSwapped = swapper(arr, res, dataSize, nxtSize);
                MPI_Send(res, nxtSize, MPI_FLOAT, rank+1, 1, new_comm);
            }
        }
    }
    // odd phase
    else{
        // even rank
        if(!(rank & 1)){
            // MPI_Send(arr, dataSize, MPI_FLOAT, rank-1, 0, new_comm);
            // MPI_Recv(arr, dataSize, MPI_FLOAT, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
            MPI_Sendrecv(arr, dataSize, MPI_FLOAT, rank-1, 0,
             arr, dataSize, MPI_FLOAT, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
            // MPI_Sendrecv_replace(arr, dataSize, MPI_FLOAT, rank-1, 0, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
        }
        // odd rank
        else{
            if(rank != 0 && rank != involveSize-1){
                MPI_Recv(res, nxtSize, MPI_FLOAT, rank+1, 0, new_comm, MPI_STATUS_IGNORE);
                nonSwapped = swapper(arr, res, dataSize, nxtSize);
                MPI_Send(res, nxtSize, MPI_FLOAT, rank+1, 1, new_comm);
            }
        }
    }
```

實作even phase和odd phase

odd phase的odd rank需要檢查第0個process不發送訊息

優化merge過程

```
int swapper(float* arr, float* res, int dataSize, int nxtSize, bool smallToLarge){
    int nonSwapped = 1;

    int arrPtr = 0, resPtr = 0, combinePtr = 0;
    // 小rank排序
    if(!smallToLarge){
        if(arr[dataSize-1] <= res[0])
            return nonSwapped;
        while(arrPtr < dataSize && resPtr < nxtSize && combinePtr < dataSize){
            if(arr[arrPtr] < res[resPtr]){
                combine[combinePtr] = arr[arrPtr];
                ++arrPtr;
            }
            else{
                combine[combinePtr] = res[resPtr];
                ++resPtr;
            }
            ++combinePtr;
        }
        while(combinePtr < dataSize){
            if(arrPtr < dataSize){
                combine[combinePtr] = arr[arrPtr];
                ++arrPtr;
            }
            else{
                combine[combinePtr] = res[resPtr];
                ++resPtr;
            }
            ++combinePtr;
        }
    }
```

```
    // 大rank排序
    else{
        if(arr[0] >= res[nxtSize-1])
            return nonSwapped;
        arrPtr = dataSize - 1;
        resPtr = nxtSize - 1;
        combinePtr = dataSize - 1;
        while(arrPtr >= 0 && resPtr >= 0 && combinePtr >= 0){
            if(arr[arrPtr] > res[resPtr]){
                combine[combinePtr] = arr[arrPtr];
                --arrPtr;
            }
            else{
                combine[combinePtr] = res[resPtr];
                --resPtr;
            }
            --combinePtr;
        }
        while(combinePtr >= 0){
            if(arrPtr >= 0){
                combine[combinePtr] = arr[arrPtr];
                --arrPtr;
            }
            else{
                combine[combinePtr] = res[resPtr];
                --resPtr;
            }
            --combinePtr;
        }
    }
    for(int i=0;i<dataSize;++i){
        arr[i] = combine[i];
    }
    return !nonSwapped;
}
```

不完全merge兩個array，而是指merge到dataSize長度就好
就可以省下一些array的讀寫
同時讓每個process都merge自己需要的array，增加資源使用效率

3. End detection

```
MPI_Barrier(new_comm);
MPI_Allreduce(&nonSwapped, &swappedTotal, 1, MPI_INT, MPI_SUM, new_comm);
if(swappedTotal == involveSize){
    if(finishOnePhase)
        break;
    else
        finishOnePhase = true;
}
else
    finishOnePhase = false;
++round;
MPI_Barrier(new_comm);
```

每次merge完都會做一次Allreduce檢查process之間是否有交換item，若都沒有交換就表示已經sort完成

4. Other implementation detail

```
// MPI_Send(arr, dataSize, MPI_FLOAT, rank-1, 0, new_comm);
// MPI_Recv(arr, dataSize, MPI_FLOAT, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
MPI_Sendrecv(arr, dataSize, MPI_FLOAT, rank-1, 0,
 arr, dataSize, MPI_FLOAT, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
```

有試著把Send和Recv合成一個Sendrecv，但好像沒有太大差別，還是需要等訊息

優化後

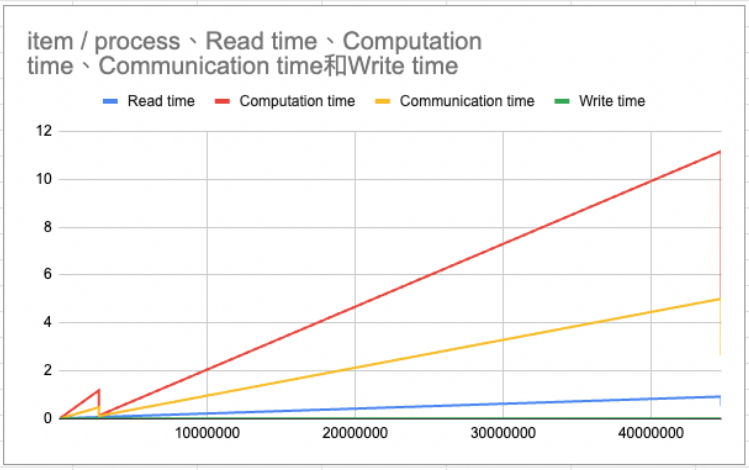
```
while(involved && size > 1){
    // start = MPI_Wtime();
    int nonSwapped = 1;
    // even phase
    if(!(round & 1)){
        // odd rank
        if(rank & 1){
            // MPI_Send(arr, dataSize, MPI_FLOAT, rank-1, 0, new_comm);
            // MPI_Recv(arr, dataSize, MPI_FLOAT, rank-1, 1, new_comm, MPI_STATUS_IGNORE);
            MPI_Sendrecv(arr, dataSize, MPI_FLOAT, rank-1, 2,
             res, preSize, MPI_FLOAT, rank-1, 2, new_comm, MPI_STATUS_IGNORE);
            // start1 = MPI_Wtime();
            nonSwapped = swapper(arr, res, dataSize, preSize, true);
            // end1 = MPI_Wtime();
        }
        // even rank
        else{
            if(rank+1 < involveSize){
                MPI_Sendrecv(arr, dataSize, MPI_FLOAT, rank+1, 2,
                    res, nxtSize, MPI_FLOAT, rank+1, 2, new_comm, MPI_STATUS_IGNORE);
                nonSwapped = swapper(arr, res, dataSize, nxtSize, false);
            }
        }
    }
```

不確定效能實際上能好多少，至少可以優化後的方法可以讓每個process都呼叫Sendrecv

Experiment & Analysis

Performance matrix

Input file	02.txt	07.txt	17.txt	27.txt	30.txt	32.txt	35.txt	40.txt
array size	15	12345	400000	191121	64123483	64123513	536869888	536869888
process	15	3	20	24	24	24	12	12
item / process	1	4115	20000	7963	2671811	2671813	44739157	44739157
node	3	3	2	2	2	2	1	3
Read time	0.023111	0.004124	0.01386	0.015251	0.073212	0.072899	0.927732	0.528008
Computation tim	0.000006	0.000232	0.004883	0.002178	1.186155	0.135036	11.160894	4.72691
Communication	0.002476	0.000364	0.003737	0.003091	0.478944	0.116216	5.008912	2.628805
Write time	0.013512	0.001121	0.009412	0.009606	0.010668	0.012587	0.012167	0.006349



時間計算方式：

Read file time: rank0讀檔案的時間

Computation time: rank0做sort和merge的總時間

Communication time: rank0做send、recv和allreduce的總時間

Write file time: rank0寫檔案的時間

簡單取幾個testcase裡的測資來做實驗

實驗一：

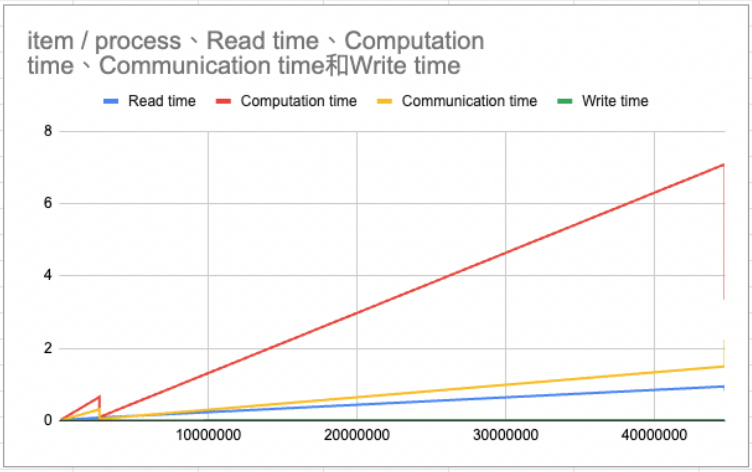
發現當每個process分到的item越多時，computation time會大幅提升，接著是communication time，然後是read file time。 Write file time沒有明顯提升。

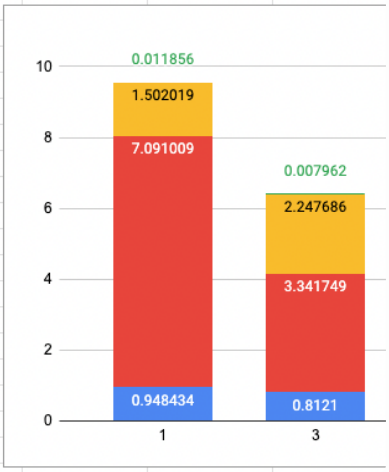
實驗二：

相同數量的item分給相同數量的process，但開的node數量不一樣會導致computation time大幅下降，應該是因為計算loading分散給其他node來做(所以以35.txt和40.txt來看，node變成三倍，計算時間也減少三倍)。但代價是communication time些微上升，大概是因為跨node的通訊overhead較大導致。

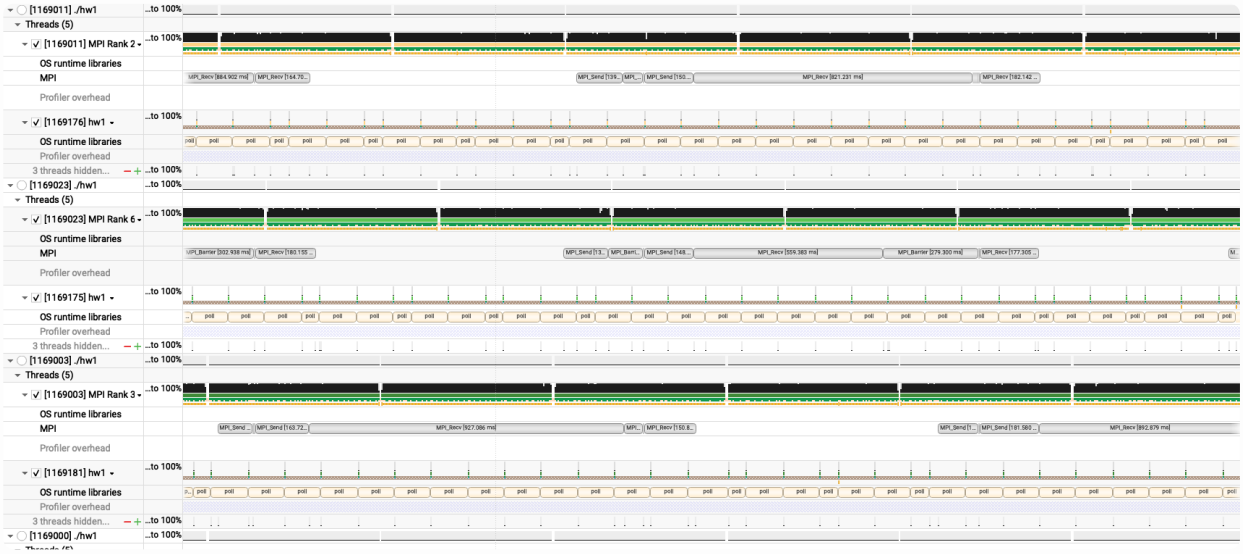
優化後

Input file	02.txt	07.txt	17.txt	27.txt	30.txt	32.txt	35.txt	40.txt
array size	15	12345	400000	191121	64123483	64123513	536869888	536869888
process	15	3	20	24	24	24	12	12
item / process	1	4115	20000	7963	2671811	2671813	44739157	44739157
node	3	3	2	2	2	2	1	3
Read time	0.016538	0.005896	0.015665	0.013488	0.08596	0.086432	0.948434	0.8121
Computation tim	0.000024	0.001169	0.003101	0.001326	0.654376	0.098135	7.091009	3.341749
Communication	0.001356	0.000248	0.003095	0.002231	0.313459	0.050539	1.502019	2.247686
Write time	0.011542	0.0002	0.009613	0.008331	0.011983	0.011009	0.011856	0.007962





Profile



使用nsys做profiling

整體看起來最大的效能瓶頸在等待計算的blocking receive，因為較大rank的process在把array傳出去之後就會陷入blocking receive狀態，也不做其他事情，導致有將近一半的計算資源閒置。因次若能有個方法能讓這些process不要陷入blocking receive的狀態，而是負責處理一部分的運算就會讓整體的運算使用率提升，並提高speedup factor。

優化後

因為寫memory的次數變少，所以等待時間有明顯變少。

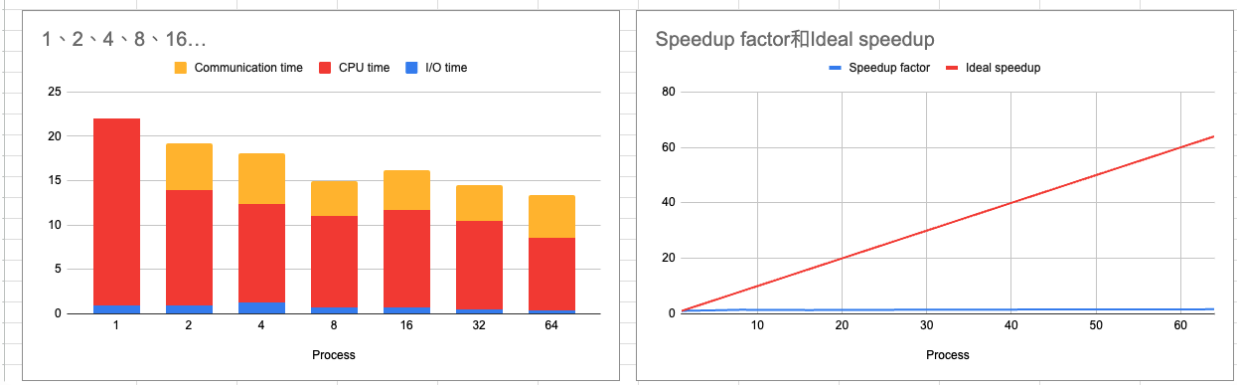
Events View

#	Name	Start	Duration	TID	Category
3478	MPI_Recv	4.64364s	1.266 s	1169003	
9494	MPI_Recv	11.4452s	927.086 ms	1169003	
7617	MPI_Recv	9.30809s	920.683 ms	1169003	
11437	MPI_Recv	13.5996s	892.879 ms	1169003	
15008	MPI_Recv	17.597s	807.056 ms	1169003	
509	MPI_File_read_at	0.727187s	805.448 ms	1169003	
13260	MPI_Recv	15.6332s	779.832 ms	1169003	
25	MPI_Init	-0.0618414s	774.164 ms	1169003	
5567	MPI_Recv	7.0474s	755.143 ms	1169003	
6297	MPI_Barrier	7.80258s	299.351 ms	1169003	
7218	MPI_Barrier	8.89375s	233.922 ms	1169003	
3291	MPI_Send	4.41783s	225.730 ms	1169003	
11260	MPI_Send	13.418s	181.580 ms	1169003	
7440	MPI_Send	9.12776s	180.281 ms	1169003	
5393	MPI_Send	6.86865s	178.693 ms	1169003	
13088	MPI_Send	15.4577s	175.463 ms	1169003	
4716	MPI_Recv	5.91215s	170.497 ms	1169003	
14847	MPI_Send	17.4327s	164.228 ms	1169003	
9334	MPI_Send	11.2814s	163.721 ms	1169003	
8531	MPI_Recv	10.2528s	155.743 ms	1169003	
5236	MPI_Send	6.71182s	154.205 ms	1169003	
10450	MPI_Recv	12.4305s	150.889 ms	1169003	
12319	MPI_Recv	14.5111s	149.794 ms	1169003	
14118	MPI_Recv	16.5201s	149.633 ms	1169003	
14700	MPI_Send	17.2818s	148.071 ms	1169003	
6588	MPI_Recv	8.10201s	144.538 ms	1169003	
12860	MPI_Send	15.214s	132.306 ms	1169003	
11142	MPI_Send	13.3013s	116.196 ms	1169003	
12980	MPI_Barrier	15.3464s	111.219 ms	1169003	
7110	MPI_Send	8.78375s	109.939 ms	1169003	
14010	MPI_Barrier	16.4131s	106.655 ms	1169003	

用event view簡單看了一下發現撇除等待時間不看，花最多時間的是Send、Recv和兩者之間的Barrier。這代表如果能減少溝通輪數，不只可以減少寫memory的次數，還能因為減少這些API call來小幅減少呼叫時間。

Speedup

Process	1	2	4	8	16	32	64
I/O time	0.985692	0.990257	1.284114	0.770806	0.733211	0.500257	0.38279
CPU time	21.057903	12.976098	11.066685	10.212945	11.006089	9.98396	8.186026
Communication time	0	5.256798	5.769207	4.007269	4.399393	4.058748	4.832951
Total time	22.043595	19.223153	18.120006	14.99102	16.138693	14.542965	13.401767
Speedup factor	1	1.146721092	1.216533538	1.470453311	1.365884771	1.515756587	1.644827507
Ideal speedup	1	2	4	8	16	32	64



Speedup factor實驗使用testcase的35.txt當作input(因為執行時間最久，但又不會太久)

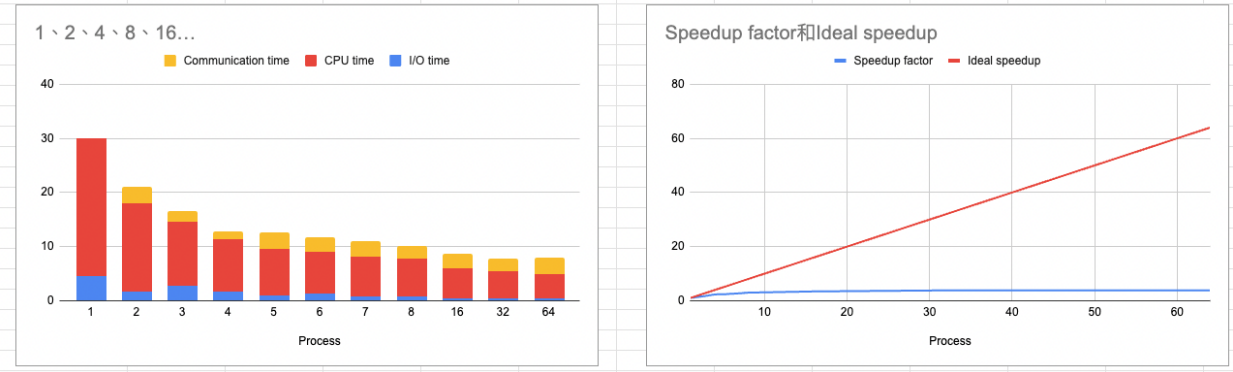
實驗條件從1個process到64個process

左圖計算不同process下的I/O time、Computation time、Communication time

右圖計算不同process下的speedup factor，並和ideal speedup factor比較

優化後

Process	1	2	3	4	5	6	7	8	16	32	64
Node	1	1	1	1	1	1	1	1	2	3	6
I/O time	4.565624	1.751629	2.805012	1.649451	1.030745	1.325191	0.812841	0.836043	0.517349	0.488115	0.41605
CPU time	25.532743	16.338816	11.836415	9.767091	8.516898	7.778968	7.420214	7.010771	5.438004	4.941334	4.566952
Communication time	0	3.031484	1.87914	1.431754	3.00817	2.630544	2.725844	2.212236	2.771544	2.425742	2.901933
Total time	30.098367	21.121929	16.520567	12.848296	12.555813	11.734703	10.958899	10.05905	8.726897	7.855191	7.884935
Speedup factor	1	1.424981923	1.821872518	2.342596014	2.397165918	2.564902324	2.746477269	2.992167948	3.448919702	3.831653107	3.817199127
Ideal speedup	1	2	3	4	5	6	7	8	16	32	64



可以明顯看出計算時間大幅度下降，通訊時間逐漸變為bottleneck

增加1~8個process的實驗用來檢查node數量相同的情況下，計算時間有逐漸下降，但通訊時間會忽高忽低

雖然speedup factor有比起優化前往上了一點，但是離ideal speedup factor還差非常多

Discussion

1. 當input逐漸scale up，computation cost會大幅上升。主要的bottleneck是對array的讀寫所造成的等待。因為大部分的merge過程都會需要對整個array做複寫，大量的複寫就會導致大量的等待時間。優化的可能考慮方式就是用vectorization，同時對多個item做操作。或是使用指標進行array element交換，避免array複寫。

2. 實驗結果指出scalability很差，因為計算輪數太多，而每一輪都會使用到memory複寫和process之間的溝通，因此computation time和communication time都降不下去。若能有個方法能讓這些process不要陷入blocking receive的狀態，而是負責處理一部分的運算就會讓整體的運算使用率提升，並提高speedup factor。我認為閒置資源越少，就越能逼近ideal speedup factor。優化後有些為提升運算資源的使用率、些微減少等待時間，但是這種實作方式會讓odd phase和even phase輪數固定。也就是必須要降低每一輪的memory讀寫時間才能夠盡量逼近ideal speedup factor。

Conclusion

隨著課堂的進行，我發現這隻程式其實有很多可以優化的地方。例如減少if else的數量，也就是減少branch的產生，不然不需要做該branch的process就必須要等待要做的process做完才能接著做下去。又或是減少複寫memory的次數也能大幅減少memory store產生的等待overhead。因此不論是使用指標交換array元素或是使用特殊register來減少複寫memory都能大幅加速程式的執行。
從這次的作業我發現即使知道效能瓶頸發生在哪裡，要如何有效去優化這些部分又是一個很大的學問。
實作中遇到的最大問題是一開始不知道是要用merge sort來實作，結果導致我寫了一個時間複雜度超高的交換演算法。所以花了很多時間重新理解題目並重新用merge sort來實作才免於超時。