HW1 report

張騰午 111065541

1. Implementation

How do you handle an arbitrary number of input items and processes?

根據輸入的 arraySize 還有 process 數量(size)，如果 size>arraySize，只要留下 arraySize 這麼多的 process，用 MPI_Comm_create 創造新的 USED_GROUP 其他的通通 MPI_Finalize()

```
if(arraySize<size){
    MPI_Comm_group(MPI_COMM_WORLD, &WORLD_GROUP);
    int range[1][3] = {{0, arraySize - 1, 1}};
    MPI_Group_range_incl(WORLD_GROUP, 1, range, &USED_GROUP);
    MPI_Comm_create(MPI_COMM_WORLD, USED_GROUP, &USED_COMM);
    if (USED_COMM == MPI_COMM_NULL){
        MPI_Finalize();
        return 0;
    }
    size = arraySize;
}
```

之後根據 rank 值用以下計算處理的數量 localSize

```
start_x = rank * ((double)arraySize / size);
int end_x = (rank + 1) * ((double)arraySize / size);
localSize = end_x - start_x;
```

How do you sort in your program?

先對每個 process 內進行 sorting，這邊使用的是 boost 函式庫裡的 spreadsort。

使用 MPI_Sendrecv();相鄰兩 processes 同時把資料交給對方，rank 小的留下較小的數值(getSmall)，rank 大的留下較大的數值(getBig)。

Other efforts you've made in your program.

另外在進行前可以先比對 rank 小的末端是否小於 rank 大的前端數值，如果成立的話表示已經排序完成，不必再進行交換；反之才進行 getSmall, getBig 計算

程式各部分:

```c
int main(int argc, char **argv)
{
    MPI_Group WORLD_GROUP, USED_GROUP;
    MPI_Comm USED_COMM = MPI_COMM_WORLD;

    if (argc != 4) {
        fprintf(stderr, "must provide exactly 3 arguments!\n");
        return 1;
    }
    MPI_Init(&argc, &argv);


    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int arraySize = atoll(argv[1]);
    char *input_filename = argv[2];
    char *output_filename = argv[3];

    MPI_File input_file, output_file;
    float *data; // Use an array to store the data
    float *data0;
    float *data_next;
    float *data_prev;
    //int maxsize = size;
    int localSize;
    int localSize_next;
    int localSize_prev;
    int start_x;
```

判斷 process 數量去掉過多的 process

```c
if(arraySize<size){
    MPI_Comm_group(MPI_COMM_WORLD, &WORLD_GROUP);
    int range[1][3] = {{0, arraySize - 1, 1}};
    MPI_Group_range_incl(WORLD_GROUP, 1, range, &USED_GROUP);
    MPI_Comm_create(MPI_COMM_WORLD, USED_GROUP, &USED_COMM);
    if (USED_COMM == MPI_COMM_NULL){
        MPI_Finalize();
        return 0;
    }
    size = arraySize;
}
```

計算 localSize 與 prev, next 的 localSize，並建立需要的空間

```
start_x = rank * ((double)arraySize / size);
int end_x = (rank + 1) * ((double)arraySize / size);
localSize = end_x - start_x;
data = new float[localSize]; // Allocate memory for the local data
data0 = new float[localSize];

//Calculate the next/prev data size for each process
if (rank == size - 1) {
    localSize_next = localSize;
} else {
    int end_next = (rank + 2) * ((double)arraySize / size);
    localSize_next = end_next - end_x;
}

if (rank == 0) {
    localSize_prev = localSize;
} else {
    int start_prev = (rank - 1) * ((double)arraySize / size);
    localSize_prev = start_x - start_prev;
}
data_next = new float[localSize_next];
data_prev = new float[localSize_prev];
```

開啟檔案讀取自己範圍的資料，並使用 spreadsort 先排序(spreadsort 測試是最快的排序方法)

```
MPI_File_open(USED_COMM, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &input_file);
MPI_Offset offset;
// Calculate the offset for each process
if(rank < arraySize){
    offset = sizeof(float) * start_x;
    MPI_File_read_at(input_file, offset, data, localSize, MPI_FLOAT, MPI_STATUS_IGNORE);
    boost::sort::spreadsort::spreadsort(data, data+localSize);
    //std::sort(data, data+localSize);
}


MPI_File_close(&input_file);
```

進入 while 迴圈，在 odd 與 even 兩個 phase 不斷和自己旁邊的 process 交換資料，依照數值大小留下屬於自己的，有發生拿走對方數據的行為則 sorted=1，最後使用 MPI_Allreduce()查看 sorted 加總是否=0，如果 sorted 都是 0 代表沒有任何交換產生，排序完成，則 while 結束。

```
int sorted = 0;
int allsorted = 1;

while(allsorted > 0){
    sorted = 0;
    //odd
    if(rank % 2 == 0 && rank == size-1) {memcpy(data0, data, localSize * sizeof(float));}
    if(rank % 2 == 0 && rank != size-1){
        MPI_Sendrecv(data, localSize, MPI_FLOAT, rank+1, 0, data_next, localSize_next, MPI_FLOAT, rank+1, 1, USED_COMM, MPI_STATUS_IGNORE);
        if(data_next[0] > data[localSize-1]){
            memcpy(data0, data, localSize * sizeof(float));
        }else{
            getSmall( data_next,data0,data, localSize,localSize_next,sorted);
        }

    }else if(rank % 2 == 1 && rank != 0){
        MPI_Sendrecv(data, localSize, MPI_FLOAT, rank-1, 1, data_prev, localSize_prev, MPI_FLOAT, rank-1, 0, USED_COMM, MPI_STATUS_IGNORE);
        if(data_prev[localSize_prev-1] < data[0]){
            memcpy(data0, data, localSize * sizeof(float));
        }else{
            getBig(data_prev, data0, data, localSize,localSize_prev, sorted);
        }
    }
```

```
    //even
    if(rank == 0) {memcpy(data, data0, localSize * sizeof(float));}
    if(rank % 2 == 1 && rank == size-1) {memcpy(data, data0, localSize * sizeof(float));}
    if(rank % 2 == 1 && rank != size-1){
        MPI_Sendrecv(data0, localSize, MPI_FLOAT, rank+1, 0, data_next, localSize_next,MPI_FLOAT, rank+1, 1, USED_COMM, MPI_STATUS_IGNORE);
        if(data_next[0] > data0[localSize-1]){
            memcpy(data, data0, localSize * sizeof(float));
        }else{
            getSmall(data_next, data, data0, localSize,localSize_next,sorted);
        }


    }else if(rank % 2 == 0 && rank != 0){
        MPI_Sendrecv(data0, localSize, MPI_FLOAT, rank-1, 1, data_prev, localSize_prev, MPI_FLOAT, rank-1, 0, USED_COMM, MPI_STATUS_IGNORE);
        if(data_prev[localSize_prev-1] < data0[0]){
            memcpy(data, data0, localSize * sizeof(float));
        }else{
            getBig(data_prev, data, data0, localSize,localSize_prev, sorted);
        }


    }
    //for(int i=0;i<localSize;i++){printf("rank %d: %f\n",rank,data[i]);}
    MPI_Allreduce(&sorted, &allsorted, 1, MPI_INT, MPI_SUM, USED_COMM);

}
```

使用 MPI_sendrecv 接收並傳出自己的 data，之後使用 getBig 或 getSmall 留下數量為 localSize 的自己需要的前幾大/小數值即可，如此可以只進行一次交換而不用 MPI_send,MPI_recv 各 call 一次。另外在接收並留下自己需要的資料時，我會使用兩組 data array。data,data0

Odd phase 時，傳出 data 資料，進行 getSmall/getBig 後存到 data0，even phase 時則將 data0 資料傳出，getSmall/getBig 後存到 data 中，若每次都把資料放在 data 中，就需要額外將資料搬運到 data0，如此可節省時間。

```
    MPI_File_open(USED_COMM, output_filename, MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &output_file);

    MPI_File_write_at(output_file, offset, data, localSize, MPI_FLOAT, MPI_STATUS_IGNORE);


    MPI_File_close(&output_file);

    delete[] data, data0, data_next, data_prev; // Free the memory

    MPI_Finalize();
    return 0;
}
```

交換數值 function:getSmall 與 getBig

```
void getBig(float data_prev[],float data1[],float data2[] ,int localSize,int localSize_prev,int &sorted){

    int i = localSize-1, j = localSize_prev-1,k=localSize-1;
    while (i>=0&&j>=0&&k >= 0) {
        if (data2[i] >= data_prev[j]) {
            data1[k--] = data2[i--];
        } else {
            data1[k--] = data_prev[j--];
            sorted=1;
        }
    }
    while (i >=0 && k >=0) {
        data1[k--] = data2[i--];
    }
    while (j >=0 && k >=0) {
        data1[k--] = data_prev[j--];
        sorted=1;
    }
}
```

```
void getSmall(float data_next[],float data1[],float data2[], int localSize,int localSize_next, int &sorted){

    int i = 0, j = 0,k=0;

    while (i < localSize && j < localSize_next && k < localSize) {
        if (data2[i] <= data_next[j]) {
            data1[k++] = data2[i++];
        } else {
            data1[k++] = data_next[j++];
            sorted=1;
        }
    }
    while (i < localSize && k < localSize) {
        data1[k++] = data2[i++];
    }
    while (j < localSize_next && k < localSize) {
        data1[k++] = data_next[j++];
        sorted=1;
    }
}
```

2. Experiment & Analysis

i、Methodology

System Spec

使用 apollo.cs.nthu.edu.tw 設備來進行我的實驗。

Performance Metrics

設置標記用 MPI_time 測量時間並加入 computing time, IO 或是 communication 的部分

IO time

在 MPI_File_read_at ()與 MPI_File_write_at()的前後加上 MPI_Wtime()取得其差值後加總,即可得到該 process 讀寫檔案的 I0 time。如下圖範例:

```
Ttemp = MPI_Wtime();
MPI_File_open(USED_COMM, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &input_file);
MPI_Offset offset;
// Calculate the offset for each process
if(rank < arraySize){
    offset = sizeof(float) * start_x;
    MPI_File_read_at(input_file, offset, data, localSize, MPI_FLOAT, MPI_STATUS_IGNORE);

    //std::sort(data, data+localSize);
}


MPI_File_close(&input_file);
TIO += MPI_Wtime() - Ttemp;
```

computing time

在 MPI_Init()後與 MPI_Finalize()前加上 MPI_Wtime()，再計算其差值即可得到整個程式的執行時間，之後減掉 IO 還有 communication time 就可得到 computing time。

communication time

在 communication 相關函式，如: MPI_Sendrecv(...)的前後加上 MPI_Wtime()，在取得其差值後再加總，即可得到該 process 用在 communication 上所花費的時間。再除以 process 數，即可得到該程式的平均 communication time。

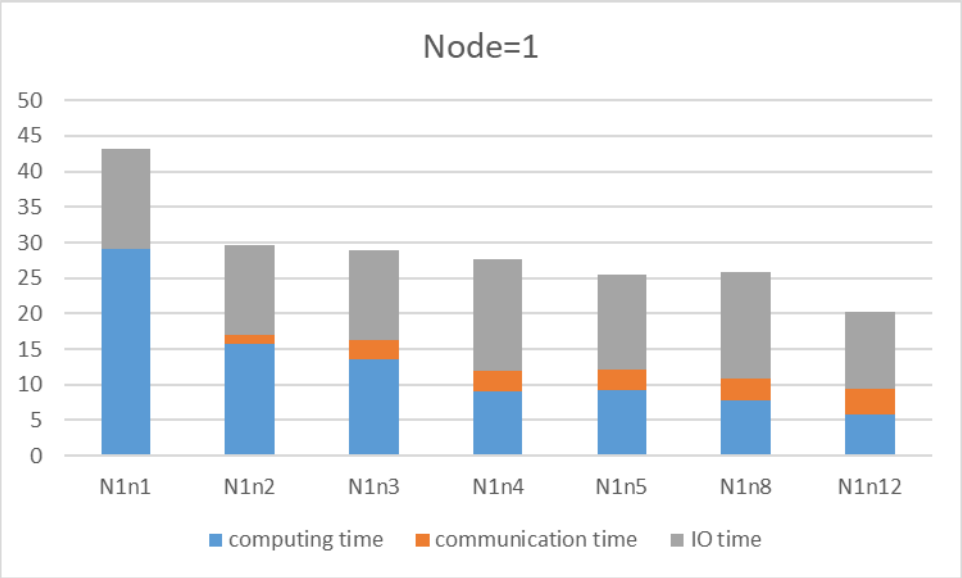
others

使用 IPM profile 在進行 hw1-judge 時產生 profile 觀察個指令時間。

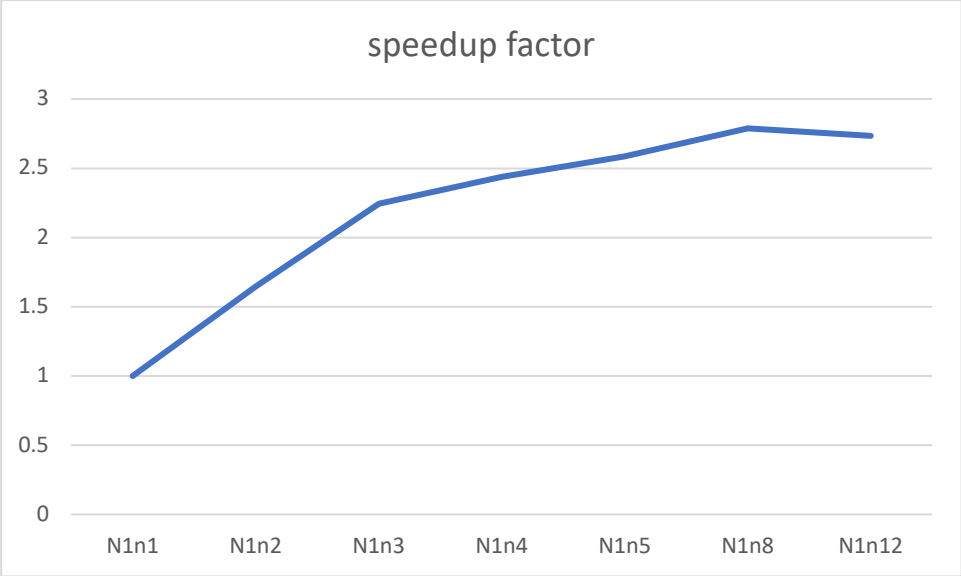
ii、Plots: Speedup Factor & Profile

使用 testcase 35 進行測試，總共資料數量為 536869888。資料數量大小剛剛好，不會大到跑太久，但也不會小到看不出來 Parallel Version 的差異。

固定 Nodes 數量等於 1 時不同 process 數量

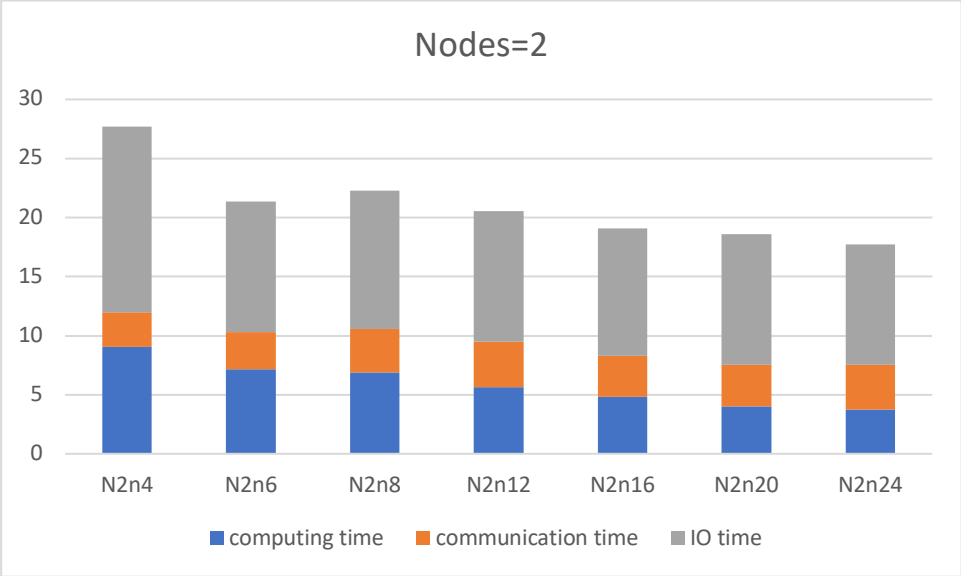


以上可以發現 computing time 在只有一個 process 時佔比很大，而隨著 process 數量增多 communication 時間明顯增加，而隨著 process 數量增加，下降最明顯的是 computing 時間。

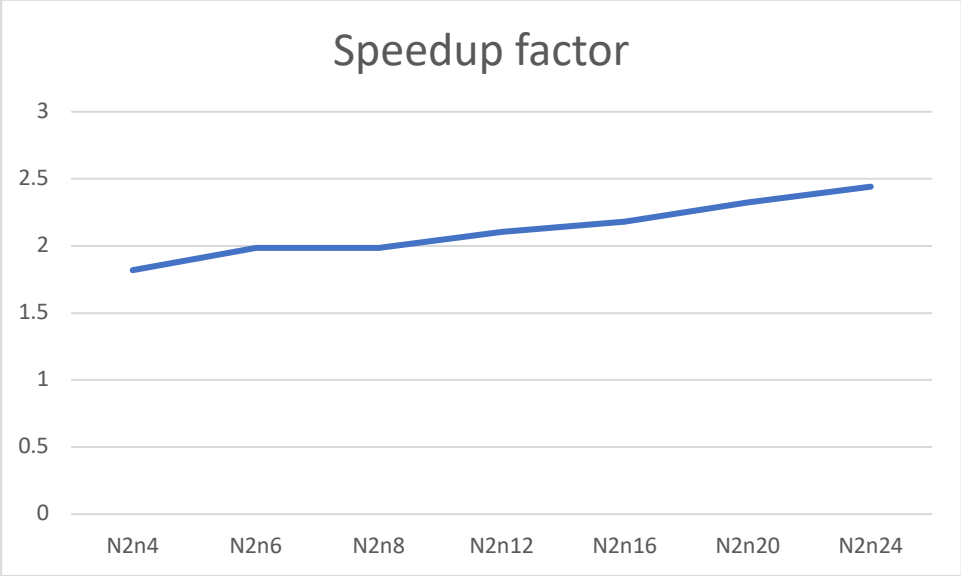


Speedup 方面可以發現除了 n=2 時的確有到大約兩倍的速度，但之後上升幅度都不高，process 數量超過 8 後甚至有點下降趨勢，可能和過多 process 要溝通導致 process 增加卻無法線性增加效能且單一 node 資源也有限，持續增加 process 數量很快會到達極限。

固定 Nodes 數量等於 2 時不同 process 數量

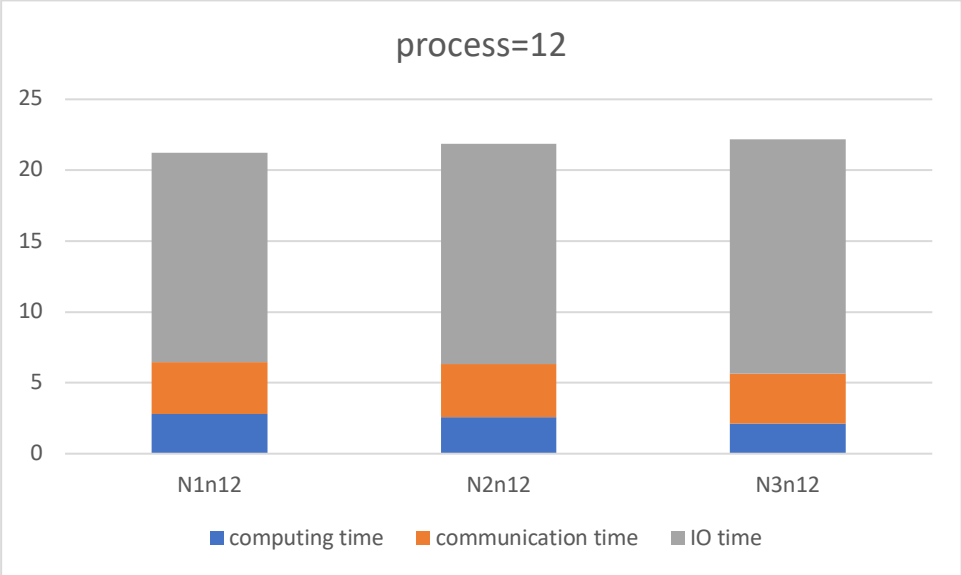


Communication time 隨 n 增加稍微增加，最主要減少的是 computing time，因為 process 分工增加降低了單個 process computing time。此外在有多個 nodes 時還增加了 network interference，



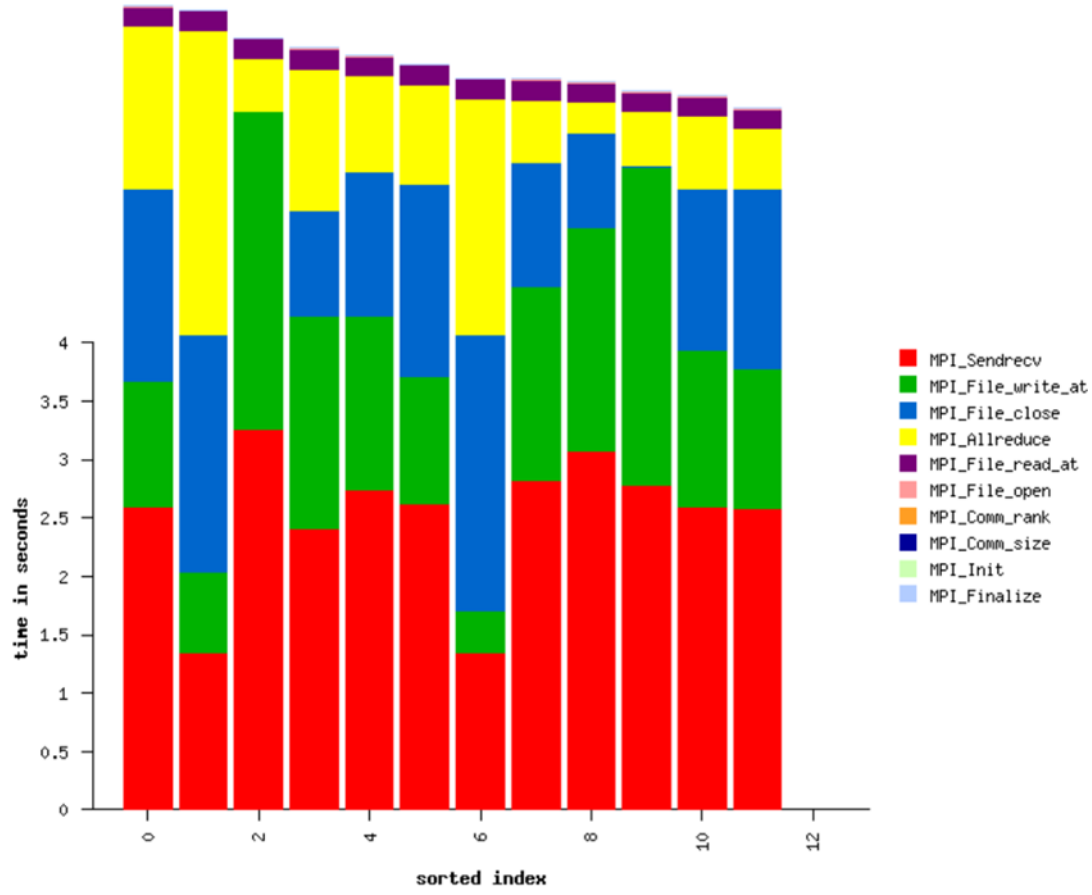
和單個 node 時相比上升趨勢更低，Scalability 效果沒有很突出，但可以持續保持上升趨勢，比單一 node 時有更多的資源讓 process 數量能持續增長。

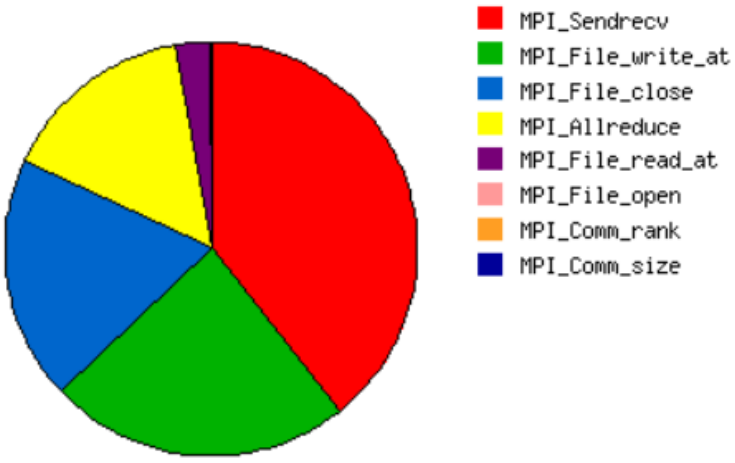
固定 Process 數量等於 12 時不同 nodes 數量



測試時 nodes 數量最多只能到 3，分別測試三種 nodes 數量時狀況，不過看起來並不顯著，但 computing time 有些微減少，IO time 則有增加，可能是因為更多 CPU 可供使用(computing time 減少)，但同時也更多 CPU 要進行讀取動作(IO time 增加)導致。

使用 IPM 產生 hw1-judge -i 35.txt 的數據





IPM 可以更精確地分析每個 MPI 指令的時間，其中可以看到在 hw1-judge 時 MPI_Sendrecv, MPI_File_close, MPI_File_write_at 都花費了非常多比例的時間。其中 MPI_Sendrecv 因為是 blocking communication，在發送或接收消息時會阻塞程序執行，直到消息傳輸完成的通信方式。MPI_Isend 和 MPI_Irecv 允許您啟動發送和接收操作，然後繼續執行其他計算任務。然後，您可以使用 MPI_Wait 或 MPI_Test 等函數來檢查操作的完成狀態。這種方式允許更好地利用計算資源，特別是在多核處理器或集群環境中。

iii、Other

std:sort()改成使用 boost::sort::spreadsort::spreadsort();
std::sort() 通常使用的是快速排序（QuickSort）演算法，
boost::sort::spreadsort::spreadsort();使用分布式排序（Spreadsort）演算法。
Spreadsort 在大量數據下有更好的效率。因此通過改使用 spreadsort 方法，讓我 hw1-judge 時間從 150 秒左右降到 130，效果顯著。

3. Experiences / Conclusion

雖然程式架構似乎想起來並不會太難，但真正動手做後面對不少沒想到的問題以及如何優化，仍令人傷腦筋。一開始優化很容易，例如：MPI_send 和 MPI_recv 改及使用 MPI_Sendrecv，或是從 std:sort()改成使用 boost::sort::spreadsort::spreadsort();部分，都能有顯著的提升，在修改程式後，可以輕易地降低 runtime，可是當 runtime 變小的時候，就不容易優化程式，很多時候修改一些覺得能有所提升的地方，反而讓 runtime 增加，或是當時 cluster , Network Interference 狀態不好，也會影響每次測試 performance，中間還經歷了停電，讓過程更加坎坷，實在花了很多心力在優化上面。但結果出來時也能對這們課程有更深了解，增加 process 數量後對程式的優化不會完全反映出來，有太多其他因素會影響了，通過這次也讓我對程式設計有更深的了解。