Bitonic Merge Sort

- Name: 盛爾葳
- email: ewinnie.sheng@gmail.com

Implementation

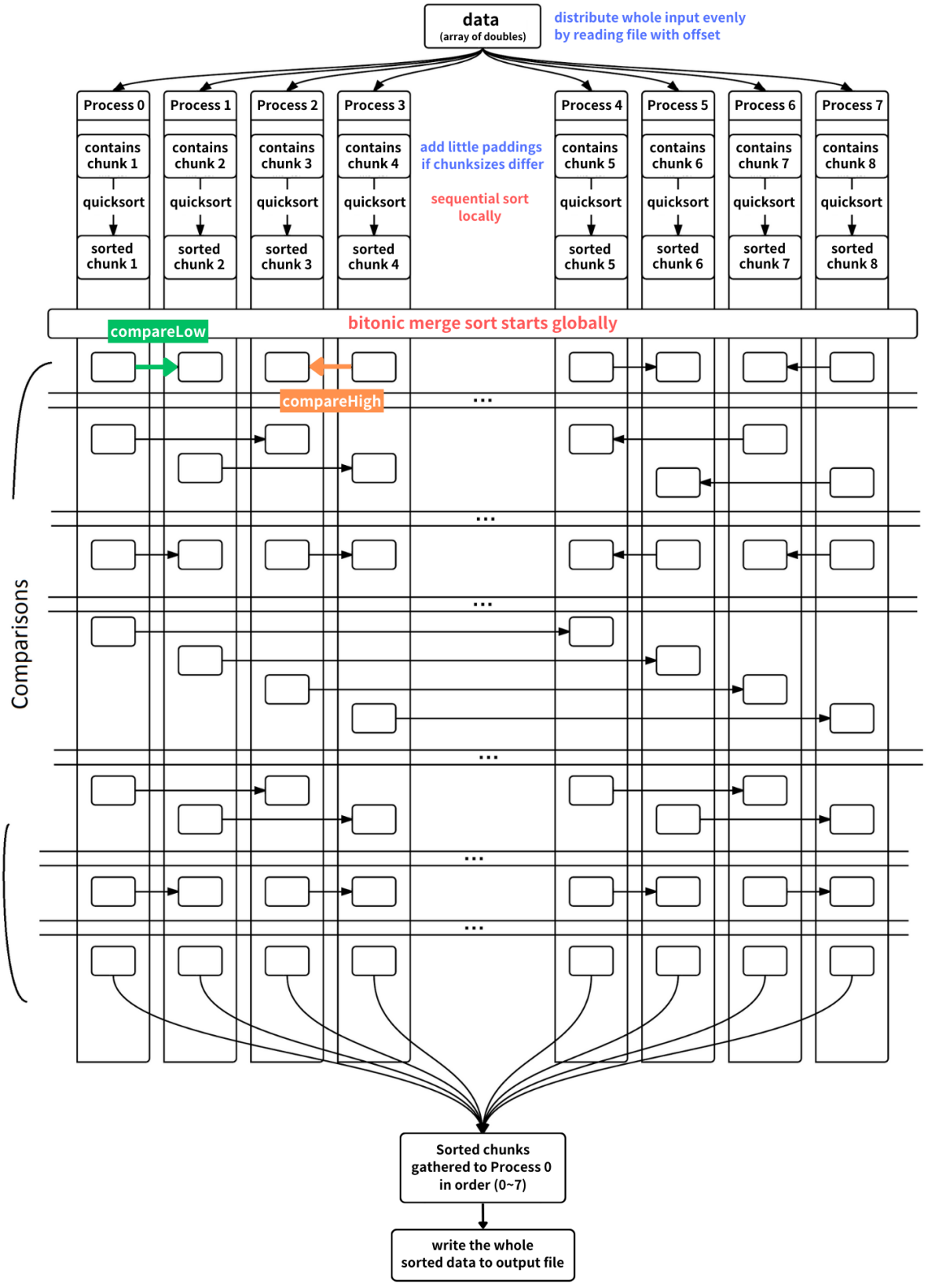
I/O

1. 若數據量 `arr_size` 不是2的次方倍，會用 `DBL_MAX` 將其大小padding至下一個最小的2的次方倍，即符合bitonic sort的定義，同時確保所有processes的分配到的elements數量(chunksize)都相等。
2. 接著是IO平行化，計算出每個process的間隔(display)就可以用 `MPI_File_read_at` 讀取各自負責的區段。

   整個sorting流程(local+bitonic)結束後，所有chunks在rank 0 ~ rank np應該已經完成排序，可以用 `MPI_Gather` 依序集合回rank 0。接著用自己寫的 `writeFile` 函式過濾掉一開始的padding elements(DBL_MAX)，並以 `MPI_File_write_at` 將 `filtered_chunk` 間隔 `filtered_size` 寫成output檔案。

Sorting logic

1. 資料分配到processes後，每個process中的chunks會先各自完成local的sequential sort(`bitonicLocal`)，一樣是用bitonic的原理進行，process之間的bitonic sort(`bitonicGlobal`)則是在merge的時候完成。
2. `bitonicGlobal` 的進行方式如下圖：



- **compareLow / compareHigh的判斷**
  - `int partner = rank ^ (1 << j)`
  - `bool ascending = ((rank >> (i + 1)) % 2 == 0)`

```
//compareLow和compareHigh的判斷
if (((rank >> j) & 1) == 0){
    if (ascending) compareLow(partner, data, chunksize,
rank);
    else compareHigh(partner, data, chunksize, rank);
}
else{
    if (ascending) compareHigh(partner, data, chunksize,
rank);
    else compareLow(partner, data, chunksize, rank);
}
```

- **與partner互換資料內容**

```
float* receivedData = (float*)malloc(chunksize *
sizeof(float));
    MPI_Sendrecv(data, chunksize, MPI_FLOAT, partner, 0,
                 receivedData, chunksize, MPI_FLOAT,
partner, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

- merge的方向不同：compareLow從小開始，保留小的那一段；compareHigh則是從大的開始，保留大的那一段。
    - compareLow

```
float* mergedData = (float*)malloc((chunksize*2) *
sizeof(float));
    int i = 0, j = 0, k = 0;
    while (i < chunksize && j < chunksize) {
        if (data[i] < receivedData[j]) {
            mergedData[k++] = data[i++];
        } else {
            mergedData[k++] = receivedData[j++];
        }
    }
    // Copy remaining elements
    while (i < chunksize) mergedData[k++] =
data[i++];
    while (j < chunksize) mergedData[k++] =
receivedData[j++];
```

- compareHigh (reverse merge)

```
float* mergedData = (float*)malloc((chunksize*2) *
sizeof(float));
    int i = chunksize - 1, j = chunksize - 1, k =
chunksize*2 - 1;
    while (i ⩾ 0 && j ⩾ 0) {
        if (data[i] > receivedData[j]) {
            mergedData[k--] = data[i--];
        } else {
            mergedData[k--] = receivedData[j--];
        }
    }
    while (i ⩾ 0) mergedData[k--] = data[i--];
    while (j ⩾ 0) mergedData[k--] = receivedData[j-
-];
```

Experiment & Analysis

System, Environment Spec

- partition: judge (才可以用到4個node)
- compile: `mpicxx -O3 -lm hw1.cc -o hw1`
- `jobscript.sh`

```
#!/bin/bash
#SBATCH -p test
#SBATCH -N 4
#SBATCH -n 32
#SBATCH -o output.log  #用來看詳細時間
#SBATCH -e error.log

module purge
module load openmpi/4.1.5

mpirun -np 32 ./hw1 536869888 test/36.in out
```

Performance Metrics

先用 `MPI_Wtime()` 計算前後時間差異，再以 `MPI_Reduce` 得到所有processes的時間總和：

- total time

  在 `MPI_Init` 後紀錄start_time， `MPI_Finalize();` 前紀錄end_time，後者減掉前者即是總時長。

- communication time

  包含 `MPI_Gather` 、 `MPI_Bcast` ，還有整個bitonic sort階段，因為 `compareLow` 、 `compareHigh` 有用 `MPI_Sendrecv` 在processes之間交換資料。

- I/O time

  `MPI_File_open` 、 `MPI_File_read_at` 、 `MPI_File_close` 、 `MPI_File_write_at` 前後紀錄時間並加總。

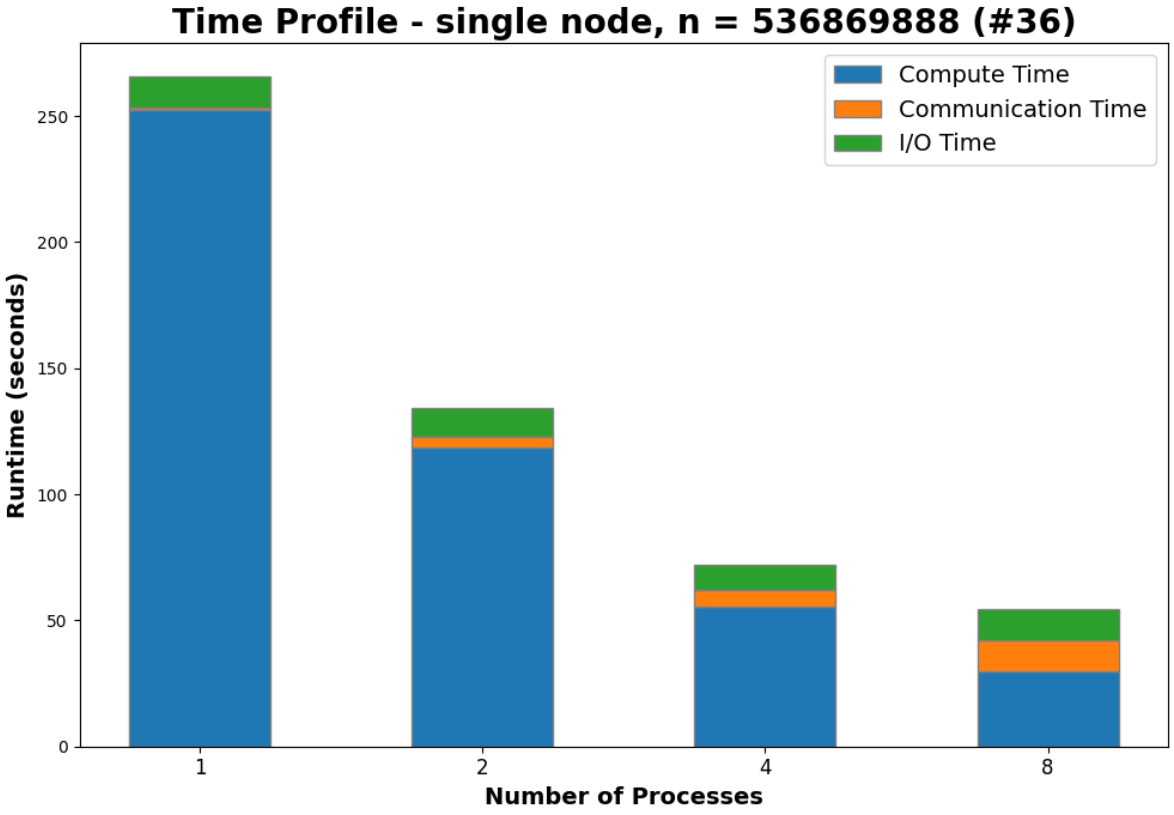
- computing time

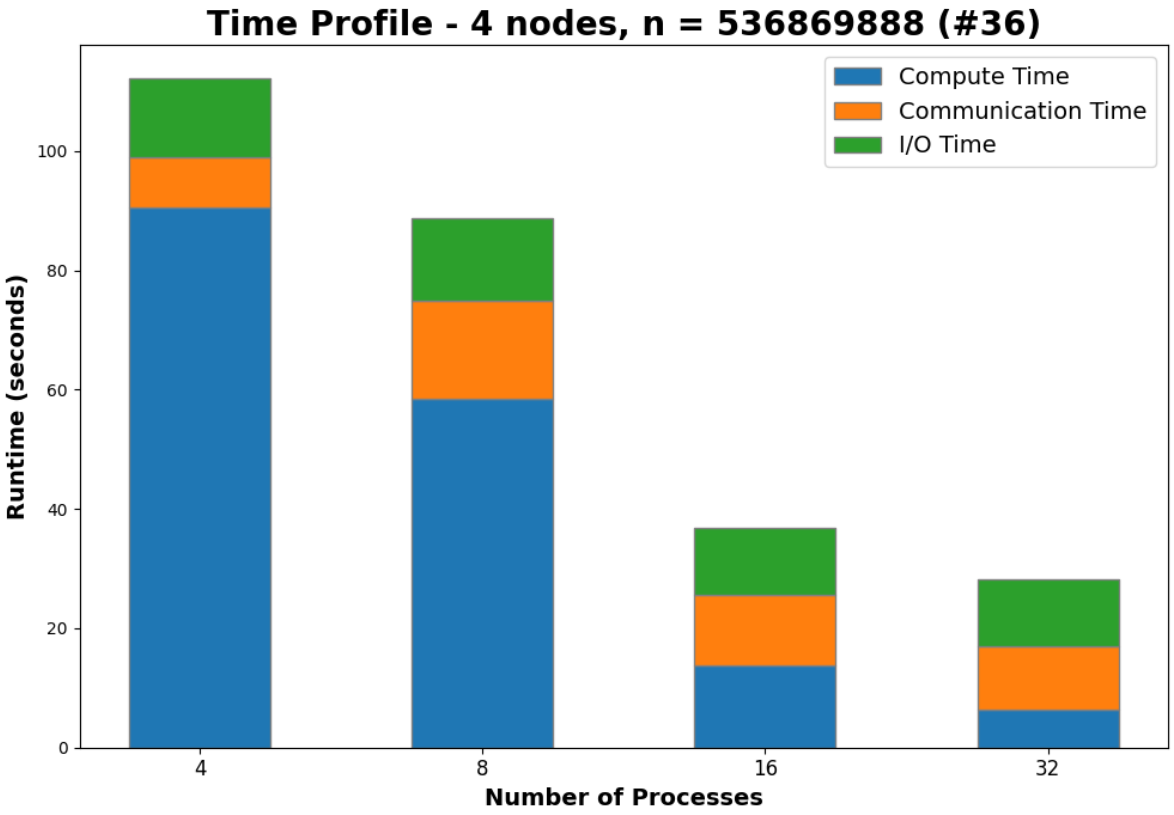
  total time扣除I/O time、communication time。主要來源是local進行的 `quicksort` 、如果需要padding也會增加時間。

Speedup Factor & Profile

> 選擇 Testcase 36: n = 536869888 (最大，且確認跑IPM沒問題)

Time profile



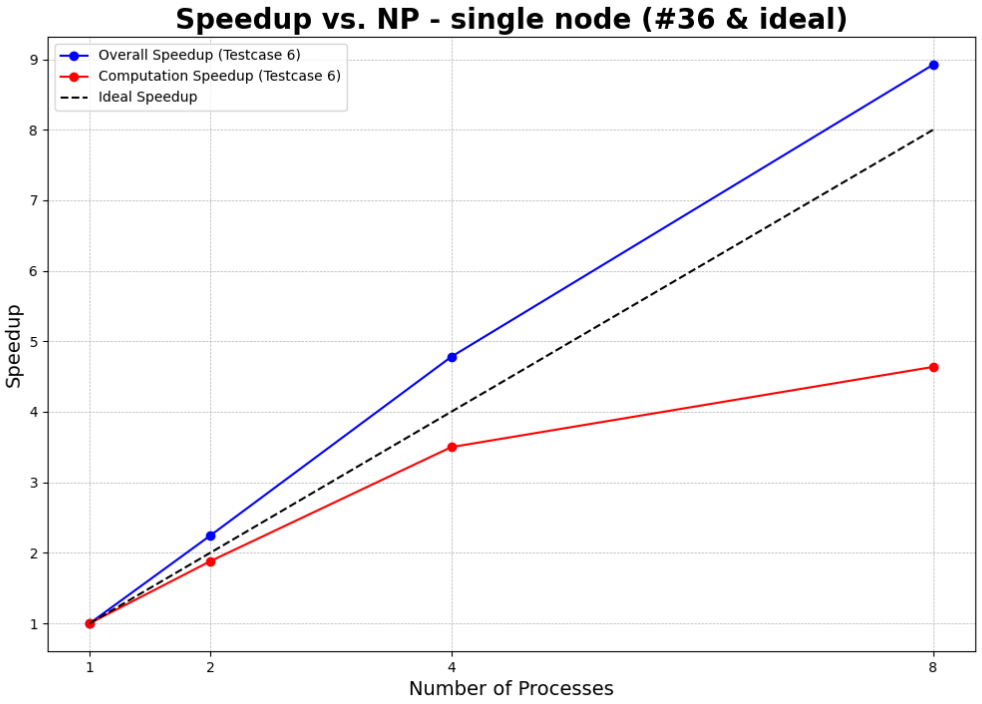


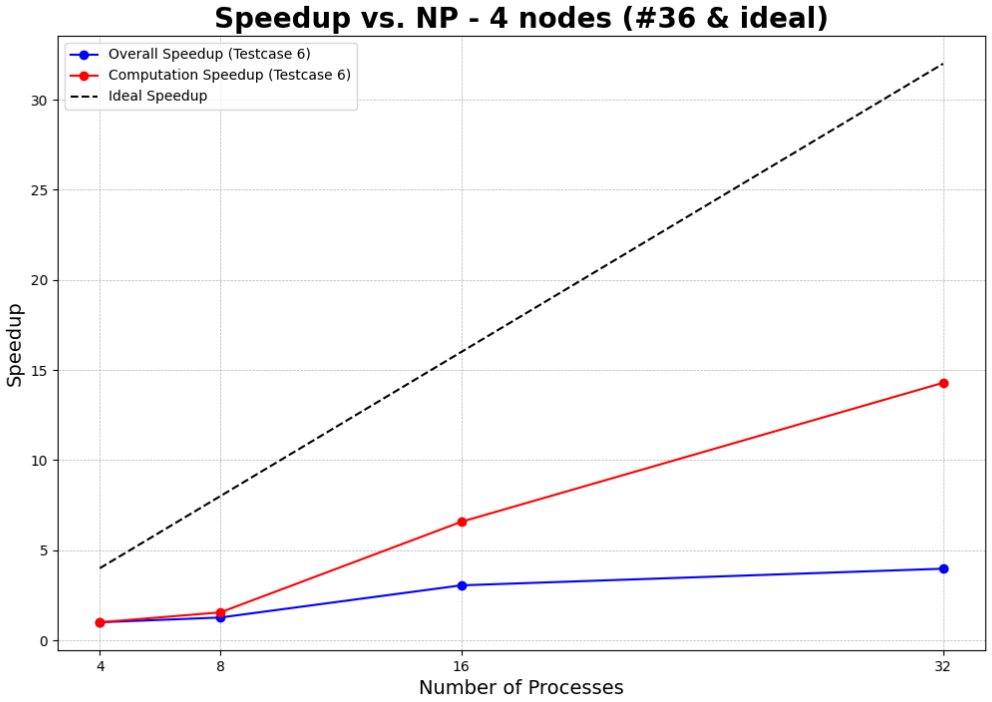
- computation time會隨著process數量增加下降，因為各自要負責local sort的數量減少，合併時也只要負責更少資料。
- I/O time都很接近，因為I/O的bandwidth有限，沒辦法提升很多。

- communication time的變化也不大（single node有稍微增加的趨勢）因為bitonic sort (global) 時process間經常需要交換資料，而數量越多越容易受網路延遲 (communication overhead)影響，尤其多nodes的時候若有跨機器運算的情形就會耗時更長且更不穩。由此可見，communication time是平行化的最大bottleneck。

Speedup

> **測試方式：**紀錄不同process數量所需的時間，與理論值做比較。





- single node的overall speedup雖然不及ideal，但是隨著process數量增加還是有上升趨勢；神奇的是computation speedup甚至有super-linear speedup的現象，推測是因資料被切分的更細後，memory與CPU cache effects的作用 (數據更可能常駐於cache，讀取快許多)更明顯，導致運算變得更快。多節點就沒有這個現象，因為每個節點都有自己的獨立記憶體。
- 另外，可以發現multiple nodes的speedup成長較single node緩慢，用到32個processes才和single node的8個processes差不多，這是因為前面提到的communication overhead，也就是節點間資料傳輸(如使用 `MPI_Sendrecv`)造成的網路延遲和bandwidth限制；不過其computation speedup表現相當不錯，32個processes時可以提升快15倍。

IPM Profile

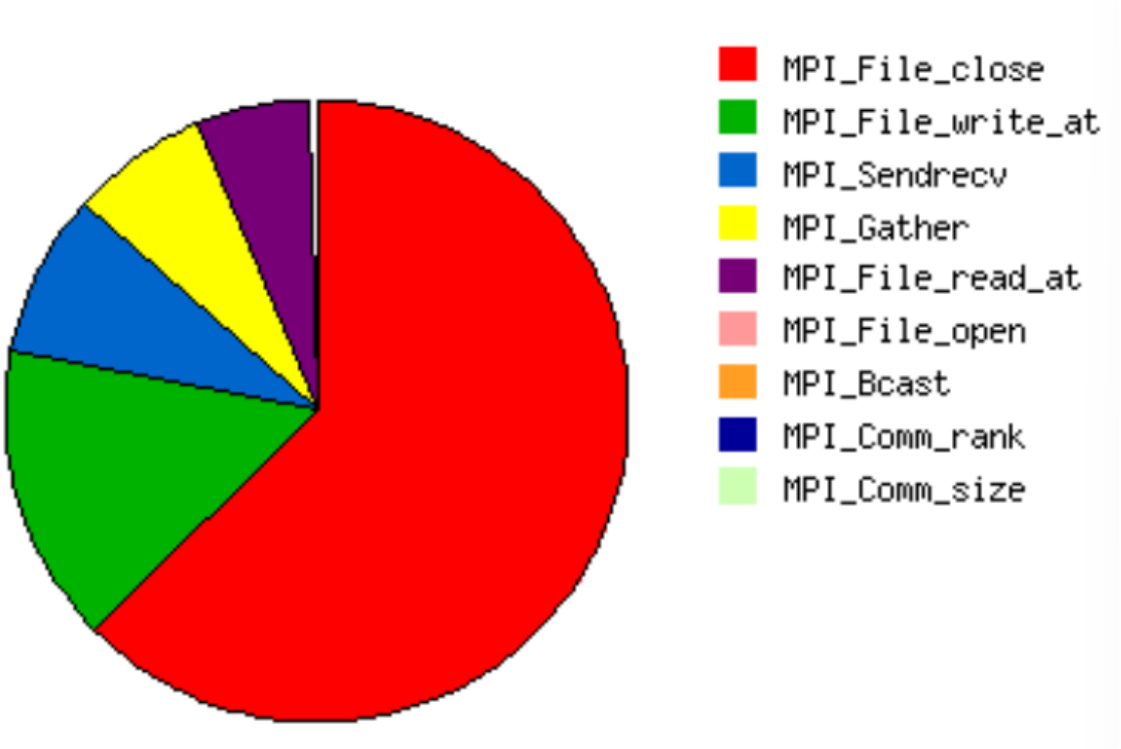
- compile: `mpicxx hw1.cc -o hw1 -L/opt/ipm/lib -lipm -lm`
- `jobscript.sh`

```
#!/bin/bash
#SBATCH -p test
#SBATCH -N 4
#SBATCH -n 32
#SBATCH -o output.log
#SBATCH -e error.log
```

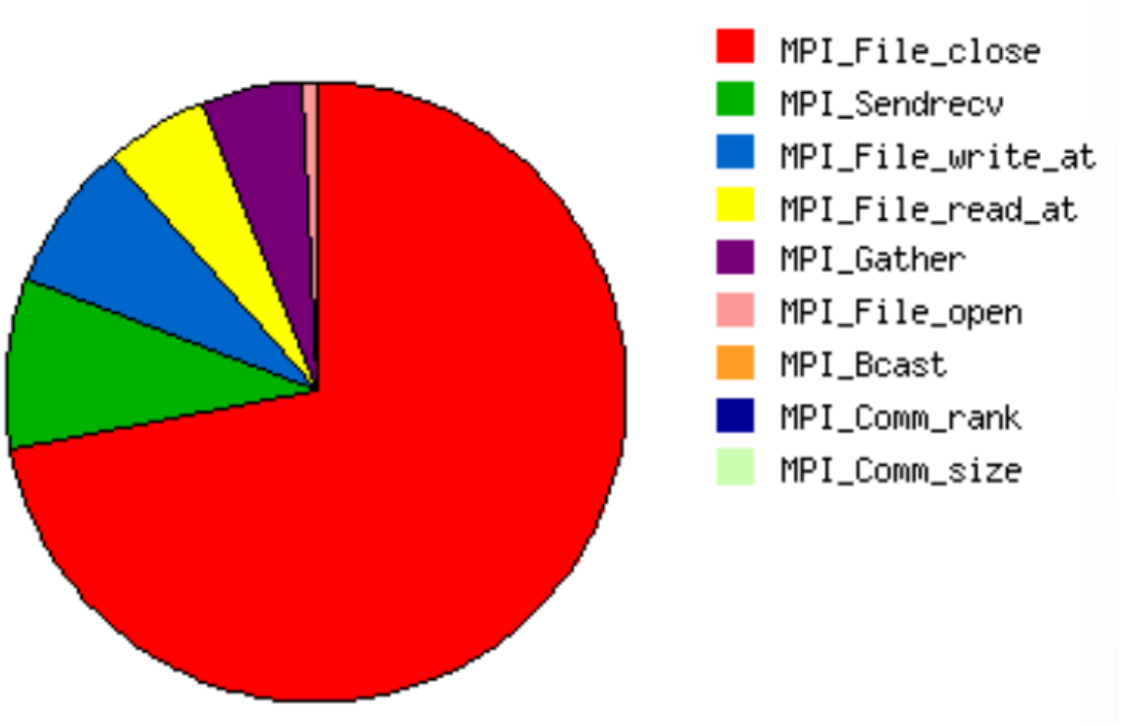
```
module purge
module load mpi ipm/mpi #作業說明提供的不行？

unset I_MPI_PMI_LIBRARY
export I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=0

mpirun -np 32 ./hw1 536869888 test/36.in out
```

Single node

- 4 proceses

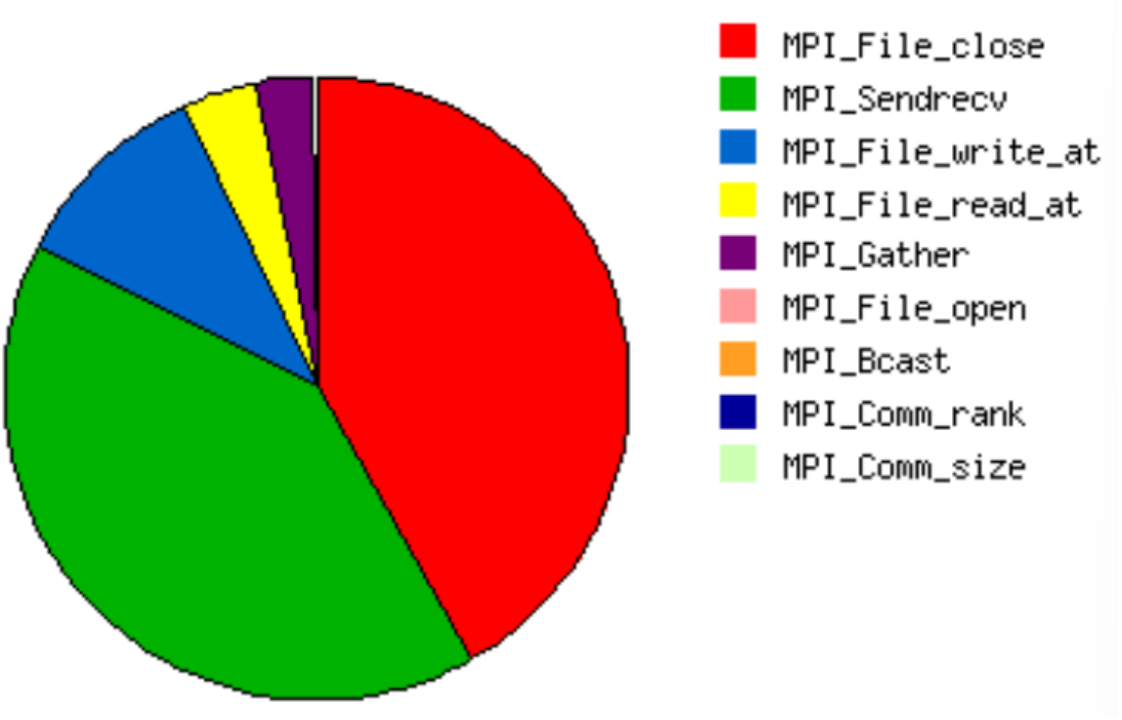


- 8 proceses

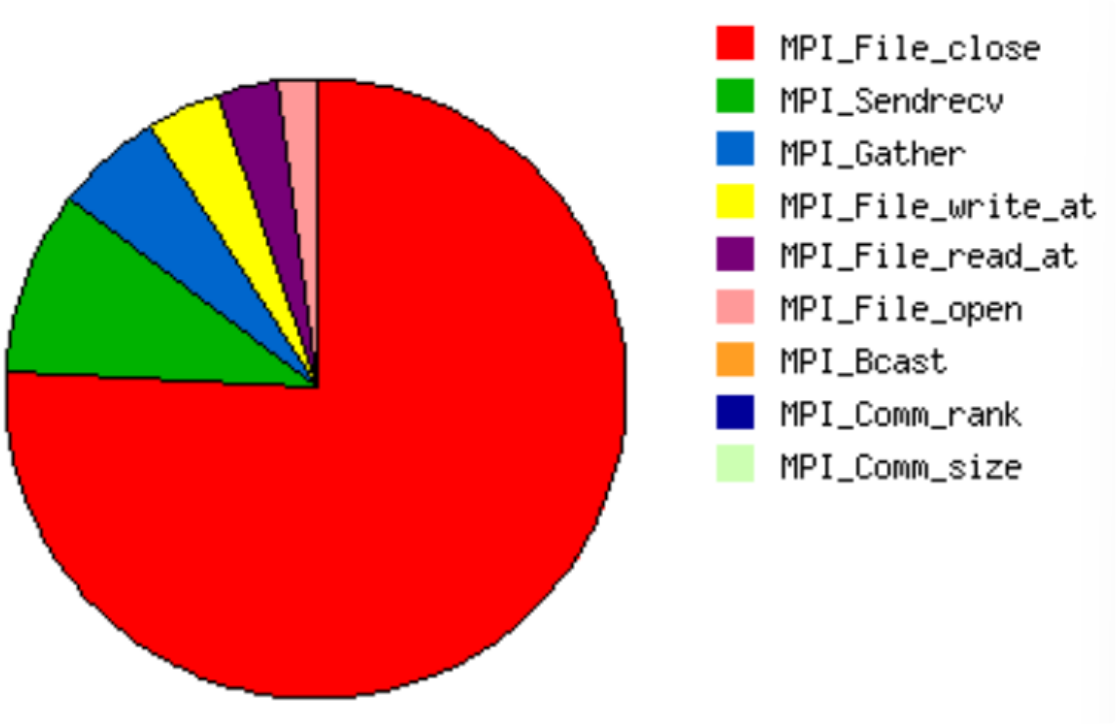


沒想到MPI_File_close是最花時間的。可能是在單一節點上使用更多processes導致I/O競爭，系統需要處理更多的同步和資源分配問題，進而增加完成操作的時間。

- **4 proceses**



- **16 proceses**



`MPI_Sendrecv` 涉及資料在processes之間的交換。當發生在多個節點間時，網路延遲和bandwidth限制會顯著影響性能。

而4個節點各運行1個process時，所有 `MPI_Sendrecv` 操作都是跨節點進行，沒有任何本地（同節點內）的數據交換，使得每次傳輸都會經歷最大的 communication overhead，因此 `MPI_Sendrecv` 會消耗更多時間。

Experiences & Conclusion

這次作業與寒假主要的差異就是padding到2的次方倍，且loocal sort也改成用 bitonic sort，可以發現效能是明顯提升的。但是paper提到的smart layout 看起來真的要整個大改code，很可惜是實在沒時間做到那部分。

我覺得比較有趣是用IPM分析的部分，我因為compile時link錯module搞了挺久，而且有些測資跑不出來全部時間(好像是IPM1本身的問題)，總之也是學到不少經驗！