Parallel Programming - Hw1 Report

CS542200 Parallel Programming Homework 1: Odd-Even Sort

107062228 陳劭愷

Implementation

本次作業中，需要實作 Odd-Even Sort 實現跨節點的排序。首先，先實作出一個無優化的版本。再逐步嘗試優化與做一些實驗上的改變。

Basic

首先將需要將資料分散在節點上，假設有 `n` 筆資料 `size` 個節點，最直觀的想法就是每個節點都分配到 `n / size` 個值，並且若 `n` 不能整除 `size`，則前

`n % size` 個節點可以多分到一個值。由此紀錄一變數 `m` 代表此節點分配到的數字量。

另外要計算出每個節點分配到哪一段序列，最簡單的做法當然是由 0 號節點開始，依序分配到 m 個值。因此紀錄一變數 `offset` 代表此節點被分配到 `[offset, offset+m)` 的值，`offset` 也被用於節點讀檔與寫檔時，可以知道要從哪一個位元開始讀取。

```
int m = n / size + (rank < n % size);
int offset = n / size * rank + min(n % size, rank); // the offset
```

```
MPI_File_read_at(in_file, sizeof(float) * offset, arr, m, MPI_FLOAT, MPI_STATUS_IGNORE);
MPI_File_close(&in_file);
```

接著是如何實現 Odd-Even Sort 的算法，最簡單的實作首先對每個 local array 做排序，可以使用 `std::sort` 或是 `boost:sort::spreadsort`，再來開始進行奇數輪與偶數輪的排序。將節點兩兩配對，配對後確認節點是在左側還是右側，左右兩側的節點互相交換資料後，左側節點保留前 `m` 小的資料，右側節點保留後 `m` 大的資料，因為資料是排序過的，所以可以使用額外的空間在 `O(m)` 時間內完成。

由於需要與兩邊的節點傳輸資料，需要在一個額外的陣列儲存，以及需要知道會接收多少數字（也就是隔壁節點被分配到的數字量）。

```
int lm = m + (rank == n % size); // the left node's size
int rm = m - (rank + 1 == n % size); // the right node's size
```

```
float* rarr = new float[lm];
```

另外觀察可以發現，Odd-Even Sort 最多執行 `size + 1` 輪就會結束。以及如果節點數量大於總數字量，可能會有節點沒有分配到數字，因此判斷也需要判斷是否有數字可以傳遞與接收。

```
bool is_left = rank & 1;
int turn = size - 1;
while (turn--) {
  if (is_left && rank != size - 1 && m > 0 && rm > 0) { ... }
  if (!is_left && rank != 0 && m > 0 && lm > 0) { ... }

  // ...

  is_left ^= 1;
}
```

每一輪中，兩個配對的節點互相傳輸彼此的資料，並且左邊節點保留前 `m` 小的資料，右邊節點保留前 `m` 大的資料。因為兩個陣列的資料都是排序過的，所以可以使用一個額外的 `tarr` 陣列做 `O(m)` 的合併，完成後再將 `tarr` 中的資料放回原本的 local 陣列。

```
MPI_Sendrecv(arr, m, MPI_FLOAT, rank + 1, 0, rarr, rm, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
int l = 0, r = 0;
for (int i = 0; i < m; ++i) {
  if (r == rm || (l < m && arr[l] < rarr[r])) tarr[i] = arr[l++];
  else tarr[i] = rarr[r++];
}
std::copy(tarr, tarr + m, arr);
```

最後完成 `size + 1` 輪排序後，即可將資料寫到 output 檔案中。

```
MPI_File_open(MPI_COMM_WORLD, argv[3], MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &out_file);
MPI_File_write_at(out_file, sizeof(float) * offset, arr, m, MPI_FLOAT, MPI_STATUS_IGNORE);
MPI_File_close(&out_file);
```

Improvement

減少 MPI 傳輸資料的量

由於資料量很大，並且傳輸通常都會是比計算更大的 overhead。因此盡量減少傳輸的量可以大幅減少運行時間。

可以發現在做 `O(m)` 合併的時候，若左邊節點的最大值 ≤ 右邊節點的最小值，則兩節點合併後結果不變。所以在兩節點互相交換所有資料之前，可以先交換其最大值/最小值給配對的節點，做為需不需要再繼續交換剩餘資料的判斷。

```
MPI_Sendrecv(arr + m - 1, 1, MPI_FLOAT, rank + 1, 0, rarr, 1, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
local_done = (arr[m - 1] <= rarr[0]);

if (!local_done) {
  // ...
}
```

不要複製 Array，交換指標

合併後需要將資料複製回原本的 local 陣列，可以改用交換指標的方式，節省複製的時間。

兩項優化後，跑出：143.39, 141.78, 148.81, 142.13, 139.89 → 143.2

嘗試使用 Non-Blocking Send/Recv 來平行運算與 I/O 時間

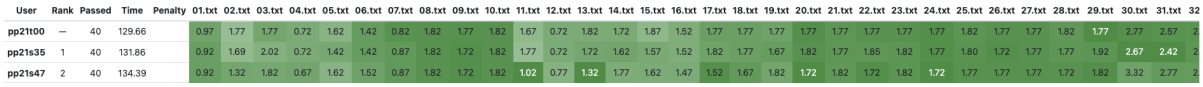
在實作完上述優化版本後，嘗試實作了幾個使用 Non-Blocking Send/Recv 的版本。核心想法為不要等待所有資料傳輸完再進行運算，而是分成多次傳輸，讓計算與傳輸時間可以重疊以降低最後的總運行時間。首先先以 blocking 互相交換一段資料，接著使用 non-block 傳輸不超過 `LIMIT` 筆資料，就可以開始處理目前已經接收到的資料。另外因為分次傳送，有可能在還沒傳送完所有資料時就已經合併完成，更可以減少 I/O 的量。

對於夠大量的資料（若每個節點有超過 500萬筆資料），控制其大約拆成 X 次傳輸，並且測量五次平均跑完全部測資來得到最好的 X。

- 3 次：142.72, 142.72, 141.66, 151.23, 139.30 → 143.526
- 4 次：147.08, 137.31, 149.27, 139.20, 138.93 → 142.358
- 5 次：142.87, 150.42, 146.31, 138.35, 148.90 → 145.370

- 6 次：138.58, 136.93, 146.41, 143.61, 139.04 → 140.914
- 7 次：157.60, 144.35, 147.11, 152.67, 145.45 → 149.436
- 8 次：137.53, 134.39, 145.30, 143.89, 138.59 → 139.940
- 9 次：144.23, 149.71, 152.27, 150.72, 146.75 → 148.736

因為 server 上面測試有點不準，有時誤差大到 20 秒，最後決定保留 6 次做為標準。事實上使用 Non-Blocking Send/Recv 並沒有與比使用 Blocking 快很多，但是跑出了 131.86 秒的成績。



User	Rank	Passed	Time	Penalty	01.txt	02.txt	03.txt	04.txt	05.txt	06.txt	07.txt	08.txt	09.txt	10.txt	11.txt	12.txt	13.txt	14.txt	15.txt	16.txt	17.txt	18.txt	19.txt	20.txt	21.txt	22.txt	23.txt	24.txt	25.txt	26.txt	27.txt	28.txt	29.txt	30.txt	31.txt	32
pp21t00	–	40	129.66		0.97	1.77	1.77	0.72	1.62	1.42	0.82	1.82	1.77	1.82	1.67	0.72	1.82	1.72	1.87	1.52	1.77	1.77	1.77	1.77	1.77	1.77	1.77	1.77	1.82	1.77	1.77	1.82	1.77	2.77	2.57	
pp21s35	1	40	131.86		0.92	1.69	2.02	0.72	1.42	1.42	0.87	1.82	1.72	1.82	1.77	0.72	1.72	1.62	1.57	1.52	1.82	1.77	1.67	1.82	1.77	1.85	1.82	1.77	1.80	1.72	1.77	1.77	1.92	2.67	2.42	
pp21s47	2	40	134.39		0.92	1.32	1.82	0.67	1.62	1.52	0.87	1.82	1.72	1.82	1.02	0.77	1.32	1.77	1.62	1.47	1.52	1.67	1.82	1.72	1.82	1.72	1.82	1.72	1.77	1.77	1.77	1.72	1.82	3.32	2.77	

Experiment & Analysis

Methodology

System Spec

學校提供的 Apollo Cluster。

Performance Metrics

使用 `std::chrono::steady_clock()` 在 IO、通訊以及運算的前後都埋一個時間戳記，使用這些時間戳記的差總和以毫秒為單位的總 IO、通訊與運算時間。

```
std::chrono::steady_clock::time_point start_time;

void start_span() {
  start_time = std::chrono::steady_clock::now();
}

void end_span(int& total) {
  total += std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::steady_clock::now() - start_time).count();
}
```

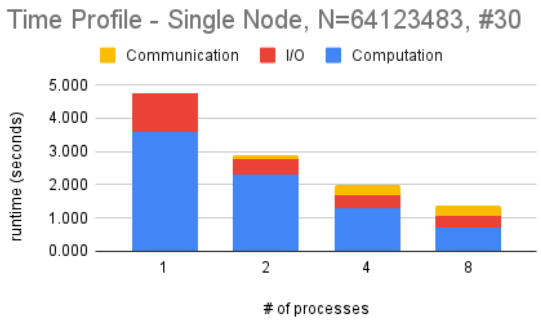
```
start_span();
MPI_Sendrecv(arr, m - 1, MPI_FLOAT, rank + 1, 0, rarr + 1, rm - 1, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
end_span(communication_time);
```

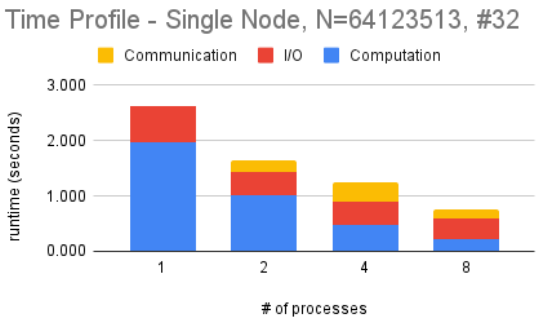
Plots: Speedup Factor & Time Profile

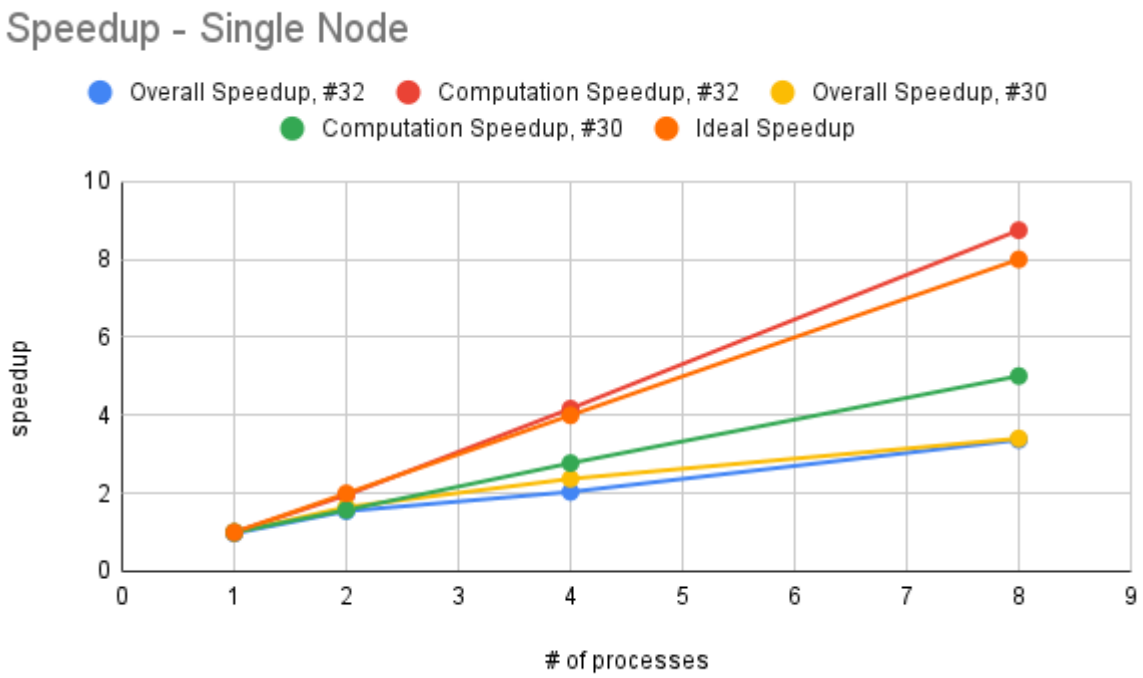
分別對 Blocking 與 Non-Blocking 版本，做 Single Node 與 Multiple Nodes 的實驗。每一組挑出兩筆測資做為實驗數據。最後觀察 overall speedup、computation speedup 與理想上的 linear speedup 做比較。

Blocking Send/Receive Version

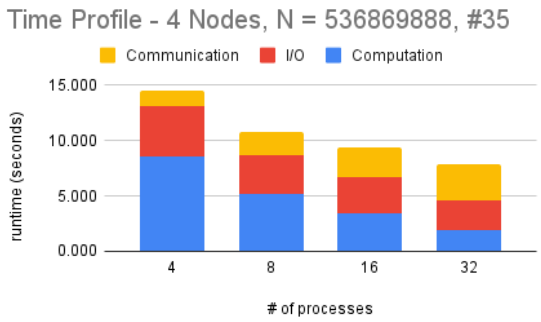
Single Node

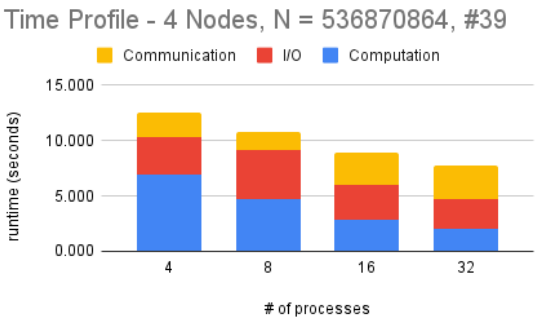


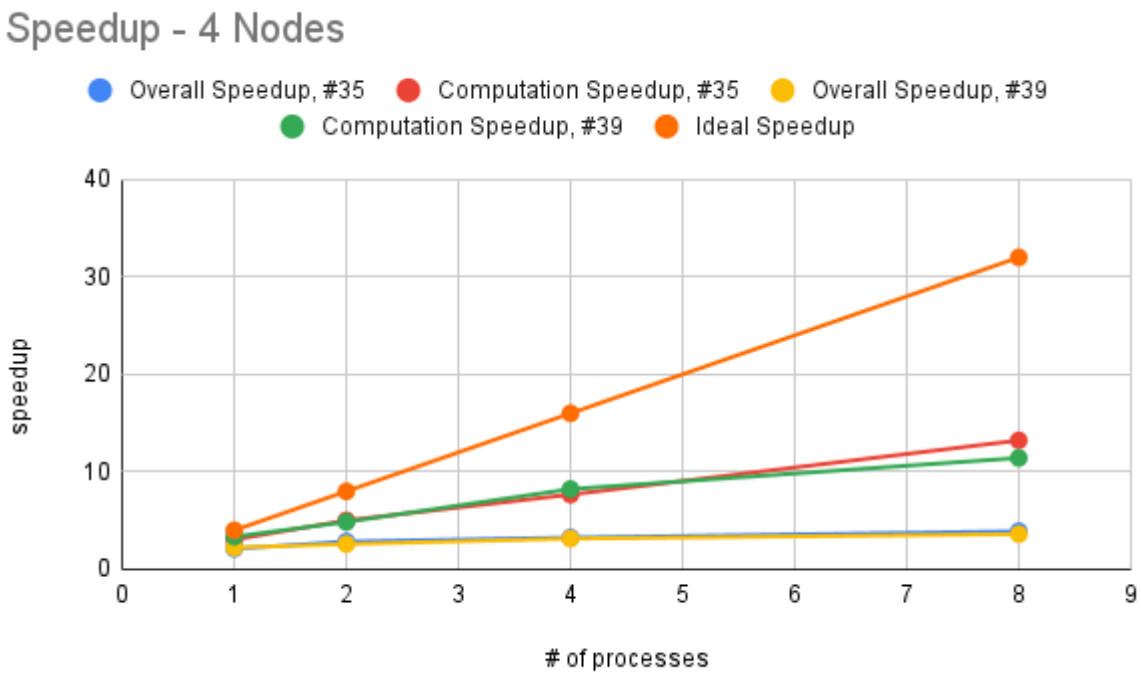




Multiple Nodes (4 Nodes)

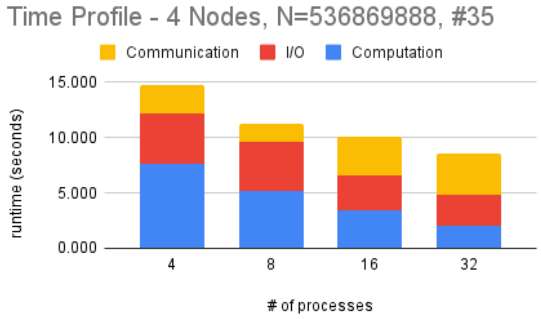


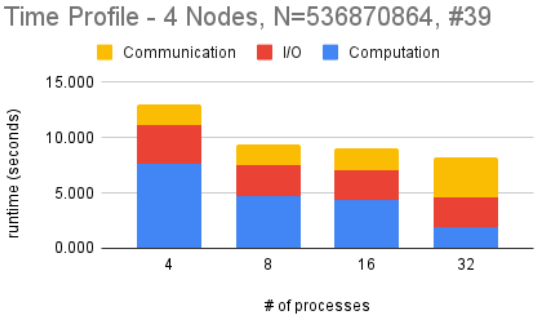


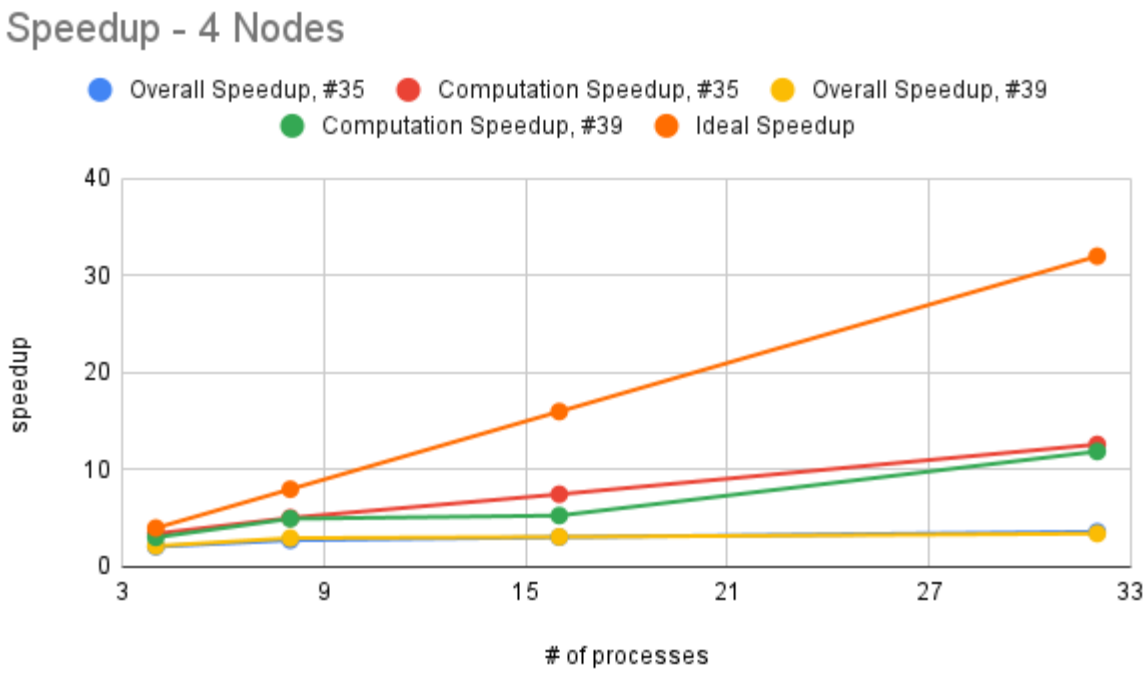


Non-Blocking Send/Receive Version

Multiple Nodes (4 Nodes)







Discussion

1. 根據 Time Profile 顯示，不管是 Non-Blocking 還是 Blocking，Single Node 還是 Multiple Nodes：

   - Computation time 都有隨著節點數量增多而下降，推測因為每個節點所需要負責做 local sort 的數量下降，合併時也只要負責更少數量的資料。
   - I/O time 也有隨著節點數量，但是下降的幅度很少。推測因為節點數量增多，所以先完成工作的人就可以先開始寫入檔案，但是 I/O 的 bandwidth 有限，所以並沒有辦法提升很多。
   - Communication time 變化量並不大，尤其 multiple nodes 時更不穩定。推測是因為可能經常溝通的兩節點之間網路延遲變化的可能性很大，若 A、B 是相鄰的兩節點，需要經常交換資料，但是被跑在不同機器上，可能就會提升 communication time。

   因此 communication time 是最大的 bottleneck，所以實作了 non-blocking 的版本希望能夠藉由 overlap 計算與通訊時間來降低總運行時間，並且與上面 improvement 提到的一樣，分次傳送資料可能可以提前結束合併，不需要把資料全部傳輸完。

但是結果並沒有如預期，推測除了因為網路因素的不穩定，而更多次的 communication 反而造成更多的不穩定之外，每次 communication 應該也會有一些準備時間的 overhead，造成效益不佳。另外可能預期可以做到的優化並不常發生，因為 merge 基本上還是很常需要互相交換完全部的資料才能完成。並且 computation 的速度還是比 communication 的速度快太多，所以 overlap 的效果不佳。

2. 整體來說，single node 的 overall speedup 雖然遠不及 ideal，但是隨著節點數量增多還是有直線上升。但是 multiple nodes 的 speedup 最多看起來只能到 single node single process 快三倍左右，再新增更多的節點應該也不會再變快。
3. 整體來說，single node 的 computation speedup 甚至有 super-linear speedup 的現象，推測是因為資料大小被分成更小之後，memory 與 CPU cache 更能發揮作用，導致運算變得更快。而 multiple nodes 的 computation speedup 表現的也不錯，32 個 processes 時可以提升約 12 倍。

Experiences & Conclusion

總結來說，這次的作業跟實驗讓我實際上體驗到要達到 ideal speedup 是很困難的，communication 的 overhead 會讓 speedup 變得困難。另外也發現平常就算時間複雜度一樣，但是只要資料量一大，每個簡單的運算都可能會被放大成好幾秒的差距。

為了不斷進步，這次作業總共實作了大大小小 8 個版本，從 blocking send/receive 的 4 個版本到 non-blocking send/receive 的 4 個版本，雖然最後 non-blocking 的版本並沒有達到預期的效果，但是讓我更了解 MPI 的各種操作，以及更能體驗到怎麼樣的改變才能真正達到顯著的優化。

另外因為每次跑實驗時都要去對比 txt 檔才能知道有多少資料，因此寫了一個腳本讓跑測試變得更簡單，再利用 file checksum 來檢查，達到全自動化測試的效果。

https://gist.github.com/justin0u0/01bdee03e751958b395957ec00be1ee9#file-runner-py

最後，因為 blocking 版本比較穩定，因此最後作業上傳的是 blocking 版本。其他版本放在 https://github.com/justin0u0/Parallel-Programming 上面，等 deadline 過後才會上傳！