Parallel Programming Homework 1: Odd-Even Sort

107060002 曹立元

Part I、Implementation

1. How I handle an arbitrary number of input items and processes

在分配待處理的數字給各 process 之前，我先將每個 process 應該處理的數字數量算出來，我的概念是每個 process 先平均分配 n / size 個數字，再將剩餘的數字（n % size 個）分給 rank 較小的 processes。

因此我定義了 local_n 及 remainder 兩個變數，local_n 即代表每個 process 會均分到多少個 process，remainder 則代表剩下沒被分到的數字的數量，若該 process 的 rank 小於 remainder 的值，則讓他的 local_n 加一，代表他要多幫忙處理一個數字，這些 rank 小於 remainder 的 processes 剛好可以把剩下要處理的數字分完。

```
int local_n = n / size; // the size for each process
int remainder = n % size; // If we assign (n / size) numbers to each process,
                          // there are still (n % size) numbers are waiting for assignment.
if(rank < remainder) local_n++; // assign one remaining numbers to processes whose rank is less than the
                                // number of remaining process
```

在知道自己所要 handle 的數字的數量之後，每個 process 也要知道在他兩側（rank +- 1）的 process 所要 handle 的量，**才能知道彼此進行溝通時的 argument count 的值**。所以我替每個 process 定義了 left_n 跟 right_n 兩個變數來記錄相鄰的 processes 所要 handle 的量。而由於 rank 小於 remainder 的 process 必須多 handle 一個數字，所以 rank == remainder 的 process 的 left_n 必須加一，而 rank == (remainder – 1)的 process 的 right_n 必須減一。

```
int left_n = local_n; // store how many numbers should be handled by the process on the left hand side
int right_n = local_n; // store how many numbers should be handled by the process on the right hand side

if(rank == remainder) left_n++;
if(rank == remainder - 1) right_n--;
```

有了以上資訊之後，各 process 也可以開始進行讀資料的部分，我宣告了 offset 這個 argument 來定義各 process 要讀寫資料的開頭位址。而因為 rank 小於 remainder 的那些 process 的 local_n 多了 1，代表那些 process 要多讀一個 float，所以對於 rank > remainder 的 processes 來說，再算 offset 時除了將 rank 乘上 local_n 之外，還要再加上 remainder * sizeof（float）個 bytes。而此 offset 值也會在各 process 將資料 sort 完準備寫入 output file 時用到，下圖為實作的部分：

```
int offset; // The position that a process read data from

if(rank < remainder){
    offset = sizeof(float) * local_n * rank;
}
else{
    offset = sizeof(float) * ((local_n * rank) + (remainder)); // Because processed whose rank < remainder reads
                                                               // sizeof(float) bytes more, the offset of others should
                                                               // be local_n * rank * sizeof(float) + remainder * sizeof(float)
}

MPI_File_read_at(f, offset, data, local_n, MPI_FLOAT, MPI_STATUS_IGNORE);

MPI_File f_out;
MPI_File_open(MPI_COMM_WORLD, argv[3], MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &f_out);
MPI_File_write_at(f_out, offset, data, local_n, MPI_FLOAT, MPI_STATUS_IGNORE);
```

2. How I sort in my program

(1) Sort the data of each process by radix sort

在進行 odd-even sort 之前，我會先將各 process 裡自己 handle 的 data sort 好，這樣在 odd-even sort 時只要和相鄰的 process 進行 merge 的動作就好，可以將 odd-even sort 的 time complexity 降到 O(local_n)。而在各 process 對自己的資料進行 sorting 時，我選擇的是 **radix sort**，因為他是個 linear time 的 sorting 演算法，會比直接呼叫 std::sort 更快，我將 32bit 的數字切成四個 8bit 的數字，並利用 counting sort 來對於每個 digit 進行 sorting。

而因爲我們這次要處理的是 floating point，所以不能直接對 raw data 進行 radix sort，必須先對浮點數進行轉換，我的做法是先將 float 轉為 unsigned int，此時若直接 sort 所有轉為 unsigned int 的 data，由小到大的順序為（小正數->大正數->大負數->小負數），所以我先將負數都先乘上(-1)，得到他們的 two's complement 值，則在排列時負數的部分就會是正確地由較小的負數排到較大的負數。再來我先 flip 負數的 MSB，讓他們保有負數 MSB 為 1 的特性，最後將所有 data 的 MSB 都 flip 一次，便可以調換正數及負數在 unsigned data 中的順序，使得最後排列出來的順序為正確的（負數小->負數大->正數小->正數大）。在做完 radix sort 後，要將 data 從 unsigned int 轉回 float，只要做上述動作的 inverse 就好，下圖為浮點數轉換的部分：

```
// convert floating point to 32 bit unsigned int
for(int i=0; i<local_n; i++){
    u_int32_t radix_num = *(u_int32_t*)&data[i];
    if(radix_num >> 31 == 1){
        radix_num *= -1;
        radix_num ^= (1 << 31);
    }
    radix_num ^= (1 << 31);
    data_for_sorting[i] = radix_num;
}

// radix sort
data_for_sorting = radix_sort(data_for_sorting, local_n);

// convert 32 bit unsigned int back to floating point
for(int i=0; i<local_n; i++){
    u_int32_t radix_num = data_for_sorting[i];
    radix_num ^= (1 << 31);
    if((radix_num >> 31) == 1){
        radix_num ^= (1 << 31);
        radix_num *= -1;
    }
    data[i] = *(float*)&radix_num;
}
```

處理好資料型態之後，就可以進行 radix sort 的部分，我將 32bit 的資料切成 4 個 8bit 的 digit 來進行 radix sort，對於每個 digit 的 sorting algorithm 我選的是 counting sort，方法就和一般的 counting sort 一樣，所以共要跑四次 counting sort，下圖為我的實作：

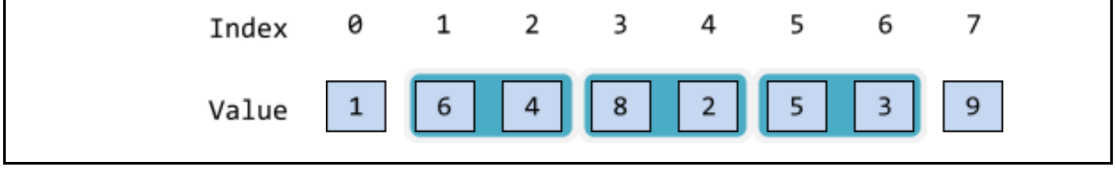
```
u_int32_t* counting_sort(u_int32_t* data_arr, int local_n, int shift_bits){
    // b is the array stores the result
    u_int32_t* b = new u_int32_t[local_n];
    // array size is 256 because we want to sort an 8-bit number
    u_int32_t* c = new u_int32_t[256];

    // Initialize counters
    memset(c, 0, 256*sizeof(u_int32_t));

    // count the number of occurrence of a particular value
    for(int i=0; i<local_n; i++){
        c[(data_arr[i] >> shift_bits) & 0xFF]++;
    }

    for(int i=1; i<256; i++){
        c[i] += c[i-1];
    }

    for(int i=local_n-1; i>=0; i--){
        b[(c[(data_arr[i] >> shift_bits) & 0xFF]--) - 1] = data_arr[i];
    }

    return b;
}

u_int32_t* radix_sort(u_int32_t* data_for_sort, int local_n){
    u_int32_t* temp = new u_int32_t[local_n];
    temp = data_for_sort;

    for(int i=0; i<4; i++){
        // shift 8 bits after doing one round of counting sort to sort another digit
        int shift_bits = i * 8;
        temp = counting_sort(temp, local_n, shift_bits);
    }
    return temp;
}
```

(2) Odd-Even Sort

我在實作 odd-even sort 時的大部分概念就和他原本的概念差不多，而在溝通時我使用的是 MPI_Sendrecv( )這個 function，他可以同時接收及傳送資料，只需溝通一次就好，send/recv count 的部分使用原本定義好的 local_n、left_n 及 right_n 來處理。而在 sorting 時我是直接使用 merge 的方法，編號較大的 process 把較大的數字拿走，編號較小的數字則把較小的數字拿走，彼此不會有重疊或衝突的狀況發生。額外 handle 的部分有 rank 為 0 或 size-1 的 process 的 idle round、在 merge 時如果兩個 process handle 的 data size 不同要注意 index 是否在正確範圍，以及在 copy data 時直接交換原本的 data array 和 sorted data array 的 pointer，避免 memory copy 造成的 overhead 等等。

最後使用 MPI_Allreduce( )這個 function 將所有的 process sorting 的結果 AND 起來，如果大家都已經 sort 好了，一整輪都沒有資料交換的動作，則可以結束 odd-even sort 的部分。因為裏面大部分的 code 概念都差不多，我就只放上一部份的實作 code（for 下圖的這個 phase 中 rank 為偶數的 processes）。

3. [Odd-phase] odd/even indexed adjacent elements are grouped into pairs.




```
// round odd
if(rank % 2 == 0){
    if(rank == 0){
        // Do nothing
    }
    else{
        // Send/recv data to/from the process on the left hand side
        MPI_Sendrecv(data, local_n, MPI_FLOAT, rank-1, 0,
                     left_buf, left_n, MPI_FLOAT, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // merge (collect bigger data)
        data_index = local_n - 1;
        left_index = left_n - 1;
        for(int i=0; i<local_n; i++){
            if(data[data_index] >= left_buf[left_index]){
                sorted_buf[local_n - 1 - i] = data[data_index];
                data_index--;
            }
            else{
                sorted_buf[local_n - 1 - i] = left_buf[left_index];
                left_index--;
                isSorted = false;
            }
        }
        // Swap pointer of old/sorted data array
        tmp = data;
        data = sorted_buf;
        sorted_buf = tmp;
    }
}
```

3. Other efforts I've made in my program

我額外做的事情大概就是想辦法減少一些不必要的時間花費，例如將一些可能不必重複進行工作想辦法移到迴圈外，或是在 merge 時透過指標交換的方式避免 memory copy 的 overhead 等等。大部分的時間還是花在想辦法成功實作 radix sort 的部分，最後使用 radix sort 跑出來的秒數也比直接使用 std::sort( )快了約二十秒。

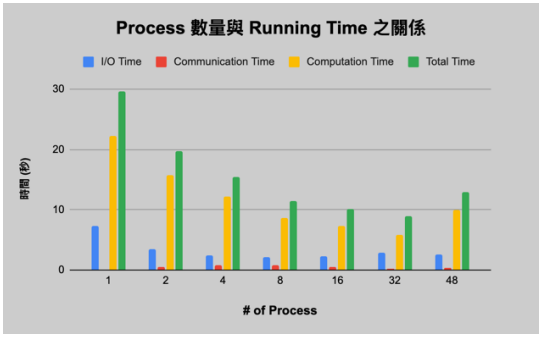
Part II、Experiment & Analysis

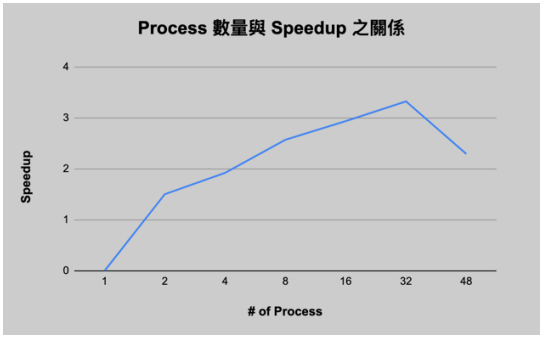
1. Methodology (Performance Metrics)

在計算 computation、communication、I/O time 的時候，我是使用 MPI_Wtime( )這個 function 來計算，分別在一個 task 的前後測量一次時間，並用結束時間減掉開始時間即可得到結果。另外，我的所有實驗都是取跑三次同樣設定的平均值。Computation time 的部分我主要是計算 radix sort 加上 odd-even sort 的時間，communication time 的部分我計算的是 odd-even sort 時各 process 之間傳遞資料的時間和最後統整個 process sorting 結果的時間，而 I/O time 的部分計算的就是一開始讀資料和最後寫資料的時間。

2. Plots: Speedup Factor & Time Profile

我在實驗時選擇的是 testcase 中的第 33 筆測資，並比較跑在 1、2、4、8、16、32、48 個 process 時各項時間的差異，下面兩張圖為實驗的結果：





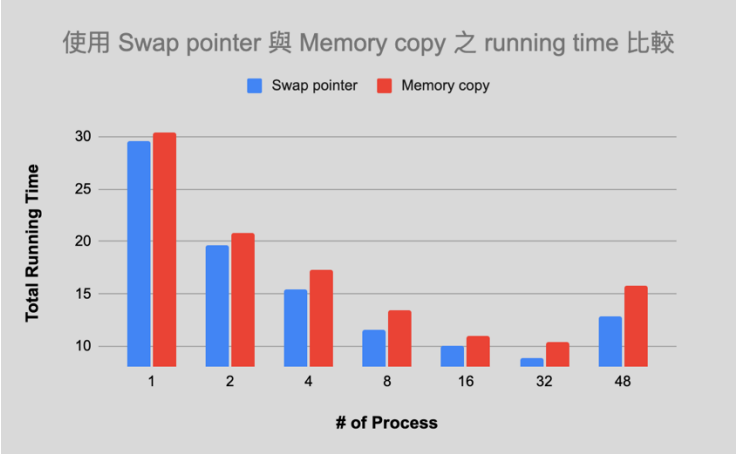
3. Discussion (Must base on the results in your plots)

由圖中可以發現這次的作業中主要的 bottleneck 是在 computation 的部分，應該是因為整支程式主要的時間就是花在大量的 sorting，而溝通和 I/O 的部分相對簡單，因此在費時的佔比上也較小。而在我將 std::sort( )改為使用 radix sort 之後，整體進步了約二十秒，我之後再想辦法精簡我的程式卻也沒有讓秒數在進步，我想如果要再進步的話，應該還是要對於資料做出更好的分析，讓 sorting 時可以避免重複處理很多已經處理好的資料。

再來是 scalability 的部分，隨著 process 數量增加有讓整體的效能更好，表示我的程式確實有利用到平行化的好處，再來我發現使用 32 個 process 能達到最好的效果，speedup 的部分約達到使用單一 process 的三倍，process 數再往上加時卻讓效果不增反減，但 bottleneck 還是在 computation 的部分，我想應該還是需要再處理資料的部分更加優化，才能讓 scale 繼續增加時有更好的表現。

4. Others

在寫這次作業中我有改善的一個地方是在 merge 兩個 process 的資料時，原先我使用的方法是 memory copy 的方法，把存入 sorted data 的 array 中的資料直接 copy 到該 process 原本的 data array，不過光想就知道很花時間，所以我也利用第 33 筆測資進行了使用 swap pointer 與 memory copy 之速度差異的實驗，同樣地我分別做了 process 數為 1、2、4、8、16、32、48 的實驗，每種實驗會做三次取平均，下圖為實驗結果，可以發現如果使用 memory copy 的方法，不論在何種情形下他的計算時間都會比 swap pointer 來得慢。



Part III、Experiences / Conclusion

我覺得在這次作業中讓我學習到滿多的，因為之前自己在寫程式時通常就是讓測資能夠全部通過就好，幾乎沒有在做優化，而平程這種對於運算時間斤斤計較的科目便能讓我從頭到尾好好思考自己的程式到底有哪些地方是可以改進的。過程中為了想讓程式不斷進步，還可以順便複習到演算法學過的東西，也改善自己的 coding style。

在這次作業中最主要還是學習到跟 MPI 有關的一些實作，無論是 I/O 或者在溝通時怎樣比較有效率，或者怎樣分配給各個 process 時可以避免造成 bottleneck 等等。除此之外，之前我也沒有自己實作過 radix sort、counting sort 的經驗，也不會特別去除多餘的迴圈、memory copy 之類的事，在這次作業中我確實在 coding 的部分學習到很多。

不過我的程式還是沒有辦法像前面那些人一樣厲害，我自己也找不太出到底還有哪些東西可以再優化了，未來還是有很多可以學習的地方！