Parallel Programming HW1

108032053 陳凱揚

Implementation

我的實作分為以下幾個步驟。

- Load Balancing

  為了解決每次執行時，不同大小的input和process的情況，我採取的方法為先平均分配相同大小的data給每個process，假設剩餘k個data，則給前k個process各一個data，使data的分配達到最平均的狀態，另外需要同時記錄下 `start` 和 `len` ，也就是此process所負責data開始的index值和數量，以便之後的MPI IO處理。

  ```
  int unit = n/size, remain = n%size;
  int start = unit*rank+std::min(remain, rank);
  int len = unit+(rank < remain);
  ```

- MPI IO

  在讀取與寫入資料的過程，我使用了 `MPI_File_read_at` 和 `MPI_File_write_at` ，設定好每個process要讀取的區間，使每段區間互不重疊，就能夠平行的讀取與寫入，提升IO的效率。

  ```
  MPI_File input_file, output_file;
  // Read data
  MPI_File_open(MPI_COMM_WORLD, input_filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &input_file);
  MPI_File_read_at(input_file, sizeof(float)*start, data, len, MPI_FLOAT, MPI_STATUS_IGNORE);
  MPI_File_close(&input_file);
  // Write data
  MPI_File_open(MPI_COMM_WORLD, output_filename, MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &output_file);
  MPI_File_write_at(output_file, sizeof(float)*start, data, len, MPI_FLOAT, MPI_STATUS_IGNORE);
  MPI_File_close(&output_file);
  ```

- Local Sort

  接著我會將每個process所負責的data都先做過一次sort，之後便以process為單位做odd-even sort，而不是以data element為單位做odd-even sort，原因在於使用這種方法能將odd-even sort的個數減少很多，不僅能夠減少sort的時間，也能減少process之間溝通的次數。

  而在這裡我上網查詢到了一種hybrid radix sort，叫做 `boost::sort::spreadsort::spreadsort` 來加速local sort的過程，比起使用傳統comparison方法的 `std::sort` 快上了約1.3倍。

  ```
  boost::sort::spreadsort::spreadsort(data, data+len);
  ```

- Odd-Even Sort

  接下來是odd-even sort的部分，我使用了 `rank_p` 和 `len_p` 來分別儲存此process在even phase及odd phase時，partner的rank和負責的data數量，而有時頭和尾會沒有partner，則設為 `MPI_PROC_NULL` ，例如：process 0在even phase時的partner是process 1，在odd phase時的partner是 `MPI_PROC_NULL` 。

  每次sort時會先使用 `MPI_Sendrecv` 將自己的data傳送給partner，並接收partner的data，接著2個process中，rank較小的process會透過 `merge` function從中找出較小的那一半data，反之rank較大的process會找出較大的那一半data。

而sort的過程最多會執行odd+even次數共process size+1次，因為交換的過程在worst case時，會從最後一個process交換到第一個process，且會因為邊界問題沒有交換到，所以最多需要size+1次才能交換完成。

```
for(int i = 0; i < size+1; i += 2){
    // Even
    MPI_Sendrecv(data, len, MPI_FLOAT, rank_p[0], 0, buf, len_p[0], \
                MPI_FLOAT, rank_p[0], 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    merge(data, tmp, len, buf, len_p[0], rank<rank_p[0]);
    std::swap(data, tmp);
    // Odd
    MPI_Sendrecv(data, len, MPI_FLOAT, rank_p[1], 0, buf, len_p[1], \
                MPI_FLOAT, rank_p[1], 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    merge(data, tmp, len, buf, len_p[1], rank<rank_p[1]);
    std::swap(data, tmp);
}
```

```
void merge(float *data, float *tmp, int len, float *buf, int len_p, bool small){
    if(small){
        // if(len_p==0 || len==0 || data[len-1]<buf[0]) return;
        int data_idx = 0, buf_idx = 0;
        for(int j = 0; j < len; j++){
            if(data_idx<len && (buf_idx>=len_p || data[data_idx]<buf[buf_idx]))
                tmp[j] = data[data_idx++];
            else
                tmp[j] = buf[buf_idx++];
        }
    }
    else{
        // if(len_p==0 || len==0 || data[0]>buf[len_p-1]) return;
        int data_idx = len-1, buf_idx = len_p-1;
        for(int j = len-1; j >= 0; j--){
            if(data_idx>=0 && (buf_idx<0 || data[data_idx]>buf[buf_idx]))
                tmp[j] = data[data_idx--];
            else
                tmp[j] = buf[buf_idx--];
        }
    }
    // memcpy(data, tmp, len*sizeof(float));
}
```

- Optimization
  - 從以data element變成以process為單位來做odd-even sort，這個改變對效能的影響非常明顯，原先的甚至在第28個testcase就會TLE了。
  - 用 `spreadsort` 替代 `std::sort` ，效能也明顯增進不少。
  - 減少迴圈內的memory allocation，把會使用到的3個data array都在一開始就先allocate好。另外應該也要避免function call，但在這裡實測將merge過程攤開在迴圈內的話對效能幾乎沒產生甚麼影響，因此為了可讀性，還是決定將merge包成一個function來使用。
  - 在merge過程結束後，需要將新的data資料重新存回原先的陣列中，這裡改由直接swap指標，減少data copy所浪費的時間。
  - 選擇最適合的MPI API來使用，像是迴圈內的send和recv的過程，我們不需要拆開成 `MPI_Send` 和 `MPI_Recv` ，而是可以直接使用他所提供的 `MPI_Sendrecv` 。

Experiment & Analysis

- Methodology

  我的測試同樣是在課程所提供的cluster上做的，並以第40個testcase作為我的測資，資料量為536869888。

  - Computing time

    我的測量方式是在 `MPI_Init` 與 `MPI_Finalize` 之間加上 `MPI_Wtime` 記錄程式總共的執行時間，再減去communication time和IO time，即為computing time。

  - Communication time

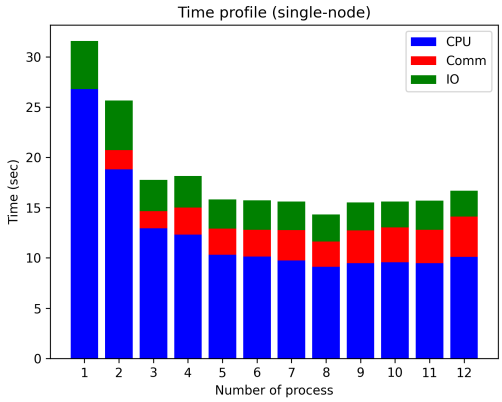
    在每個 `MPI_Sendrecv` 的前後加上 `MPI_Wtime` 記錄時間，並全部加總起來即為communication time。

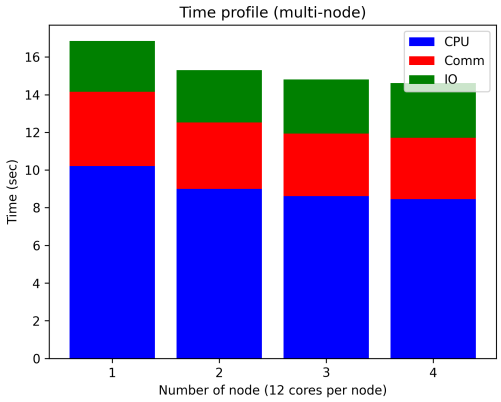
  - IO time

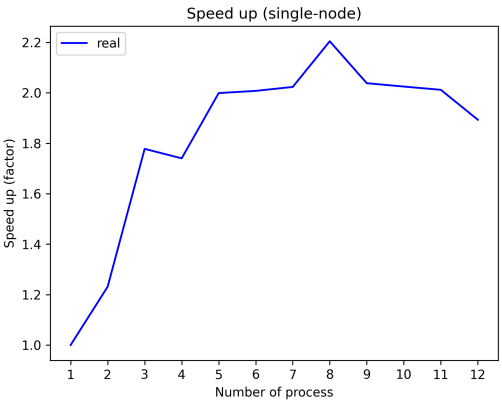
    如同communication time，在每個 `MPI_File` 相關的function前後加上 `MPI_Wtime` 記錄時間並加總即為IO time。

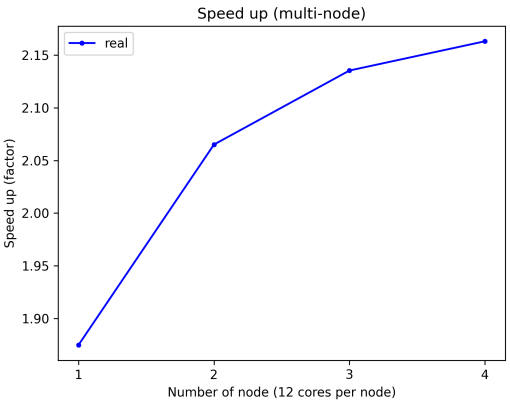
  此外，在每個process都會有各自的computing, communication, IO time，所以我會將每個process的3種時間各自加總起來取平均，得到此次執行的3種時間。

- Plots: Speedup Factor & Time Profile









- Discussion
  - Time Profile

    在single node的部分可以看到隨著process的增加，總時間和CPU time都有隨之減少的趨勢，而Communication time也隨著增加，這符合我們的預期，IO的部分則是沒有太大的差異，另外可以注意到在4個process左右之後，執行的總時間便沒有太大的下降了，這是因為CPU time可以平行化的部分已經平行到相對小於其他CPU time，因此再增加process沒辦法有太大的下降，甚至可能因為溝通時間而讓總時間增加，像是使用12個process時，此時的bottleneck就會從CPU time變成Communication time。

    而在multi-node的部分，也是因為平行化的部分已經越來越小了，所以雖然process增加很多，但時間卻只下降一點點，而IO time和Communication time並沒有太大的差異，這是因為溝通的部分只存在於每次向左右兩個process做溝通，因此不會隨著process的增加而大量下降CPU time，所以能夠優化的地方會變成在IO和Communication上，像是加強硬體的效能或是採取更好的protocol。

  - Speedup Factor

    在single node的部分，可以看到加速比最大只到了約2倍，代表可能這隻程式能夠平行化的部分並沒有很大，或是我寫出來的方式沒有讓他最大的平行化。在multi-node時也只到了2倍多。我覺得除了IO和Communication的固定時間造成scaling的能力下降外，也可能是因為單個process在處理data時，執行時間與data量沒有很接近線性的關係，使data量下降很多時，執行時間卻只有下降一點點。若要再想辦法獲得更好的speed up的話，可能需要在IO或Communication上做優化。

Conclusion

在這次的作業中，我覺得對於了解MPI的運作很有幫助，之前雖然也有使用過mpirun來跑平行程式，但對其中的運作和概念都不太清楚，只知道下指令後，效能就會隨著提升了。而在作業中，因為整份code是自己寫出來的，所以概念和想法上都很清楚這支平行程式是怎麼跑起來的，又是甚麼原因讓他可以平行化，還有其中的瓶頸可能卡在哪邊。

另外，我覺得這次作業最大的困難在於不斷的精進優化程式效能，一開始要寫出一個基本版能夠平行化的程式不會太過於困難，接著找出一些明顯能夠改進的地方也不會太難，但是在這之後的優化就很困難了，像是不同的程式架構可能影響編譯出來的組語等較底層的問題，連帶也影響到了平行程式的平行效果，這個部份很難去發現並做優化，需要累積不少經驗才能夠即時改進。像是在這次作業中，感覺已經做了所有我能看見可能能優化的地方，但在排行榜中仍然只能排在中段處，很好奇排名前段的同學是怎麼再繼續做優化的。