Parallel Programming HW2

108032053 陳凱揚

Implementation

我的實作分為以下幾個步驟。

- Pthread
  - Load Balancing

    我分配工作給thread的方式不是採用static固定數量的points，而是讓每個thread執行完目前的工作後，呼叫 `get_chunk` 來取得下一塊chunk來計算，以 `idx` 來記錄目前取到了第幾個點，又因為每個thread都會去呼叫這個function而修改到 `idx` ，所以這邊使用 `pthread_mutex_t` 來保護這個變數；而 `CHUNK` 是每塊chunk的大小，在這裡我也不是直接設定他的數值大小，而是以能夠使用的cpu數量來決定，下面除50的意思是將所有points分成50個chunk，讓thread去自由取得，且size最小只能為2，這和後面使用到的SIMD有關。

    ```
    CHUNK = max(2, total/cpu_num/50);

    Pair get_chunk(){
        pthread_mutex_lock(&mutex);
        Pair res = {idx, min(CHUNK, total-idx)};
        idx += res.S;
        pthread_mutex_unlock(&mutex);
        return res;
    }
    ```

  - SIMD

    使用 `cat /proc/cpuinfo` 指令後，可以看到我們所使用的 `Intel(R) Xeon(R) CPU X5670` 有支援 `sse2` ，代表我們可以使用 `__m128d` 來同時做2個雙精度浮點數(double)的計算。

    當每個thread取到一塊chunk後，我採用了以上SIMD的方式來做計算，而且我使用了類似pipeline的方式，當兩個點有其中一個點算完後，馬上遞補下一個點，而不是兩兩計算完後，才開始下兩個的計算。

    程式碼的部分有點冗長，主要是將兩組數字都包進 `__m128d` 裡，像是 `x0` , `y0` , `x` , `y` , `squared` ，並以提供的函數來做加減乘除，如 `_mm_add_pd` ；並以 `end1` 和 `end2` 分別紀錄這兩點是否計算完成，當完成時就load進下一個數字直到結束為止。

```
__m128d SIMD_2 = _mm_set1_pd(2);
while(true){
    Pair p = get_chunk();
    if(p.S == 0) break;

    ll idx1 = p.F, idx2 = p.F+1;
    bool end1 = false, end2 = (idx2==p.F+p.S);
    if(end2) idx2 = p.F+p.S-1;

    int j1 = idx1/width, i1 = idx1%width;
    int j2 = idx2/width, i2 = idx2%width;
    double y0_1 = j1 * ((upper - lower) / height) + lower;
    double x0_1 = i1 * ((right - left) / width) + left;
    double y0_2 = j2 * ((upper - lower) / height) + lower;
    double x0_2 = i2 * ((right - left) / width) + left;

    int repeats1 = 0, repeats2 = 0;
    double tmp;

    __m128d SIMD_y0 = _mm_set_pd(y0_1, y0_2);
    __m128d SIMD_x0 = _mm_set_pd(x0_1, x0_2);
    __m128d SIMD_x = _mm_set1_pd(0);
    __m128d SIMD_y = _mm_set1_pd(0);
    __m128d SIMD_squared = _mm_set1_pd(0);

    while(!end1 || !end2){
        __m128d SIMD_tmp = _mm_add_pd(_mm_sub_pd(_mm_mul_pd(SIMD_x, SIMD_x), \
                            _mm_mul_pd(SIMD_y, SIMD_y)), SIMD_x0);
        SIMD_y = _mm_add_pd(_mm_mul_pd(_mm_mul_pd(SIMD_x, SIMD_y), SIMD_2), SIMD_y0);
        SIMD_x = SIMD_tmp;
        SIMD_squared = _mm_add_pd(_mm_mul_pd(SIMD_x, SIMD_x), _mm_mul_pd(SIMD_y, SIMD_y));
        if(!end1){
            _mm_storeh_pd(&tmp, SIMD_squared);
            if(++repeats1==iters || tmp>=4){
                image[j1 * width + i1] = repeats1;
                if(max(idx1, idx2)+1 != p.F+p.S){
                    idx1 = max(idx1, idx2)+1;
                    j1 = idx1/width, i1 = idx1%width;
                    y0_1 = j1 * ((upper - lower) / height) + lower;
                    x0_1 = i1 * ((right - left) / width) + left;
                    SIMD_y0 = _mm_set_pd(y0_1, y0_2);
                    SIMD_x0 = _mm_set_pd(x0_1, x0_2);
                    repeats1 = 0;

                    _mm_storel_pd(&tmp, SIMD_x);
                    SIMD_x = _mm_set_pd(0, tmp);
                    _mm_storel_pd(&tmp, SIMD_y);
                    SIMD_y = _mm_set_pd(0, tmp);
                    _mm_storel_pd(&tmp, SIMD_squared);
                    SIMD_squared = _mm_set_pd(0, tmp);
                }
                else end1 = true, idx1 = p.F+p.S-1;
            }
        }
        if(!end2){
            // similar to if(!end1)
        }
    }
}
```

- Hybrid
  - Load Balancing

    在hybrid版本中，有兩個地方需要做好load balancing，第一個是分配給每個rank的工作量，第二個是在每個rank底下，分配給每個thread的工作量。

    - Rank

      原先最好的計畫為實作出 dynamic 方式的 centralized work pool，並在其中一個rank裡開一個thread來負責工作的分配；或是採取static的加強版，像是以間隔的距離取點、間隔的row取點，因為Mandelbrot Set的圖形大多具有locality，意思是鄰近點的repeats會相近，因此間隔取點的方式便能讓工作量較平均的分配於thread間。

      比較可惜的是此兩種方法實作出來後有些許bug，因此最後採取的是較簡單的數量平均分配，雖然是簡單的平均分配，但因為是將2D的圖形攤平成1D來分配，還是稍微的分散了前面提到的locality，效能表現上中規中矩，並不會太差，但可能在worst case時就會很差了。

      ```
      unit = total/size, remain = total%size;
      start = unit*rank+min(remain, rank);
      len = unit+(rank < remain);
      ```

    - Thread

      而在thread的部分，我採取了與pthread相同的方法來分配，也就是每個thread做完工作後，就會去取得下一塊chunk來做計算；而這裡同樣也需要有個lock來保護 `idx` ，在這裡會使用的是 `omp_lock_t` ；最後在 `#pragma` 的部分就只使用了簡單的 `parallel` 和設定 `num_threads` 數量。

      ```
      CHUNK = max(2, len/cpu_num/30);

      Pair get_chunk(){
          omp_set_lock(&lock);
          Pair res = {idx, min(CHUNK, start+len-idx)};
          idx += res.S;
          omp_unset_lock(&lock);
          return res;
      }

      #pragma omp parallel num_threads(cpu_num)
      {
          __m128d SIMD_2 = _mm_set1_pd(2);
          while(true){
              Pair p = get_chunk();
              if(p.S == 0) break;

              // similar to pthread
          }
      }
      ```

- SIMD

  與pthread一樣，使用了 `__m128d` 來進行SIMD的計算。

Experiment & Analysis

- Methodology
  - System Spec

    我的測試是在課程所提供的cluster上做的，並以 `strict23` 作為我的測資。

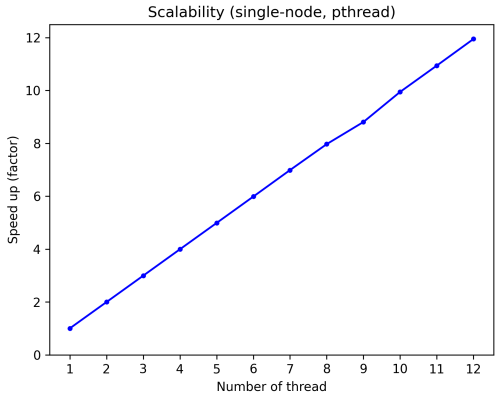
    ```
    srun ./exe $out 10000 0.3182631 0.3182632 -0.4128295 -0.4128296 2575 3019
    ```

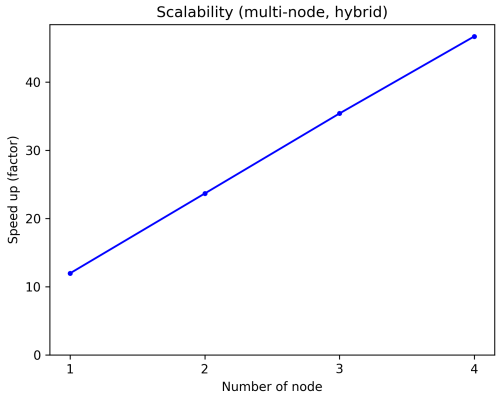
  - Performance Metrics

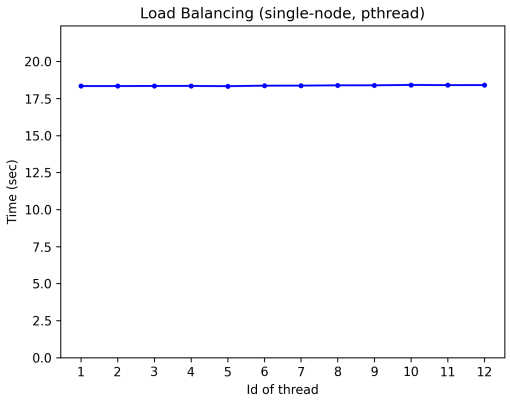
    在pthread中，我是使用 `<sys/time.h>` 裡的 `gettimeofday()` 來計算時間，可以算出每個thread單獨的執行時間和程式的總時間，而總時間裡我去掉了較不重要的 `write_png` 。

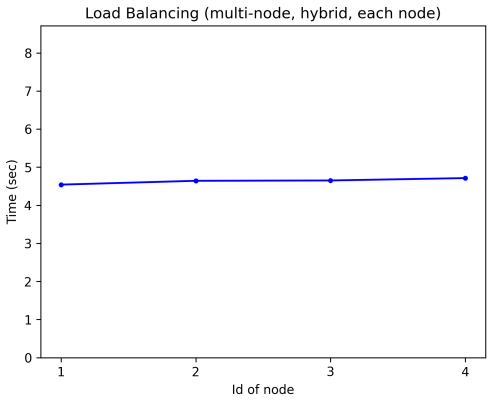
    在hybrid裡，我用 `MPI_Wtime()` 來取得MPI的執行時間，同樣是去掉了較不重要的 `write_png` ；而在thread裡我使用 `omp_get_thread_num` 先取得thread id，再用 `omp_get_wtime` 計算時間。

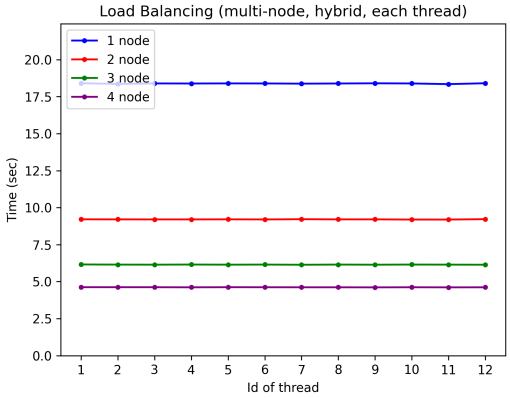
- Plots: Scalability & Load Balancing











- Discussion
  - Scalability

    從圖中可以看出我們平行的相當成功，不管是在single-node，還是在multi-node，scalability都幾乎是完成的線性上升，這應該是因為 Mandelbrot Set 的計算量非常重，而且整支程式幾乎從頭到尾都在做計算，不但很少溝通，也沒有IO，而且又是 Embarrassingly Parallel，每個計算都互相獨立，這讓我們很容易地去做平行化，也很容易達到很好的效果，只有在 `get_chunk()` 時的 `mutex` 可能稍微減少了一些平行的部分。

  - Load Balancing

    兩張圖中也能夠看出工作量分配得相當平均，這歸功於dynamic的分配工作量給thread，減少thread空閒的情況，而在multi-node的MPI執行時間，雖然在這裡還算平均，但因為沒有實現出dynamic的分配，所以可能在一些測資上會不夠平均，不像pthread能夠在各種測資中都有一定程度的表現；但multi-node的thread執行時間因為與pthread採取相同方法，所以同樣有不錯的結果。

Conclusion

在這次的作業中，我感受到了 Embarrassingly Parallel 的計算平行化效果非常好，又因為缺少了IO和溝通，讓整體的bottle neck幾乎只卡在了 load balancing 上，只要有做好 dynamic 的分配，整體效能就很容易大幅提升，不像在HW1的odd-even sort中，因為溝通成本很重，平行化的部分又不多，平行起來的scalability就相當差。另外這次還有個很有趣的地方在於SIMD的實作，第一次知道了他的實作細節，也注意到他與組語的觀念息息相關，像是資料儲存的排序就不像平常array那樣，在 `_mm_storel_pd` 和 `_mm_storeh_pd` 就卡了一段時間才成功做對。