多處理機平行程式設計 HW1

報告說明：這次的作業主要練習使用 point-to-point communication 與 tree-structured communication 使各process之間溝通時間降到 O(log(n))並仍然可以得到與 serial computing 相同的結果。

Problem1：Circuit Satisfiability Problem

1. What have you done?

a. 撰寫 serial code 與 使用 MPI function 的code 進行處理時間的比較。

b. 在使用 MPI function 的code 中，將point-to-point communication 組織成tree-structured communication 使所有 process 傳遞 local sum 給 process 0之間的速度增加，若是使用 for 迴圈單一傳遞的話時間複雜度為 O(n)，但使用 tree-structured communication 則可將時間壓縮到 O(log(n))

c. 額外寫了一個 funtion(calculate_legal_circuit_count(int id,int process_size)讓每個 process 可以執行 checkCircuit 這個 function 到達分配的次數，並加總計算符合電路圖規定的數量有幾種。

2. Analysis on your result

Serial Code

```
#include <stdio.h>       // printf()
#include <limits.h>      // UINT_MAX
#include <time.h>

int checkCircuit (int, int);

int main (int argc, char *argv[]) {

   int i;                /* loop variable (32 bits) */
   int id = 0;           /* process id */
   int count = 0;        /* number of solutions */
   double START,END;     /* time stamp */

   START = clock();
```

```
  /* Calculate legal circuit count */
  for (i = 0; i <= USHRT_MAX; i++) {
      count += checkCircuit (id, i);
  }
  END = (clock() - START) / CLOCKS_PER_SEC;


  printf ("Excute time: %f",END );
  printf ("Process %d finished.\n", id);
  printf("\nA total of %d solutions were found.\n\n", count);
  fflush (stdout);
  return 0;
}

#define EXTRACT_BIT(n,i) ( (n & (1<<i) ) ? 1 : 0)
#define SIZE 16

int checkCircuit (int id, int bits) {
  int v[SIZE];          /* Each element is a bit of bits */
  int i;

  for (i = 0; i < SIZE; i++)
    v[i] = EXTRACT_BIT(bits,i);

  if (   (v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
      && (!v[3] || !v[4]) && (v[4] || !v[5])
      && (v[5] || !v[6]) && (v[5] || v[6])
      && (v[6] || !v[15]) && (v[7] || !v[8])
      && (!v[7] || !v[13]) && (v[8] || v[9])
      && (v[8] || !v[9]) && (!v[9] || !v[10])
      && (v[9] || v[11]) && (v[10] || v[11])
      && (v[12] || v[13]) && (v[13] || !v[14])
      && (v[14] || v[15])   )
  {
      printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d \n", id,
        v[15],v[14],v[13],v[12],
        v[11],v[10],v[9],v[8],v[7],v[6],v[5],v[4],v[3],v[2],v[1],v[0]);
      fflush (stdout);
```

```
        return 1;
    } else {
        return 0;
    }
}
```

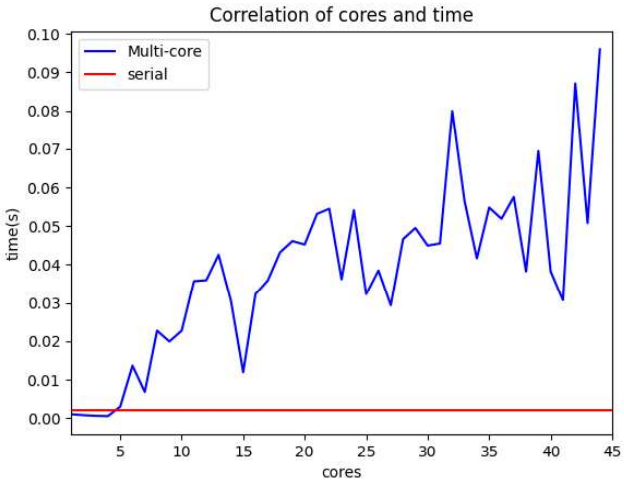
執行時間與結果：

```
問題  輸出  偵錯主控台  終端機  JUPYTER

$ ./problem1.exe
0) 1001100111110101
0) 1001100111110110
0) 1001100111110111
0) 1001101111110101
0) 1001101111110110
0) 1001101111110111
0) 1001110111110101
0) 1001110111110110
0) 1001110111110111
Excute time: 0.002000Process 0 finished.

A total of 9 solutions were found.
```

用 serial code 運算符合題目中電路圖所有可能跑出來的時間 0.002 秒左右。取三次執行時間平均為 0.001972。但若使用 MPI_function 並運用 tree-structured communication 的話，呼叫不同數量的核心運算時間也不同。

以下為使用 cluster 後最慢 process 結束時間三次所需的平均運算時間畫成的折線圖與詳細數據。

Core	AVG	Core	AVG	Core	AVG
1	0.002043	16	0.011962	31	0.044983
2	0.001049	17	0.032349	32	0.045517
3	0.000811	18	0.035913	33	0.079910
4	0.000638	19	0.043246	34	0.056451
5	0.000556	20	0.046133	35	0.041674
6	0.003019	21	0.045253	36	0.054817
7	0.013663	22	0.053167	37	0.051907
8	0.006834	23	0.054501	38	0.057543
9	0.022765	24	0.036240	39	0.038290
10	0.019941	25	0.054129	40	0.069413
11	0.022756	26	0.032329	41	0.038246
12	0.035739	27	0.038523	42	0.030730
13	0.035991	28	0.029385	43	0.087102
14	0.042594	29	0.046646	44	0.050800
15	0.030601	30	0.049513	45	0.095969



從上圖可知，使用 MPI function 但沒有平行運算(n=1) 時整體運行時間會較一般 serial code 慢上一些(0.002043)，若以 Multi-core 同時進行運算，以這個題目在 n = 2 ~ n = 5 左右時，運行時間都是可以較一般 serial code 還快上一些，但當 n > 5 時，或許是 core 的數量增多，造成需要 communication 的時間增加，導致整體運行時間開始上升，反而效率沒有來得 serial code 佳。因此 core 的使用數量在執行平行運算時也會是重要的考量，並不是越多 core 同時運行所需時間就會較少，還要考量到 process 之間的 communication overhead。

3. Any difficulties?

在將 Problem1 改為使用 MPI function 平行計算時，比較主要遇到的困難是一開始不知道 MPI_Send() 與 MPI_Recv() 是阻塞式通訊，會擔心 process 尚未接收到訊息或是順利傳送訊息到其他 process 時，其他 process 就繼續進行下一個指令，造成 tree-structured communication 時，接收的 process 可能會還沒接收到訊息導致每一個階段接收到的 sum 會有誤。

一開始試著在每層結束時使用 MPI_Barrier()希望能確保每層 sum 的正確性，但最後發現加入 MPI_Barrier()後即便 multi-core 去計算結果很難比 serial code 快速。最後試著去理解 MPI_Send()與 MPI_Recv()的傳輸方式才發現這兩個 funtion 是阻塞式的 function，不論 MPI_Recv()先執行，或是 MPI_Send()先執行，先執行的 process 都會確保對方收到訊息才會繼續執行剩餘的 code，因此得到的 global sum 都會是正確的。

Problem2：Monte Carlo method

1. What have you done?

a. 撰寫 serial code 與 使用 MPI function 的 code 進行處理時間的比較。

b. 在使用 MPI function 的 code 中，將 point-to-point communication 組織成 tree-structured communication 使 process 0 broadcast total tosses 至所有 process 以及其他 process 傳遞 local sum 給 process 0 之間的速度增加，若是使用 for 迴圈單一傳遞的話時間複雜度為 O(n)，但使用 tree-structured communication 則可將時間壓縮到 O(log(n))

c. 完成可以隨機產生 x,y 的參數當作投擲點並判斷投擲點是否有在圓圈內的 function

2. Analysis on your result

Serial code

```c
#include<stdio.h>
#include <stdlib.h>
#include <time.h>

long long int is_in_circle();

int main() {

    long long int number_in_circle = 0;
    long long int total_tosses = 0;

    // Make every round would get difference number of tosses in the circle
    time_t t;
    srand((unsigned) time(&t));

    // Scan total tosses number from keyboard
    scanf_s("%lld", &total_tosses);

    double START,END;
    START = clock();

    for (int i = 0; i < total_tosses; i++) {
            number_in_circle += is_in_circle();
    }

    END = (double)(clock() - START) / CLOCKS_PER_SEC;
    printf ("Excute time: %.6f\n",END );

    double ans =  (double)(4 * number_in_circle) / total_tosses;
    printf("PI = %.6f\n",ans);

}
```

```c
// Calculate how many tosses in the circle within distributed tosses of the process
long long int is_in_circle() {

    int min_value = -1;
    int max_value = 1;

    double x = ((max_value - min_value) * (double)rand() / (RAND_MAX)) + min_value;
    double y = ((max_value - min_value) * (double)rand() / (RAND_MAX)) + min_value;
    double distance_squared = x * x + y * y;

    return  (distance_squared <= 1) ? 1 : 0;

}
```

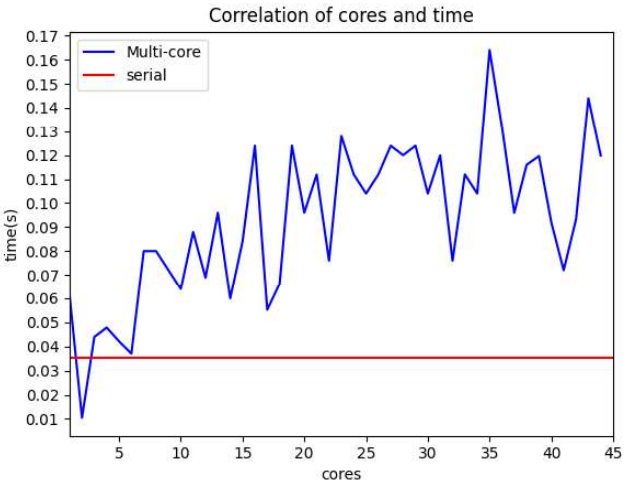
Total tosses = 1,000,000

Serial code 執行時間與結果:

```
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
1000000
785011
Excute time: 0.032119
PI = 3.140044
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
1000000
785880
Excute time: 0.040418
PI = 3.143520
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
1000000
785773
Excute time: 0.033621
PI = 3.143092
```

Serial code 在 total tosses = 1,000,000 時，執行三次平均時間為 0.035386。
和 Multi-core 進行比較結果如下:

Core	AVG	Core	AVG	Core	AVG
1	0.060886	16	0.083980	31	0.103990
2	0.060958	17	0.123963	32	0.119942
3	0.010462	18	0.055244	33	0.075966
4	0.043976	19	0.066136	34	0.111973
5	0.047842	20	0.123983	35	0.103983
6	0.042124	21	0.095980	36	0.163996
7	0.037037	22	0.111901	37	0.131984
8	0.079976	23	0.075963	38	0.095949
9	0.079974	24	0.128000	39	0.115974
10	0.071979	25	0.111978	40	0.119640
11	0.063957	26	0.103977	41	0.091967
12	0.087963	27	0.111980	42	0.071995
13	0.068864	28	0.123971	43	0.093211
14	0.095974	29	0.120006	44	0.143978
15	0.059993	30	0.123977	45	0.119862



由上可知當 total tosses = 1,000,000 時，只有 n = 3 時使用 multi-core 可以使運算速度超越 serial code，推測或許是在剛開始 Process 0 傳遞 total tosses 給其他 Process 與最後加總丟進 circle 的 local sum 時，經過兩次 process 通訊導致執行時間變慢。n=1 時比一般 serial code 慢的原因可能是有呼叫許多 MPI function 導致。

Total tosses = 10,000,000

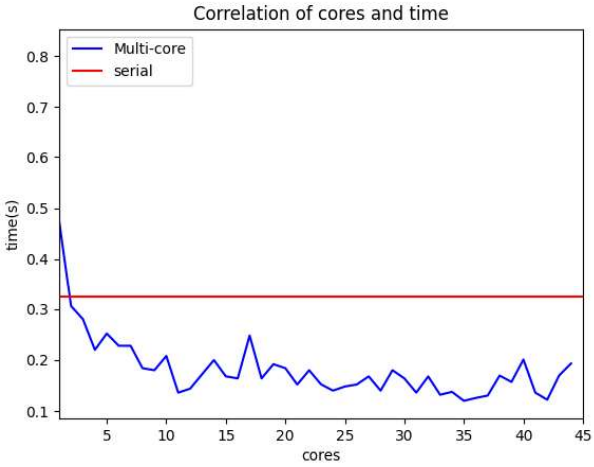
接下來試著將 total tosses 放大試試看，設 total tosses = 10,000,000 時對 serial code 與 multi-core 進行分析。

Serial code 執行時間與結果:

```
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
10000000
7854953
Excute time: 0.314979
PI = 3.141981
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
10000000
7855877
Excute time: 0.333518
PI = 3.142351
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
10000000
7853281
Excute time: 0.323859
PI = 3.141312
```

Serial code 在 total tosses = 10,000,000 時，執行三次平均時間為 0.324119。
和 Multi-core 進行比較結果如下:

Core	AVG	Core	AVG	Core	AVG
1	0.818171	16	0.167978	31	0.163959
2	0.474170	17	0.163982	32	0.136019
3	0.306017	18	0.247989	33	0.167853
4	0.279932	19	0.163984	34	0.131908
5	0.219963	20	0.191831	35	0.137585
6	0.251972	21	0.183974	36	0.119977
7	0.227973	22	0.151953	37	0.125595
8	0.227951	23	0.179970	38	0.130273
9	0.183974	24	0.152314	39	0.223195
10	0.179986	25	0.139973	40	0.251979
11	0.207988	26	0.148023	41	0.263984
12	0.135997	27	0.151977	42	0.135987
13	0.143770	28	0.167974	43	0.122178
14	0.171977	29	0.139985	44	0.169639
15	0.199983	30	0.179974	45	0.347975



由上可知當 total tosses = 10,000,000 時, 除了 n = 1 與 n= 2 外，其餘運算得到結果的時間都比 serial code 快速，即便 core 的數量越多，communication overhead 越大，運算的時間也是 serial code 的 2-3 倍。

n = 1 時比一般 serial code 慢的原因仍然可能是有呼叫許多 MPI function 導致。
n = 2 也沒有像 Problem1 一樣運算時間較 n=1 大幅下降的原因可能為，由於這次 input 的大小是 process 0 傳遞給其餘 process，但只有兩個 Process 的情況下，無法達到樹狀傳遞的效益，也會使 Process 2 開始運算的時間變慢，因此整個過程結束的時間也會較慢。

Total tosses = 100,000,000

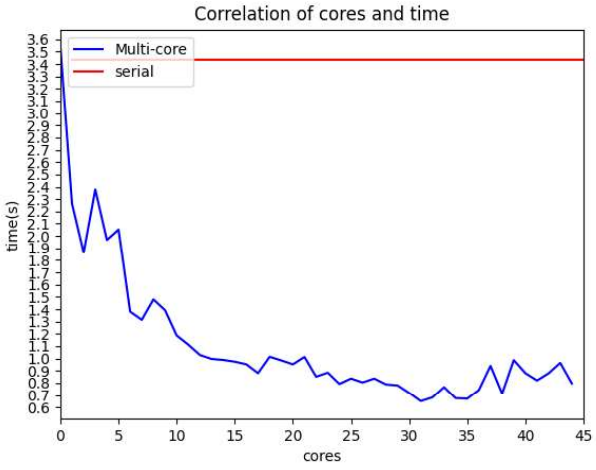
Serial code 執行時間與結果:

```
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
100000000
78535782
Excute time: 3.385916
PI = 3.141431
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
100000000
78530805
Excute time: 3.546329
PI = 3.141232
P76101322@sivslab-pn1:~/HW1$ ./serial_problem2
100000000
78539728
Excute time: 3.371873
PI = 3.141589
```

再試著將 input 大小放大 10 倍做觀察，設定 total tosses = 100,000,000 時，執行三次平均時間為 3.434706。

和 Multi-core 進行比較結果如下:

Core	AVG	Core	AVG	Core	AVG
1	3.534224	16	0.973388	31	0.717746
2	2.261083	17	0.951977	32	0.649681
3	1.863406	18	0.879978	33	0.679941
4	2.378093	19	1.013649	34	0.765236
5	1.967073	20	0.983967	35	0.673722
6	2.051982	21	0.951979	36	0.669773
7	1.379848	22	1.011980	37	0.741831
8	1.313497	23	0.850888	38	0.939460
9	1.479398	24	0.883938	39	0.711982
10	1.391971	25	0.791982	40	0.986324
11	1.186765	26	0.835976	41	0.879983
12	1.111985	27	0.803988	42	0.819978
13	1.027973	28	0.835780	43	0.878773
14	0.996000	29	0.787973	44	0.963965
15	0.987979	30	0.779977	45	0.795950



由上可知當 total tosses = 100,000,000 時, 除了 n = 1 外，其餘運算得到結果的時間都比 serial code 快速，即便 core 的數量越多，communication overhead 越大，運算的時間也到達 serial code 的數倍。n = 1 較一般 serial code 慢的原因同上段所描述。

3. Any difficulties?

在完成 Problem2 的過程中，主要遇到的困難是在如何讓 process 0 用樹狀的方式傳遞 total tosses 給其他 process。最後是考慮讓 process 0 傳給 process 1,2，process 1, 2 分別傳給 3,4 與 5,6，按照 full binary tree 的方式來傳遞以達到最佳效益。

過程中，一直遇到觸發中斷點的狀況，最後才發現是沒有考慮到 n = 1 的狀況時，process 不會傳遞 total tosses 出去才造成觸發。

結論：

在 problem1 時，由於要計算的數量都是固定的，因此當 Process 超過一定的數量時，由於 communication overhead 會變大，因此 process 數量越多反而總須完成任務的時間越長。

而 problem2，則是 total tosses 在到達一定的數量前，serial code 的表現反而會較 multi-core 表現好，因為 multi-core 要花費一些時間在 process communication 上。

而隨著 total tosses 數量越大時，multi-core 的優勢有明顯出現了，分配 tosses 讓各個 process 加總計算最後計算完成的時間會較 serial 好上許多，但由於要 total tosses 大到一定程度才有明顯效果，故屬於 weak scaling 的範圍。

另外，Cluster 的使用也似乎會受到在線人數的多寡影響到效能，測出來的數據偶爾會有些浮動，在這次報告中的數據都是以當下抓取到的數據作為紀錄。