嵌入式系統軟體設計

Embedded System Software

PA1

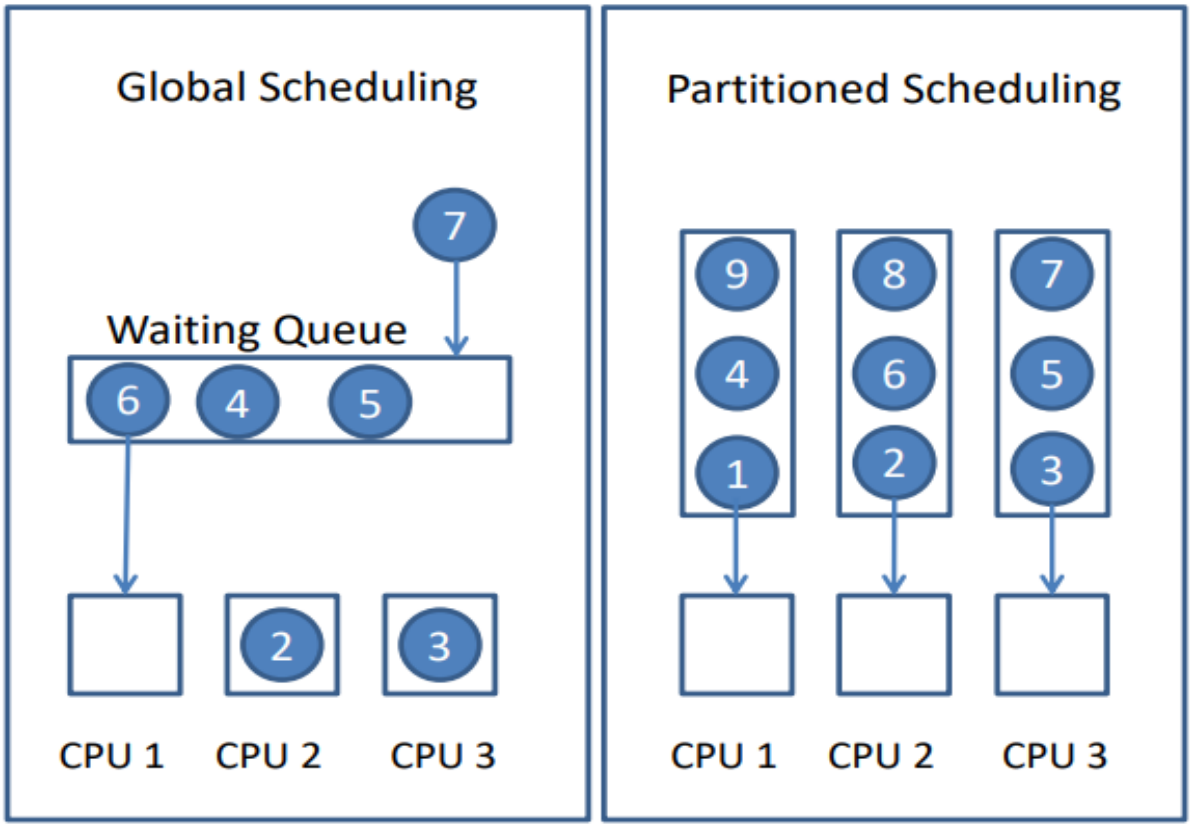
教授: 陳雅淑

學號: M11007328

姓名: 陳柏華

Part 1:

Divide the matrix convolution into four independent parts which could execute concurrently. Then use multi-thread execution with Global and Partition scheduling to boost the performance of matrix convolution. The execution result is demonstrated in Figure



Global Scheduling: 所有任務都先不分配 Core，之後看哪顆 Core 有空就照優先度排入空的 Core。

Partition Scheduling: 所有任務都先分配好要排進哪一個 Core，選定後不換 Core 執行。

[Global Scheduling. 10%]

▪ Describe how to implement Global scheduling by using pthread. 5%

~~~~~~ system.cpp ~~~~~~

```
9   #if (PART == 1)
10          /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~~*/
11          // For part1, we assign the matrix0 into all threads
12          threadSet[i].init(multiResult[0], matrix[0], mask[0]);
13          // Set up the caculation range of each thread matrix
14          if (i == 0){
15              threadSet[i].setStartCalculatePoint((threadSet[i].matrixSize() / 10) * 0);
16              threadSet[i].setEndCalculatePoint((threadSet[i].matrixSize() / 10) * 1);
17          }
18          else if (i == 1){
19              threadSet[i].setStartCalculatePoint((threadSet[i].matrixSize() / 10) * 1);
20              threadSet[i].setEndCalculatePoint((threadSet[i].matrixSize() / 10) * 3);
21          }
22          else if (i == 2){
23              threadSet[i].setStartCalculatePoint((threadSet[i].matrixSize() / 10) * 3);
24              threadSet[i].setEndCalculatePoint((threadSet[i].matrixSize() / 10) * 6);
25          }
26          else{
27              threadSet[i].setStartCalculatePoint((threadSet[i].matrixSize() / 10) * 6);
28              threadSet[i].setEndCalculatePoint((threadSet[i].matrixSize() / 10) * 10);
29          }
30
31          /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
32  #else
```

將大小為 10000 的輸入資料分配給 4 個 Thread (Thread 0: 1000、Thread 1: 2000、Thread 3: 3000、Thread 4: 4000)

分配方法為呼叫 Thread.h 裡面的 setStartCalculatePoint 與 setEndCalculatePoint 兩個涵式，透過這兩項涵式可以設定 Thread.h 的私有變數 startCalculatePoint 與 endCalculatePoint，以便在等等做 convolution 時可以分辨每個 Thread 的執行範圍。

透過迴圈執行需要 Thread 數量的次數(Part1 為 4 個 Thread)，每回圈一次即初始化並設定範圍給一個 Thread。變數 i 代表第幾個 Thread 當 i = 0 起始設位置為 0 終止位置設為 1000。i = 1 起始設位置為 1000 終止位置設為 3000 後面以此類推。彼此 Thread 的處理範圍不重疊。
~~~~~~

```
118  void
119  System::globalMultiCoreConv ()
120  {
121      std::cout << "\n==========Start Global Multi-Thread Convolution==========" << std::endl;
122      check->setCheckState(GLOBAL);
123      setStartTime();
124
125      /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~*/
126      // Create thread and join
127      for (int i = 0; i < numThread; i++){
128          pthread_create (&threadSet[i]._thread, NULL, &threadSet[i].convolution, &threadSet[i]);
129      }
130      for (int j = 0; j < numThread; j++)
131          pthread_join (threadSet[j]._thread, NULL);
132      /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
133
134      setEndTime();
135      std::cout << "Global Multi Thread Spend time : " << _timeUse << std::endl;
136      cleanMultiResult();
137  }
```

先用 pthread_create 創建所有需要用到的 Thread (Part1 為 4 個 Thread)，接著以相同的方式呼叫 pthread_join 用來等待 Thread 結束。

~~~~~~ Thread.cpp ~~~~~~

```
33  void*
34  Thread::convolution(void* args)
35  {
36    Thread *obj = (Thread*) args;
37
38  #if (PART == 3)
39      obj->setUpScheduler();
40  #endif
41    /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~*/
42    // Set up the affinity mask
43    obj->setUpCPUAffinityMask(obj->core);
44    /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
```

```
118  Thread::setUpCPUAffinityMask (int core_num)
119  {
120    /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~*/
121    // Pined the thread to core.
122    cpu_set_t set;
123    CPU_ZERO (&set);
124    CPU_SET (core_num, &set);
125    sched_setaffinity (0, sizeof(set), &set);
126    /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
127  }
```

透過 system.cpp 裡面 pthread_create 創建的 Thread 會呼叫 convolution 函數並輸入一開始各 Thread 所分配到的資料執行卷積運算，在運算前須先設定 CPU Affinity 讓 Thread 盡量長時間的在指定的 CPU 上運作，這部分程式為老師提供的方法。
~~~~~~

▪ **Describe how to observe task migration. 5%**

```
46  /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~~~*/
47  // Edit the function into partial multiplication.
48  // Hint : Thread::startCalculatePoint & Thread::endCalculatePoint
49  int curPID = syscall (SYS_gettid);
50  obj->cur_core = sched_getcpu();
51  obj->Get_PID(curPID);
52  pthread_mutex_lock( &count_Mutex );
53    obj->printThreadInfo();
54  pthread_mutex_unlock( &count_Mutex );
55  int shift = (MASK_SIZE-1)/2;
56  // std::cout << "startCalculatePoint " << obj->startCalculatePoint << " to " << obj->endCalculatePoint << std::endl;
57  for (int i = obj->startCalculatePoint; i < obj->endCalculatePoint; i++)
58  {
59    for (int j = 0; j < obj->_matrixSize; j++)
60    {
61      for (int k = -shift; k <= shift; k++)
62      {
63        for (int l = -shift; l <= shift; l++)
64        {
65          if( i + k < 0 ||  i + k >= obj->_matrixSize || j + l < 0 ||  j + l >= obj->_matrixSize)
66            continue;
67          obj->multiResult[i][j] += obj->matrix[i+k][j+l] * obj->mask[k+shift][l+shift];
68        }
69      }
70      /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~~~*/
71      // Observe the thread migration
72      int newCpu = sched_getcpu();
73      if (obj->cur_core != newCpu){
74        std::cout << "The thread " << obj->_ID;
75        std::cout << " PID " << obj->PID;
76        std::cout << " is moved from CPU " << obj->cur_core;
77        std::cout << " to " << newCpu << std::endl;
78        obj->cur_core = newCpu;
79      }
80      /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
81    }
```

```
196  void
197  Thread::Get_PID (int pid)
198  {
199    PID = pid;
200  }
```

這部分為 convolution 的算法與如何觀察 **task migration**，49 行是為找到當前 PID 為多少，50 行是為了抓到當前使用到的 CPU 為哪一個 core，51 行是我自己寫的涵式它的目的是為了把當前 PID 的值寫到 Thread.h 的私有變數 PID 裡面方便後續呼叫。後續透過預設涵式 printThreadInfo 滿足顯示上的條件，這部分加入 Mutex 是因為沒加的時候有發現不同的 Thread 可能會同時使用這個涵數，造成顯示上某些 Thread 的結果會出現在同一行，加上 Mutex 後可以解決這個問題。

在第 57 行將迴圈 i 的初始值與結束位置設改成之前對個別 Thread 設定的 startCalculatePoint 與 endCalculatePoint，這樣可以讓涵式知道每個 Thread 該執行的範圍。

72 行是為了知道 Thread 在不同 CPU 中的移動狀況，每一輪偵測一次當前的 CPU 為哪一顆，當偵測到 CPU 在不同顆時(與 Thread.h 裡的 cur_core 不同)就顯示出來。78 行為將當前新的 CPU 改寫到 Thread.h 裡的 cur_core 繼續執行下一次。

[Partition Scheduling. 5%]

▪ Describe how to implement partition scheduling by using pthread.

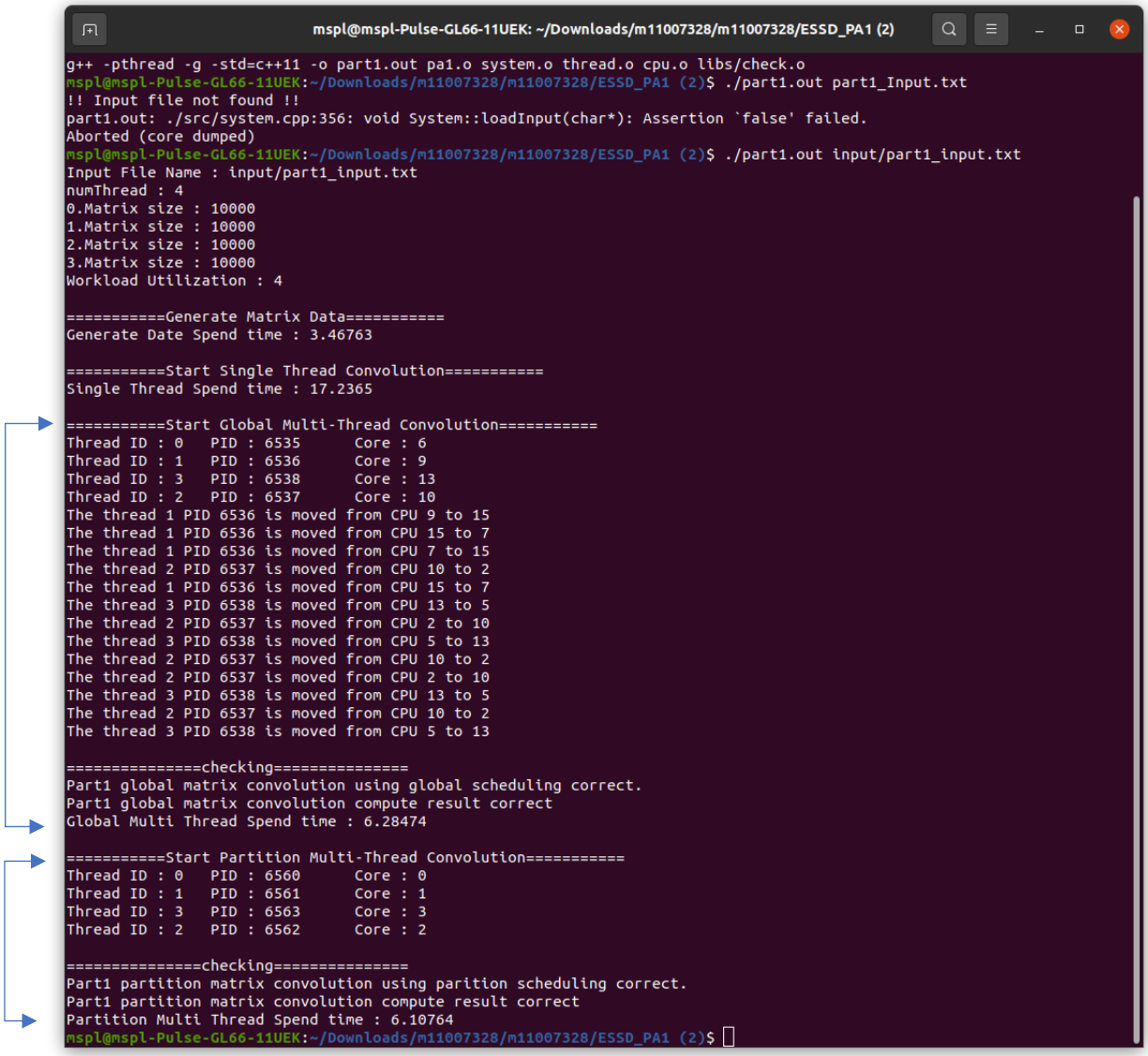
~~~~~~ system.cpp ~~~~~~

```
139  void
140  System::partitionMultiCoreConv ()
141  {
142  #if (PART == 1)
143      std::cout << "\n==========Start Partition Multi-Thread Convolution==========" << std::endl;
144      check->setCheckState(PARTITION);
145      for (int i = 0; i < numThread; i++)
146          threadSet[i].setCore(i);
147  #endif
148      setStartTime();
149
150      /*~~~~~~~~~~~~~~Your code(PART1)~~~~~~~~~~~~*/
151      // Create thread and join
152      for (int i = 0; i < numThread; i++){
153          pthread_create (&threadSet[i]._thread, NULL, &threadSet[i].convolution, &threadSet[i]);
154      }
155      for (int j = 0; j < numThread; j++)
156          pthread_join (threadSet[j]._thread, NULL);
157      /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
158
159      setEndTime();
160      std::cout << "Partition Multi Thread Spend time : " << _timeUse << std::endl;
161      cleanMultiResult();
162  }
```

由於在 Part1 中 Global Scheduling 與 Partition Scheduling 使用的資料相同的且只會設定一次，但會先執行 Global Scheduling 所以我在進入到 Partition Scheduling 之前，先透過迴圈加上 Thread.h 裡的 setCore 涵式將 Thread 先分配到指定的 Core 上藉此規定後續只讓該 Thread 在指定的 Core 上運作。
~~~~~~

[Result. 10%]

▪ Show the scheduling states of tasks. (You have to show the screenshot result of using the input part1_input.txt)

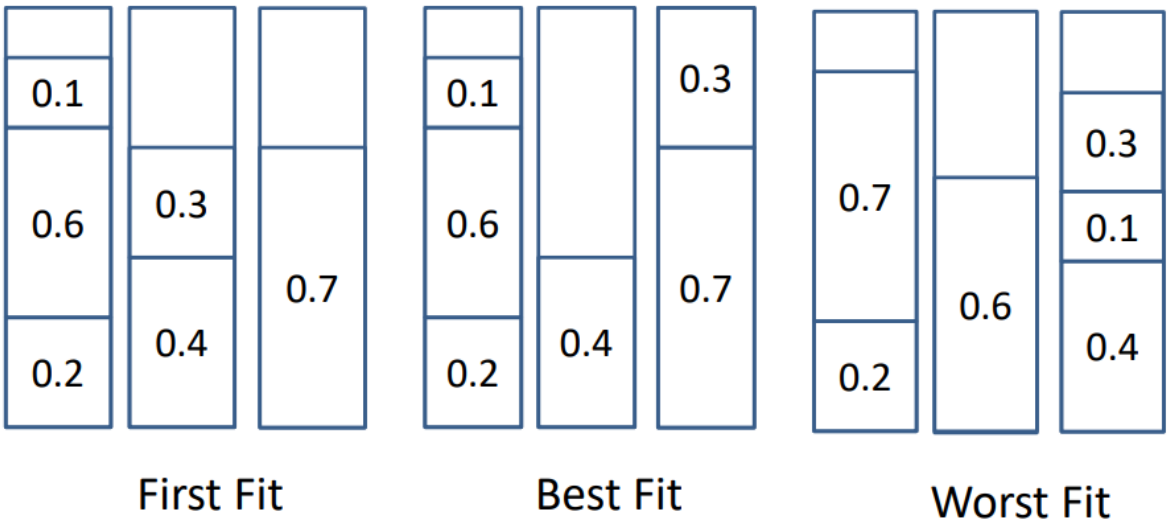


Part 2:

[Partition method Implementation. 10%]

▪ Describe how to implement the three different partition methods (First-Fit, Best-Fit, Worst-Fit) in partition scheduling.

- 0.2 -> 0.6 -> 0.4 -> 0.7 -> 0.1 -> 0.3



First Fit: 所有情況都優先排第一個 Core，當第一個排不進去(Utilization > 1)時換第二個 Core 以此類推。

Best Fit: 在 Utilization < 1 的情況下優先考慮 Utilization 最高的 Core。

Worst Fit: 在 Utilization < 1 的情況下讓所有的 Utilization 盡可能的低，優先排任務到 Utilization 最低的 Core。

~~~~~~ system.cpp ~~~~~~

[First-Fit]

```
164  void
165  System::partitionFirstFit ()
166  {
167      std::cout << "\n==========Partition First-Fit Multi Thread Matrix Multiplication==========" << std::endl;
168  #if (PART == 2)
169      check->setCheckState(PARTITION_FF);
170  #endif
171      threadSet -> setcurrent_PID(-1);
172      threadSet -> restart_fn(0);
173      threadSet -> totoalThread_fn(numThread);
174      /*~~~~~~~~~~~~~~Your code(PART2)~~~~~~~~~~~~~*/
175      // Implement parititon first-fit and print result.
176      for (int i = 0; i < CORE_NUM; i++)
177          cpuSet[i].emptyCPU();
178      int usedList[numThread] = {0};
179      for (int i = 0; i < CORE_NUM; i++){
180          for (int j = 0; j < numThread; j++){
181              if (usedList[j] == 1)
182                  continue;
183              else if ((cpuSet[i].utilization() + threadSet[j].utilization()) <= 1){
184                  cpuSet[i].push_thread(threadSet[j].ID(), threadSet[j].utilization());
185                  threadSet[j].setCore(i);
186                  usedList[j] = 1;
187              }
188          }
189      }
190      for (int i = 0; i < numThread; i++){
191          if (usedList[i] == 0)
192              std::cout << "Thread-" << threadSet[i].ID() << " not schedulable." << std::endl;
193         }
194      for (int i = 0; i < CORE_NUM; i++){
195          std::cout << "CPU" << i << " : ";
196          cpuSet[i].printCPUInformation();
197      }
198      /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
199
200      partitionMultiCoreConv();
201      cleanMultiResult();
202  }
```

首先會先清空所有 CPU 避免上一個排程殘留的資料影響結果，之後我會設一個大小為 numThread 的 usedList 零矩陣用來存放使用過的 Thread，這個方法類似 One-hot encoder 矩陣內的元素為 0 代表該位置的 Thread 沒被使用過，為 1 代表該位置的 Thread 被使用過。

接著透過兩個迴圈將 Thread 排進 Core，外部迴圈是 Core，內部迴圈是 Thread，當 Core 的 Utilization 加上 Thread 的 Utilization 可以小於 1 時代表是放得進去的，就將 Thread 放進 Core 裡面，使用的方法是 push_thread 這個涵式與 setCore，完成後再將 usedList 裡對應 Thread 位置的 index 設為 1 用來代表該 Thread 已經成功排進去，下一個 Core 要排程時可以忽略這個 Thread，另外因為迴圈有順序關係，所以符合 First Fit 的要求。

接下來再找出 usedList 當中為 0 的位置，該位置代表排不進去的 Thread 將他印出符合 Result 要求。最後印出個別 Core 的排程與 Utilization。
~~~~~~

[Best-Fit]

```
204  void
205  System::partitionBestFit ()
206  {
207      std::cout << "\n==========Partition Best-Fit Multi Thread Matrix Multiplication==========" << std::endl;
208  #if (PART == 2)
209      check->setCheckState(PARTITION_BF);
210  #endif
211      threadSet -> setcurrent_PID(-1);
212      threadSet -> restart_fn(0);
213      /*~~~~~~~~~~~~~~Your code(PART2)~~~~~~~~~~~~~~*/
214      // Implement partition best-fit and print result.
215      float utilList[CORE_NUM] = {0};
216      int usedList[numThread] = {0};
217      int maxCpu = 0;
218      float maxValeue = 0;
219      for (int i = 0; i < CORE_NUM; i++)
220          cpuSet[i].emptyCPU();
221      for (int i = 0; i < numThread; i++){
222          for (int j = 0; j < CORE_NUM; j++){
223              utilList[j] = cpuSet[j].utilization() + threadSet[i].utilization();
224          }
225          // std::cout << "utilList = [" << cpuSet[0].utilization() << ", " << cpuSet[1].utilization() << ", " <<
226
227          for (int j = 0; j < CORE_NUM; j++){
228              if (utilList[j] <= 1){
229                  if (utilList[j] > maxValeue){
230                      maxValeue = utilList[j];
231                      maxCpu = j;
232                  }
233              }
234          }
235          if ((cpuSet[maxCpu].utilization() + threadSet[i].utilization()) <= 1){
236              cpuSet[maxCpu].push_thread(threadSet[i].ID(), threadSet[i].utilization());
237              threadSet[i].setCore(maxCpu);
238              usedList[i] = 1;
239          }
240          maxValeue = 0;
241      }
242      for (int i = 0; i < numThread; i++){
243          if (usedList[i] == 0)
244              std::cout << "Thread-" << threadSet[i].ID() << " not schedulable." << std::endl;
245        }
```

```
246      for (int i = 0; i < CORE_NUM; i++){
247          std::cout << "CPU" << i << " : ";
248          cpuSet[i].printCPUInformation();
249      }
250      /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
251
252      partitionMultiCoreConv();
253      cleanMultiResult();
254  }
```

首先清空所有 CPU 避免上一個排程的值影響結果，另外我用矩陣 utilList 存放每個 Core 的 Utilization 加上當前 Thread 的 Utilization。接著用迴圈搭配條件涵式 if 的方式找出 utilList 中小於 1 的最大值，代表 utilList 那個位置的 Core 符合最大 Utilization 且不超過 1，接著將找到的最大 Core 也就是變數 maxCpu 傳給將 Thread 輸入 Core 的涵式 push_thread 與 setCore，最後重置 maxCpu 避免這顆 Core 塞不下然後別顆 Core 的 Utilization 比這顆 Core 低導致找不到的情況，接著印出的方法與 First Fit 相同。

[Worst-Fit]

```
256  void
257  System::partitionWorstFit ()
258  {
259      std::cout << "\n==========Partition Worst-Fit Multi Thread Matrix Multiplication==========" << std::endl;
260  #if (PART == 2)
261      check->setCheckState(PARTITION_WF);
262  #endif
263      threadSet -> setcurrent_PID(-1);
264      threadSet -> restart_fn(0);
265      /*~~~~~~~~~~~~~~Your code(PART2)~~~~~~~~~~~~~~*/
266      // Implement partition worst-fit and print result.
267      float utilList[CORE_NUM] = {0};
268      int usedList[numThread] = {0};
269      int maxCpu = 0;
270      float maxValeue = 1;
271      for (int i = 0; i < CORE_NUM; i++)
272          cpuSet[i].emptyCPU();
273      for (int i = 0; i < numThread; i++){
274          for (int j = 0; j < CORE_NUM; j++){
275              utilList[j] = cpuSet[j].utilization() + threadSet[i].utilization();
276          }
277          // std::cout << "utilList = [" << cpuSet[0].utilization() << ", " << cpuSet[1].utilization() << ", " <<
278
279          for (int j = 0; j < CORE_NUM; j++){
280              if (utilList[j] <= 1){
281                  if (utilList[j] < maxValeue){
282                      maxValeue = utilList[j];
283                      maxCpu = j;
284                  }
285              }
286          }
287          if ((cpuSet[maxCpu].utilization() + threadSet[i].utilization()) <= 1){
288              cpuSet[maxCpu].push_thread(threadSet[i].ID(), threadSet[i].utilization());
289              threadSet[i].setCore(maxCpu);
290              usedList[i] = 1;
291          }
292          maxValeue = 1;
293      }
294      for (int i = 0; i < numThread; i++){
295          if (usedList[i] == 0)
296              std::cout << "Thread-" << threadSet[i].ID() << " not schedulable." << std::endl;
297        }
```

```
298      for (int i = 0; i < CORE_NUM; i++){
299          std::cout << "CPU" << i << " : ";
300          cpuSet[i].printCPUInformation();
301      }
302      /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
303
304      partitionMultiCoreConv();
305      cleanMultiResult();
306  }
```

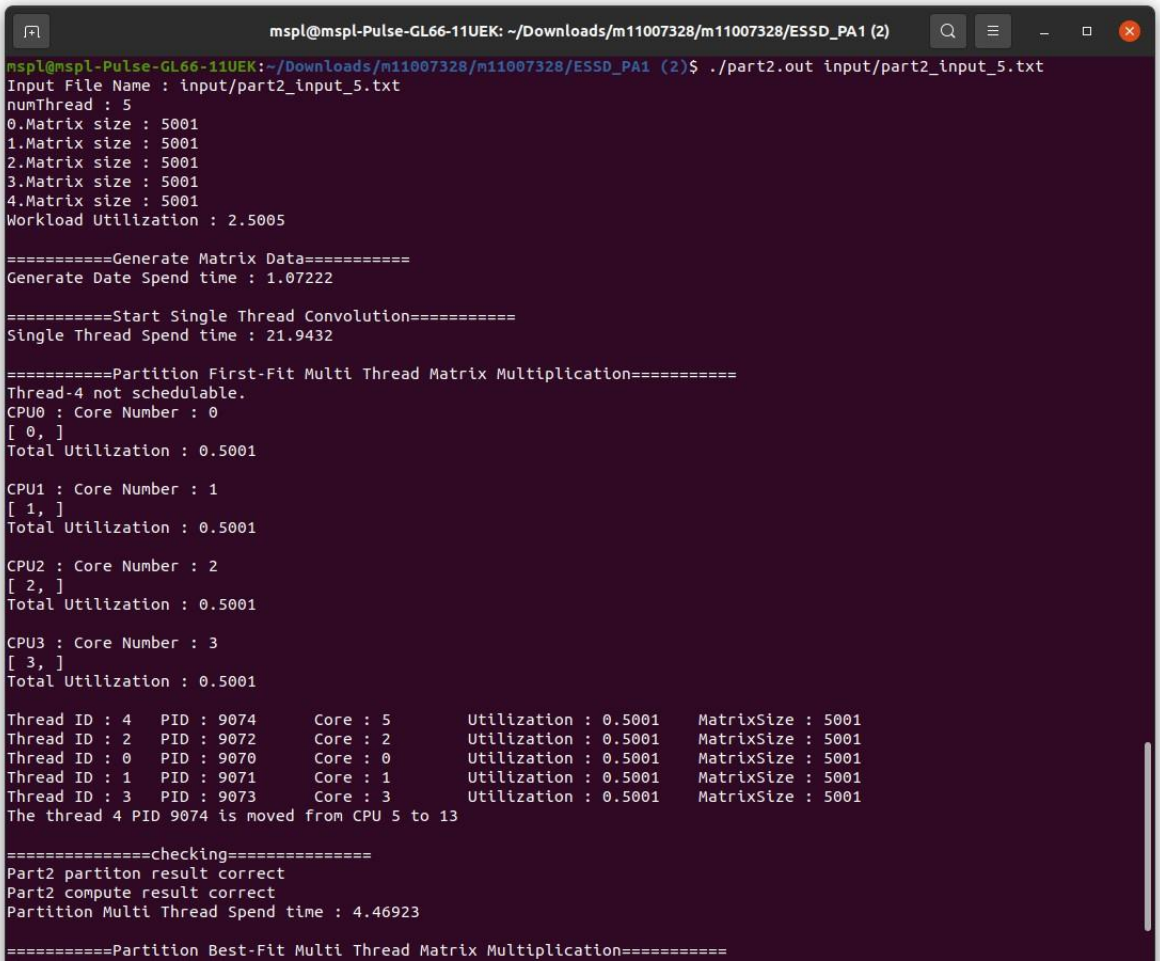
方法部分與 Best Fit 相同，但是改變了 270, 292 行的 maxValue 為 1 這樣做可以讓程式變成找出最小 Utilization 的 Core，由於 Best Fit 與 Worst Fit 我將 Thread 的迴圈放在最外面，所以每一個 Thread 都可以判別當前 Core 的 Utilization。

[Result. 10%]

▪ Show the scheduling states of tasks. (You have to show the screenshot result of using input part2_input_5.txt and part2_input_10.txt)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ **part2_input_5.txt** ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[First Fit]


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[Best Fit]

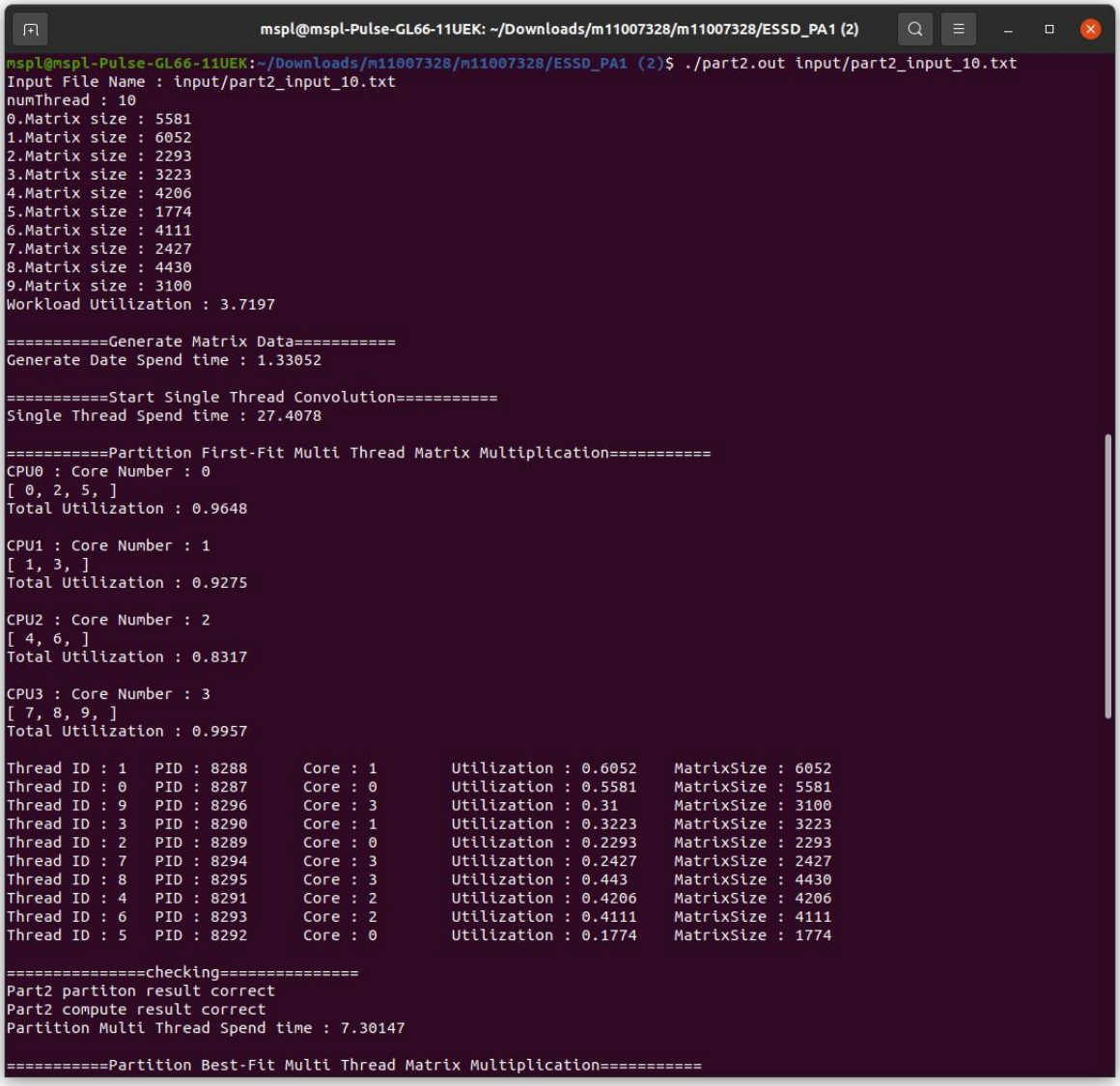
```
===========Partition Best-Fit Multi Thread Matrix Multiplication===========
Thread-4 not schedulable.
CPU0 : Core Number : 0
[ 0, ]
Total Utilization : 0.5001

CPU1 : Core Number : 1
[ 1, ]
Total Utilization : 0.5001

CPU2 : Core Number : 2
[ 2, ]
Total Utilization : 0.5001

CPU3 : Core Number : 3
[ 3, ]
Total Utilization : 0.5001

Thread ID : 0   PID : 9075      Core : 0        Utilization : 0.5001    MatrixSize : 5001
Thread ID : 1   PID : 9076      Core : 1        Utilization : 0.5001    MatrixSize : 5001
Thread ID : 3   PID : 9078      Core : 3        Utilization : 0.5001    MatrixSize : 5001
Thread ID : 4   PID : 9079      Core : 4        Utilization : 0.5001    MatrixSize : 5001
Thread ID : 2   PID : 9077      Core : 2        Utilization : 0.5001    MatrixSize : 5001

===============checking===============
Part2 partiton result correct
Part2 compute result correct
Partition Multi Thread Spend time : 4.39996
```

[Worst Fit]

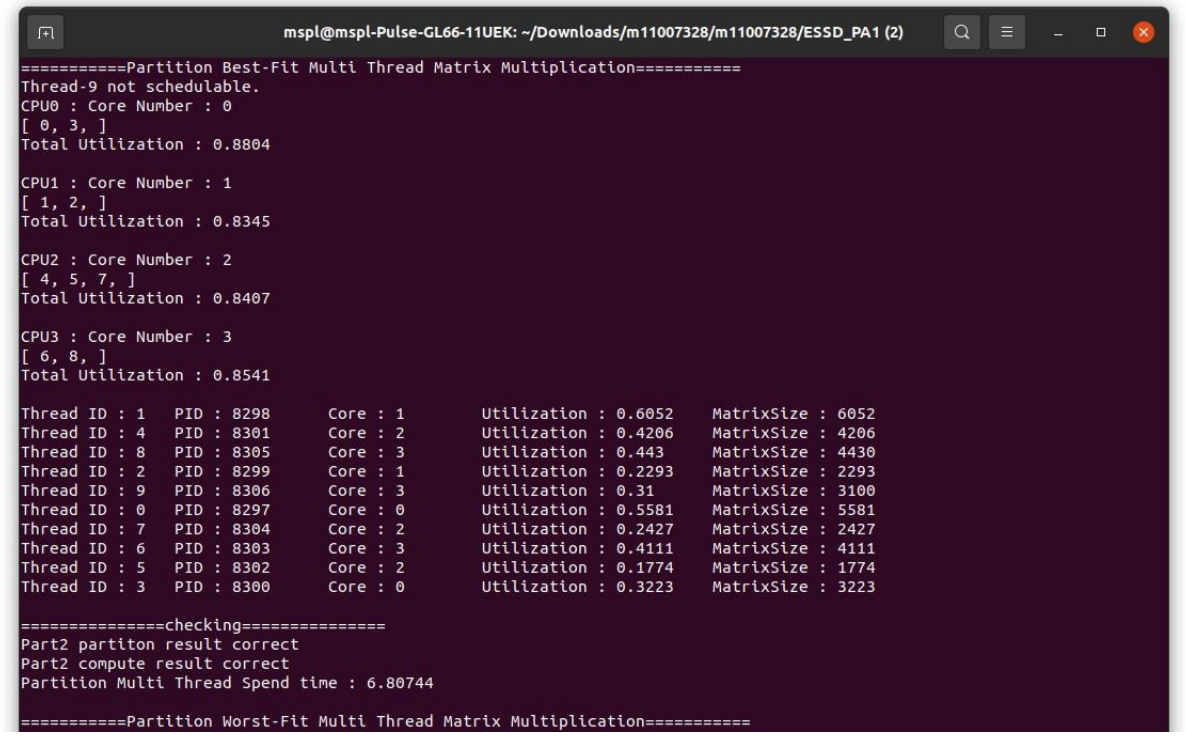
```
===========Partition Worst-Fit Multi Thread Matrix Multiplication===========
Thread-4 not schedulable.
CPU0 : Core Number : 0
[ 0, ]
Total Utilization : 0.5001

CPU1 : Core Number : 1
[ 1, ]
Total Utilization : 0.5001

CPU2 : Core Number : 2
[ 2, ]
Total Utilization : 0.5001

CPU3 : Core Number : 3
[ 3, ]
Total Utilization : 0.5001

Thread ID : 0   PID : 9080      Core : 0        Utilization : 0.5001    MatrixSize : 5001
Thread ID : 1   PID : 9081      Core : 1        Utilization : 0.5001    MatrixSize : 5001
Thread ID : 4   PID : 9084      Core : 5        Utilization : 0.5001    MatrixSize : 5001
Thread ID : 2   PID : 9082      Core : 2        Utilization : 0.5001    MatrixSize : 5001
Thread ID : 3   PID : 9083      Core : 3        Utilization : 0.5001    MatrixSize : 5001
The thread 4 PID 9084 is moved from CPU 5 to 13
The thread 4 PID 9084 is moved from CPU 13 to 5

===============checking===============
Part2 partiton result correct
Part2 compute result correct
Partition Multi Thread Spend time : 4.47731
mspl@mspl-Pulse-GL66-11UEK:~/Downloads/m11007328/m11007328/ESSD_PA1 (2)$ ./part2.out input/part2_input_5.txt
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ **part2_input_10.txt** ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

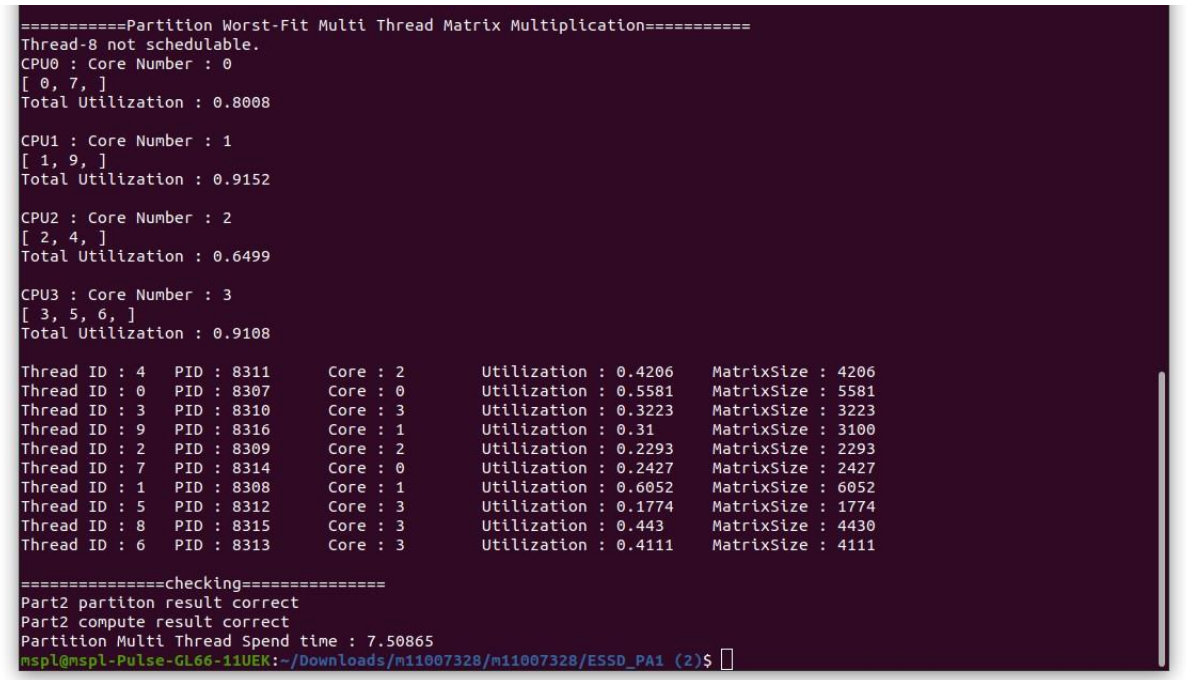
[First Fit]


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[Best Fit]



[Worst Fit]



Part 3:

[Scheduler Implementation. 10%]

▪ Describe how to implement the scheduler setting in partition scheduling. (FIFO with FF, RR with FF)

~~~~~~ **system.cpp** ~~~~~~

```
37  #if (PART == 3)
38          /*~~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~~~*/
39          // Set the scheduling policy for thread.
40          for (int i = 0; i < numThread; i++)
41              threadSet[i].setSchedulingPolicy(SCHEDULING);
42              // std::cout << threadSet[i].schedulingPolicy() << std::endl;
43          /*~~~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~~~*/
44  #endif
```

~~~~~~ **thread.cpp** ~~~~~~

```
183 void
184 Thread::setSchedulingPolicy (int schedulingPolicy)
185 {
186   _schedulingPolicy = schedulingPolicy;
187 }
```

在 **system.cpp** 裡面將 SCHEDULING 這個變數傳給每一個 ThreadSet，SCHEDULING 會隨著第三題 Make 出來的結果改變數值，預設 0 為 SCHED_OTHER，1 為 SCHED_FIFO，2 為 SCHED_RR。

透過 setSchedulingPolicy 去設定_schedulingPolicy 這個私有變數。

```
129 void
130 Thread::setUpScheduler()
131 {
132   /*~~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~~~*/
133   // Set up the scheduler for current thread
134   struct sched_param param;
135   param.sched_priority = sched_get_priority_max(schedulingPolicy());
136   sched_setscheduler(0, schedulingPolicy(), &param);
137   /*~~~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~~~*/
138 }
```

接著在進行 convolution 之前設定排序方法給 Thread，並以 schedulingPolicy()呼叫 SCHEDULING 來確定方法。

```
#if (PART == 3)
    /*~~~~~~~~~~~~~~Your code(PART3)~~~~~~~~~~~~~~*/
    /* Obaserve the execute thread on core-0 */
    // std::cout << "totoalThread: " << totoalThread << std::endl;
    pthread_mutex_lock( &count_Mutex );
    if (total != totoalThread-1){
      total++;
      pthread_cond_wait(&count_Total, &count_Mutex);
    }
    else{

      pthread_cond_broadcast(&count_Total);
    }
    pthread_mutex_unlock( &count_Mutex );

    if (obj->cur_core == 0 && current_PID == -1){
      current_PID = obj->PID;
      std::cout << "Core" << obj->cur_core << " start PID-" << current_PID << std::endl;
    }
    else if (obj->cur_core == 0 && current_PID != -1 && current_PID != obj->PID){
      std::cout << "Core" << obj->cur_core << " context switch from PID-" << current_PID << " to PID-" << obj->PID << std::endl;
      current_PID = obj->PID;
    }
    /*~~~~~~~~~~~~~~~~~~~~END~~~~~~~~~~~~~~~~~~~~*/
#endif
```

顯示的部分透過 pthread_mutex_lock、pthread_cond_wait 與 pthread_conda_broadcast 三種 mutex 的方法達到顯示時可以不會穿插顯式的情況。totalThread 是我自己設的變數，用來呼叫當前排程使用到的 Thread 數目，當 pthread_cond_wait 等到所有的 Thread 時再透過 pthread_conda_broadcast 將所有的鎖解開。

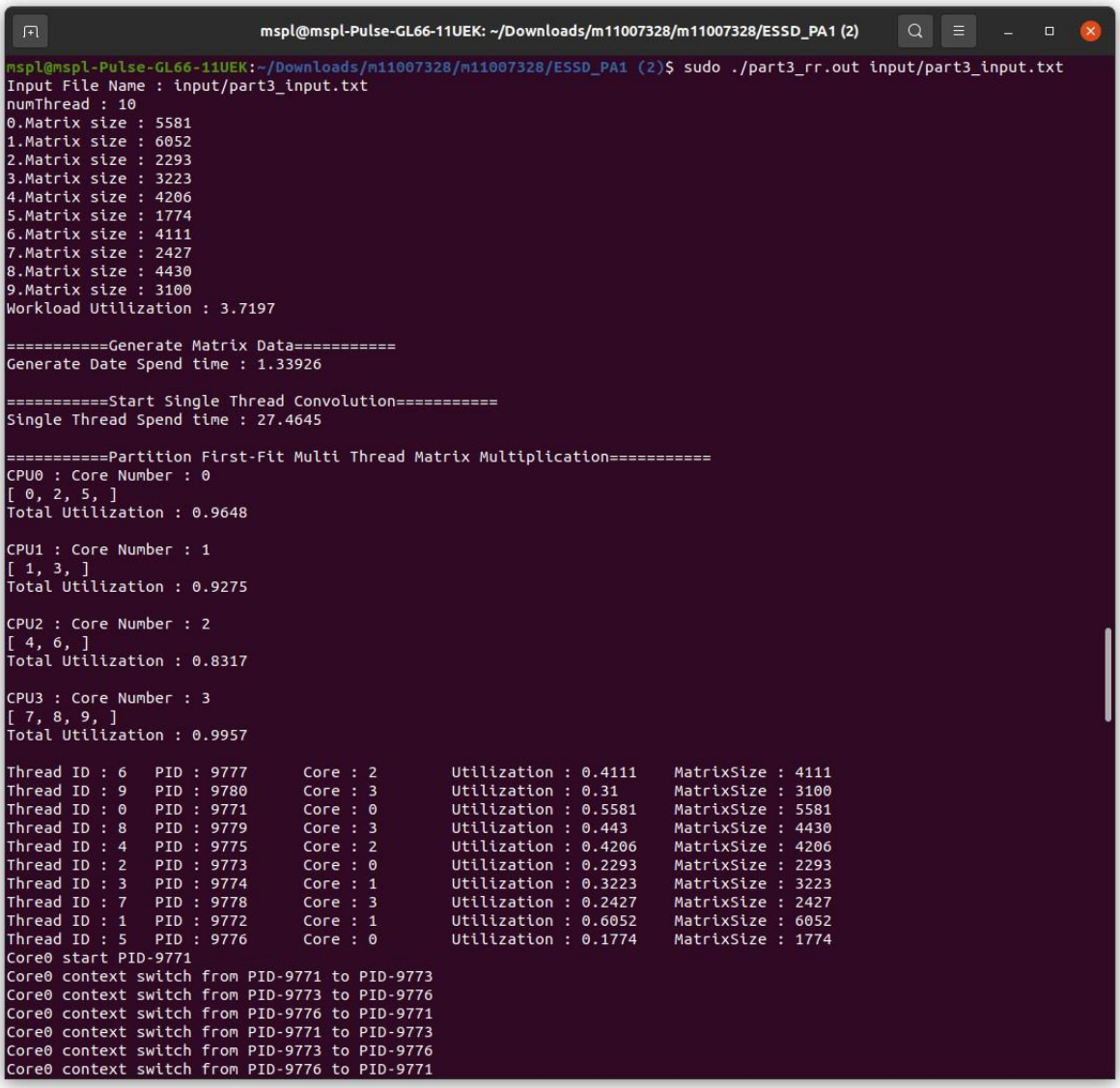
當前 PID 有改變時，將 current_PID 改成新的 PID，並且在各別 Fit 方法的一開始重設這個變數為-1 使顯示不會延續上一個排程的 PID。

[Result. 10%]

▪ Show the process execution states of tasks. (You have to show the screenshot result of using input part3_input.txt

~~~~~~ **RR** ~~~~~~

[First Fit]


~~~~~~

mspl@mspl-Pulse-GL66-11UEK: ~/Downloads/m11007328/m11007328/ESSD_PA1 (2)

```
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9776
Core0 context switch from PID-9776 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9771
Core0 context switch from PID-9771 to PID-9773
Core0 context switch from PID-9773 to PID-9771

===============checking===============
Part3 change scheduler correct
Part3 compute result correct
Partition Multi Thread Spend time : 12.3945

===========Partition Best-Fit Multi Thread Matrix Multiplication===========
```

[Best Fit]

```
mspl@mspl-Pulse-GL66-11UEK: ~/Downloads/m11007328/m11007328/ESSD_PA1 (2)
===========Partition Best-Fit Multi Thread Matrix Multiplication===========
Thread-9 not schedulable.
CPU0 : Core Number : 0
[ 0, 3, ]
Total Utilization : 0.8804

CPU1 : Core Number : 1
[ 1, 2, ]
Total Utilization : 0.8345

CPU2 : Core Number : 2
[ 4, 5, 7, ]
Total Utilization : 0.8407

CPU3 : Core Number : 3
[ 6, 8, ]
Total Utilization : 0.8541

Thread ID : 6   PID : 9787      Core : 3        Utilization : 0.4111    MatrixSize : 4111
Thread ID : 2   PID : 9783      Core : 1        Utilization : 0.2293    MatrixSize : 2293
Thread ID : 5   PID : 9786      Core : 2        Utilization : 0.1774    MatrixSize : 1774
Thread ID : 1   PID : 9782      Core : 1        Utilization : 0.6052    MatrixSize : 6052
Thread ID : 3   PID : 9784      Core : 0        Utilization : 0.3223    MatrixSize : 3223
Thread ID : 9   PID : 9790      Core : 3        Utilization : 0.31      MatrixSize : 3100
Thread ID : 0   PID : 9781      Core : 0        Utilization : 0.5581    MatrixSize : 5581
Thread ID : 4   PID : 9785      Core : 2        Utilization : 0.4206    MatrixSize : 4206
Thread ID : 8   PID : 9789      Core : 3        Utilization : 0.443     MatrixSize : 4430
Thread ID : 7   PID : 9788      Core : 2        Utilization : 0.2427    MatrixSize : 2427
Core0 start PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
```

```
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781
Core0 context switch from PID-9781 to PID-9784
Core0 context switch from PID-9784 to PID-9781

===============checking===============
Part3 change scheduler correct
Part3 compute result correct
Partition Multi Thread Spend time : 11.7587

===========Partition Worst-Fit Multi Thread Matrix Multiplication===========
```

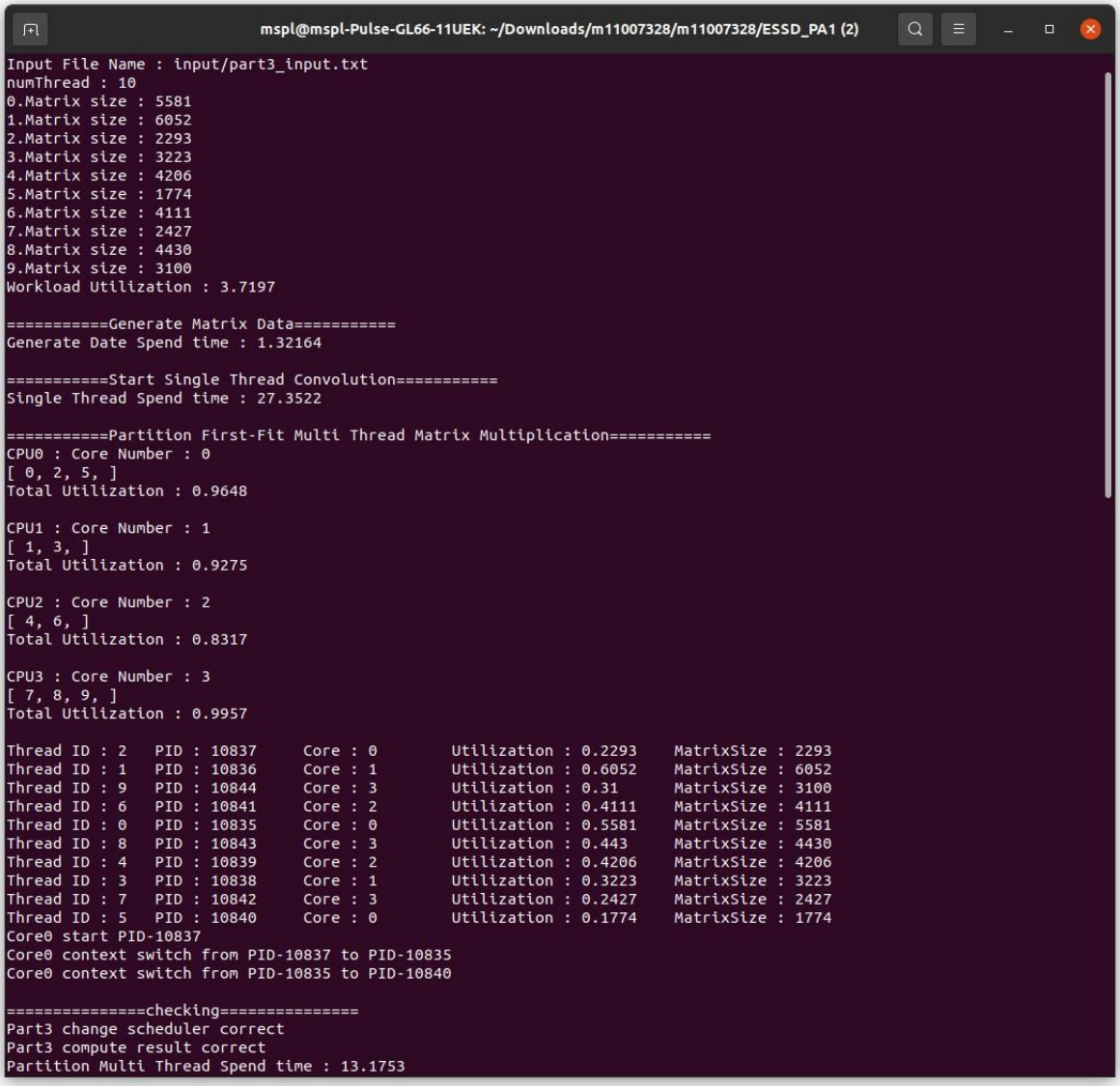
[Worst Fit]

```
mspl@mspl-Pulse-GL66-11UEK: ~/Downloads/m11007328/m11007328/ESSD_PA1 (2)
===========Partition Worst-Fit Multi Thread Matrix Multiplication===========
Thread-8 not schedulable.
CPU0 : Core Number : 0
[ 0, 7, ]
Total Utilization : 0.8008

CPU1 : Core Number : 1
[ 1, 9, ]
Total Utilization : 0.9152

CPU2 : Core Number : 2
[ 2, 4, ]
Total Utilization : 0.6499

CPU3 : Core Number : 3
[ 3, 5, 6, ]
Total Utilization : 0.9108

Thread ID : 0   PID : 9791     Core : 0       Utilization : 0.5581    MatrixSize : 5581
Thread ID : 9   PID : 9800     Core : 1       Utilization : 0.31      MatrixSize : 3100
Thread ID : 1   PID : 9792     Core : 1       Utilization : 0.6052    MatrixSize : 6052
Thread ID : 4   PID : 9795     Core : 2       Utilization : 0.4206    MatrixSize : 4206
Thread ID : 3   PID : 9794     Core : 3       Utilization : 0.3223    MatrixSize : 3223
Thread ID : 7   PID : 9798     Core : 0       Utilization : 0.2427    MatrixSize : 2427
Thread ID : 6   PID : 9797     Core : 3       Utilization : 0.4111    MatrixSize : 4111
Thread ID : 8   PID : 9799     Core : 3       Utilization : 0.443     MatrixSize : 4430
Thread ID : 5   PID : 9796     Core : 3       Utilization : 0.1774    MatrixSize : 1774
Thread ID : 2   PID : 9793     Core : 2       Utilization : 0.2293    MatrixSize : 2293
Core0 start PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791
Core0 context switch from PID-9791 to PID-9798
Core0 context switch from PID-9798 to PID-9791

===============checking==============
```

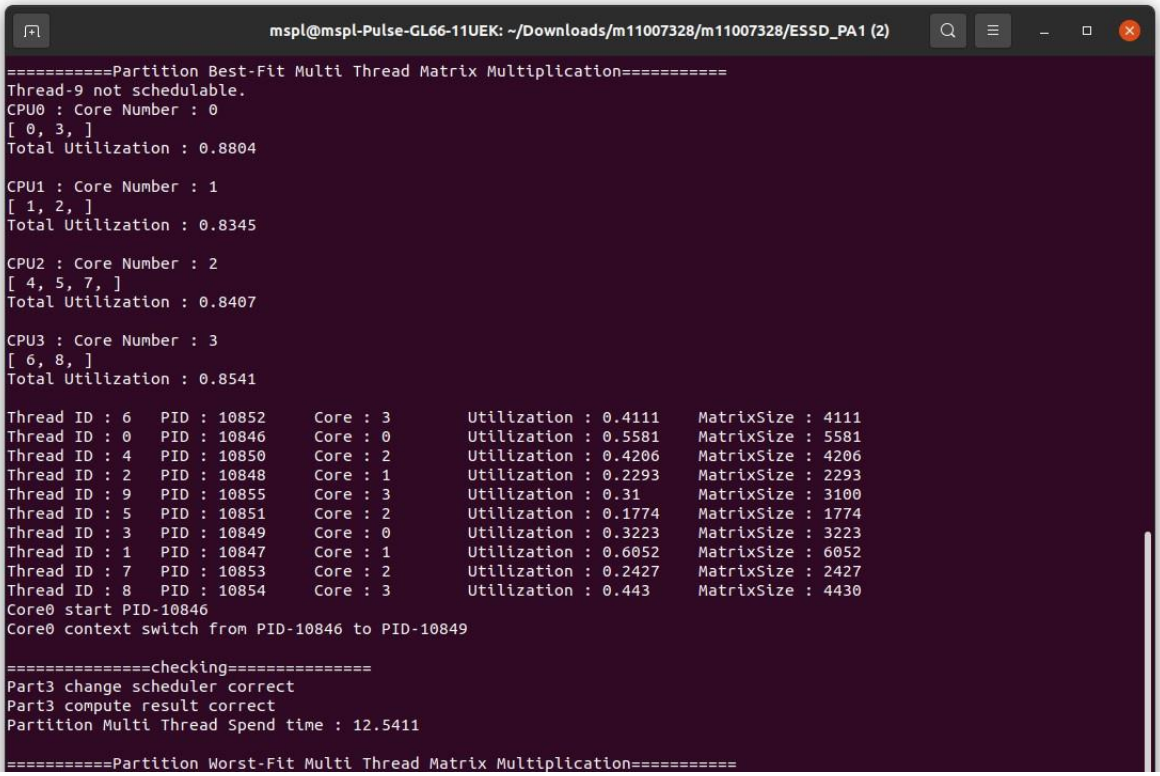
```
===============checking==============
Part3 change scheduler correct
Part3 compute result correct
Partition Multi Thread Spend time : 13.7099
mspl@mspl-Pulse-GL66-11UEK:~/Downloads/m11007328/m11007328/ESSD_PA1 (2)$
```

~~~~~~ **FIFO** ~~~~~~

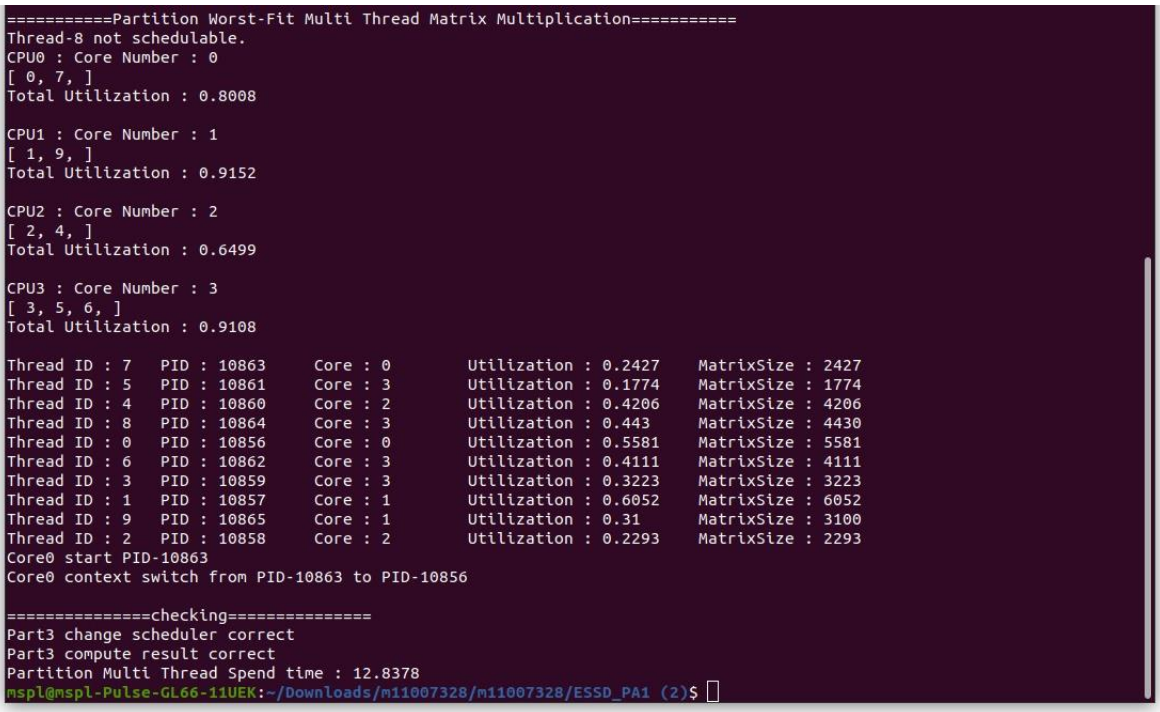
[First Fit]


~~~~~~

[Best Fit]



[Worst Fit]



Discussion:

▪ Analyze and compare the response time of the program, with single thread and multi-thread using in part 1 and part 2. (Including Single, Global, First-Fit, Best-Fit, Worst-Fit) 5%

Part 1			
	Single	Global	Partition
response time	17.2365	6.28474	6.10764

經過實驗得知，在 Part1 中的 Global 與 Partition 皆能產生比 Single 快速的執行速度，而當中 Partition 運算速度又會再比 Global 快一點，我想可能的原因是因為 Partition 省去了找空閒 Core 的步驟，在一開始就將哪些 Core 要執行哪些任務給分配好，而且切分資料上較為平均因此才能有較佳效能。

Part 2				
	Single	First-Fit	Best-Fit	Worst-Fit
response time	27.4078	7.30147	6.80744	7.50865

(資料選用: **part2_input_10.txt)**

透過 Part2 可以得知三種排程方法皆能有效提升運算效能，相比於 Single 皆能提升近 3 倍的速率，而當中效能最好的是 Best Fit 他主要是優先選用 Utilization 高的 Core 去安排任務。

Part2 同時也可以觀察到各種排程方法的差異，舉例像是 First-Fit 在這個資料中所有 Thread 皆能排進去，Best Fit 在編號 9 的 Thread 無法排進去 (各別 Core 的 Utilization 皆 > 1)，Worst Fit 在編號 8 的 Thread 無法排進去。由此可知排程方法不同，可能會導致原本可以排的 Thread 變成無法安排。

▪ Analyze and compare the characteristic of the three different partition methods (First-Fit, Best-Fit, Worst-Fit) 5%

First-Fit: 特色是優先排滿第一個 Core，當剩餘的 Thread 都無法排進當前的 Core 時才考慮放到其他的 Core 上，優點是 Overhead 低與功耗低。

Best-Fit: 特色是優先排滿 Utilization 最高的 Core，當 Utilization 最高的 Core 排不進去時會去試著放入第二高 Utilization 的 Core，以此類推。較不公平，因為資料分配會不均勻，高的高低得很低，但透過實驗結果卻發現 Worst-Fit 的排序方法排出來的結果比 Best-Fit 還不平均，因此推斷可能是排序方法可能有特例情況。

Worst-Fit: 特色是優先排入 Utilization 最低的 Core，這樣的排法通常資料分布較平均，因為有發現最低的就放入資料補齊，容易調頻降頻因此功耗較低。

▪ Analyze and compare the response time of the program, with two different schedulers. (FIFO with FF, RR with FF) 5%

Part 3		
response time	FIFO with FF	RR with FF
First-Fit	13.1753	12.3945
Best-Fit	12.5411	11.7587
Worst Fit	12.8378	13.7099

FIFO with FF: 特色為先進入的先執行，當執行完成後才會給其他人做，在 Worst Fit 的時候有較佳的表現，但整體來看與 RR 方法的結果差不多。

RR with FF: 特色為給每個任務分配的時間都一樣，當時間一到就換下一個任務執行，當任務提早結束就提早釋出 Core 給其他任務執行，在 First-Fit 與 Best-Fit 的時候皆取得較佳的表現。