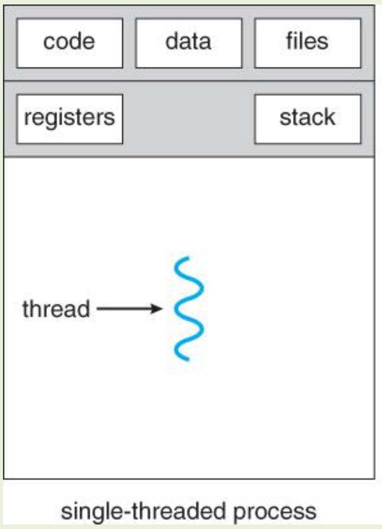
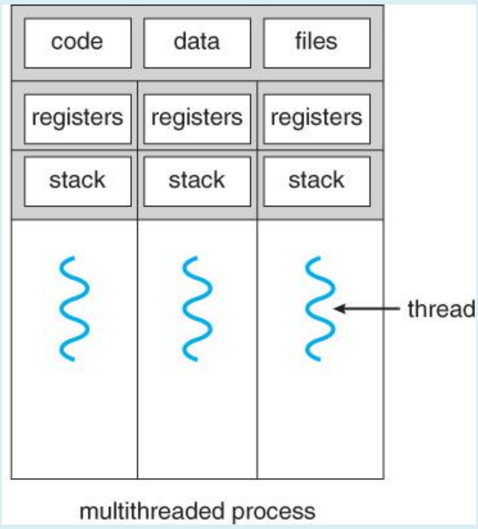
National Taipei University of Technology (NTUT)

Operating System (OS) 109-2 Homework II

FOR-LOOP	PARALLEL SKILL	
		OpenMP Microsoft Parallel Computing Platform Parallel Patterns Library(PPL) Windows API Win32 API

Class	電子四丙
Student ID	106360701
Student Name	游煒亘
Guiding mentor	段裘慶 教授

1. Coding independent threads can speed up the running of programs. Let us conduct an experiment to experience the features and skill of multithreading. You need to code two different styles of programs to perform a "matrix multiplication" in the same programming language (e.g. C language), as follows.

$$C_{35\times35} = A_{35\times60} \times B_{60\times35}$$

$$\left[a_{ij}\right] = 3.5i - 6.6j,\ \left[b_{ij}\right] = 6.6 + 8.8i - 3.5j$$

- In the first program, you just code it following the traditional for-looping skill
- In the second program, you need trying to code it by using the new multithreading skill

Q1: Point out the major parts in the threaded program to highlight its differences with for-loops.

For-loop

```
void Matrix_mul(array_two arrayA, array_two arrayB, array_two & arrayC) {
    for (int i = 0; i < arrayA.size(); i++)
        for (int j = 0; j < arrayB[0].size(); j++)
            for (int k = 0; k < arrayB.size(); k++)
                arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
}
```

Multithread (PPL)

```
void parallel_matrix_mul_ppl(array_two arrayA, array_two arrayB, array_two & arrayC) {
    parallel_for(size_t(0), arrayA.size(), [&](size_t i) {
        for (int j = 0; j < arrayB[0].size(); j++)
            for (int k = 0; k < arrayB.size(); k++)
                arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
    });
}
```

Multithread (OpenMP)

```cpp
void parallel_matrix_mul_omp(array_two arrayA, array_two arrayB, array_two & arrayC) {
    #pragma omp parallel num_threads(MAX_THREADS)
    {
        #pragma omp for
        for (int i = 0; i < arrayA.size(); i++)
            for (int j = 0; j < arrayB[0].size(); j++)
                for (int k = 0; k < arrayB.size(); k++)
                    arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
    }
}
```

Multithread (Win32 API)

Matrix_mul Function

```cpp
DWORD WINAPI parallel_matrix_mul_WIN32(LPVOID Param) {
  PMYDATA arrays_P = (PMYDATA)Param;
  for (int i = 0; i < arrays_P->arrayA.size(); i++) {
    for (int j = 0; j < arrays_P->arrayB[0].size(); j++) {
      for (int k = 0; k < (arrays_P->arrayB.size() / MAX_THREADS); k++) {
        arrays_P->arrayC[i][j] += \
        arrays_P->arrayA[i][k+(arrays_P->arrayB.size()/MAX_THREADS)*arrays_P->number] *
         arrays_P->arrayB[k+(arrays_P->arrayB.size()/MAX_THREADS)*arrays_P->number][j];
      }
    }
  }
  if ((arrays_P->arrayB.size() % MAX_THREADS) != 0) {
    if ((arrays_P->arrayB.size() % MAX_THREADS) > arrays_P->number) {
      for (size_t i = 0; i < arrays_P->arrayA.size(); i++) {
        for (size_t j = 0; j < arrays_P->arrayB[0].size(); j++) {
          arrays_P->arrayC[i][j] += arrays_P->arrayA[i][arrays_P->number + \
             (arrays_P->arrayB.size() / MAX_THREADS) * MAX_THREADS] *
            arrays_P->arrayB[arrays_P->number + \
            (arrays_P->arrayB.size() / MAX_THREADS) * MAX_THREADS][j];
        }
      }
    }
```

```
  }
  return 0;
}
```

Main Program

```
void parallel_matrix_mul_WIN32_main(array_two arrayA, array_two arrayB, array_two& arrayC) {
  DWORD ThreadId[MAX_THREADS];
  HANDLE ThreadHandle[MAX_THREADS];
  PMYDATA arrays_P[MAX_THREADS];
  /* create the thread */
  for (size_t i = 0; i < MAX_THREADS; i++) {
     // Allocate memory for thread data.
     arrays_P[i]=(PMYDATA)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(MYDATA));
     // If the array allocation fails, the system is out of memory
     // so there is no point in trying to print an error message.
     // Just terminate execution.
    if (arrays_P == NULL)
      ExitProcess(2);
    arrays_P[i]->arrayA = arrayA;
    arrays_P[i]->arrayB = arrayB;
    arrays_P[i]->arrayC = arrayC;
    arrays_P[i]->number = i;
    ThreadHandle[i] = CreateThread(NULL, 0, parallel_matrix_mul_WIN32,
                                   arrays_P[i], 0, &ThreadId[i]);
    }
    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS,ThreadHandle,TRUE ,INFINITE);
    for (size_t i = 0; i < MAX_THREADS; i++) {
        CloseHandle(ThreadHandle[i]);
        if (arrays_P[i] != NULL) {
           HeapFree(GetProcessHeap(), 0, arrays_P[i]);
           arrays_P[i] = NULL;
        }
    }
    return;
}
```

Q2: Record your experimental results at least 3 rounds execution in the below table, and state how you can count the running time of programs in ms.

Matrix size : $A_{1000\times219}$, $B_{219\times1000}$

For Loop Code
(code below)

```
//Metric multiplication
void Matrix_mul(array_two arrayA, array_two arrayB, array_two & arrayC) {
    for (int i = 0; i < arrayA.size(); i++)
        for (int j = 0; j < arrayB[0].size(); j++)
            for (int k = 0; k < arrayB.size(); k++)
                arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
}
```

Win32 API part:

Win32 API					
Coding Skill	Line of Code	Execution Time (s)			Average Execution Time
		1-roumd	2-round	3-round	
【A1】 *For-loops*	6	42.6436s	40.294s	39.2891s	44.9005s
【B1】 by cells ***Multithread*** Numbers : 2	55	24.2085s	24.1016s	23.472s	23.9274s
Differences 【A1-B1】	-49	18.4351s	16.1924s	15.8119s	16.8191s
【A2】 *For-loops*	6	39.1398s	45.4014s	44.9218s	43.1543s
【B2】 by cells ***Multithread*** Numbers : 4	55	13.9212s	15.7591s	15.0389s	14.9064s
Differences 【A2-B2】	-49	25.2186s	29.6423s	29.8829s	28.2479s
【A3】 *For-loops*	6	41.5103s	42.9053s	43.4709s	42.6289s
【B3】 by cells ***Multithread*** Numbers : 32	55	13.7245s	15.1131s	13.2517s	14.0298s
Differences 【A3-B3】	-49	27.7858s	27.7922s	30.2192s	28.5991s

Parallel by Win32 Code (main)

```
void parallel_matrix_mul_WIN32_main(array_two arrayA, array_two arrayB, array_two& arrayC) {
    DWORD ThreadId[MAX_THREADS];
    HANDLE ThreadHandle[MAX_THREADS];
    PMYDATA arrays_P[MAX_THREADS];
    /* create the thread */
    for (size_t i = 0; i < MAX_THREADS; i++) {
        // Allocate memory for thread data.
        arrays_P[i] = (PMYDATA)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(MYDATA));
        // If the array allocation fails, the system is out of memory
        // so there is no point in trying to print an error message.
        // Just terminate execution.
        if (arrays_P == NULL)
            ExitProcess(2);
        arrays_P[i]->arrayA = arrayA;
        arrays_P[i]->arrayB = arrayB;
        arrays_P[i]->arrayC = arrayC;
        arrays_P[i]->number = i;
        ThreadHandle[i] = CreateThread(NULL, 0, parallel_matrix_mul_WIN32,
            arrays_P[i], 0, &ThreadId[i]);
    }
    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS, ThreadHandle, TRUE, INFINITE);
    /*for (size_t i = 0; i < 4; i++)
        arrayC[0][0] += arrays_P[i]->arrayC[0][0];
    printf("%f\n", arrayC[0][0]);*/
    // Close the all thread handles and free memory allocations
    for (size_t i = 0; i < MAX_THREADS; i++) {
        CloseHandle(ThreadHandle[i]);
        if (arrays_P[i] != NULL) {
            HeapFree(GetProcessHeap(), 0, arrays_P[i]);
            arrays_P[i] = NULL;
        }
    }
    return;
}
```

Parallel by Win32 Code (matrix_mul)

```
DWORD WINAPI parallel_matrix_mul_WIN32(LPVOID Param) {
    PMYDATA arrays_P = (PMYDATA)Param;
    for (int i = 0; i < arrays_P->arrayA.size(); i++) {
        for (int j = 0; j < arrays_P->arrayB[0].size(); j++) {
            for (int k = 0; k < (arrays_P->arrayB.size() / MAX_THREADS); k++) {
                arrays_P->arrayC[i][j] += \
                    arrays_P->arrayA[i][k + (arrays_P->arrayB.size() / MAX_THREADS) * arrays_P->number] *
                    arrays_P->arrayB[k + (arrays_P->arrayB.size() / MAX_THREADS) * arrays_P->number][j];
            }
        }
    }
    if ((arrays_P->arrayB.size() % MAX_THREADS) != 0) {
        if ((arrays_P->arrayB.size() % MAX_THREADS) > arrays_P->number) {
            for (size_t i = 0; i < arrays_P->arrayA.size(); i++) {
                for (size_t j = 0; j < arrays_P->arrayB[0].size(); j++) {
                    arrays_P->arrayC[i][j] += arrays_P->arrayA[i][arrays_P->number + \
                        (arrays_P->arrayB.size() / MAX_THREADS) * MAX_THREADS] *
                        arrays_P->arrayB[arrays_P->number + \
                        (arrays_P->arrayB.size() / MAX_THREADS) * MAX_THREADS][j];
                }
            }
        }
    }
    return 0;
}
```

OpenMP part

OpenMP API					
Coding Skill	Line of Code	Execution Time (s)			Average Execution Time
		1-roumd	2-round	3-round	
【A1】 *For-loops*	6	42.6436s	40.294s	39.2891s	44.9005s
【B1】 by cells ***Multithread*** Numbers : 2	10	22.4869s	22.8791s	21.6103s	22.3254s
Differences 【A1-B1】	-4	20.1561s	17.4149s	17.6788	22.5751s
【A2】 *For-loops*	6	39.1398s	45.4014s	44.9218s	43.1543s
【B2】 by cells ***Multithread*** Numbers : 4	10	10.0415s	12.9001s	13.8233s	12.255s
Differences 【A2-B2】	-4	29.0983s	32.5013s	31.0985s	30.8993s
【A3】 *For-loops*	6	41.5103s	42.9053s	43.4709s	42.6289s
【B3】 by cells ***Multithread*** Numbers : 32	10	12.5478s	13.7086s	10.7021s	12.3195s
Differences 【A3-B3】	-4	28.9625s	29.1967s	32.7688s	30.3094s

Parallel by OpenMP code

```
void parallel_matrix_mul_omp(array_two arrayA, array_two arrayB, array_two& arrayC) {
    #pragma omp parallel num_threads(MAX_THREADS)
    {
    #pragma omp for
        for (int i = 0; i < arrayA.size(); i++)
            for (int j = 0; j < arrayB[0].size(); j++)
                for (int k = 0; k < arrayB.size(); k++)
                    arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
    }
}
```

PPL part:

PPL API					
Coding Skill	Line of Code	Execution Time (s)			Average Execution Time
		1-roumd	2-round	3-round	
【A1】 *For-loops*	6	42.6436s	40.294s	39.2891s	44.9005s
【B1】 by cells ***Multithread***	7	11.1173s	10.55s	9.9346s	10.534s
Differences 【A1-B1】	-1	31.5263s	29.744s	29.3545s	34.4665s
【A2】 *For-loops*	6	39.1398s	45.4014s	44.9218s	43.1543s
【B2】 by cells ***Multithread***	7	9.9844s	13.6329s	10.2084s	11.2752s
Differences 【A2-B2】	-1	29.1554s	31.7685s	34.7134s	31.8791s
【A3】 *For-loops*	6	41.5103s	42.9053s	43.4709s	42.6289s
【B3】 by cells ***Multithread***	7	11.1294s	17.0227s	12.6441s	14.0298s
Differences 【A3-B3】	-1	30.3809s	25.8826s	30.8268s	28.5991s

Parallel by PPL code (main)

```cpp
//Parallel by ppl API
void parallel_matrix_mul_ppl(array_two arrayA, array_two arrayB, array_two& arrayC) {
    parallel_for(size_t(0), arrayA.size(), [&](size_t i) {
        for (int j = 0; j < arrayB[0].size(); j++)
            for (int k = 0; k < arrayB.size(); k++)
                arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
        });
}
```

Calculate of Time method

Time calculation method

$$Current\ CPU\ Frequecy = F, Start\ PerformanceCounter\ Number = \ SPCN$$
$$End\ PerformanceCounter\ Number = \ EPCN$$

$$function\ cost\ time = \frac{(EPCN - SPCN)}{F}$$

Instruction

我主要是透過 Win32 API 的 QueryPerformanceFrequency 來查詢目前 CPU 的頻率，在使用 QueryPerformanceCounter 來獲取函數開始前和結束的效能計數器目前計數的數量，並且將兩者做相減，最後除於頻率如上面的公式來獲取函數執行所花費的時間。

I mainly use Win32 API's QueryPerformanceFrequency to query the current CPU frequency, and use QueryPerformanceCounter to get the current count of the performance counter before and after the function starts, subtract the two, and finally divide by the frequency as the above formula. Get the time taken by the function to execute.

Code

```
double test_and_calculate_cost_time(void (*func)(array_two, array_two, array_two&),
    array_two* paramter) {
    LARGE_INTEGER StartingTime, EndingTime;
    LARGE_INTEGER Frequecy;

    QueryPerformanceFrequency(&Frequecy);
    QueryPerformanceCounter(&StartingTime);
    func(paramter[0], paramter[1], paramter[2]);
    QueryPerformanceCounter(&EndingTime);
    return (double)(EndingTime.QuadPart - StartingTime.QuadPart) / (double)Frequecy.QuadPart;
}
```

Q3: State your discovering and commends on this exercise of coding threaded programs

這次主要使用的硬體如下表:

The main hardware used this time is as follows:

CPU	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
Main memory	16.0GB
Core number	4
L1 Cache	256KB
L2 Cache	1.0MB
L3 Cache	6.0MB

這次實驗我主要是使用 3 種並行處理的方式來和傳統 for-loop 方式做比較，透過更改 thread 的數量可以發現執行的速度比 for-loop 會上升很多，但是就如老師上課所說當 thread 數到達一定的數量時，執行速度就會停止上升甚至下降因為 overhead 的問題。

In this experiment, I mainly used three parallel processing methods to compare with the traditional for loop method. By changing the number of threads, you can find that the speed is much higher than the for loop, but as the teacher said, when the number of threads reaches a certain level, the execution speed will stop rising or even falling, due to overhead。

Summary

下表是傳統 for-loop 以及三種並行方式用不同 thread 數的平均時間

For-loop			
Average time	44.9005s	43.1543s	42.6289s
Parallel			
PPL			
Average time	10.534s	11.2752s	14.0298s
	Number of threasds 2	Number of threasds 4	Number of threasds 16
Win32 API			
Average time	23.9274s	14.9064s	14.0298s
OpenMP API			
Average time	22.3254s	12.255s	12.3195s
Difference with for-loop			
PPL	34.4665s	31.8791s	28.5991s
	Number of threasds 2	Number of threasds 4	Number of threasds 16
Win32 API	16.8191s	28.2479s	28.5991s
OpenMP API	22.5751s	30.8993s	30.3094s

Number of threasds : 2

```
(1)[for-loop  ]Cost time :    42.6436s
(2)[for-loop  ]Cost time :     40.294s
(3)[for-loop  ]Cost time :    39.2891s
Average time :     40.7422s

(1)[ppl API    ]Cost time :    11.1173s
(2)[ppl API    ]Cost time :      10.55s
(3)[ppl API    ]Cost time :    9.93458s
Average time :      10.534s

(1)[OpenMP API]Cost time :    22.4869s
(2)[OpenMP API]Cost time :    22.8791s
(3)[OpenMP API]Cost time :    21.6103s
Average time :     22.3254s

(1)[Win32 API ]Cost time :    24.2085s
(2)[Win32 API ]Cost time :    24.1016s
(3)[Win32 API ]Cost time :     23.472s
Average time :     23.9274s
```

Number of threasds : 4

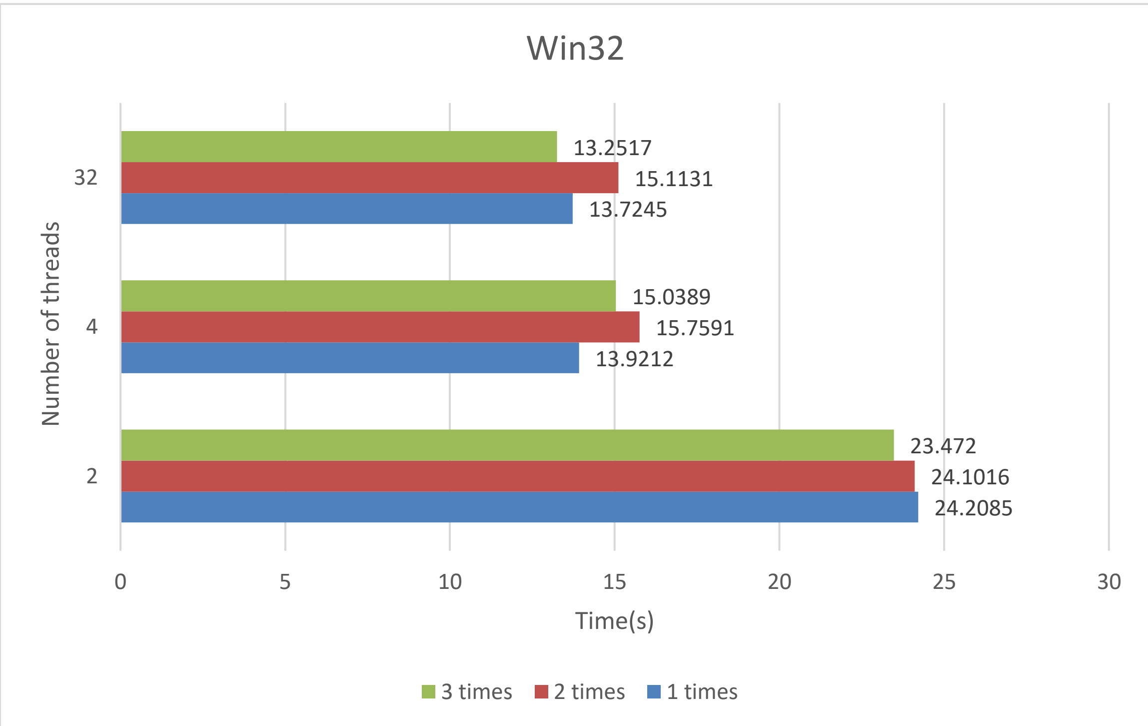
```
(1)[for-loop  ]Cost time :    39.1398s
(2)[for-loop  ]Cost time :    45.4014s
(3)[for-loop  ]Cost time :    44.9218s
Average time :    43.1543s

(1)[ppl API   ]Cost time :    9.98438s
(2)[ppl API   ]Cost time :    13.6329s
(3)[ppl API   ]Cost time :    10.2084s
Average time :    11.2752s

(1)[OpenMP API]Cost time :    10.0415s
(2)[OpenMP API]Cost time :    12.9001s
(3)[OpenMP API]Cost time :    13.8233s
Average time :     12.255s

(1)[Win32 API ]Cost time :    13.9212s
(2)[Win32 API ]Cost time :    15.7591s
(3)[Win32 API ]Cost time :    15.0389s
Average time :    14.9064s
```

Number of threasds : 32

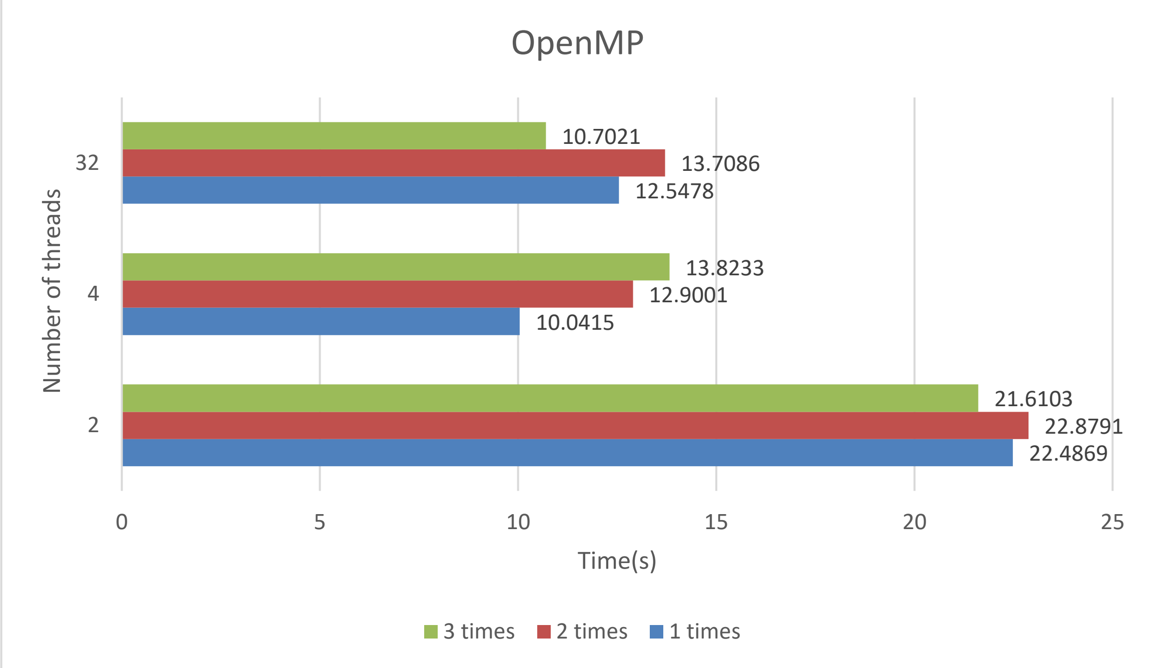
```
Microsoft Visual Studio Debug Console
(1)[for-loop  ]Cost time :    41.5103s
(2)[for-loop  ]Cost time :    42.9053s
(3)[for-loop  ]Cost time :    43.4709s
Average time :    42.6289s

(1)[ppl API   ]Cost time :    11.1294s
(2)[ppl API   ]Cost time :    17.0227s
(3)[ppl API   ]Cost time :    12.6441s
Average time :    13.5987s

(1)[OpenMP API]Cost time :    12.5478s
(2)[OpenMP API]Cost time :    13.7086s
(3)[OpenMP API]Cost time :    10.7021s
Average time :    12.3195s

(1)[Win32 API ]Cost time :    13.7245s
(2)[Win32 API ]Cost time :    15.1131s
(3)[Win32 API ]Cost time :    13.2517s
Average time :    14.0298s
```

Trend

Win32



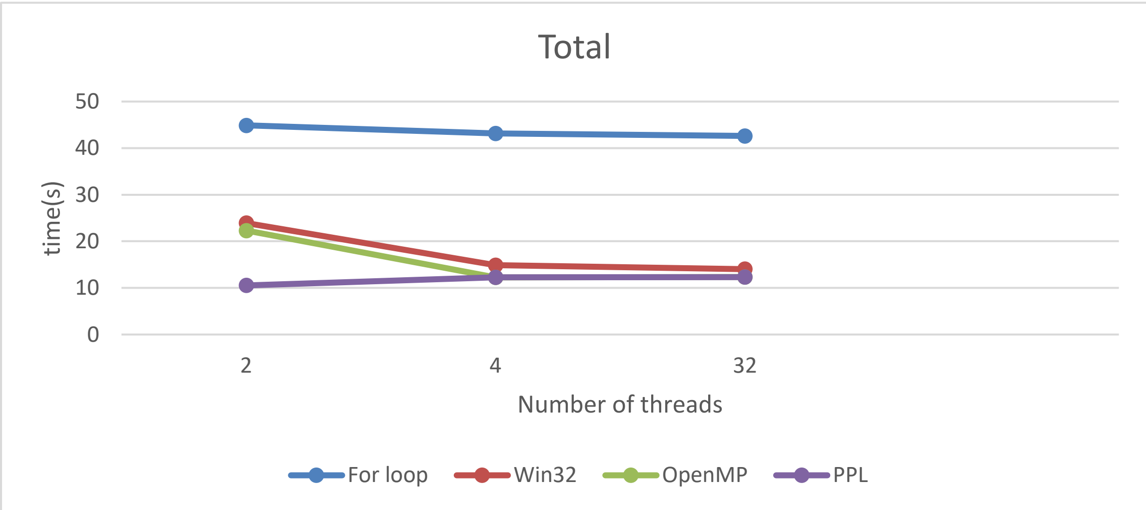
OpenMP

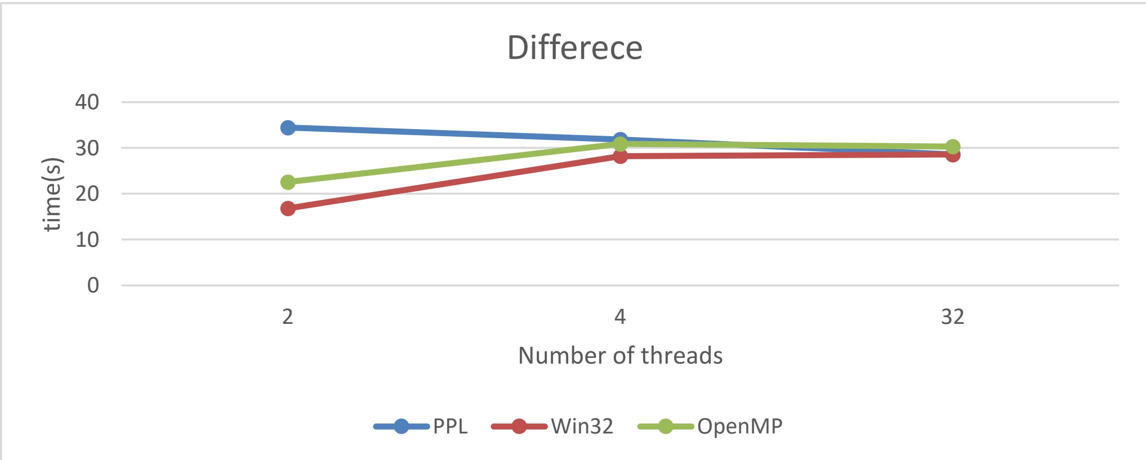


Total

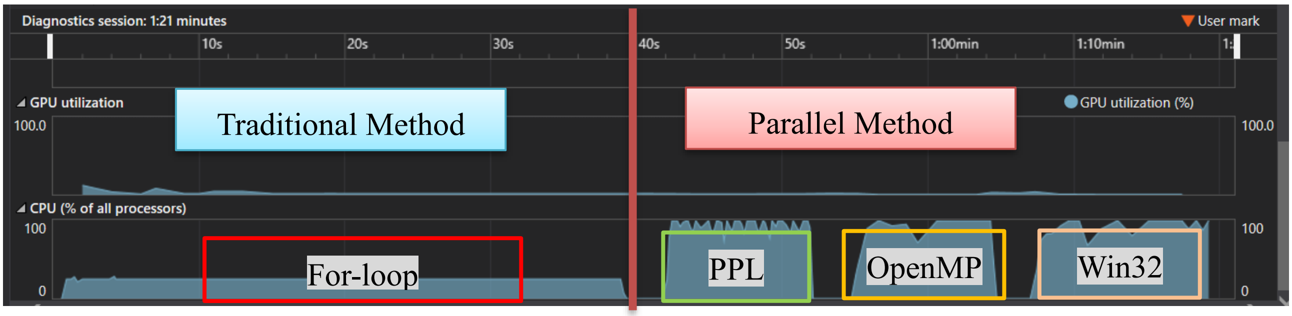
PPL 和 For-loop 的 x 軸不是 threads number 是執行回合數(1, 2, 3)

The x-axis of PPL and For-loop is not the number of threads, but the number of executions





CPU Usage



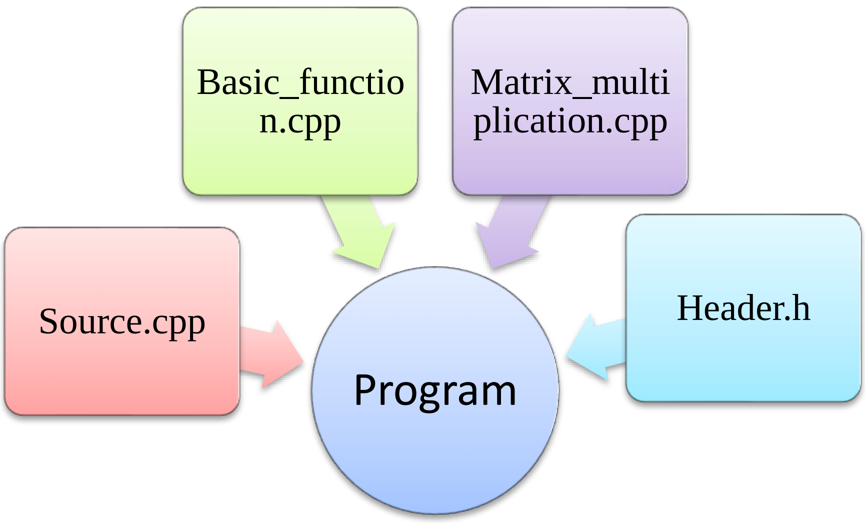
我使用 Visual Studio 的 Performance tools 來獲取 CPU 和 GPU 的使用量，透過上圖可以觀察到傳統方式 For-loop(single thread)的 CPU 使用量相較於其他三種並行處理的方式來說非常低。

I use the performance tools of Visual Studio to get the usage of the traditional for-loop(single thread) method can be observed to be very low compared to the other three parallel processing methods.

心得：

透過這次實驗，我了解 Multi Thread 的重要性，如果不了解其就沒辦法更有效的使用硬體資源進一步導致浪費時間以及讓大部分硬體資源空閒沒事。但是使用 Multi Thread 並不會因為你使用比原本多兩倍的 thread 而加快兩倍，以及 thread 的數不是越大越好因為會有 overhead 的問題，而且也要看 OS 是以哪種 Multi Thread 的類型例如多對一、一對一以及多對多等去設計才能夠有效使用硬體資源來加速運算時間，感謝老師讓我有機會做這個實驗，使我了解 Multi Thread 的重要性。

Additional



Source.cpp

```
#include "Header.h"
int main() {

    //Public variables
    array_two A(a_row, std::vector<double>(a_column));
    array_two B(b_row, std::vector<double>(b_column));
    array_two parameter[3];
    string  time_groups_name[] = { "for-loop", "ppl API", "OpenMP API" ,"Win32 API"};
    //Build time groups
    time_groups groups[4];
    build_time_groups(groups,time_groups_name);
    //Build Matrix A and B
    BuildArray( A, B);
    parameter[0] = A; parameter[1] = B;
    // Metrix C and C1 and C2 initialization
    array_two C(A.size(), std::vector<double>(B[0].size()));
    array_two C1(A.size(), std::vector<double>(B[0].size()));
    array_two C2(A.size(), std::vector<double>(B[0].size()));
    array_two C3(A.size(), std::vector<double>(B[0].size()));
    parameter[2] = C;

    // Repeat 3 times
    for (int i = 0; i < 1; i++) {
        //For loop part
        parameter[2] = C;
        //Calculate cost time of for-loop
        groups[0].times[i] = test_and_calculate_cost_time(Matrix_mul, parameter);
        Sleep(3000);
```

```
        parameter[2] = C1;
        //Calculate the time spent in parallel through the PPL API
        groups[1].times[i] = test_and_calculate_cost_time(parallel_matrix_mul_ppl, parameter);
        Sleep(3000);

        //OpenMP API part
        parameter[2] = C2;
        //Calculate
        groups[2].times[i] = test_and_calculate_cost_time(parallel_matrix_mul_omp, parameter);
        Sleep(3000);

        //Win32 API part
        parameter[2] = C3;
        //Calculate
        groups[3].times[i] = test_and_calculate_cost_time(parallel_matrix_mul_WIN32_main, parameter);
    }
    return 0;

    //Calculate average operating time
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 3; j++)
            groups[i].average_time += groups[i].times[j];
        groups[i].average_time /= 3;
    }
    //Print result
    print_table(groups);
    //std::cout << "Error rate : " << verification(C,C1) * 100 << "%" << std::endl;*/
}
```

Basic_function.cpp

```cpp
#include "Header.h"

void build_time_groups(time_groups* groups, string* time_groups_name) {
    for (int i = 0; i < 4; i++)
        groups[i].name = time_groups_name[i];
}

void print_table(time_groups* groups) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 3; j++) {
            cout << '(' << j + 1 << ')';
            cout << '['; cout.width(10); cout << left << groups[i].name;
            cout << ']' << "Cost time : ";
            cout.width(10); cout << right << groups[i].times[j] << "s" << endl;
        }
        cout << "Average time : "; cout.width(10); cout << right << groups[i].average_time << "s" << endl << endl;
    }
}
//
double verification(array_two C, array_two C1) {
    double Error_val = 0;
    for (int i = 0; i < C1.size(); i++)
        for (int j = 0; j < C1[0].size(); j++)
            Error_val = (C1[i][j] - C[i][j]) == 0 ? Error_val : Error_val + 1;
    return (double)Error_val / (C1.size() * C1[0].size());
}
```

Matrix_multiplication.cpp

```cpp
#include "Header.h"
void parallel_matrix_mul_omp(array_two arrayA, array_two arrayB, array_two& arrayC) {
    #pragma omp parallel num_threads(MAX_THREADS)
    {
    #pragma omp for
        for (int i = 0; i < arrayA.size(); i++)
            for (int j = 0; j < arrayB[0].size(); j++)
                for (int k = 0; k < arrayB.size(); k++)
                    arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
    }
}


DWORD WINAPI parallel_matrix_mul_WIN32(LPVOID Param) {
    PMYDATA arrays_P = (PMYDATA)Param;
    for (int i = 0; i < arrays_P->arrayA.size(); i++) {
        for (int j = 0; j < arrays_P->arrayB[0].size(); j++) {
            for (int k = 0; k < (arrays_P->arrayB.size() / MAX_THREADS); k++) {
                arrays_P->arrayC[i][j] += \
                    arrays_P->arrayA[i][k + (arrays_P->arrayB.size() / MAX_THREADS) * arrays_P->number] *
                    arrays_P->arrayB[k + (arrays_P->arrayB.size() / MAX_THREADS) * arrays_P->number][j];
            }
        }
    }
    if ((arrays_P->arrayB.size() % MAX_THREADS) != 0) {
        if ((arrays_P->arrayB.size() % MAX_THREADS) > arrays_P->number) {
            for (size_t i = 0; i < arrays_P->arrayA.size(); i++) {
                for (size_t j = 0; j < arrays_P->arrayB[0].size(); j++) {
                    arrays_P->arrayC[i][j] += arrays_P->arrayA[i][arrays_P->number + \
                        (arrays_P->arrayB.size() / MAX_THREADS) * MAX_THREADS] *
                        arrays_P->arrayB[arrays_P->number + \
                        (arrays_P->arrayB.size() / MAX_THREADS) * MAX_THREADS][j];
                }
            }
        }
    }
    return 0;
}
```

```
void parallel_matrix_mul_WIN32_main(array_two arrayA, array_two arrayB, array_two& arrayC) {
    DWORD ThreadId[MAX_THREADS];
    HANDLE ThreadHandle[MAX_THREADS];
    PMYDATA arrays_P[MAX_THREADS];
    /* create the thread */
    for (size_t i = 0; i < MAX_THREADS; i++) {
        // Allocate memory for thread data.
        arrays_P[i] = (PMYDATA)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(MYDATA));
        // If the array allocation fails, the system is out of memory
        // so there is no point in trying to print an error message.
        // Just terminate execution.
        if (arrays_P == NULL)
            ExitProcess(2);
        arrays_P[i]->arrayA = arrayA;
        arrays_P[i]->arrayB = arrayB;
        arrays_P[i]->arrayC = arrayC;
        arrays_P[i]->number = i;
        ThreadHandle[i] = CreateThread(NULL, 0, parallel_matrix_mul_WIN32,
            arrays_P[i], 0, &ThreadId[i]);
    }
    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS, ThreadHandle, TRUE, INFINITE);
    /*for (size_t i = 0; i < 4; i++)
        arrayC[0][0] += arrays_P[i]->arrayC[0][0];
    printf("%f\n", arrayC[0][0]);*/
    // Close the all thread handles and free memory allocations
    for (size_t i = 0; i < MAX_THREADS; i++) {
        CloseHandle(ThreadHandle[i]);
        if (arrays_P[i] != NULL) {
            HeapFree(GetProcessHeap(), 0, arrays_P[i]);
            arrays_P[i] = NULL;
        }
    }
    return;
}
```

```
//Parallel by ppl API
void parallel_matrix_mul_ppl(array_two arrayA, array_two arrayB, array_two& arrayC) {
    parallel_for(size_t(0), arrayA.size(), [&](size_t i) {
        for (int j = 0; j < arrayB[0].size(); j++)
            for (int k = 0; k < arrayB.size(); k++)
                arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
        });
}
//Metric multiplication
void Matrix_mul(array_two arrayA, array_two arrayB, array_two& arrayC) {
    for (int i = 0; i < arrayA.size(); i++)
        for (int j = 0; j < arrayB[0].size(); j++)
            for (int k = 0; k < arrayB.size(); k++)
                arrayC[i][j] += arrayA[i][k] * arrayB[k][j];
}
//Build Array A and B
void  BuildArray(array_two& A, array_two& B) {
    for (int i = 0; i < A.size(); i++) {
        for (int j = 0; j < B.size(); j++) {
            A[i][j] = 3.5 * i - 6.6 * j;
            B[j][i] = 6.6 + 8.8 * j - 3.5 * i;
        }
    }
}
//
double test_and_calculate_cost_time(void (*func)(array_two, array_two, array_two&),
    array_two* paramter) {
    LARGE_INTEGER StartingTime, EndingTime;
    LARGE_INTEGER Frequecy;

    QueryPerformanceFrequency(&Frequecy);
    QueryPerformanceCounter(&StartingTime);
    func(paramter[0], paramter[1], paramter[2]);
    QueryPerformanceCounter(&EndingTime);
    return (double)(EndingTime.QuadPart - StartingTime.QuadPart) / (double)Frequecy.QuadPart;
}
```

Header.h

```cpp

#include<iostream>
#include<vector>
#include <ppl.h>
#include<omp.h>
#include<windows.h>
#include <stdio.h>
#include <thread>

using namespace std;
using namespace concurrency;
typedef std::vector<std::vector<double>> array_two;

//Setting array A and B size
enum array_A_size { a_row = 1000, a_column = 219 };
enum array_B_size { b_row = 219, b_column = 1000 };

//setting thread numbers
#define MAX_THREADS 32

//time groups
struct time_groups {
    double times[3] = {};
    string name;
    double average_time = 0;
};

///Parallel by Win32API
typedef struct array_groups {
    array_two arrayA;
    array_two arrayB;
    array_two arrayC;
    int number;
}MYDATA, * PMYDATA;

//Display result and
void build_time_groups(time_groups*, string* );
void print_table(time_groups* );
double verification(array_two, array_two);

// matrix multiplication
void parallel_matrix_mul_omp(array_two, array_two, array_two&);
DWORD WINAPI parallel_matrix_mul_WIN32(LPVOID);
void parallel_matrix_mul_WIN32_main(array_two , array_two , array_two& );
void parallel_matrix_mul_ppl(array_two , array_two , array_two &);
void Matrix_mul(array_two, array_two, array_two &);
void  BuildArray(array_two & , array_two &);

// test and caculate time
double test_and_calculate_cost_time(void (*func)(array_two, array_two, array_two&), array_two* );
```

Reference:

[1]Microsoft, **https://docs.microsoft.com/en-us/visualstudio/profiling/cpu-usage?view=vs-2019**

[2] Microsoft, **https://docs.microsoft.com/en-us/windows/win32/procthread/creating-threads**

[3] cplusplus.com, **https://www.cplusplus.com/reference/vector/vector/assign/**

[4] bisqwit.iki.fi, **https://bisqwit.iki.fi/story/howto/openmp/**

[5] geeksforgeeks.org ,**https://www.geeksforgeeks.org/2d-vector-in-cpp-with-user-defined-size/**

[6] openmp.org ,**https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf**

[7] Wikipedia , **https://en.wikipedia.org/wiki/OpenMP**

[8] Mincrosft, **https://docs.microsoft.com/en-us/cpp/parallel/parallel-programming-in-visual-cpp?view=msvc-160**