BÀI THỰC HÀNH SỐ 2

NỘI DUNG: HPC VỚI OPENMP

A. THÔNG TIN CHUNG

Yêu cầu lý thuyết:

- HPC với tính toán song song sử dụng mô hình chia sẻ bộ nhớ
- Các lệnh chỉ dẫn/mệnh đề, hàm, môi trường OpenMP
- Các thuật toán nhân ma trận Cannon và Fox
- Các thuật toán Linear Search/Merge Sort/Select Sort

Yêu cầu báo cáo:

- Kết quả chạy các chương trình
- Trả lời các yêu cầu của từng bài

Sản phẩm nộp cuối buổi:

- Báo cáo
- Code các chương trình

A. NỘI DUNG THỰC HÀNH

Bài 1:

- Viết chương trình nhân 2 ma trận AxB kiểu tuần tự.
- Tính thời gian thực hiện nhân 2 ma trận (sử dụng hàm *GetTickCount()*/*omp_get_wtime()* để lấy thời gian bắt đầu và kết thúc xử lý – tham khảo bài phần III của bài thực hành số 1)

Bài 2: Lập trình song song với OpenMP để nhân 2 ma trận với thuật toán Fox. Phân tích chương trình (chú thích cạnh các dòng lệnh).

- ***Trình bày lại thuật toán Fox và công thức tính độ phức tạp.***
- Tính thời gian thực hiện xử lý song song theo lý thuyết (công thức tính độ phức tạp) và thời gian thu được từ thực nghiệm.
- So sánh với thời gian thực hiện tuần tự theo lý thuyết (công thức tính độ phức tạp) và thời gian thu được từ thực nghiệm.
- Tính hệ số speedup theo lý thuyết và theo thực nghiệm
- Giải thích kết quả.
- Kết quả được thể hiện trong bảng sau:

- STT	Kích thước ma trận NxN	Tuần tự (s)		Song song (s)		Speedup		Ghi chú
		LT	TN	LT	TN	LT	TN	
1	100x100							
2	1000x1000							
3	10000x10000							
4	1000000 x100000							

LT*: Lý thuyết*; TN: *Thực nghiệm*

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>


void matrix_creation(double** pA, double** pB, double** pC, int size)
{
    *pA = (double*)malloc(size * size * sizeof(double));
    *pB = (double*)malloc(size * size * sizeof(double));
    *pC = (double*)calloc(size * size, sizeof(double));
}
void matrix_initialization(double* A, double* B, int size, int sup)
{
    srand(time(NULL));
    for (int i = 0; i < size * size; ++i) {
        *(A + i) = rand() % sup + 1;
        *(B + i) = rand() % sup + 1;
    }
}

void matrix_dot(double* A, double* B, double* C, int n)
{
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            for (int k = 0; k < n; ++k) {
                *(C + i * n + j) += *(A + i * n + k) * *(B + k * n + j);
            }
        }
    }
}
int matrix_check(double* A, double* B, int n)
{
    for (int i = 0; i < n * n; ++i) {
        if (*(A + i) != *(B + i)) {
            return 0;
```

```
        }
    }
    return 1;
}
void matrix_print(double* A, int n)
{
    printf("---~---~---~---~---\n");
    for (int i = 0; i < n * n; ++i) {
        printf("%.2lf ", *(A + i));
        if ((i + 1) % n == 0) {
            printf("\n");
        }
    }
    printf("---~---~---~---~---\n");
}
void matrix_removal(double** pA, double** pB, double** pC)
{
    free(*pA);
    free(*pB);
    free(*pC);
}
/* ----------------------------- FoxAlgorithm -------------------------
----- */

void FoxAlgorithm(double* A, double* B, double* C, int m_size, int*
save)
{
    int stage;
#pragma omp parallel private(stage) shared(A, B, C)
    {
        int n_threads = omp_get_num_threads();
        *save = n_threads;  // used for printing in main()
        int n_blocks = sqrt(n_threads);
        int block_size = m_size / n_blocks;
        int PrNum = omp_get_thread_num();
        int i1 = PrNum / n_blocks, j1 = PrNum % n_blocks;
        double* A1, * B1, * C1;
        for (stage = 0; stage < n_blocks; ++stage) {
            A1 = A + (i1 * m_size + ((i1 + stage) % n_blocks)) *
block_size;
            B1 = B + (((i1 + stage) % n_blocks) * m_size + j1) *
block_size;
            C1 = C + (i1 * m_size + j1) * block_size;
            for (int i = 0; i < block_size; ++i) {
                for (int j = 0; j < block_size; ++j) {
                    for (int k = 0; k < block_size; ++k) {
                        *(C1 + i * m_size + j) += *(A1 + i * m_size + k)
* *(B1 + k * m_size + j);
                    }
                }
```

```
            }
        }
    }
}

/* ----------------------------------------------------------------------
----- */
int main(int argc, char** argv)
{
    int m_size = 100, n_threads = -1;

    if (argc == 2) {
        sscanf_s(argv[1], "%d", &m_size);
    }

    double* pA, * pB, * pC;
    matrix_creation(&pA, &pB, &pC, m_size);
    matrix_initialization(pA, pB, m_size, 1000);

    double start_time = omp_get_wtime();
    FoxAlgorithm(pA, pB, pC, m_size, &n_threads);
    double end_time = omp_get_wtime() - start_time;

    // printf("FoxAlgorithm_time: %.5lf n_threads: %d m_size: %d \n",
end_time, n_threads, m_size);
    printf("%.5lf, %d, %d\n", end_time, n_threads, m_size);
    matrix_removal(&pA, &pB, &pC);
    return 0;
}
```

Bài 3: Tương tự như bài 2 nhưng sử dụng thuật toán **Cannon**
(Code tham khảo kèm theo hoặc:
https://github.com/jorgeramirez/gcannon/blame/master/openmp/cannon.c)

PHẦN B: BÀI TẬP TỰ THỰC HÀNH.

Bài 4: Tìm kiếm tuyến tính

- Nghiên cứu thuật toán tìm kiếm tuyến tính tuần tự và cài đặt chương trình thực hiện thuật toán tuần tự.
- Nhập code chương trình thuật toán song song và chạy
- Chạy với N khác nhau trong 2 trường hợp và tổng kết như bảng ví dụ sau sau:

Table -1 Time taken by linear search and parallel linear search in worst case

N	Linear search(seconds)	Parallel linear search(seconds)
10000	0.000289	0.000288
20000	0.000568	0.000553
30000	0.000782	0.000677
40000	0.001064	0.000836
50000	0.001219	0.000947

(Lưu ý: Cả 2 thuật toán được thực hiện trên cùng dữ liệu trên máy tính. (cùng cấu hình, trạng thái).

-Thực hiện tính hệ số speedup và hệ số hiệu quả và so sánh với lý thuyết và giải thích.

//Code with the Parallel ALgorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "omp.h"
clock_t t;
double cpu_time_used;

int linearSearch(int* A, int n, int tos);

int main() {

	int number, iter = 0, find;
	int* Arr;

	Arr = (int*)malloc(number * sizeof(int));

	scanf("%d", &number);

	for (; iter < number; iter++) {
		scanf("%d", &Arr[iter]);
	}

	scanf("%d", &find);
	printf("\nTo find: %d\n", find);

	t = clock();
	int indexx = linearSearch(Arr, number, find);
	t = clock() - t;

	if (indexx == -1) {
		printf("Not found");
	}
	else
		printf("Found at %d\n", indexx);

	cpu_time_used = ((double)t) / CLOCKS_PER_SEC;

	printf("\nTime taken for search: %f", cpu_time_used);
```

```
	return 0;

}

// Linear serach beigns here
int linearSearch(int* A, int n, int tos) {

	int foundat = -1;

	//Simple OpenMP for loop in parallel
#pragma omp parallel for
	for (int iter = 0; iter < n; iter++) {
		if (A[iter] == tos)
			// DO not return since it will result in an invalid branch.
			foundat = iter + 1;
	}
	// Return the index finally, after each and every element has been checked.
	return foundat;
}
```

Bài 5: Thực hiện tương tự như bài 4

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"
void mergesort(int a[], int i, int j);
void merge(int a[], int i1, int j1, int i2, int j2);
int main()
{
    int* a, num, i;
    scanf("%d", &num);

    a = (int*)malloc(sizeof(int) * num);
    for (i = 0; i < num; i++)
        scanf("%d", &a[i]);

    mergesort(a, 0, num - 1);

    printf("\nSorted array :\n");
    for (i = 0; i < num; i++)
        printf("%d ", a[i]);

    return 0;
}

void mergesort(int a[], int i, int j)
{
    int mid;

    if (i < j)
    {
        mid = (i + j) / 2;

#pragma omp parallel sections
        {

#pragma omp section
            {
                mergesort(a, i, mid);        //left recursion
```

```
            }

#pragma omp section
            {
                mergesort(a, mid + 1, j);    //right recursion
            }
        }

        merge(a, i, mid, mid + 1, j);    //merging of two sorted sub-arrays
    }
}

void merge(int a[], int i1, int j1, int i2, int j2)
{
    int temp[1000];    //array used for merging
    int i, j, k;
    i = i1;    //beginning of the first list
    j = i2;    //beginning of the second list
    k = 0;

    while (i <= j1 && j <= j2)    //while elements in both lists
    {
        if (a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }

    while (i <= j1)    //copy remaining elements of the first list
        temp[k++] = a[i++];

    while (j <= j2)    //copy remaining elements of the second list
        temp[k++] = a[j++];

    //Transfer elements from temp[] back to a[]
    for (i = i1, j = 0; i <= j2; i++, j++)
        a[i] = temp[j];
}
```

Bai 5: Thực hiện tương tự như bài 4

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "omp.h"
clock_t t, end;
double cpu_time_used;
// Structure for enabling reduction on the index of elements
struct Compare { int val; int index; };
// Custom reduction for finding hte index of the max element.
#pragma omp declare reduction(maximum : struct Compare : omp_out = omp_in.val >
omp_out.val ? omp_in : omp_out)
void swap(int* a, int* b);
void selectionSort(int* A, int n);
void verify(int* A, int n);
```

```
int main() {

	int number, iter = 0;
	int* Arr;

	Arr = (int*)malloc(number * sizeof(int));

	scanf("%d", &number);

	for (; iter < number; iter++) {
		scanf("%d", &Arr[iter]);
	}

	t = clock();
	selectionSort(Arr, number);
	t = clock() - t;

	for (int iter = 0; iter < number; iter++) {
		printf("%d ", Arr[iter]);
	}

	cpu_time_used = ((double)t) / CLOCKS_PER_SEC;

	// Verify if the algorithm works as advised
	verify(Arr, number);

	printf("\nTime taken for sort: %f", cpu_time_used);
	return 0;

}

void selectionSort(int* A, int n) {

	for (int startpos = 0; startpos < n; startpos++) {
		// Declare the structure required for reduction
		struct Compare max;
		// Initialize the variables
		max.val = A[startpos];
		max.index = startpos;

// Parallel for loop with custom reduction, at the end of the loop, max will have
//the max element and its index.
#pragma omp parallel for reduction(maximum:max)
		for (int i = startpos + 1; i < n; ++i) {
```

```
                if (A[i] > max.val) {
                        max.val = A[i];
                        max.index = i;
                }
        }

        swap(&A[startpos], &A[max.index]);
    }
}

// Verification function
void verify(int* A, int n) {
    int failcount = 0;
    for (int iter = 0; iter < n - 1; iter++) {
        if (A[iter] < A[iter + 1]) {
            failcount++;
        }
    }
    printf("\nFail count: %d\n", failcount);
}

//Swap function
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Bài 6. Cho các thuật toán song song:

- Thuật toán sắp xếp: *Enumeration Sort/ Odd-Even Transposition Sort*
- Thuật toán tìm kiếm: *Binary search/Depth-First Search/Breadth-First Search/Best-First Search*

Yêu cầu:

- Xem lại thuật toán tuần tự và viết chương trình tuần tự
- Phân tích các thuật toán song song (dạng Pseudo-code) cho sau và viết chương trình C với OpenMP
- Đánh giá hiệu quả của thuật toán song song trên máy tính cụ thể

// Enumeration Sort

```
procedure ENUM_SORTING (n)

begin
```

```
  for each process P1,j do
    C[j] := 0;

  for each process Pi, j do

    if (A[i] < A[j]) or A[i] = A[j] and i < j) then
      C[j] := 1;
    else
      C[j] := 0;

  for each process P1, j do
    A[C[j]] := A[j];

end ENUM_SORTING
```

// Odd-Even Transposition Sort

```
procedure ODD-EVEN_PAR (n)

begin
  id := process's label

  for i := 1 to n do
  begin

    if i is odd and id is odd then
      compare-exchange_min(id + 1);
    else
      compare-exchange_max(id - 1);

    if i is even and id is even then
      compare-exchange_min(id + 1);
    else
      compare-exchange_max(id - 1);

  end for
```

// Binary search

```
Binarysearch(a, b, low, high)

if low < high then
  return NOT FOUND
else
  mid ← (low+high) / 2
```

```
  if b = key(mid) then
    return key(mid)
  else if b < key(mid) then
    return BinarySearch(a, b, low, mid−1)
  else

    return BinarySearch(a, b, mid+1, high)
```

//DFS

```
DFS(G,v)

  Stack S := {};

  for each vertex u, set visited[u] := false;
  push S, v;
  while (S is not empty) do
   u := pop S;

    if (not visited[u]) then
      visited[u] := true;
      for each unvisited neighbour w of u
        push S, w;
    end if

  end while
```

//BFS

```
BFS(G,v)

  Queue Q := {};

  for each vertex u, set visited[u] := false;
  insert Q, v;
  while (Q is not empty) do
    u := delete Q;

    if (not visited[u]) then
      visited[u] := true;
      for each unvisited neighbor w of u
        insert Q, w;
    end if

  end while
```

```
END BFS()
```

Sinh viên tham khảo các thuật toán song song khác theo link:

https://www.tutorialspoint.com/parallel_algorithm/index.htm

https://www.cs.cmu.edu/~scandal/nesl/algorithms.html#scan