Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

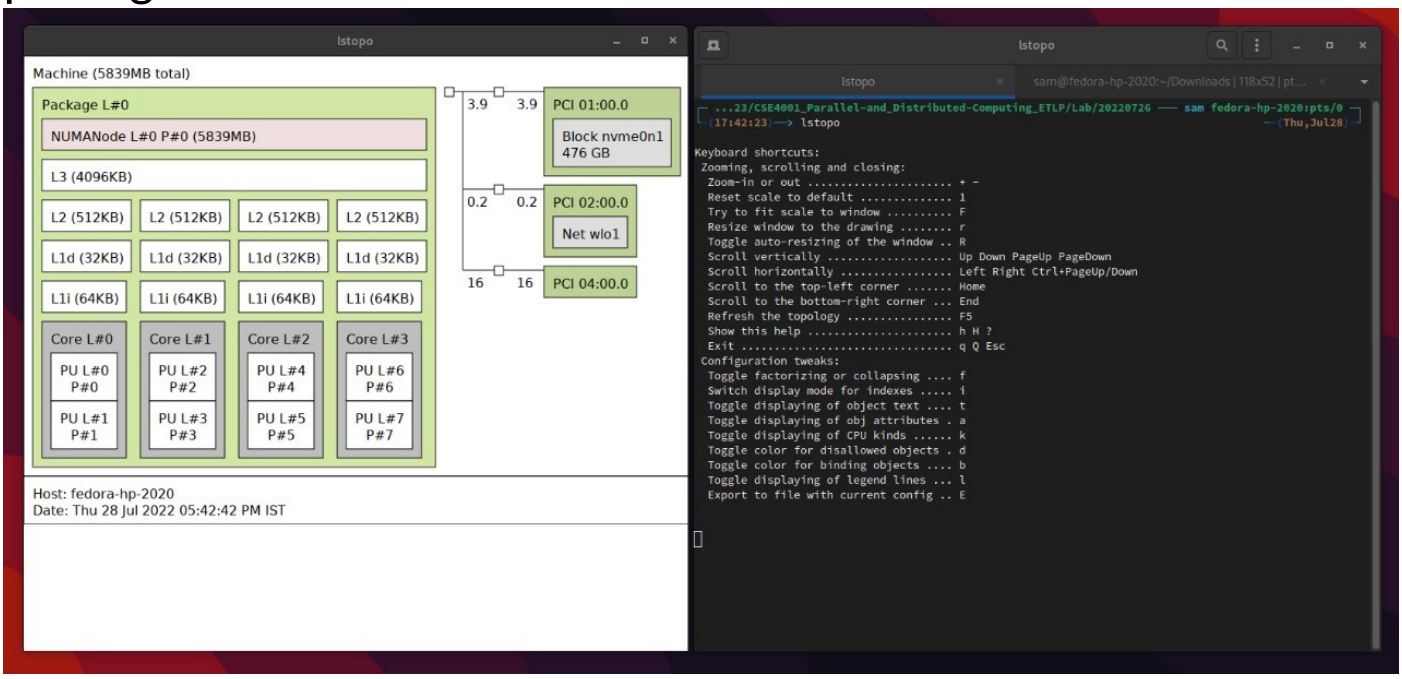
Consolidated Lab Report

Name: Samridh Anand Paatni
Registration Number: 20BCE1550
Subject: CSE4001 Parallel Distributed Computing
Slot: L9+L10
Professor: Dr. Kumar R

Lab 01: PThreads

<u>Q1.</u> Display the processors layout of your system.

The output for the command **`lstopo`**, after installing the hwloc package:



<u>Q2.</u> Write a multithreaded program in C to create 10k, 20k and 50k threads and measure the time taken for each thread group.

<u>C Code:</u>

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

void * void_function(void *message) {}

int main(int argc, char * argv[]) {
    pthread_t * threads;
    int num_threads = atoi(argv[1]); // because the cli argument is an ASCII code

    threads = (pthread_t *) calloc(num_threads, sizeof(pthread_t));

    clock_t t = clock();

    for (int i = 0; i < num_threads; i++) {
        pthread_create(&threads[i], NULL, void_function, NULL);
    }

    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    t = clock() - t;

    printf(
        "took %f seconds to make %d threads\n",
        ((double) t)/CLOCKS_PER_SEC,
        num_threads
    );

    free(threads);

    return 0;
}
```

Output:

Creating 5000 pthreads resulted in a segmentation fault.

```
sam@fedora-hp-2020:~/VIT/year3/sem5_fall-22-23/CSE4001_Parallel-and_...
┌(...d_Distributed-Computing_ETLP/Lab/20220726 ── sam fedora-hp-2020:pts/0 ┐
└─(18:40:07)──> gcc q2.c -o q2 -lpthread                    ──(Thu,Jul28)─┘
┌(...d_Distributed-Computing_ETLP/Lab/20220726 ── sam fedora-hp-2020:pts/0 ┐
└─(18:40:17)──> ./q2 10000                                  ──(Thu,Jul28)─┘
took 0.441445 seconds to make 10000 threads
┌(...d_Distributed-Computing_ETLP/Lab/20220726 ── sam fedora-hp-2020:pts/0 ┐
└─(18:40:22)──> ./q2 20000                                  ──(Thu,Jul28)─┘
took 0.965726 seconds to make 20000 threads
┌(...d_Distributed-Computing_ETLP/Lab/20220726 ── sam fedora-hp-2020:pts/0 ┐
└─(18:40:28)──> ./q2 50000                                  ──(Thu,Jul28)─┘
[1]    33889 segmentation fault (core dumped)  ./q2 50000
┌(...d_Distributed-Computing_ETLP/Lab/20220726 ── sam fedora-hp-2020:pts/0 ┐
└─(18:40:34)──>                                        139 ↵ ──(Thu,Jul28)─┘
```

Q3. Write a program to create 2 threads. Thread 1 has to print "PDC" and thread 2 has to print "lab".

C Code:

```c
#include<stdio.h>
#include<pthread.h>

void * message_function(void *message) {
    printf("%s\n", ((char *) message));
}

int main() {
    pthread_t t1, t2;

    char *m1 = "Thread 1: PDC";
    char *m2 = "Thread 2: Lab";

    pthread_create(&t1, NULL, message_function, (void *) m1);
    pthread_create(&t2, NULL, message_function, (void *) m2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

Output:

```
sam@fedora-hp-2020:~/VIT/year3/sem5_fall-22-23/CSE4001_Parallel-and_Distributed-Computing...
┌─(...d_Distributed-Computing_ETLP/Lab/20220726)── sam fedora-hp-2020:pts/0 ─┐
└─(17:55:41)──> gcc q3.c -o q3 -lpthread                          ──(Thu,Jul28)─┘
┌─(...d_Distributed-Computing_ETLP/Lab/20220726)── sam fedora-hp-2020:pts/0 ─┐
└─(17:55:44)──> ./q3                                              ──(Thu,Jul28)─┘
Thread 1: PDC
Thread 2: Lab
┌─(...d_Distributed-Computing_ETLP/Lab/20220726)── sam fedora-hp-2020:pts/0 ─┐
└─(17:55:47)──>                                                   ──(Thu,Jul28)─┘
```

Lab 02: Open MP Programming

Q1. Create a hundred threads using:

a) Runtime Library Routines

Code:

```
// using runtime library routines

#include<stdio.h>
#include<omp.h>

// compile using: `gcc filename -fopenmp`

int main(int argc, char *argv[]) {
    int tid, numThreads;
    omp_set_num_threads(100);
    # pragma omp parallel private (tid, numThreads)
    {
        tid = omp_get_thread_num();
        printf("welcome to PDC %d\n", tid);

        if (tid == 0) {
            numThreads = omp_get_num_threads();
            printf("%d threads have been created\n", numThreads);
        }
    }
    return 0;
}
```

Output:

```
sam@fedora-hp-2020:~/VIT/year3/sem5_fall-22-23/CSE...
┌──(.../20220802)── sam fedora-hp-2020:pts/0 ──┐
└─(10:10:54)──> make case1        ──(Fri,Aug05)─┘
gcc case1.c -fopenmp -o case1.out && \
./case1.out
welcome to PDC 1
welcome to PDC 15
welcome to PDC 20
welcome to PDC 22
welcome to PDC 23
welcome to PDC 4
welcome to PDC 26
welcome to PDC 27
welcome to PDC 8
welcome to PDC 29
welcome to PDC 31
welcome to PDC 30
welcome to PDC 41
welcome to PDC 43
welcome to PDC 40
welcome to PDC 11
welcome to PDC 13
welcome to PDC 12
welcome to PDC 51
welcome to PDC 14
welcome to PDC 56
welcome to PDC 54
```

```
sam@fedora-hp-2020:~/VIT/year3/s
welcome to PDC 56
welcome to PDC 54
welcome to PDC 57
welcome to PDC 58
welcome to PDC 60
welcome to PDC 61
welcome to PDC 62
welcome to PDC 64
welcome to PDC 65
welcome to PDC 24
welcome to PDC 68
welcome to PDC 69
welcome to PDC 70
welcome to PDC 9
welcome to PDC 76
welcome to PDC 28
welcome to PDC 81
welcome to PDC 82
welcome to PDC 33
welcome to PDC 89
welcome to PDC 90
welcome to PDC 92
welcome to PDC 39
welcome to PDC 93
welcome to PDC 94
welcome to PDC 97
```

```
sam@fedora-hp-2020:~/VIT/yea
welcome to PDC 94
welcome to PDC 97
welcome to PDC 42
welcome to PDC 98
welcome to PDC 45
welcome to PDC 48
welcome to PDC 46
welcome to PDC 47
welcome to PDC 5
welcome to PDC 52
welcome to PDC 50
welcome to PDC 17
welcome to PDC 18
welcome to PDC 16
welcome to PDC 55
welcome to PDC 59
welcome to PDC 6
welcome to PDC 21
welcome to PDC 19
welcome to PDC 63
welcome to PDC 53
welcome to PDC 67
welcome to PDC 3
welcome to PDC 66
welcome to PDC 72
welcome to PDC 2
```

```
sam@fedora-hp-2020:~/VIT/year3/sem5_fall-22-23/CSE
welcome to PDC 75
welcome to PDC 73
welcome to PDC 77
welcome to PDC 71
welcome to PDC 7
welcome to PDC 78
welcome to PDC 83
welcome to PDC 80
welcome to PDC 84
welcome to PDC 79
welcome to PDC 35
welcome to PDC 32
welcome to PDC 85
welcome to PDC 86
welcome to PDC 91
welcome to PDC 87
welcome to PDC 10
welcome to PDC 34
welcome to PDC 88
welcome to PDC 95
welcome to PDC 36
welcome to PDC 0
100 threads have been created
welcome to PDC 96
welcome to PDC 38
welcome to PDC 37
welcome to PDC 44
welcome to PDC 99
welcome to PDC 49
┌─(.../20220802)──(sam㉿fedora-
└─(10:13:38)──>
```

b)Compiler Directives

Code:

```
// using compiler directives

#include<stdio.h>
#include<omp.h>

// compile using: `gcc filename -fopenmp`

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

    # pragma omp parallel private (tid, numThreads) num_threads(100)
    {
        tid = omp_get_thread_num();
        printf("welcome to PDC %d\n", tid);

        if (tid == 0) {
            numThreads = omp_get_num_threads();
            printf("%d threads have been created\n", numThreads);
        }
    }
    return 0;
}
```

Output:

```
sam@fedora-hp-2020:~/VIT/year3/sem5_fall-22-23/CSE...
┌─(.../20220802)──(sam@fedora-hp-2020:pts/0)─┐
└─(10:19:35)──> make case2        ──(Fri,Aug05)─┘
gcc case2.c -fopenmp -o case2.out && \
./case2.out
welcome to PDC 2
welcome to PDC 40
welcome to PDC 44
welcome to PDC 47
welcome to PDC 52
welcome to PDC 61
welcome to PDC 58
welcome to PDC 64
welcome to PDC 62
welcome to PDC 14
welcome to PDC 68
welcome to PDC 11
welcome to PDC 71
welcome to PDC 70
welcome to PDC 3
welcome to PDC 12
welcome to PDC 80
welcome to PDC 79
welcome to PDC 15
welcome to PDC 81
welcome to PDC 4
welcome to PDC 85
welcome to PDC 78
welcome to PDC 20
```

```
sam@fedora-hp-2020:~/VIT/yea
welcome to PDC 20
welcome to PDC 23
welcome to PDC 24
welcome to PDC 26
welcome to PDC 25
welcome to PDC 5
welcome to PDC 22
welcome to PDC 28
welcome to PDC 27
welcome to PDC 30
welcome to PDC 31
welcome to PDC 29
welcome to PDC 32
welcome to PDC 34
welcome to PDC 36
welcome to PDC 35
welcome to PDC 33
welcome to PDC 37
welcome to PDC 38
welcome to PDC 39
welcome to PDC 41
welcome to PDC 43
welcome to PDC 42
welcome to PDC 45
welcome to PDC 7
welcome to PDC 9
welcome to PDC 46
welcome to PDC 48
```

```
sam@fedora-hp-2020:~/VIT/yea
welcome to PDC 48
welcome to PDC 49
welcome to PDC 50
welcome to PDC 51
welcome to PDC 53
welcome to PDC 10
welcome to PDC 54
welcome to PDC 55
welcome to PDC 56
welcome to PDC 17
welcome to PDC 59
welcome to PDC 57
welcome to PDC 6
welcome to PDC 60
welcome to PDC 63
welcome to PDC 65
welcome to PDC 66
welcome to PDC 1
welcome to PDC 67
welcome to PDC 69
welcome to PDC 73
welcome to PDC 13
welcome to PDC 72
welcome to PDC 75
welcome to PDC 8
welcome to PDC 77
welcome to PDC 76
welcome to PDC 74
```

```
sam@fedora-hp-2020:~/VIT/year3/sem5_fall-22-23/CS
welcome to PDC 77
welcome to PDC 76
welcome to PDC 74
welcome to PDC 16
welcome to PDC 84
welcome to PDC 82
welcome to PDC 83
welcome to PDC 18
welcome to PDC 91
welcome to PDC 87
welcome to PDC 0
100 threads have been created
welcome to PDC 88
welcome to PDC 90
welcome to PDC 89
welcome to PDC 86
welcome to PDC 93
welcome to PDC 92
welcome to PDC 19
welcome to PDC 95
welcome to PDC 94
welcome to PDC 97
welcome to PDC 96
welcome to PDC 98
welcome to PDC 99
welcome to PDC 21
┌─(.../20220802)──(sam@fedora-
└─(10:19:38)──>
```

c) Environment Variables

Code:

```
// using environment variables

#include<stdio.h>
#include<omp.h>

// compile using: `gcc filename -fopenmp`
// before running, give the command: `export OMP_NUM_THREADS=100` in bash

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

    # pragma omp parallel private (tid, numThreads)
    {
        tid = omp_get_thread_num();
        printf("welcome to PDC %d\n", tid);

        if (tid == 0) {
            numThreads = omp_get_num_threads();
            printf("%d threads have been created\n", numThreads);
        }
    }
    return 0;
}
```

Output:

```
sam@fedora-hp-2020:~/VIT/year3/sem5_fall-22-23/CSE...
┌─(.../20220802)──(sam㉿fedora-hp-2020:pts/0)─┐
└─(10:23:53)──> make case3      ──(Fri,Aug05)─┘
export OMP_NUM_THREADS=100 && \
gcc case3.c -fopenmp -o case3.out && \
./case3.out
welcome to PDC 5
welcome to PDC 29
welcome to PDC 38
welcome to PDC 40
welcome to PDC 6
welcome to PDC 42
welcome to PDC 7
welcome to PDC 2
welcome to PDC 44
welcome to PDC 47
welcome to PDC 3
welcome to PDC 52
welcome to PDC 59
welcome to PDC 58
welcome to PDC 12
welcome to PDC 10
welcome to PDC 62
welcome to PDC 61
welcome to PDC 64
welcome to PDC 67
welcome to PDC 69
welcome to PDC 70
welcome to PDC 76
```

```
sam@fedora-hp-2020:~/VIT/year3/sem5_fall-22-23/CS
welcome to PDC 76
welcome to PDC 68
welcome to PDC 73
welcome to PDC 80
welcome to PDC 20
welcome to PDC 23
welcome to PDC 84
welcome to PDC 81
welcome to PDC 82
welcome to PDC 90
welcome to PDC 95
welcome to PDC 96
welcome to PDC 98
welcome to PDC 99
welcome to PDC 30
welcome to PDC 0
welcome to PDC 32
100 threads have been created
welcome to PDC 35
welcome to PDC 37
welcome to PDC 33
welcome to PDC 39
welcome to PDC 4
welcome to PDC 41
welcome to PDC 46
welcome to PDC 9
welcome to PDC 45
welcome to PDC 48
```

```
sam@fedora-hp-2020:~/VIT/ye
welcome to PDC 48
welcome to PDC 50
welcome to PDC 43
welcome to PDC 8
welcome to PDC 53
welcome to PDC 54
welcome to PDC 11
welcome to PDC 49
welcome to PDC 56
welcome to PDC 55
welcome to PDC 51
welcome to PDC 13
welcome to PDC 63
welcome to PDC 60
welcome to PDC 15
welcome to PDC 14
welcome to PDC 16
welcome to PDC 57
welcome to PDC 17
welcome to PDC 65
welcome to PDC 71
welcome to PDC 21
welcome to PDC 66
welcome to PDC 19
welcome to PDC 72
welcome to PDC 18
welcome to PDC 77
welcome to PDC 74
```

```
sam@fedora-hp-2020:~/VIT/ye
welcome to PDC 77
welcome to PDC 74
welcome to PDC 22
welcome to PDC 75
welcome to PDC 79
welcome to PDC 78
welcome to PDC 25
welcome to PDC 83
welcome to PDC 24
welcome to PDC 88
welcome to PDC 85
welcome to PDC 89
welcome to PDC 86
welcome to PDC 26
welcome to PDC 91
welcome to PDC 28
welcome to PDC 87
welcome to PDC 92
welcome to PDC 94
welcome to PDC 93
welcome to PDC 27
welcome to PDC 97
welcome to PDC 31
welcome to PDC 1
welcome to PDC 36
welcome to PDC 34
┌──(.../20220802)──
└─(10:23:55)──>
```

<u>Q2.</u> Implement vector addition in serial and parallel and compare the results. Do the parallel computation using a 1000 threads.

Code:

Serial addition:

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

#define VECTOR_SIZE 100000

int main() {
    // make the vectors
    int a[VECTOR_SIZE], b[VECTOR_SIZE], c[VECTOR_SIZE];
    for (int i = 0; i < VECTOR_SIZE; i++) {
        a[i] = VECTOR_SIZE - i;
        b[i] = i;
    }

    // serially add the vectors
    clock_t tSerial = clock();
    for (int i = 0; i < VECTOR_SIZE; i++) {
        c[i] = a[i] + b[i];
    }
    tSerial = clock() - tSerial;

    // show the output
    printf(
        "Serial addition took %f seconds\n",
        ((double) tSerial)/CLOCKS_PER_SEC
    );

    return 0;
}
```

Parallel addition:

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

#define VECTOR_SIZE 100000

int main(int argc, char * argv[]) {
    // make the vectors
    int a[VECTOR_SIZE], b[VECTOR_SIZE], c[VECTOR_SIZE];
```

```
    for (int i = 0; i < VECTOR_SIZE; i++) {
        a[i] = VECTOR_SIZE - i;
        b[i] = i;
    }

    int nThreads = atoi(argv[1]);

    // paralelly add the vectors:

    clock_t tPar = clock();

    // the part of the vector one thread will access
    int slice_size = VECTOR_SIZE / nThreads;

    int slice_start, slice_end;
    int tid;

    // make threads
    omp_set_num_threads(nThreads);
    #pragma omp parallel private (tid, slice_start, slice_end)
    {
        // allot a slice to the particular thread
        tid = omp_get_thread_num();
        slice_start = tid * slice_size;
        slice_end = slice_start + slice_size;

        // perform addition for the elements in the allotted slice_size
        for (int i = slice_start; i < slice_end; i++) {
            c[i] = a[i] + b[i];
        }
    }

    tPar = clock() - tPar;

    // show the output
    printf(
        "Parallel addition took %f seconds with %d threads\n",
        ((double) tPar)/CLOCKS_PER_SEC,
        nThreads
    );

    return 0;
}
```

Output:

Running the computation with 4 threads is the fastest (because I have a quad-core laptop). The parallel computation with a 100 more threads is slower than the serial execution.

```
┌─( ...01_Parallel-and_Distributed-Computing_ETLP/Lab/20220802 ── ( sam fedora-hp-2020:pts/0 ─┐
└─(11:31:53)──> make vadds                                          ──(Fri,Aug05)─┘
gcc vadd_serial.c -fopenmp -o vadds.out && \
./vadds.out
Serial addition took 0.000916 seconds
┌─( ...01_Parallel-and_Distributed-Computing_ETLP/Lab/20220802 ── ( sam fedora-hp-2020:pts/0 ─┐
└─(11:31:55)──> make vaddp                                          ──(Fri,Aug05)─┘
gcc vadd_parallel.c -fopenmp -o vaddp.out && \
./vaddp.out 4
Parallel addition took 0.000544 seconds with 4 threads
./vaddp.out 1000
Parallel addition took 0.080599 seconds with 1000 threads
```

Lab 4: Sections

Q1.

Code:

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

void function1() {
    // int n = 100000;
    int n = 16;
    int *nums = (int *)calloc(n, sizeof(int));

    for (int i = 0; i < n; i++) {
        nums[i] = rand() % 100;
    }

    int i = 0;
    int local_min_1, local_min_2, local_min_3, local_min_4;
    int min = 100;

    #pragma omp parallel shared(nums, n, min) private(i, local_min_1, local_min_2,
local_min_3, local_min_4) num_threads(4)
    {
        #pragma omp sections
        {
            min = 100;
            #pragma omp section
            {
                local_min_1 = 100;
                for (i = 0; i < n/4; i++) {
                    if (nums[i] < local_min_1) local_min_1 = nums[i];
                }
                #pragma omp critical
                {
                    if (local_min_1 < min) min = local_min_1;
                }
            }
            #pragma omp section
            {
                local_min_2 = 100;
                for (i = n/4; i < 2 * n/4; i++) {
                    if (nums[i] < local_min_2) local_min_2 = nums[i];
                }
                #pragma omp critical
                {
                    if (local_min_2 < min) min = local_min_2;
                }
            }
```

```
            #pragma omp section
            {
                local_min_3 = 100;
                for (i = 2 * n/4; i < 3 * n/4; i++) {
                    if (nums[i] < local_min_3) local_min_3 = nums[i];
                }
                #pragma omp critical
                {
                    if (local_min_3 < min) min = local_min_3;
                }
            }
            #pragma omp section
            {
                local_min_4 = 100;
                for (i = 3 * n/4; i < n; i++) {
                    if (nums[i] < local_min_4) local_min_4 = nums[i];
                }
                #pragma omp critical
                {
                    if (local_min_4 < min) min = local_min_4;
                }
            }
        }
    }

        for (i = 0; i < n; i++) {
            printf(
                "%s%d%s",
                i == 0 ? "[" : " ",
                nums[i],
                i == n - 1 ? "]\n" : ","
            );
        }
        printf("\n%d is the minimum element", min);
        free(nums);
}

int main() {
    function1();

    printf("\n");
    return 0;
}
```

Output:

```
gcc q1.c -fopenmp -o q1.out
./q1.out
[83, 86, 77, 15, 93, 35, 86, 92, 49, 21, 62, 27, 90, 59, 63, 26]

15 is the minimum element
```

Q2.

Code:

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

typedef int ** matrix;
#define N 1100 // the size of the matrices

void multiply(matrix A, matrix B, matrix ans) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                ans[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// case 1 = only the outermost loop is parallelized
void multiplyCase1(matrix A, matrix B, matrix ans, int numThreads) {
    printf("The outermost loop is parallelized\n");
    int i = 0;
    #pragma omp parallel for shared(numThreads, A, B, ans) private(i)
    for (i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                ans[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void multiplyCase2(matrix A, matrix B, matrix ans, int numThreads) {
    printf("The outer 2 loops are parallelized\n");
    int i = 0, j = 0;
    #pragma omp parallel for shared(numThreads, A, B, ans) private(i)
    for (i = 0; i < N; i++) {
        #pragma omp parallel for
```

```
        for (j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                ans[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void multiplyCase3(matrix A, matrix B, matrix ans, int numThreads) {
    printf("All loops are parallelized\n");
    int i = 0, j = 0, k = 0;
    #pragma omp parallel for shared(numThreads, A, B, ans) private(i)
    for (i = 0; i < N; i++) {
        #pragma omp parallel for
        for (j = 0; j < N; j++) {
            #pragma omp parallel for
            for (k = 0; k < N; k++) {
                ans[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void multiplyCase4(matrix A, matrix B, matrix ans, int numThreads) {
    printf("All loops are parallelized\n");
    int i = 0, j = 0, k = 0;
    #pragma omp parallel for shared(numThreads, A, B, ans) private(i) collapse(3)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                ans[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void print(matrix A) {
    for (int i = 0; i < N; i++) {
        printf("%s", i == 0 ? "[\n " : " ");
        for (int j = 0; j < N; j++) {
            printf(
                "%s%d%s",
                j == 0 ? "[" : " ",
                A[i][j],
                j == N - 1 ? "]" : ","
            );
        }
        printf("%s", i == N - 1 ? "\n]\n" : ",\n");
    }
}

int main(int argc, char *argv[]) {
    int numThreads = atoi(argv[1]);
    int type = atoi(argv[2]);
```

```
    printf("multiplying 2 %dx%d matrices using %d threads and case %d:\n", N, N,
numThreads, type);

    matrix A, B, C;

    A = (int **)calloc(N, sizeof(int*));
    B = (int **)calloc(N, sizeof(int*));
    C = (int **)calloc(N, sizeof(int*));

    for (int i = 0; i < N; i++) {
        A[i] = (int *) calloc(N, sizeof(int));
        B[i] = (int *) calloc(N, sizeof(int));
        C[i] = (int *) calloc(N, sizeof(int));

        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
            C[i][j] = 0;
        }
    }

    // print(A);
    // print(B);

    double t = omp_get_wtime();

    switch (type)
    {
    case 0:
        multiply(A, B, C);
        break;

    case 1:
        multiplyCase1(A, B, C, numThreads);
        break;

    case 2:
        multiplyCase2(A, B, C, numThreads);
        break;

    case 3:
        multiplyCase3(A, B, C, numThreads);
        break;

    case 4:
        multiplyCase4(A, B, C, numThreads);
        break;

    default:
    }

    t = omp_get_wtime() - t;
```

```
    // print(C);

    printf("took %f seconds\n", t);

    for (int i = 0; i < N; i++) {
        free(A[i]);
        free(B[i]);
    }
    free(A);
    free(B);
    free(C);

    printf("\n");
    return 0;
}
```

Output:

```
gcc q2.c -fopenmp -o q2.out
./q2.out 4 1
multiplying 2 1100x1100 matrices using 4 threads and case 1:
The outermost loop is parallelized
took 3.952004 seconds

./q2.out 8 1
multiplying 2 1100x1100 matrices using 8 threads and case 1:
The outermost loop is parallelized
took 4.079892 seconds

./q2.out 16 1
multiplying 2 1100x1100 matrices using 16 threads and case 1:
The outermost loop is parallelized
took 4.190457 seconds

./q2.out 4 2
multiplying 2 1100x1100 matrices using 4 threads and case 2:
The outer 2 loops are parallelized
took 3.976589 seconds

./q2.out 8 2
multiplying 2 1100x1100 matrices using 8 threads and case 2:
The outer 2 loops are parallelized
took 4.423549 seconds

./q2.out 16 2
multiplying 2 1100x1100 matrices using 16 threads and case 2:
The outer 2 loops are parallelized
took 4.767388 seconds
```

```
./q2.out 4 3
multiplying 2 1100x1100 matrices using 4 threads and case 3:
All loops are parallelized
took 4.870290 seconds

./q2.out 8 3
multiplying 2 1100x1100 matrices using 8 threads and case 3:
All loops are parallelized
took 5.937704 seconds

./q2.out 16 3
multiplying 2 1100x1100 matrices using 16 threads and case 3:
All loops are parallelized
took 6.497956 seconds

./q2.out 4 4
multiplying 2 1100x1100 matrices using 4 threads and case 4:
All loops are parallelized
took 5.845127 seconds

./q2.out 8 4
multiplying 2 1100x1100 matrices using 8 threads and case 4:
All loops are parallelized
took 7.457370 seconds

./q2.out 16 4
multiplying 2 1100x1100 matrices using 16 threads and case 4:
All loops are parallelized
took 6.505237 seconds
```

Lab 05: OMP Synchronization Constructs

Q1.

Code:

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

#define N 10

#define OMP_NUM_THREADS 4

/*
1. Write your own code snippet to demonstrate the following
    a. Barrier
    b. Master
    c. Single
    d. Critical
    e. Ordered
*/

void barrier() {
    printf("\na. Barrier:\n");
    int a[1000], b[1000], i = 0, sum = 0;

    for (i = 0; i < 1000; i++) {
        a[i] = rand() % 100;
        b[i] = rand() % 10;
    }

    #pragma omp parallel private(i)
    {
        for (i = 0; i < 1000; i++) {
            a[i] = a[i] - b[i];
        }

        #pragma omp barrier

        #pragma omp for reduction(+ : sum)
        for (i = 0; i < 1000; i++)
        {
            sum += a[i];
        }
    }
    printf("sum: %d\n", sum);
}

void master() {
    printf("\nb. Master\nwithout 'master':\n");
    #pragma omp parallel
    {
```

```
        printf("hello, from thread %d\n", omp_get_thread_num());
    }

    printf("\nwith master:\n");
    #pragma omp parallel
    {
        #pragma omp master
        {
            printf("hello, from thread %d\n", omp_get_thread_num());
        }
    }
}

void single() {
    printf("\nc. Single:\n");
    int a=0, b=0;
    #pragma omp parallel num_threads(4)
    {
        #pragma omp single
        a++;
        #pragma omp critical
        b++;
    }
    printf("single: %d | critical: %d\nsingle runs once, critical is run once per
thread\n", a, b);
}

void critical() {
    printf("\nd. Critical:\n");

    int i; int max; int a[N];
    for (i = 0; i < N; i++) {
        a[i] = rand();
        printf(
            "a[%d] = %d\tthread no %d\n",
            i,
            a[i],
            omp_get_num_threads()
        );
    }
    max = a[0];
    #pragma omp parallel for
    for (i = 1; i < N; i++) {
        if (a[i] > max) {
            #pragma omp critical
            {
                if (a[i] > max) max = a[i];
            }
        }
    }
    printf("\nmax = %d\t%d threads\n", max, omp_get_num_threads());
}

void ordered() {
```

```
    printf("\ne. Ordered:\nwithout ordered:\n");

    int i = 0;
    int n = 10;

    #pragma omp parallel shared(n) private(i)
    {
        #pragma omp for
        for (i = 0; i < n; i++) {
            printf("thread %d at index %d\n", omp_get_thread_num(), i);
        }
    }

    printf("\nwith ordered:\n");

    #pragma omp parallel shared(n) private(i)
    {
        #pragma omp for ordered
        for (i = 0; i < n; i++) {
            #pragma omp ordered
            {
                printf("thread %d at index %d\n", omp_get_thread_num(), i);
            }
        }
    }
}

int main() {
    barrier();
    master();
    single();
    critical();
    ordered();

    printf("\n");
    return 0;
}
```

Output:

```
gcc q1.c -o q1.out -fopenmp
./q1.out

a. Barrier:
sum: 18404

b. Master
without 'master':
hello, from thread 1
hello, from thread 2
hello, from thread 5
hello, from thread 6
hello, from thread 3
hello, from thread 0
hello, from thread 4
hello, from thread 7

with master:
hello, from thread 0
```

```
c. Single:
single: 1 | critical: 4
single runs once, critical is run once per thread

d. Critical:
a[0] = 184794536        thread no 1
a[1] = 388450127        thread no 1
a[2] = 915736906        thread no 1
a[3] = 101072999        thread no 1
a[4] = 659067697        thread no 1
a[5] = 1777483316       thread no 1
a[6] = 1906889260       thread no 1
a[7] = 113766839        thread no 1
a[8] = 111387570        thread no 1
a[9] = 1883555567       thread no 1

max = 1906889260        1 threads
```

```
e. Ordered:
without ordered:
thread 7 at index 9
thread 1 at index 2
thread 1 at index 3
thread 3 at index 5
thread 4 at index 6
thread 0 at index 0
thread 0 at index 1
thread 5 at index 7
thread 6 at index 8
thread 2 at index 4
```

```
with ordered:
thread 0 at index 0
thread 0 at index 1
thread 1 at index 2
thread 1 at index 3
thread 2 at index 4
thread 3 at index 5
thread 4 at index 6
thread 5 at index 7
thread 6 at index 8
thread 7 at index 9
```

Q2

Code:

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

#define MAX_SLEEP 10

omp_lock_t lock;
int cr = 0;

void reader(int i) {
    #pragma omp critical
    {
        cr++;
        if (cr == 1) {
            omp_set_lock(&lock);
            printf("lock set by reader %d \n", i);
        }
    }

    printf(
        "reader %d (on thread %d) is reading\n",
        i,
        omp_get_thread_num()
    );

    sleep(rand() % MAX_SLEEP);

    #pragma omp critical
    {
        cr--;
        if (cr == 0) {
            omp_unset_lock(&lock);
            printf("lock unset by reader %d\n", i);
        }
    }
}

void writer(int i) {
    omp_set_lock(&lock);
    printf("lock set by writer %d\n", i);

    printf(
        "writer %d (on thread %d) is writing\n",
        i,
        omp_get_thread_num()
    );

    sleep(rand() % MAX_SLEEP);
```

```
    omp_unset_lock(&lock);
    printf("lock unset by writer %d\n", i);
}

int main(int argc, char *argv[]) {
    printf("Readers-writers in parallel\n");

    srand(clock());
    omp_init_lock(&lock);

    #pragma omp parallel sections num_threads(8)
    {
        #pragma omp section
        {
            writer(0);
        }
        #pragma omp section
        {
            reader(0);
        }
        #pragma omp section
        {
            reader(1);
        }
        #pragma omp section
        {
            reader(2);
        }
        #pragma omp section
        {
            writer(1);
        }
        #pragma omp section
        {
            writer(2);
        }
        #pragma omp section
        {
            reader(3);
        }
        #pragma omp section
        {
            reader(4);
        }
    }

    return 0;
}
```

Output:

```
gcc q2.c -o q2.out -fopenmp
./q2.out
Readers-writers in parallel
lock set by writer 0
writer 0 (on thread 2) is writing
lock unset by writer 0
lock set by reader 1
reader 1 (on thread 3) is reading
reader 0 (on thread 0) is reading
reader 3 (on thread 1) is reading
reader 2 (on thread 5) is reading
reader 4 (on thread 7) is reading
lock unset by reader 1
lock set by writer 2
writer 2 (on thread 4) is writing
lock unset by writer 2
lock set by writer 1
writer 1 (on thread 6) is writing
lock unset by writer 1
```

Lab 06: Reductions

Q1. Write a program in OpenMP to find out the largest number in an array of 1000000 randomly generated numbers from 1 to 100000 using reduction clause. Compare the versions of serial, parallel for and reduction clause.

Code:

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

#define N 1000000
#define MAX 100000

void findMinSerial(int ar[]) {
    int max = 0;
    for (int i = 0; i < N; i++) if (ar[i] > max) max = ar[i];

    printf("%d is the maximum element\n", max);
}

void findMinParallel(int ar[]) {
    int max = 0, i;

    #pragma omp parallel for shared(ar, max) private(i)
    for (i = 0; i < N; i++) {
        if (ar[i] > max) {
            #pragma omp critical
            {
                max = ar[i];
            }
        }
    }
    printf("%d is the maximum element\n", max);
}

void findMinReduction(int ar[]) {
    int max = 0, i;

    #pragma omp parallel for shared(ar) private(i) reduction(max: max)
    for (i = 0; i < N; i++) if (ar[i] > max) max = ar[i];

    printf("%d is the maximum element\n", max);
}

int main() {
```

```
    int ar[N];

    srand(clock());

    for (int i = 0; i < N; i++) ar[i] = rand() % MAX;

    double t = omp_get_wtime();
    findMinSerial(ar);
    t = omp_get_wtime() - t;
    printf("Serial execution took %fs\n", t);

    t = omp_get_wtime();
    findMinParallel(ar);
    t = omp_get_wtime() - t;
    printf("Parallel execution using a critical section took %fs\n", t);

    t = omp_get_wtime();
    findMinReduction(ar);
    t = omp_get_wtime() - t;
    printf("Parallel execution using a reduction took %fs\n", t);

    return 0;
}
```

Output:

```
gcc q1.c -o q1.out -fopenmp
./q1.out
9999 is the maximum element
Serial execution took 0.002375s
9999 is the maximum element
Parallel execution using a critical section took 0.001397s
9999 is the maximum element
Parallel execution using a reduction took 0.001132s
```

Q2. Write a program in OpenMP to find out the standard deviation of 1000000 randomly generated numbers using reduction clause. Document the development versions of serial, parallel for and reduction clause.

<u>Code:</u>

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

#define N 1000000
#define MAX 100000

void sdSerial(int ar[]) {
    double mean = 0;
    for (int i = 0; i < N; i++) mean += ar[i];
    mean /= N;

    double runningSum = 0;
    for (int i = 0; i < N; i++) {
        runningSum += pow(ar[i] - mean, 2);
    }

    double sd = sqrt(runningSum / N);
    printf("sd = %f\n", sd);
}

void sdParallel(int ar[]) {
    double mean = 0;
    int i = 0;

    #pragma omp parallel for shared(ar, mean) private(i)
    for (i = 0; i < N; i++) {
        #pragma omp critical
        {
            mean += ar[i];
        }
    }
    mean /= N;

    double runningSum = 0;

    #pragma omp parallel for shared(runningSum, mean) private(i)
    for (i = 0; i < N; i++) {
        #pragma omp critical
        {
            runningSum += pow(ar[i] - mean, 2);
```

```
        }
    }

    double sd = sqrt(runningSum / N);
    printf("sd = %f\n", sd);
}

void sdReduction(int ar[]) {
    double mean = 0;
    int i = 0;

    #pragma omp parallel for shared(ar) private(i) reduction(+ : mean)
    for (i = 0; i < N; i++) mean += ar[i];
    mean /= N;

    double runningSum = 0;

    #pragma omp parallel for shared(ar, mean) private(i) reduction(+ : runningSum)
    for (i = 0; i < N; i++) runningSum += pow(ar[i] - mean, 2);

    double sd = sqrt(runningSum / N);
    printf("sd = %f\n", sd);
}

int main() {
    int ar[N];

    srand(clock());

    for (int i = 0; i < N; i++) ar[i] = rand() % MAX;

    double t = omp_get_wtime();
    sdSerial(ar);
    t = omp_get_wtime() - t;
    printf("Serial execution took %fs\n\n", t);

    t = omp_get_wtime();
    sdParallel(ar);
    t = omp_get_wtime() - t;
    printf("Parallel execution using a critical section took %fs\n\n", t);

    t = omp_get_wtime();
    sdReduction(ar);
    t = omp_get_wtime() - t;
    printf("Parallel execution using a reduction took %fs\n", t);

    return 0;
}
```

Output:

```
gcc q2.c -lm -o q2.out -fopenmp
./q2.out
sd = 28852.563893
Serial execution took 0.027027s

sd = 28852.563892
Parallel execution using a critical section took 0.361950s

sd = 28852.563892
Parallel execution using a reduction took 0.008412s
```

Q3. Write a multithreaded program using OpenMP to implement sequential and parallel version of the Monte Carlo algorithm for approximating Pi. Compare the results of sequential, loop-level parallelism and reduction clause with 10000000 samples.

<u>Code:</u>

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

#define N 100000
#define M 100000

void serialMonteCarloPi(double xSamples[], double ySamples[]) {
    int counter = 0;
    for (int i = 0; i < N; i++) {
        double x = xSamples[i];
        double y = ySamples[i];

        if (x * x + y * y < 1) counter++;
    }
    printf("pi = %f\n", 4.0 * (double) counter / (double) N);
}

void parallelMonteCarloPi(double xSamples[], double ySamples[]) {
    int counter = 0, i;
    double x, y;

    #pragma omp parallel for shared(xSamples, ySamples, counter) private(i, x, y)
    for (i = 0; i < N; i++) {
        x = xSamples[i];
        y = ySamples[i];

        if (x * x + y * y < 1) {
            #pragma omp critical
            {
                counter++;
            }
        }
    }
    printf("pi = %f\n", 4.0 * (double) counter / (double) N);
}

void reductionMonteCarloPi(double xSamples[], double ySamples[]) {
    int counter = 0, i;
    double x, y;
```

```
    #pragma omp parallel for shared(xSamples, ySamples) private(i, x, y) reduc-
tion(+: counter)
    for (i = 0; i < N; i++) {
        x = xSamples[i];
        y = ySamples[i];

        if (x * x + y * y < 1) counter += 1;
    }
    printf("pi = %f\n", 4.0 * (double) counter / (double) N);
}

int main() {
    srand(clock());

    double xSamples[N], ySamples[N];
    for (int i = 0; i < N; i++) {
        xSamples[i] = (double)(rand() % M) / M;
        ySamples[i] = (double)(rand() % M) / M;
    }

    double t = omp_get_wtime();
    serialMonteCarloPi(xSamples, ySamples);
    t = omp_get_wtime() - t;
    printf("Serial execution took %fs\n\n", t);

    t = omp_get_wtime();
    parallelMonteCarloPi(xSamples, ySamples);
    t = omp_get_wtime() - t;
    printf("Parallel execution using a critical section took %fs\n\n", t);

    t = omp_get_wtime();
    reductionMonteCarloPi(xSamples, ySamples);
    t = omp_get_wtime() - t;
    printf("Parallel execution using a reduction took %fs\n", t);

}
```

Output:

```
gcc q3.c -lm -o q3.out -fopenmp
./q3.out
pi = 3.147320
Serial execution took 0.001995s

pi = 3.147320
Parallel execution using a critical section took 0.021563s

pi = 3.147320
Parallel execution using a reduction took 0.000500s
┌─(~/vit/CSE4001 Parallel-and Distributed-Computing ETLP/La
```

Lab 07: Profiling

Matrix Multiplication

Nested parallelism is not supported in ompP, so the ‘collapse’ keyword has been used.

Code:

```
#include <stdio.h>

#include <stdlib.h>
#include <omp.h>


typedef int ** matrix;
#define N 1100 // the size of the matrices


void multiply(matrix A, matrix B, matrix ans, int numThreads) {
int i = 0, j = 0, k = 0;
#pragma omp parallel shared(A, B, ans) private(i, j, k) num_threads(numThreads)
{
#pragma omp for collapse(3)
for (i = 0; i < N; i++) {
for (j = 0; j < N; j++) {
for (k = 0; k < N; k++) {
ans[i][j] += A[i][k] * B[k][j];
}
}
}
}
}


int main(int argc, char *argv[]) {
int numThreads = atoi(argv[1]);
matrix A, B, C;


A = (int **)calloc(N, sizeof(int*));
B = (int **)calloc(N, sizeof(int*));
C = (int **)calloc(N, sizeof(int*));


for (int i = 0; i < N; i++) {
A[i] = (int *) calloc(N, sizeof(int));
B[i] = (int *) calloc(N, sizeof(int));
```

```
C[i] = (int *) calloc(N, sizeof(int));
for (int j = 0; j < N; j++) {
A[i][j] = rand() % 10;
B[i][j] = rand() % 10;
C[i][j] = 0;
}
}

double t = omp_get_wtime();

multiply(A, B, C, numThreads);

t = omp_get_wtime() - t;

printf("took %f seconds\n", t);

for (int i = 0; i < N; i++) {
free(A[i]);
free(B[i]);
}
free(A);
free(B);
free(C);

printf("\n");
return 0;
}
```

OmpP Profiler Output:

```
----------------------------------------------------------------------
---------ompP General Information----------------------------------
----------------------------------------------------------------------
Start Date      : Thu Sep 15 21:31:53 2022
End Date        : Thu Sep 15 21:31:56 2022
Duration        : 2.94 sec
Application Name: unknown
Type of Report  : final
User Time       : 11.24 sec
System Time     : 0.00 sec
Max Threads     : 8
ompP Version    : 0.8.99
ompP Build Date : Sep 13 2022 10:59:18
PAPI Support    : not available

----------------------------------------------------------------------
---------ompP Region Overview----------------------------------------
----------------------------------------------------------------------
PARALLEL: 1 region:
  * R00001 q1.c (10-20)

LOOP: 1 region:
  * R00002 q1.c (12-19)
```

```
----------------------------------------------------------------------
----      ompP Flat Region Profile (inclusive data)      ---------------
----------------------------------------------------------------------
R00001 q1.c (10-20) PARALLEL
 TID      execT      execC      bodyT   exitBarT   startupT   shutdwnT      taskT
   0       2.94          1       2.94       0.00       0.00       0.00       0.00
   1       2.94          1       2.94       0.00       0.00       0.00       0.00
   2       2.94          1       2.94       0.00       0.00       0.00       0.00
   3       2.94          1       2.94       0.00       0.00       0.00       0.00
   4       0.00          0       0.00       0.00       0.00       0.00       0.00
   5       0.00          0       0.00       0.00       0.00       0.00       0.00
   6       0.00          0       0.00       0.00       0.00       0.00       0.00
   7       0.00          0       0.00       0.00       0.00       0.00       0.00
 SUM      11.77          4      11.77       0.00       0.00       0.00       0.00

R00002 q1.c (12-19) LOOP
 TID      execT      execC      bodyT   exitBarT      taskT
   0       2.94          1       2.87       0.08       0.00
   1       2.94          1       2.80       0.14       0.00
   2       2.94          1       2.74       0.20       0.00
   3       2.94          1       2.94       0.00       0.00
   4       0.00          0       0.00       0.00       0.00
   5       0.00          0       0.00       0.00       0.00
   6       0.00          0       0.00       0.00       0.00
   7       0.00          0       0.00       0.00       0.00
 SUM      11.77          4      11.34       0.42       0.00
```

```
----      ompP Callgraph Region Profiles (incl./excl. data)      -------
----------------------------------------------------------------------

[*00] unknown
[+01] R00001 q1.c (10-20) PARALLEL
 TID      execT      execC    bodyT/I    bodyT/E   exitBarT   startupT   shutdwnT      taskT
   0       2.94          1       2.94       0.00       0.00       0.00       0.00       0.00
   1       2.94          1       2.94       0.00       0.00       0.00       0.00       0.00
   2       2.94          1       2.94       0.00       0.00       0.00       0.00       0.00
   3       2.94          1       2.94       0.00       0.00       0.00       0.00       0.00
   4       0.00          0       0.00       0.00       0.00       0.00       0.00       0.00
   5       0.00          0       0.00       0.00       0.00       0.00       0.00       0.00
   6       0.00          0       0.00       0.00       0.00       0.00       0.00       0.00
   7       0.00          0       0.00       0.00       0.00       0.00       0.00       0.00
 SUM      11.77          4      11.77       0.00       0.00       0.00       0.00       0.00

[*00] unknown
[+01] R00001 q1.c (10-20) PARALLEL
[=02] R00002 q1.c (12-19) LOOP
 TID      execT      execC    bodyT/I    bodyT/E   exitBarT      taskT
   0       2.94          1       2.87       2.87       0.08       0.00
   1       2.94          1       2.80       2.80       0.14       0.00
   2       2.94          1       2.74       2.74       0.20       0.00
   3       2.94          1       2.94       2.94       0.00       0.00
   4       0.00          0       0.00       0.00       0.00       0.00
   5       0.00          0       0.00       0.00       0.00       0.00
   6       0.00          0       0.00       0.00       0.00       0.00
   7       0.00          0       0.00       0.00       0.00       0.00
 SUM      11.77          4      11.34      11.34       0.42       0.00
```

```
----------------------------------------------------------------------
----        ompP Overhead Analysis Report        ---------------------------
----------------------------------------------------------------------
Total runtime (wallclock)   : 2.94 sec [8 threads]
Number of parallel regions  : 1
Parallel coverage           : 2.94 sec (99.97%)

Parallel regions sorted by wallclock time:
        Type                          Location      Wallclock (%)
R00001  PARALLEL                    q1.c (10-20)      2.94 (99.97)
                                                SUM   2.94 (99.97)

Overheads wrt. each individual parallel region:
        Total       Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)   +    Mgmt (%)
R00001   23.54    0.42 ( 1.80)   0.00 ( 0.00)   0.42 ( 1.80)   0.00 ( 0.00)   0.00 ( 0.00)

Overheads wrt. whole program:
        Total       Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)   +    Mgmt (%)
R00001   23.54    0.42 ( 1.80)   0.00 ( 0.00)   0.42 ( 1.80)   0.00 ( 0.00)   0.00 ( 0.00)
   SUM   23.54    0.42 ( 1.80)   0.00 ( 0.00)   0.42 ( 1.80)   0.00 ( 0.00)   0.00 ( 0.00)
```

Lab 8: Simple MPI Program

Code:

ex8.c > main(int, char * [])

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    double starttime = MPI_Wtime(), endtime;
    int procRank, procNum;
    int message;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &procNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);

    if (procRank == 0) {
        message = 1550;
        MPI_Send(&message, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("proc %d sent message %d to id:1\n", procRank, message);
    } else if (procRank == 1) {
        MPI_Recv(&message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Recieved %d message by proc%d\n",message, procRank);
    }
    endtime = MPI_Wtime();
    MPI_Finalize();
    printf("proc %d took %f seconds\n", procRank, endtime - starttime); return 0;
}
```

Output:

```
┌─(~/vit/CSE4001_Parallel-and_Dis
└─(18:17:00)──> make ex8
mpicc ex8.c -o ex8.out
mpiexec -np 2 ex8.out
Recieved 1550 message by proc1
proc 0 sent message 1550 to id:1
proc 1 took 0.633166 seconds
proc 0 took 0.636244 seconds
┌─(~/vit/CSE4001_Parallel-and_Dis
```

Lab 9: MPI Point to Point Communication

a:

Code:

ex9a.c > main()

```c
/*
a. Write a program in MPI to create two processes in two
    different machines. Process 0 pings Process 1 and
    awaits for return ping using Non-blocking message
    passing routines. Execute your code on MPI cluster.
*/

#include <stdio.h>
#include <mpi.h>

int main(){
    int rank, size;
    int tag, destination, count;
    int buffer;

    tag = 1234;
    destination = 1;
    count = 1;

    MPI_Status status;
    MPI_Request request = MPI_REQUEST_NULL;
    MPI_Init(NULL,NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        buffer=10;
        MPI_Isend(&buffer, count, MPI_INT, destination, tag,
            MPI_COMM_WORLD, &request);
    }
```

```c
    if (rank == 0) {
        buffer=10;
        MPI_Isend(&buffer, count, MPI_INT, destination, tag,
        MPI_COMM_WORLD, &request);
    }
    if (rank == destination) {
        MPI_Irecv(&buffer, count, MPI_INT, 0, tag,
        MPI_COMM_WORLD, &request);
    }
    MPI_Wait(&request, &status);
    if (rank == 0) {
        printf("proc %d sent %d\n", rank, buffer);

    }else if (rank == destination) {
        printf("proc %d got %d\n", rank, buffer);
    }
    MPI_Finalize();
    return 0;
}
```

-

Output:

```
┌─ ~/vit/CSE4001_Parallel-ar
└─(18:25:51)──> make ex9a
mpicc ex9a.c -o ex9a.out
mpiexec -np 2 ex9a.out
proc 0 sent 10
proc 1 got 10
┌─ ~/vit/CSE4001_Parallel-ar
└─(18:28:26)──>
```

b:

Code:

ex9b.c > main(int, char * [])

```c
/*
b. Write a program in MPI to create 10 tasks.
    Construct a ring topology to exchange message to its
    nearest neighbour in the ring using blocking massage
    passing routines. Execute your code on MPI cluster.
*/

#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int myid, numprocs, left, right;
    int buffer = 1550, buffer2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    MPI_Request request[2];
    MPI_Status status;

    right = (myid+ 1) % numprocs;

    left = myid-1; if (left < 0)
    left = numprocs-1;

    MPI_Isend(&buffer, 1, MPI_INT, right, 123,
    MPI_COMM_WORLD, &request[0]);
    printf("proc %d to proc %d sent number: %d\n", myid,
    right, buffer);
```

```c
    MPI_Irecv(&buffer2, 1, MPI_INT, left, 123,
    MPI_COMM_WORLD, &request[1]);
    MPI_Wait(&request[0], &status);
    MPI_Wait(&request[1], &status);
    printf("proc %d from proc %d, received number: %d\n",myid,
    left,buffer2);
    MPI_Finalize();
    return 0;
}
```

Output:

```
┌─(~/vit/CSE4001_Parallel-and_Distributed-Co
└─(18:37:11)──> make ex9b
mpicc ex9b.c -o ex9b.out
mpiexec -np 3 ex9b.out
proc 0 to proc 1 sent number: 1550
proc 1 to proc 2 sent number: 1550
proc 1 from proc 0, received number: 1550
proc 2 to proc 0 sent number: 1550
proc 0 from proc 2, received number: 1550
proc 2 from proc 1, received number: 1550
┌─(~/vit/CSE4001_Parallel-and_Distributed-Co
└─(18:37:25)──>
```

Lab 10: MPI Collective Communication

a:

Code:

ex10a.c > main(int, char * [])

```c
/*
a. Write a program in MPI to generate 'n' random float
    numbers and send 'k' of those to each node and make
    them compute the average and send it back to the
    master which computes the average of those averages.
*/
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char *argv[]){
    int procRank, procNum;
    int n, k, numSend, numRecv;
    double avg;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &procNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);

    k = 4;
    n = k * procNum;

    if (procRank == 0) {
        for (int dest = 1; dest < procNum; dest++){
            for (int j = 0; j < k; j++) {
                numSend = rand() % 100;
                printf("send: P%d(%d)->P%d\n", procRank, numSend, dest);
                MPI_Send(&numSend, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
            }
        }
```

```
    } else {
        int count = 0, sum = 0, from = 0;
        for (int i = 0; i < k; i++){
            MPI_Recv(&numRecv, 1, MPI_INT, from, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            count++;
            sum += numRecv;
            printf("recv: P0->P%d(%d) [%d/%d]\n", procRank, numRecv, count, k);
        }
        avg = (double)sum / count;
        printf("calc: P%d average=%3f\n", procRank, avg);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if (procRank != 0) {
        MPI_Send(&avg, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
        printf("P%d(%f)->P0\n", procRank, avg);
    } else {
        int count = 0;
        double sum = 0.0;
        for (int i = 1; i < procNum; i++) {
            MPI_Recv(&avg, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("P%d->P0(%f) [%d/%d]\n", i, avg, count, procNum-1);
            count++;
            sum += avg;
```

```
        }
        printf("\nFinal average = %f\n", sum / count);
    }

    MPI_Finalize();
    return 0;
}
```

Output:

```
mpicc ex10a.c -o ex10a.out
mpiexec -np 4 ex10a.out
send: P0(83)->P1
send: P0(86)->P1
send: P0(77)->P1
send: P0(15)->P1
send: P0(93)->P2
send: P0(35)->P2
send: P0(86)->P2
send: P0(92)->P2
send: P0(49)->P3
send: P0(21)->P3
send: P0(62)->P3
send: P0(27)->P3
recv: P0->P1(83) [1/4]
recv: P0->P1(86) [2/4]
recv: P0->P1(77) [3/4]
recv: P0->P1(15) [4/4]
calc: P1_average=65.250000
recv: P0->P2(93) [1/4]
recv: P0->P2(35) [2/4]
recv: P0->P2(86) [3/4]
recv: P0->P2(92) [4/4]
calc: P2_average=76.500000
recv: P0->P3(49) [1/4]
recv: P0->P3(21) [2/4]
recv: P0->P3(62) [3/4]
recv: P0->P3(27) [4/4]
calc: P3_average=39.750000
P3(39.750000)->P0
P2(76.500000)->P0
P1->P0(65.250000) [0/3]
P2->P0(76.500000) [1/3]
P3->P0(39.750000) [2/3]

Final average = 60.500000
P1(65.250000)->P0
```

b:

Code:

ex10b.c > main(int, char * [])

```c
/*
b. Write a MPI program to compute PI using “dartboard”
    technique for 1000 rounds by using reduction
    collective computation.
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

double getRandom() {
    return (double)(rand() % 1000000) / 500000 - 1;
}

int main(int argc, char *argv[]) {
    srand(clock());
    int procRank, procNum;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &procNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);

    int count = 0, total = 0;
    int finalCount = 0, finalTotal = 0;

    if (procRank != 0) {
        for (int i = 0; i < 10000; i++) {
            double x = getRandom();
            double y = getRandom();
            if ((x*x)+(y*y)< 1) count++;
            total++;
```

```c
            if ((x*x)+(y*y)< 1) count++;
            total++;
        }
        printf("P%d: %d/%d\n", procRank, count, total);
    }
    MPI_Reduce(&count, &finalCount, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&total, &finalTotal, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (procRank == 0) {
        printf(
            "Throws inside circle = %d\nTotal throws = %d\npi = %f\n",
            finalCount,
            finalTotal,
            (double)finalCount/finalTotal * 4
        );
    }
    MPI_Finalize();
}
```

Output:

```
┌─(~/vit/CSE4001_Parallel-and_D
└─(20:00:00)──> make ex10b
mpicc ex10b.c -o ex10b.out
mpiexec -np 4 ex10b.out
P2: 7837/10000
P3: 7886/10000
P1: 7895/10000
Throws inside circle = 23618
Total throws = 30000
pi = 3.149067
┌─(~/vit/CSE4001_Parallel-and_D
```

c:

Code:

```
/*
c. Write a MPI program to perform matrix multiplication
(1000x1000) using scatter and gather routines.
*/

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <mpi.h>
#include <stdio.h>

#define SIZE 8

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

void fill_matrix(int m[SIZE][SIZE])
{
static int n=0;
int i, j;
for (i=0; i<SIZE; i++)
for (j=0; j<SIZE; j++)
m[i][j] = n++;
}

void print_matrix(int m[SIZE][SIZE])
{
int i, j = 0;
for (i=0; i<SIZE; i++) {
printf("\n\t| ");
for (j=0; j<SIZE; j++)
printf("%2d ", m[i][j]);
printf("|");
}
}

int main(int argc, char *argv[])
{
int myrank, P, from, to, i, j, k;
int tag = 666; /* any value will do */
MPI_Status status;
MPI_Init (&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* who am i */
```

```
MPI_Comm_size(MPI_COMM_WORLD, &P); /* number of processors */

/* Just to use the simple variants of MPI_Gather and MPI_Scatter we */
/* impose that SIZE is divisible by P. By using the vector versions, */
/* (MPI_Gatherv and MPI_Scatterv) it is easy to drop this restriction. */

if (SIZE%P!=0) {
if (myrank==0) printf("Matrix size not divisible by number of processors\n");
MPI_Finalize();
exit(-1);
}

from = myrank * SIZE/P;
to = (myrank+1) * SIZE/P;

/* Process 0 fills the input matrices and broadcasts them to the rest */
/* (actually, only the relevant stripe of A is sent to each process) */

if (myrank==0) {
fill_matrix(A);
fill_matrix(B);
}

MPI_Bcast (B, SIZE*SIZE, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter (A, SIZE*SIZE/P, MPI_INT, A[from], SIZE*SIZE/P, MPI_INT, 0, MPI_COMM_WORLD);

printf("computing slice %d (from row %d to %d)\n", myrank, from, to-1);
for (i=from; i<to; i++)
for (j=0; j<SIZE; j++) {
C[i][j]=0;
for (k=0; k<SIZE; k++)
C[i][j] += A[i][k]*B[k][j];
}

MPI_Gather (C[from], SIZE*SIZE/P, MPI_INT, C, SIZE*SIZE/P, MPI_INT, 0, MPI_COMM_WORLD);

if (myrank==0) {
printf("\n\n");
print_matrix(A);
printf("\n\n\t * \n");
print_matrix(B);
printf("\n\n\t = \n");
print_matrix(C);
printf("\n\n");
}
```

```
MPI_Finalize();
return 0;
}
```

Output:

```
┌─(~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/MPI-
└─(20:00:09)──> make ex10c
mpicc ex10c.c -o ex10c.out
mpiexec -np 4 ex10c.out
computing slice 0 (from row 0 to 1)



        |  0  1  2  3  4  5  6  7 |
        |  8  9 10 11 12 13 14 15 |
        | 16 17 18 19 20 21 22 23 |
        | 24 25 26 27 28 29 30 31 |
        | 32 33 34 35 36 37 38 39 |
computing slice 1 (from row 2 to 3)
computing slice 2 (from row 4 to 5)
computing slice 3 (from row 6 to 7)
        | 40 41 42 43 44 45 46 47 |
        | 48 49 50 51 52 53 54 55 |
        | 56 57 58 59 60 61 62 63 |

              *

        | 64 65 66 67 68 69 70 71 |
        | 72 73 74 75 76 77 78 79 |
        | 80 81 82 83 84 85 86 87 |
        | 88 89 90 91 92 93 94 95 |
        | 96 97 98 99 100 101 102 103 |
        | 104 105 106 107 108 109 110 111 |
        | 112 113 114 115 116 117 118 119 |
        | 120 121 122 123 124 125 126 127 |

              =

        | 2912 2940 2968 2996 3024 3052 3080 3108 |
        | 8800 8892 8984 9076 9168 9260 9352 9444 |
        | 14688 14844 15000 15156 15312 15468 15624 15780 |
        | 20576 20796 21016 21236 21456 21676 21896 22116 |
        | 26464 26748 27032 27316 27600 27884 28168 28452 |
        | 32352 32700 33048 33396 33744 34092 34440 34788 |
        | 38240 38652 39064 39476 39888 40300 40712 41124 |
        | 44128 44604 45080 45556 46032 46508 46984 47460 |

┌─(~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/MPI-
└─(20:14:29)──>
```

Lab 11: CUDA Programming

Setup:

In [ ]:
```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
```

```
Reading package lists... Done
```

In [ ]:
```
!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-repo-ubuntu1604-9-2-local
!dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
!apt-get update
!apt-get install cuda-9.2
```

```
2022-10-18 04:41:54    https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-repo-u
```

In [ ]:
```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git
```

```
%load_ext nvcc_plugin
```

Code:

Print name and capability of gpu

In [ ]:
```
%%cu
#include<stdio.h>
#include<cuda.h>

int main()
{
    cudaDeviceProp p;
    int device_id;
    int major;
    int minor;


    cudaGetDevice(&device_id);
    cudaGetDeviceProperties(&p,device_id);

    major=p.major;
    minor=p.minor;

    printf("Name of GPU on your system is %s\n",p.name);

    printf("\n Compute Capability of a current GPU on your system is %d.%d",major,minor);

    return 0;
}
```

```
Name of GPU on your system is Tesla T4

 Compute Capability of a current GPU on your system is 7.5
```

Hello World

In [ ]:
```
%%cu
#include <stdio.h>
#include<cuda.h>
__global__ void Hellokernel()
{

}
main()
{
Hellokernel << <1, 1 >> > ();
 printf("Hello World\n");
return 0;
}
```

```
Hello World
```

In [ ]:
```
%%cu
#include <stdio.h>
#include<cuda.h>
__global__ void add(int a, int b, int *c)
{
*c = a + b;
}
int main(void)
{
int c;
int *dev_c;
cudaMalloc((void**)&dev_c, sizeof(int));
add << <1, 1 >> > (2, 7, dev_c);
cudaMemcpy(&c, dev_c, sizeof(int),
cudaMemcpyDeviceToHost);
printf("2 + 7 = %d\n", c);
cudaFree(dev_c);
return 0;
}
```

```
2 + 7 = 9
```

```
    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = sin(i)*sin(i);
        h_b[i] = cos(i)*cos(i);
    }

    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;

    // Number of threads in each thread block
    blockSize = 1024;

    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    // Sum up vector c and print result divided by n, this should equal 1 within error
    double sum = 0;
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

```
final result: 1.000000
```

Lab 12: Java RMI

Code:

Adder.java

```java
import java.rmi.*;

public interface Adder extends Remote {
    public int add(int x, int y) throws RemoteException;
}
```

AdderRemote.java

```java
import java.rmi.*;
import java.rmi.server.*;

public class AdderRemote extends UnicastRemoteObject implements Adder {
    AdderRemote() throws RemoteException {
        super();
    }
    public int add(int x, int y) {
        return x + y;
    }
}
```

MyServer.java

```java
MyServer.java
import java.rmi.*;
import java.rmi.registry.*;

public class MyServer {
    public static void main(String args[]) {
        try {
            Adder stub = new AdderRemote();
            Naming.rebind("rmi://localhost:5000/lab12", stub);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

MyClient.java

```java
MyClient.java
import java.rmi.*;

public class MyClient {
    public static void main(String args[]) {
        try {
            Adder stub = (Adder)Naming.lookup("rmi://localhost:5000/lab12");
            System.out.println(stub.add(34, 4));
        } catch (Exception e) {}
    }
}
```

Steps:

1. Create all the files and compile them all:

```
~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/20221101 ──────────── sam
(11:26:08)──> /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.345.b01-1.fc36.x86_64/bin/javac *.java
```

2. Create the stub and skeleton for AdderRemote:

```
~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/20221101 ──────────── sam fedo
(11:26:12)──> /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.345.b01-1.fc36.x86_64/bin/rmic AdderRemote
```

3. Create the RMI Registry:

```
~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/20221101 ──────────── sam fedo
(11:26:22)──> /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.345.b01-1.fc36.x86_64/bin/rmigeistry 5000 &
[1] 303551
```

4. Run MyServer:

```
~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/20221101 ──────────── sam fed
(11:26:41)──> /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.345.b01-1.fc36.x86_64/bin/java MyServer
```

5. Run MyClient:

```
~/vit/CSE4001_Parallel-and_Distributed-Computing_ETLP/Lab/20221101 ──────────── sam fedo
(11:27:16)──> /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.345.b01-1.fc36.x86_64/bin/java MyClient
38
```