Practical No 5

PRN : 21510111

Github Repo : https://github.com/Sidd-77/hpc-lab/tree/main/as5

Q1. Write an OpenMP program such that, it should print the name of your family members, such that the names should come from different threads/cores. Also print the respective job id.

1.c

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

int main(int argc, char *argv[])
{
  char* arr[] = {"Siddharth", "Shrinivas", "Avishkar", "Rupali", "Shamrao", "Kamal"};
  #pragma omp parallel for
  for (int i=0; i<6; i++) {
    printf("%s from thread %d \n", arr[i], omp_get_thread_num());
  }
  return EXIT_SUCCESS;
}
```

Q2. Write an OpenMP program such that, it should print the sum of square of the thread id's. Also make sure that, each thread should print the square value of their thread id.

1.c 2.c

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

int main(int argc, char *argv[]) {
    int sum_of_squares = 0;

    #pragma omp parallel reduction(+:sum_of_squares)
    {
        int thread_id = omp_get_thread_num();
        int square = thread_id * thread_id;

        printf("Thread %d: square of thread ID = %d\n", thread_id, square);

        sum_of_squares += square;
    }

    printf("Sum of squares of thread IDs = %d\n", sum_of_squares);

    return EXIT_SUCCESS;
}

```

Q3. Consider a variable called "Aryabhatta" declared as 10 (i.e int Arbhatta=10).Write an OpenMP program which should print the result of multiplication of thread id and value of the above variable.

Note*: The variable "Aryabhatta" should be declared as private

Practical No 5

1.c 2.c 3.c

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

int main(int argc, char *argv[]) {
    int Aryabhatta = 10;

    #pragma omp parallel private(Aryabhatta)
    {
        Aryabhatta = 10; // setting value again cause it gives garbage value otherwise
        int thread_id = omp_get_thread_num();
        int result = thread_id * Aryabhatta;
        printf("Thread %d: result of thread ID * Aryabhatta = %d\n", thread_id, result);
    }

    return EXIT_SUCCESS;
}

```

Q4. Write an OpenMP program that calculates the partial sum of the first 20 natural numbers using parallelism. Each thread should compute a portion of the sum by iterating through a loop. Implement the program using the lastprivate clause to ensure that the final total sum is correctly computed and printed outside the parallel region.

Hint:

1.Utilize OpenMP directives to parallelize the summation process.

2.Ensure that each thread has its private copy of partial sum.

3.Use the lastprivate clause to assign the value of the last thread's partial sum to the final total sum after the parallel region.

1.c 2.c 3.c 4.c

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

int main(int argc, char *argv[]) {
    int total_sum = 0;

    #pragma omp parallel reduction(+:total_sum)
    {
        int partial_sum = 0;

        #pragma omp for
        for (int i = 1; i <= 20; i++) {
            partial_sum += i;
            printf("%d by thread %d\n", partial_sum, omp_get_thread_num());
        }

        #pragma omp lastprivate(partial_sum)
        {
            total_sum = partial_sum;
        }
    }

    printf("Total sum of the first 20 natural numbers = %d\n", total_sum);

    return EXIT_SUCCESS;
}

```

Q5. Consider a scenario where you have to parallelize a program that performs matrix multiplication using OpenMP. Your task is to implement parallelization using both static and dynamic scheduling, and compare the execution time of each approach.

Note*:

- Implement a serial version of matrix multiplication in C/C++.
- Parallelize the matrix multiplication using OpenMP with static scheduling.
- Parallelize the matrix multiplication using OpenMP with dynamic scheduling.
- Measure the execution time of each parallelized version for various matrix sizes.
- Compare the execution times and discuss the advantages and disadvantages of static and dynamic scheduling in this context.

Practical No 5

1.c 2.c 3.c 4.c 5.c

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

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

void parallelMatrixMultiplicationStatic(int **A, int **B, int **C, int N) {
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void parallelMatrixMultiplicationDynamic(int **A, int **B, int **C, int N) {
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Q6. Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads !

1.c 2.c 3.c 4.c 5

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

void printSeries(int startValue, int increment, int count) {
    for (int i = 0; i < count; i++) {
        printf("%d(%d) ", startValue + i * increment, startValue);
    }
    printf("\n \n");
}

int main() {
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            printSeries(2, 2, 20);
        }

        #pragma omp section
        {
            printSeries(4, 4, 20);
        }
    }

    return 0;
}

```

Q7. Consider a scenario where you have a shared variable total_sum that needs to be updated concurrently by multiple threads in a parallel program. However, concurrent updates to this variable can result in data races and incorrect results. Your task is to modify the program to ensure correct synchronization using OpenMP's critical and atomic constructs.

Note*:

- Implement a simple parallel program in C that initializes an array of integers and calculates the sum of its elements concurrently using OpenMP.
- Identify potential issues with concurrent updates to the total_sum variable in the parallelized version of the program.
- Modify the program to use OpenMP's critical/atomic directive to ensure synchronized access to the total_sum variable.
- Measure and compare the performance of synchronized versions against the unsynchronized implementation.

```
*[main][~/acad/hpc_lab/as5]$ gcc -fopenmp 7.c -o 7 && ./7
Serial Total Sum: 10000
Serial Time: 0.000033 seconds
Parallel Total Sum (with critical): 10000
Parallel Time (with critical): 0.005101 seconds
*[main][~/acad/hpc_lab/as5]$ gcc -fopenmp 7.c -o 7 && ./7
Serial Total Sum: 50000
Serial Time: 0.000160 seconds
Parallel Total Sum (with critical): 50000
Parallel Time (with critical): 0.040970 seconds
*[main][~/acad/hpc_lab/as5]$
```

6.c 7.c

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

void parallel(int N) {
    int *array = (int *)malloc(N * sizeof(int));
    long long total_sum = 0;

    for (int i = 0; i < N; i++) {
        array[i] = 1;
    }

  clock_t start = clock();

  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    #pragma omp critical
    {
      total_sum += array[i];
    }
  }

  clock_t end = clock();

  double criticalTime = (double)(end - start) / CLOCKS_PER_SEC;
  printf("Parallel Total Sum (with critical): %lld\n", total_sum);
  printf("Parallel Time (with critical): %f seconds\n", criticalTime);

  free(array);
  return;
}

void serial(int N) {
  int *array = (int *)malloc(N * sizeof(int));
  long long total_sum = 0;

  for (int i = 0; i < N; i++) {
    array[i] = 1;
  }

  clock_t start = clock();
  for (int i = 0; i < N; i++) {
    total_sum += array[i];
  }
  clock_t end = clock();

  double serialTime = (double)(end - start) / CLOCKS_PER_SEC;
  printf("Serial Total Sum: %lld\n", total_sum);
  printf("Serial Time: %f seconds\n", serialTime);

  free(array);
  return;
}
```

Q8. Consider a scenario where you have a large array of integers, and you need to find the sum of all its elements in parallel using OpenMP. The array is shared among multiple threads, and parallelism is needed to expedite the computation process. Your task is to write a parallel program that calculates the sum of all elements in the array using OpenMP's reduction clause.

Note*:

Practical No 5

- Implement a sequential version of the program that calculates the sum of all elements in the array without using any parallelism.
- Identify potential bottlenecks and limitations of the sequential implementation in handling large arrays efficiently.
- Modify the program to utilize OpenMP's reduction clause to parallelize the summation process across multiple threads.
- Test the program with different array sizes and thread counts to evaluate its scalability and performance.
- Discuss the advantages of using the reduction clause for parallel summation and its impact on program efficiency.

```
*[main][~/acad/hpc_lab/as5]$ gcc -fopenmp 8.c -o 8 && ./8
Sequential Total Sum: 100000
Sequential Time: 0.000320 seconds
Parallel Total Sum: 100000
Parallel Time: 0.000291 seconds
*[main][~/acad/hpc_lab/as5]$ gcc -fopenmp 8.c -o 8 && ./8
Sequential Total Sum: 1000000
Sequential Time: 0.003268 seconds
Parallel Total Sum: 1000000
Parallel Time: 0.099433 seconds
*[main][~/acad/hpc_lab/as5]$
```

Practical No 5

6.c 7.c 8.c

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

void parallel(int N) {
    int *array = (int *)malloc(N * sizeof(int));
    long long total_sum = 0;

    for (int i = 0; i < N; i++) {
        array[i] = 1; // Using 1 for simplicity
    }

    clock_t start = clock();
    #pragma omp parallel reduction(+:total_sum)
    {
        #pragma omp for
        for (int i = 0; i < N; i++) {
            total_sum += array[i];
        }
    }
    clock_t end = clock();

    double parallelTime = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Parallel Total Sum: %lld\n", total_sum);
    printf("Parallel Time: %f seconds\n", parallelTime);

    free(array);
    return;
}

void sequential(int N) {
    int *array = (int *)malloc(N * sizeof(int));
    long long total_sum = 0;

    for (int i = 0; i < N; i++) {
        array[i] = 1; // Using 1 for simplicity
    }

    clock_t start = clock();
    for (int i = 0; i < N; i++) {
        total_sum += array[i];
    }
    clock_t end = clock();

    double serialTime = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Sequential Total Sum: %lld\n", total_sum);
    printf("Sequential Time: %f seconds\n", serialTime);

    free(array);
    return;
}

```