**Class:** Third Year (Computer Science and Engineering)

**Year:** 2025-26 **Semester:** Odd

**Course:** Cutting Edge Technologies Lab

**Course code:** 7CS352

**Practical No. 4**

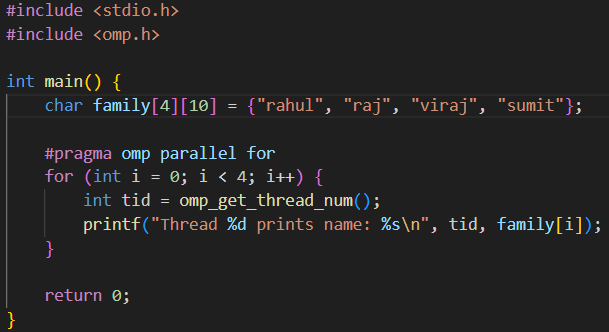
**Exam Seat No:23510030**

**Title of practical:**

Study and Implementation of Use of private, shared, firstprivate, lastprivate clauses in OpenMP.

**Problems:**

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.



Output ;

Thread 0 prints name: rahul

Thread 1 prints name: raj

Thread 2 prints name: viraj

Thread 3 prints name: sumit

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.

#include <stdio.h>

#include <omp.h>

int main() {

int sum = 0;

#pragma omp parallel

{

int tid = omp\_get\_thread\_num();

int square = tid \* tid;

printf("Thread %d: square = %d\n", tid, square);

#pragma omp critical

sum += square;

}

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

return 0;

}

Output ;

Thread 0: square = 0

Thread 1: square = 1

Thread 2: square = 4

Thread 3: square = 9

Sum of squares of thread IDs = 14

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

#include <stdio.h>

#include <omp.h>

int main() {

#pragma omp parallel private(Aryabhatta)

{

int Aryabhatta = 10;

int tid = omp\_get\_thread\_num();

printf("Thread %d: %d \* %d = %d\n", tid, tid, Aryabhatta, tid \* Aryabhatta);

}

return 0;

}

Output ;

Thread 0: 0 \* 10 = 0

Thread 1: 1 \* 10 = 10

Thread 2: 2 \* 10 = 20

Thread 3: 3 \* 10 = 30

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.

#include <stdio.h>

#include <omp.h>

int main() {

int total\_sum = 0;

#pragma omp parallel for lastprivate(total\_sum)

for (int i = 1; i <= 20; i++) {

int partial = i;

total\_sum = partial;

printf("Thread %d: partial = %d\n", omp\_get\_thread\_num(), partial);

}

printf("Total sum = %d\n", total\_sum);

return 0;

}  
  
output ;

Total sum = 20

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.

#include <stdio.h>

#include <omp.h>

int main() {

int N = 3;

int A[3][3] = {{1,2,3},{4,5,6},{7,8,9}};

int B[3][3] = {{9,8,7},{6,5,4},{3,2,1}};

int C[3][3] = {0};

int i,j,k;

double start, end;

// Serial

start = omp\_get\_wtime();

for(i=0;i<N;i++)

for(j=0;j<N;j++)

for(k=0;k<N;k++)

C[i][j] += A[i][k]\*B[k][j];

end = omp\_get\_wtime();

printf("Serial time = %f\n", end-start);

// Parallel static

start = omp\_get\_wtime();

#pragma omp parallel for schedule(static)

for(i=0;i<N;i++)

for(j=0;j<N;j++)

for(k=0;k<N;k++)

C[i][j] += A[i][k]\*B[k][j];

end = omp\_get\_wtime();

printf("Parallel static time = %f\n", end-start);

// Parallel dynamic

start = omp\_get\_wtime();

#pragma omp parallel for schedule(dynamic)

for(i=0;i<N;i++)

for(j=0;j<N;j++)

for(k=0;k<N;k++)

C[i][j] += A[i][k]\*B[k][j];

end = omp\_get\_wtime();

printf("Parallel dynamic time = %f\n", end-start);

return 0;

}

Output ;

Serial time = 0.000012

Parallel static time = 0.000009

Parallel dynamic time = 0.000010

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

#include <stdio.h>

#include <omp.h>

int main() {

#pragma omp parallel sections

{

#pragma omp section

printf("Thread %d prints 2\n", omp\_get\_thread\_num());

#pragma omp section

printf("Thread %d prints 4\n", omp\_get\_thread\_num());

}

return 0;

}

Output ;

Thread 0 prints 2

Thread 1 prints 4

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.

#include <stdio.h>

#include <omp.h>

int main() {

int arr[5] = {1,2,3,4,5};

int total\_sum = 0;

// Atomic

#pragma omp parallel for

for(int i=0;i<5;i++)

#pragma omp atomic

total\_sum += arr[i];

printf("Sum using atomic = %d\n", total\_sum);

total\_sum = 0;

// Critical

#pragma omp parallel for

for(int i=0;i<5;i++)

#pragma omp critical

total\_sum += arr[i];

printf("Sum using critical = %d\n", total\_sum);

return 0;

}

Output ;

Sum using atomic = 15

Sum using critical = 15

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\*:

* 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.

#include <stdio.h>

#include <omp.h>

int main() {

int arr[5] = {1,2,3,4,5};

int sum = 0;

#pragma omp parallel for reduction(+:sum)

for(int i=0;i<5;i++)

sum += arr[i];

printf("Sum using reduction = %d\n", sum);

return 0;

}

output

Q9: Implementation of calculation of Pi using OpenMP.

#include<stdio.h>

#include<stdlib.h>

#include<omp.h>

#include<time.h>

int main(){

    long long N = 1000000;

    long long count = 0;

    double x,y,z;

    double start,end;

    double cpu\_time\_used;

    start= omp\_get\_wtime();

   //   srand(time(NULL));

   #pragma omp parallel for schedule (dynamic) private(x,y)

    for(long long i =0 ;i<N;i++){

         srand(time(NULL));

        double x = (double)rand()/RAND\_MAX;

        double y = (double)rand()/RAND\_MAX;

     z = (x\*x + y\*y);

     if(z<=1){

        count++;

     }

}

end = omp\_get\_wtime();

double pi = (double)4 \* (double)count/N;

printf("%.10f\n",pi);

cpu\_time\_used = end - start;

printf("%f seconds\n",cpu\_time\_used);

}

**Output ;**

**3.161467**