



Veermata Jijabai Technological Institute, Mumbai 400019

Experiment No.: 04

Aim : Implementation using OpenMP.

- i. Fork Join model,
- ii. Producer Consumer problem,
- iii. Matrix Multiplication,
- iv. find prime number,
- v. Largest Element in an array and
- vi. Pi calculation

Name : Kiran K Patil

Enrolment No.: 211070904

Branch: Computer Engineering

Batch: IV

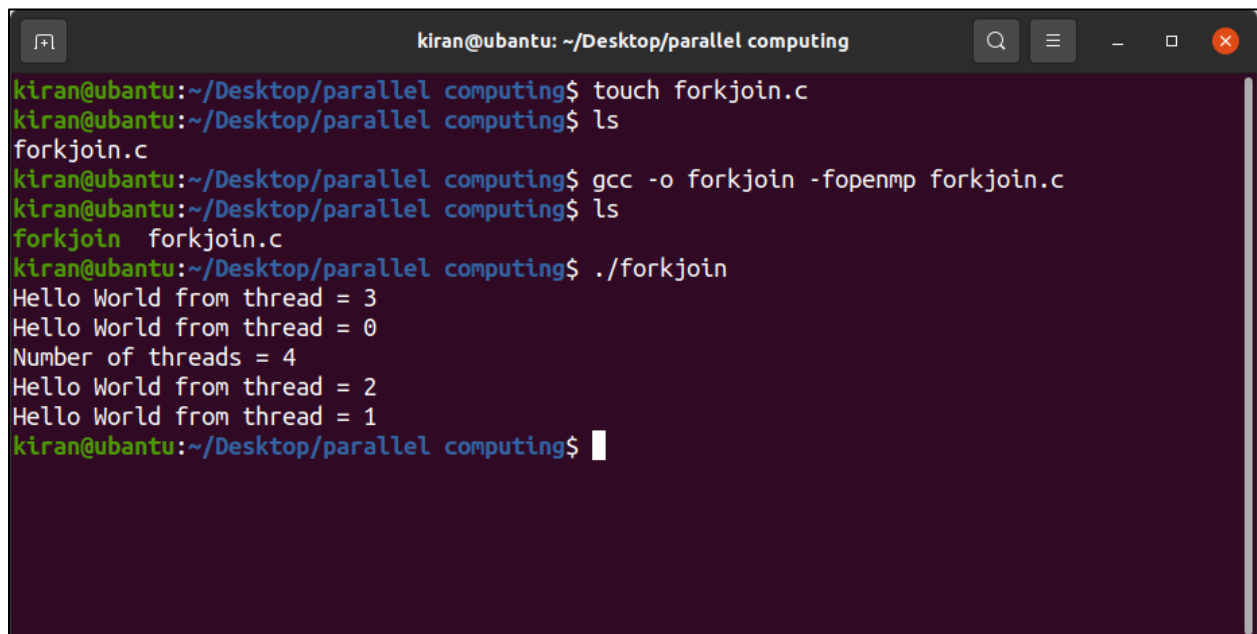
1. Fork Join Model :

OpenMP uses a fork-join model of parallel execution. When a thread encounters a parallel construct, the thread creates a team composed of itself and some additional (possibly zero) number of threads

Program :

```
#include<stdio.h>
#include<omp.h>
int main(){
int nthreads, tid;
/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
{
/* Obtain thread number */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
/* Only master thread does this */
if (tid == 0)
{
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
} /* All threads join master thread and disband */
}
```

Output:

A terminal window titled "kiran@ubuntu: ~/Desktop/parallel computing" displays the execution of a C program. The user enters commands to create the file, list files, compile with gcc, list files again, and run the program. The output shows five lines of "Hello World from thread = " followed by thread IDs 3, 0, 2, 1, and 4, and a line "Number of threads = 4".

```
kiran@ubuntu:~/Desktop/parallel computing$ touch forkjoin.c
kiran@ubuntu:~/Desktop/parallel computing$ ls
forkjoin.c
kiran@ubuntu:~/Desktop/parallel computing$ gcc -o forkjoin -fopenmp forkjoin.c
kiran@ubuntu:~/Desktop/parallel computing$ ls
forkjoin  forkjoin.c
kiran@ubuntu:~/Desktop/parallel computing$ ./forkjoin
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 1
kiran@ubuntu:~/Desktop/parallel computing$
```

2. Producer Consumer problem

In this program, the master thread will act as a producer while the other threads will wait until the master thread creates buffer, and once added the master notifies the threads using a shared variable and all other threads will consume the data.

Program :

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int i=0;
    int x=0;
    #pragma omp parallel shared(i)
    {
        if(omp_get_thread_num()==0)
        {
            printf("Master thread with Thread ID:%d\n", omp_get_thread_num());
            printf("Since it is the producer thread It is adding some data to be consumed by other consumer threads\n");
            i+=10;
            x=1;
        }

        else
        {
            while(x==0)
            printf("Waiting for buffer to be filled. Thread ID: %d\n",omp_get_thread_num());
            #pragma critical
            {
                if(i>0){
                    printf("Data is consumed by Consumer with Thread ID: %d\n",omp_get_thread_num());
                    i-=5;
                } else {
                    printf("Could not find any data for thread ID: %d\n",omp_get_thread_num());
                }
            }
        }
    }
}
```

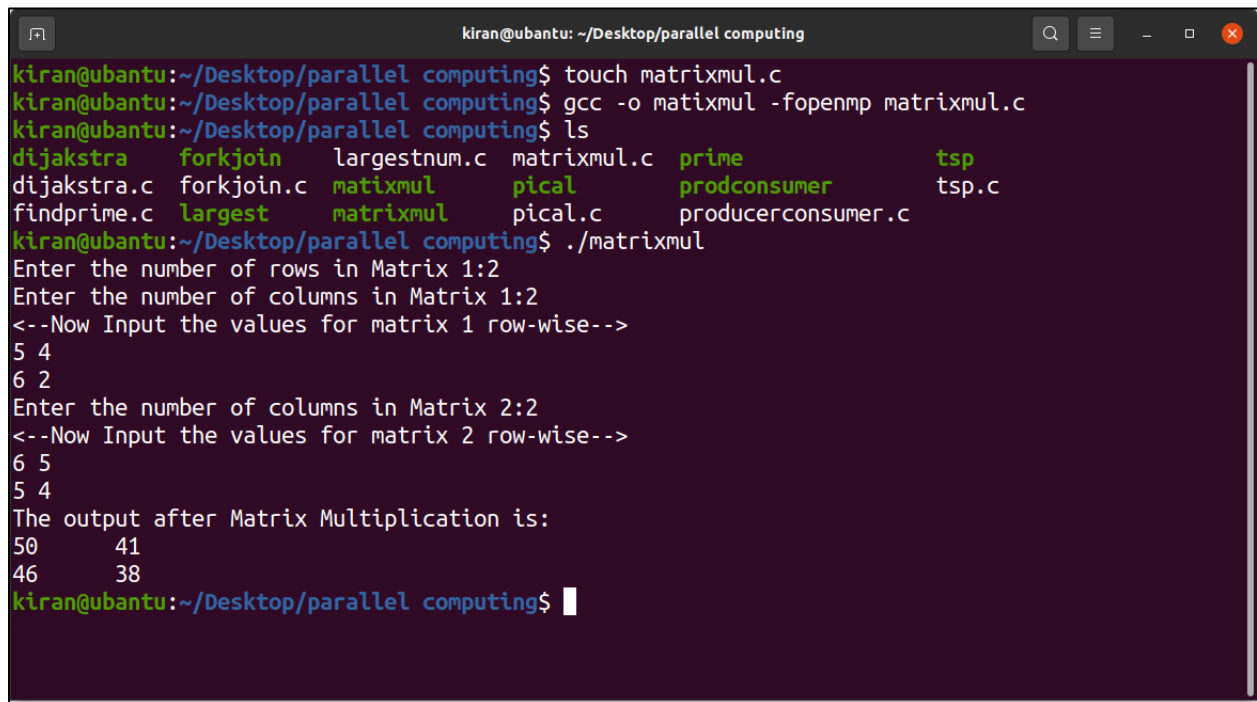

3. Matrix multiplication

Program :

```
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>
int main(){
int i,j,k,m,n,p;
printf("Enter the number of rows in Matrix 1:");
scanf("%d",&m);
int *matrixA[m];
printf("Enter the number of columns in Matrix 1:");
scanf("%d",&n);
for(i=0;i<m;i++){
matrixA[i] = (int *)malloc(n*sizeof(int));
}
printf("<--Now Input the values for matrix 1 row-wise-->\n");
for(i=0;i<m;i++){
for(j=0;j<n;j++){
scanf("%d",&matrixA[i][j]);
}
}
printf("Enter the number of columns in Matrix 2:");
scanf("%d",&p);
int *matrixB[n];
for(i=0;i<n;i++){
matrixB[i] = (int *)malloc(p*sizeof(int));
}
printf("<--Now Input the values for matrix 2 row-wise-->\n");
for(i=0;i<n;i++){
for(j=0;j<p;j++){
scanf("%d",&matrixB[i][j]);
}
}
int matrixC[m][p];
#pragma omp parallel private(i,j,k) shared(matrixA,matrixB,matrixC)
{
#pragma omp for schedule(static)
for (i=0; i<m; i=i+1){
for (j=0; j<p; j=j+1){
matrixC[i][j] = 0;
for (k=0; k<n; k=k+1){
matrixC[i][j]=(matrixC[i][j])+((matrixA[i][k])*(matrixB[k][j]));
}
}
}
```

```
}  
}  
}  
printf("The output after Matrix Multiplication is: \n");  
for(i=0;i<m;i++){  
    for(j=0;j<p;j++){  
        printf("%d \t",matrixC[i][j]);  
    }  
    printf("\n");  
}  
return 0;  
}
```

Output :



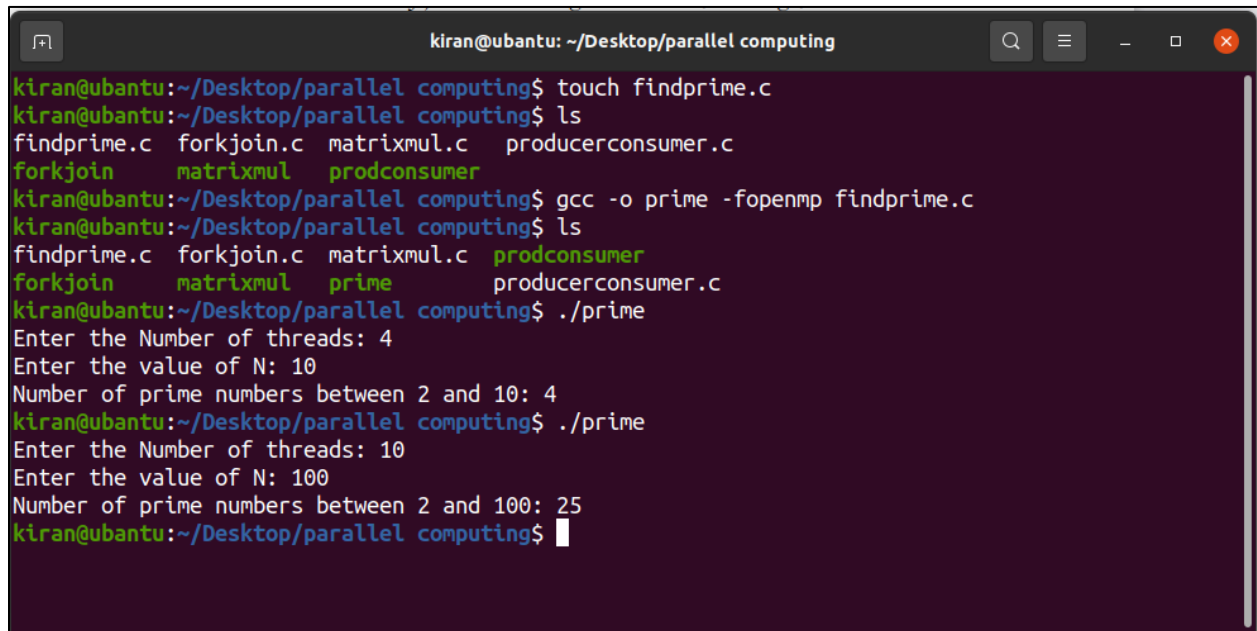
```
kiran@ubuntu: ~/Desktop/parallel computing  
kiran@ubuntu:~/Desktop/parallel computing$ touch matrixmul.c  
kiran@ubuntu:~/Desktop/parallel computing$ gcc -o matixmul -fopenmp matrixmul.c  
kiran@ubuntu:~/Desktop/parallel computing$ ls  
dijkstra      forkjoin      largestnum.c  matrixmul.c  prime          tsp  
dijkstra.c    forkjoin.c    matixmul      pical        prodconsumer   tsp.c  
findprime.c   largest       matrixmul     pical.c      producerconsumer.c  
kiran@ubuntu:~/Desktop/parallel computing$ ./matrixmul  
Enter the number of rows in Matrix 1:2  
Enter the number of columns in Matrix 1:2  
<--Now Input the values for matrix 1 row-wise-->  
5 4  
6 2  
Enter the number of columns in Matrix 2:2  
<--Now Input the values for matrix 2 row-wise-->  
6 5  
5 4  
The output after Matrix Multiplication is:  
50      41  
46      38  
kiran@ubuntu:~/Desktop/parallel computing$
```

4. Find Prime

Program :

```
#include<stdio.h>
#include<omp.h>
int IsPrime(int number) {
    int i;
    for (i = 2; i < number; i++) {
        if (number % i == 0 && i != number) return 0;
    }
    return 1;
}
int main(){
    int noOfThreads,valueN,indexCount=0,arrayVal[10000],tempValue;
    printf("Enter the Number of threads: ");
    scanf("%d",&noOfThreads);
    printf("Enter the value of N: ");
    scanf("%d",&valueN);
    omp_set_num_threads(noOfThreads);
    #pragma omp parallel for reduction(+:indexCount)
    for(tempValue=2;tempValue<=valueN;tempValue++){
        if(IsPrime(tempValue)){
            arrayVal[indexCount] = tempValue;
            indexCount++;
        }
    }
    printf("Number of prime numbers between 2 and %d: %d\n",valueN,indexCount)
    ;
    return 0;
}
```

Output :



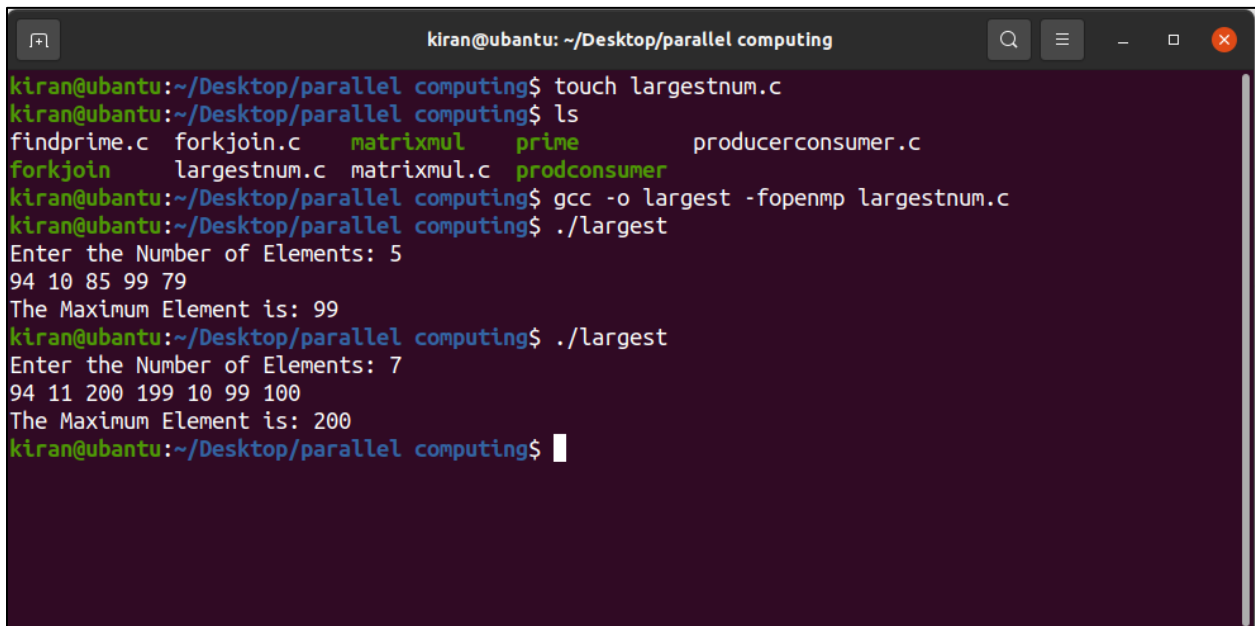
```
kiran@ubuntu: ~/Desktop/parallel computing
kiran@ubuntu:~/Desktop/parallel computing$ touch findprime.c
kiran@ubuntu:~/Desktop/parallel computing$ ls
findprime.c  forkjoin.c  matrixmul.c  producerconsumer.c
forkjoin     matrixmul   prodconsumer
kiran@ubuntu:~/Desktop/parallel computing$ gcc -o prime -fopenmp findprime.c
kiran@ubuntu:~/Desktop/parallel computing$ ls
findprime.c  forkjoin.c  matrixmul.c  prodconsumer
forkjoin     matrixmul   prime         producerconsumer.c
kiran@ubuntu:~/Desktop/parallel computing$ ./prime
Enter the Number of threads: 4
Enter the value of N: 10
Number of prime numbers between 2 and 10: 4
kiran@ubuntu:~/Desktop/parallel computing$ ./prime
Enter the Number of threads: 10
Enter the value of N: 100
Number of prime numbers between 2 and 100: 25
kiran@ubuntu:~/Desktop/parallel computing$
```


5. Largest Element in an array

Program :

```
#include<stdio.h>
#include<omp.h>
int main(){
int numberOfElements,currentMax=-1,iIterator,arrayInput[10000];
printf("Enter the Number of Elements: ");
scanf("%d",&numberOfElements);
for(iIterator=0;iIterator<numberOfElements;iIterator++){
scanf("%d",&arrayInput[iIterator]);
}
#pragma omp parallel for shared(currentMax)
for(iIterator=0;iIterator<numberOfElements;iIterator++){
#pragma omp critical
if(arrayInput[iIterator] > currentMax){
currentMax = arrayInput[iIterator];
}
}
printf("The Maximum Element is: %d\n",currentMax);
return 0;
}
```

Output :



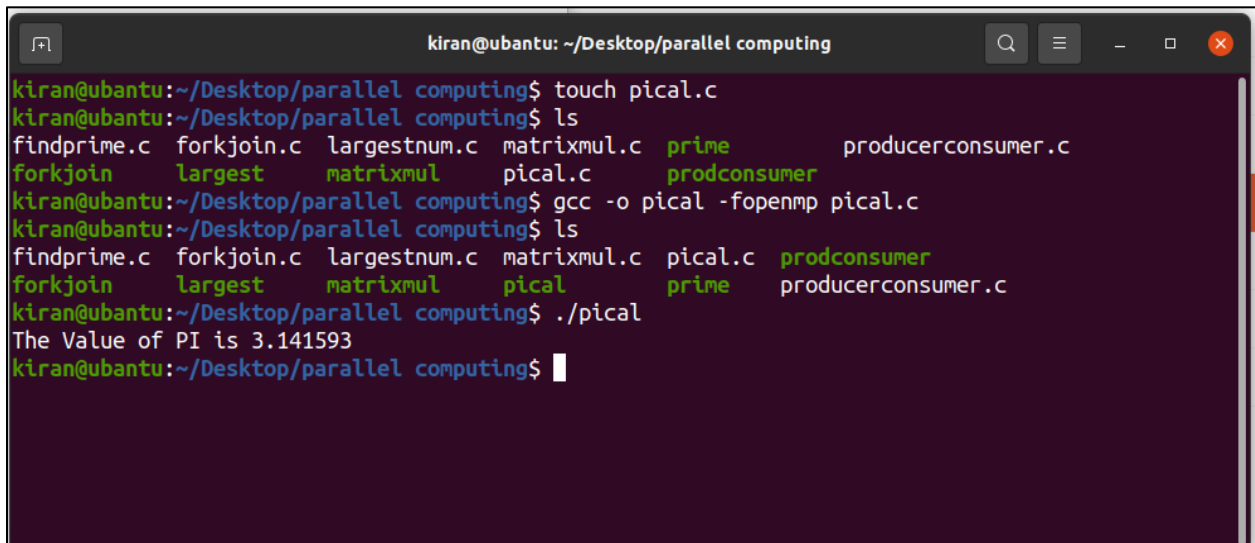
```
kiran@ubuntu: ~/Desktop/parallel computing
kiran@ubuntu:~/Desktop/parallel computing$ touch largestnum.c
kiran@ubuntu:~/Desktop/parallel computing$ ls
findprime.c  forkjoin.c  matrixmul  prime  producerconsumer.c
forkjoin     largestnum.c  matrixmul.c  prodconsumer
kiran@ubuntu:~/Desktop/parallel computing$ gcc -o largest -fopenmp largestnum.c
kiran@ubuntu:~/Desktop/parallel computing$ ./largest
Enter the Number of Elements: 5
94 10 85 99 79
The Maximum Element is: 99
kiran@ubuntu:~/Desktop/parallel computing$ ./largest
Enter the Number of Elements: 7
94 11 200 199 10 99 100
The Maximum Element is: 200
kiran@ubuntu:~/Desktop/parallel computing$
```

6. Pi calculation

Program :

```
#include<stdio.h>
#include<omp.h>
int main(){
int num_steps=10000,i;
double aux,pi,step = 1.0/(double) num_steps,x=0.0,sum = 0.0;
#pragma omp parallel private(i,x,aux) shared(sum)
{
#pragma omp for schedule(static)
for (i=0; i<num_steps; i=i+1){
x=(i+0.5)*step;
aux=4.0/(1.0+x*x);
#pragma omp critical
sum = sum + aux;
}
}
pi=step*sum;
printf("The Value of PI is %lf\n",pi);
return 0;
}
```

Output :



```
kiran@ubuntu: ~/Desktop/parallel computing
kiran@ubuntu:~/Desktop/parallel computing$ touch pical.c
kiran@ubuntu:~/Desktop/parallel computing$ ls
findprime.c  forkjoin.c  largestnum.c  matrixmul.c  prime          producerconsumer.c
forkjoin     largest     matrixmul    pical.c      prodconsumer
kiran@ubuntu:~/Desktop/parallel computing$ gcc -o pical -fopenmp pical.c
kiran@ubuntu:~/Desktop/parallel computing$ ls
findprime.c  forkjoin.c  largestnum.c  matrixmul.c  pical.c  prodconsumer
forkjoin     largest     matrixmul    pical        prime    producerconsumer.c
kiran@ubuntu:~/Desktop/parallel computing$ ./pical
The Value of PI is 3.141593
kiran@ubuntu:~/Desktop/parallel computing$
```

Conclusion:

- In conclusion, OpenMP (Open Multi-Processing) is a widely used application programming interface (API) that enables parallel programming in shared-memory architectures. OpenMP provides a set of compiler directives, runtime libraries, and environment variables that simplify the development of parallel applications.
- By using OpenMP, programmers can explicitly specify which parts of their code can be executed in parallel, and how the parallelism should be managed. This can lead to significant performance improvements, especially in applications that perform intensive computations or data processing.