NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL
DEPARTMENT OF INFORMATION TECHNOLOGY

IT 301 Parallel Computing LAB 4

02nd September 2020

Faculty: Dr. Geetha V and Mrs. Tanmayee

Execute following programs and put screen shots of the output. Write analysis of the result before uploading in IRIS as a single pdf file. for programming exercises, write the code and also put screenshot of the results.

1. Program 1

Execute following code and observe the working of task directive.
Check the result by removing if() clause with task.

```
#include<stdio.h>
#include<omp.h>
int fibo(int n);
int main(void)
{
int n,fib;
double t1,t2;
printf("Enter the value of n:\n");
scanf("%d",&n);
t1=omp_get_wtime();
#pragma omp parallel shared(n)
{
#pragma omp single
{
fib=fibo(n);
}
}
t2=omp_get_wtime();
printf("Fib is %d\n",fib);
printf("Time taken is %f s \n",t2-t1);
return 0;
}
```

```
int fibo(int n)
{
int a,b;
if(n<2)
return n;
```

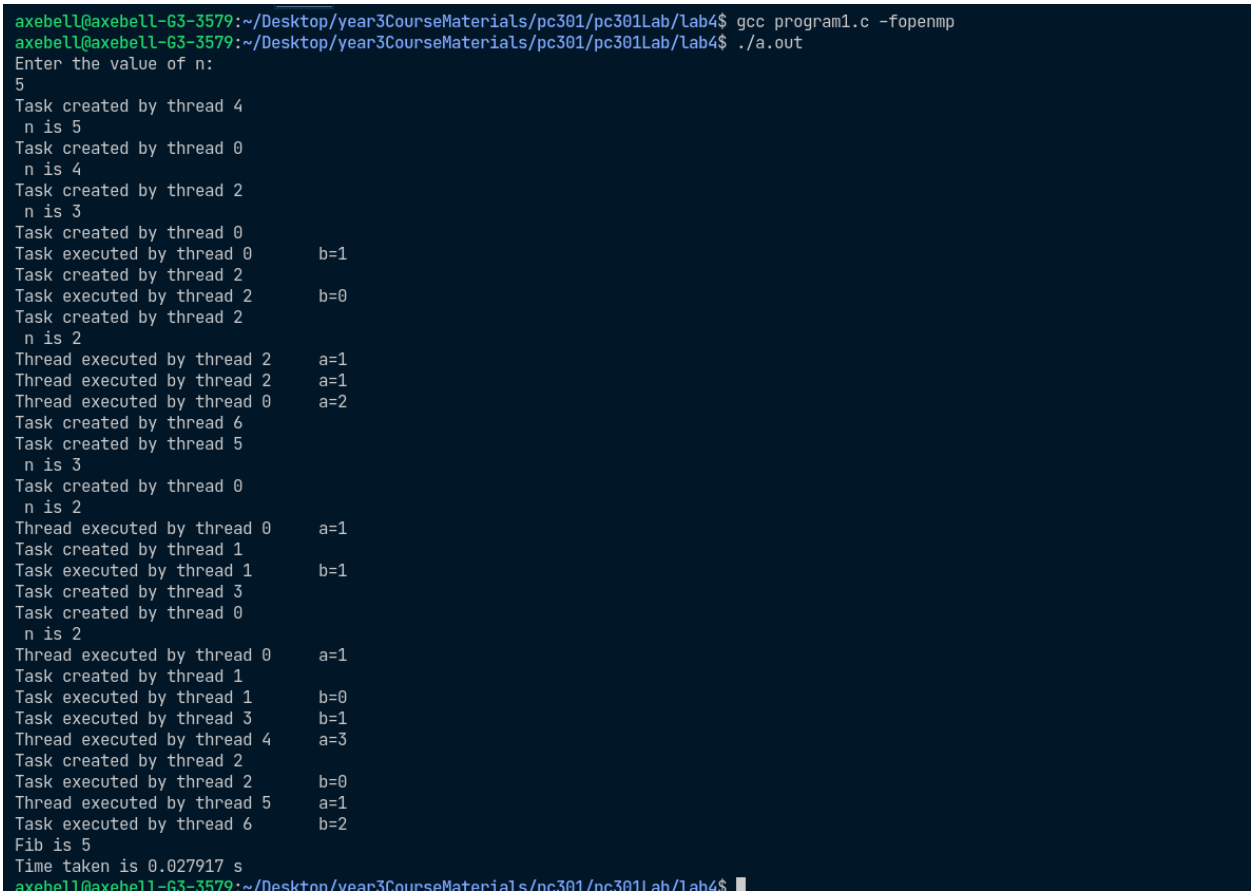
```
else
{
#pragma omp task shared(a) if(n>5)
{
printf("Task Created by Thread %d\n",omp_get_thread_num());
a=fibo(n-1);
printf("Task Executed by Thread %d \ta=%d\n",omp_get_thread_num(),a);
}
#pragma omp task shared(b) if(n>5)
{
printf("Task Created by Thread %d\n",omp_get_thread_num());
b=fibo(n-2);
printf("Task Executed by Thread %d \tb=%d\n",omp_get_thread_num(),b);
}
#pragma omp taskwait
return a+b;
}
```

* The parallel region is executed only if the if clause evalutated to true.
* if the if(n>5) clause is removed then the executions are done by a single thread. Its more like sequential execution.
* if the clause is there the calculation is done parallely for n>5 otherwise its executed by a single thread.

IF the if clause is removed

```
axebell@axebell-G3-3579:~/Desktop/year3CourseMaterials/pc301/pc301Lab/lab4$ ./a.out
Enter the value of n:
5
Task created by thread 2
 n is 5
Task created by thread 2
 n is 4
Task created by thread 2
 n is 3
Task created by thread 2
 n is 2
Thread executed by thread 2     a=1
Task created by thread 2
Task executed by thread 2       b=0
Thread executed by thread 2     a=1
Task created by thread 2
Task executed by thread 2       b=1
Thread executed by thread 2     a=2
Task created by thread 2
Task created by thread 2
 n is 2
Thread executed by thread 2     a=1
Task created by thread 2
Task executed by thread 2       b=0
Task executed by thread 2       b=1
Thread executed by thread 2     a=3
Task created by thread 2
Task created by thread 2
 n is 3
Task created by thread 2
 n is 2
Thread executed by thread 2     a=1
Task created by thread 2
Task executed by thread 2       b=0
Thread executed by thread 2     a=1
Task created by thread 2
Task executed by thread 2       b=1
Task executed by thread 2       b=2
Fib is 5
Time taken is 0.000792 s
axebell@axebell-G3-3579:~/Desktop/year3CourseMaterials/pc301/pc301Lab/lab4$
```

IF the if clause is not removed for n > 5



Programming exercises in OpenMP

2.Write a C/C++ OpenMP program to find ROWSUM and COLUMNSUM of a matrix a[n][n]. Compare the time of parallel execution with sequential execution.

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#define SIZE 10000

int main() {
int* rowsum = (int*) malloc (SIZE * sizeof(int));
int* colsum = (int*) malloc (SIZE * sizeof(int));

int** matrix = (int**) malloc (SIZE * sizeof(int*));

for(int i=0; i<SIZE; ++i) {
matrix[i] = (int*) malloc (SIZE * sizeof(int));
}

for(int i=0; i<SIZE; ++i) {
for(int j=0; j<SIZE; ++j) {
matrix[i][j] = rand() % 100;
}
}

printf("SEQUENTIAL CALCUALTION\n");
double st = omp_get_wtime();
for(int i=0; i<SIZE; ++i) {
rowsum[i] = 0;
colsum[i] = 0;
for(int j=0; j<SIZE; ++j) {
rowsum[i] += matrix[i][j];
colsum[i] += matrix[j][i];
}
}
double et = omp_get_wtime();
printf("total time=%f\n", et - st);

printf("PARALLEL CALCULATION\n");
st = omp_get_wtime();
#pragma omp parallel for shared(rowsum, colsum)
for(int i=0; i<SIZE; i++) {
rowsum[i] = 0;
colsum[i] = 0;
```

```
int tmp1=0, tmp2=0;
for(int j=0; j<SIZE; ++j) {
tmp1 += matrix[i][j];
tmp2 += matrix[j][i];
}
#pragma omp critical
{
rowsum[i] = tmp1;
colsum[i] = tmp2;
}
}
et = omp_get_wtime();
printf("total time=%f\n", et - st);
}
```

I have used a tmp variable to record the sum of elements so as to eliminate **FALSE SHARING.** So that there's no writing to and reloading from memory every time there is an addition to make.

The sequential execution is initially faster than the parallel execution (I think) because of the overhead of creating threads and also because of false sharing.

When a thread encounters #pragma omp critical only one thread can execute the structured block inside.

There is false sharing during writing to rowsum[i] and colsum[i]. This might add to some overhead and possibly might also be the reason for sequential program being faster initially.

SIZE OF MATRIX	**PARALLEL EXECUTION in seconds**	**SEQUENTIAL EXECUTION in seconds**
5	0.009629 s	0.000002
10	0.006345	0.000003
100	0.006878	0.000178
1000	0.003109	0.005510
5000	0.048757	0.215523

3. Write a C/C++ OpenMP program to perform matrix multiplication. Compare the time of parallel execution with sequential execution.

SEQUENTIAL PROGRAM FOR MATRIX MULTIPLICATION

```
//without using threads sequential;
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>

#define SIZE 1000
int main() {
printf("SEQUENTIAL MATRIX MULTIPLICATION: matrices of size: %d\n", SIZE);
int **mat1 = (int **) malloc (SIZE * sizeof(int*));
int **mat2 = (int **) malloc (SIZE * sizeof(int*));
int **res = (int **) malloc (SIZE * sizeof(int*));
for(int i=0; i<SIZE; ++i) {
mat1[i] = (int*) malloc (SIZE * sizeof(int));
mat2[i] = (int*) malloc (SIZE * sizeof(int));
res[i] = (int*) malloc (SIZE * sizeof(int));
}

for(int i=0; i<SIZE; ++i) {
for(int j=0; j<SIZE; ++j) {
mat1[i][j] = rand()%100;
mat2[i][j] = rand()%100;
}
}

//perform matrix multiplication
double st = omp_get_wtime();
for(int i=0; i<SIZE; ++i) {
for(int j=0; j<SIZE; ++j) {
for(int k=0; k<SIZE; ++k) {
res[i][j] = mat1[i][k] * mat2[k][j];
}
}
}
double et = omp_get_wtime();
printf("total time=%f\n", et - st);
}
```

PARALLEL PROGRAM FOR MATRIX MUlTIPLICATION

```
#include<stdlib.h>
#include<omp.h>
#include<stdio.h>
#define SIZE 1000

int main() {
printf("PARALLEL MATRIX MULTIPLICATION: matrices of size=%d\n", SIZE);
int** mat1 = (int **) malloc (SIZE * sizeof(int*));
int** mat2 = (int **) malloc (SIZE * sizeof(int*));
int** res = (int **) malloc (SIZE * sizeof(int*));
for(int i=0; i<SIZE; ++i) {
mat1[i] = (int*) malloc (SIZE * sizeof(int));
mat2[i] = (int*) malloc (SIZE * sizeof(int));
res[i] = (int*) malloc (SIZE * sizeof(int));
}

//initialize matrices
for(int i=0; i<SIZE; ++i) {
for(int j=0; j<SIZE; ++j) {
mat1[i][j] = rand()%100;
mat2[i][j] = rand()%100;
}
}

//perform parallel matrix multiplication
double st = omp_get_wtime();
#pragma omp parallel for shared(mat1, mat2, res)
for(int i=0; i<SIZE; ++i) {
for(int j=0; j<SIZE; ++j) {
int tmp = 0;
for(int k=0; k<SIZE; ++k) {
tmp += mat1[i][k] * mat2[k][j]; // used tmp to eliminate FALSE SHARING
}
#pragma omp critical
{
res[i][j] = tmp;
}
}
}
double et = omp_get_wtime();
printf("total time=%f\n", et - st);
}
```

SIZE	PARALLEL in seconds	SEQUENTIAL in seconds

10	0.005988	0.000004
100	0.007747	0.004699
1000	1.263195	3.810736
2000	12.926811	55.960864
3000	52.312749	Took too long. So I stopped.