## Write an OpenMP program to print "Hello World" with thread number and total number of thread usedFile

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
int main() {
        int i;
        omp_set_num_threads(10);
        #pragma omp parallel
        {
                i=omp_get_num_threads();
                int num=omp_get_thread_num();
                printf("Hello World - %d\n",num);
        }
        printf("Number of threads = %d\n",i);
}
```

## Write an OpenMP program to print message according to even and odd id number of thread.File

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
int main() {
        omp_set_num_threads(10);
        #pragma omp parallel
        {
                int i=omp_get_thread_num();
                if(i%2==0)
            printf("Hello World from %d, I am even !\n",i);
        else
            printf("Hello World from %d, I am odd !\n",i);
        }
}
```

SAXPY stands for Single Precision A*X+Y , a function in standard basic linear algebra subroutine. It is a combination of scalar multiplication and vector addition represented as Z->A*X+Y. Write an OpenMP program for SAXPY

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
void main() {
    int n;
    printf("Enter the size of vector : ");
    scanf_s("%d",&n);
    int *a,x,*y,*z,i=0;
    a=(int*)malloc(n*sizeof(int));
    y=(int*)malloc(n*sizeof(int));
    z=(int*)malloc(n*sizeof(int));
    omp_set_num_threads(n);
    x=rand()%100;
    #pragma omp parallel
    {
        #pragma omp for
        for(i=0;i<n;i++) {
            srand(i);
            a[i]=rand()%100;
            y[i]=rand()%100;
            z[i]=a[i]*x+y[i];
        }
    }
    printf(" A   *   X   +   Y   =   Z \n");
    for(i=0;i<n;i++) {
        if(i==n/2)
            printf("%3d * %3d + %3d = %3d\n",a[i],x,y[i],z[i]);
        else
            printf("%3d         %3d   %3d\n",a[i],y[i],z[i]);
    }
}
```

- Write an OpenMP program to multiply two 2D matrix and display themFile

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
void main() {
        int m,n,p,q;
        printf("Enter the size of 1st matrix : ");
        scanf_s("%d%d",&m,&n);
    printf("Enter the size of 2nd matrix : ");
        scanf_s("%d%d",&p,&q);
        if (n!=p) {
                printf("Multiplication is not possible.\n");
                exit(0);
        }
        int i=0,j=0,k=0;
        int **arr1=(int**)malloc(m*sizeof(int*));
        int **arr2=(int**)malloc(p*sizeof(int*));
        int **res=(int**)malloc(m*sizeof(int*));
        omp_set_num_threads(m);
        #pragma omp parallel private(j)
        {
                #pragma omp for
                for (i=0;i<m;i++) {
                        srand(i);
                        arr1[i]=(int*)malloc(n*sizeof(int));
             res[i]=(int*)malloc(q*sizeof(int));
                        for (j=0;j<n;j++)
                                arr1[i][j]=rand()%100;
                }
        }
        omp_set_num_threads(p);
    #pragma omp parallel private(j)
        {
                #pragma omp for
                for (i=0;i<p;i++) {
                        srand(i);
                        arr2[i]=(int*)malloc(q*sizeof(int));
                        for (j=0;j<q;j++)
                                arr2[i][j]=rand()%100;
                }
        }
        omp_set_num_threads(m);
        #pragma omp parallel private(j,k)
        {
                #pragma omp for
                for(i=0;i<m;i++) {
                        for(j=0;j<q;j++) {
                 res[i][j]=0;
                 for(k=0;k<n;k++)
                     res[i][j]+=arr1[i][k]*arr2[k][j];
             }
                }
        }
        printf("Matrix-1 :\n");
        for(i=0;i<m;i++) {
                for(j=0;j<n;j++)
                        printf("%3d ",arr1[i][j]);
        printf("\n");
        }
    printf("Matrix-2 :\n");
    for(i=0;i<p;i++) {
        for(j=0;j<q;j++)
            printf("%3d ",arr2[i][j]);
        printf("\n");
    }
    printf("Resultant Matrix :\n");
    for(i=0;i<m;i++) {
        for(j=0;j<q;j++)
            printf("%6d ",res[i][j]);
        printf("\n");
    }
}
```

Given an n*n matrix arr and a vector vec of length n, their product res=arr*vec . Write a program to implement the multiplication using OpenMP PARALLEL directive

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
void main() {
        int m,n;
        printf("Enter the size of square matrix : ");
        scanf_s("%d",&n);
        printf("Enter the size of vector : ");
        scanf_s("%d", &m);
        if (m!=n) {
                printf("Multiplication is not possible.\n");
                exit(0);
        }
        int i=0,j=0;
        int **arr=(int**)malloc(n*sizeof(int*));
        int *vec=(int*)malloc(n*sizeof(int));
        int *res=(int*)malloc(n*sizeof(int));
        omp_set_num_threads(n);
        #pragma omp parallel private(j)
        {
                #pragma omp for
                for (i=0;i<n;i++) {
                        srand(i);
                        arr[i]=(int*)malloc(n*sizeof(int));
                        vec[i]=rand()%100;
                        for (j=0;j<n;j++)
                                arr[i][j]=rand()%100;
                }
        }
        #pragma omp parallel private(j)
        {
                #pragma omp for
                for(i=0;i<n;i++) {
                        res[i]=0;
                        for(j=0;j<n;j++)
                                res[i]+=arr[i][j]*vec[j];
                }
        }
        printf("Matrix * Vector = Resultant Matrix\n");
        for(i=0;i<n;i++) {
                for(j=0;j<n;j++)
                        printf("%3d ",arr[i][j]);
                if(i==n/2)
                        printf("  *  %3d  = %6d\n",vec[i],res[i]);
                else
                        printf("     %3d    %6d\n",vec[i],res[i]);
        }
}
```

Consider a scenario where a person visits a supermarket for shopping. He purhases various items in different sections such as clothing, gaming, grocery, stationary.

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

```
void main() {
    int r,i,ans=0;
    printf("Enter number of sections : ");
    scanf_s("%d",&r);
    int **arr=(int**)malloc(r*sizeof(int*));
    int *size=(int*)malloc(r*sizeof(int));
    omp_set_num_threads(r);
        #pragma omp parallel
    {
        #pragma omp for
        for (i=0;i<r;i++) {
            srand(i);
            int j,sum=0;
            size[i]=rand()%20;
                    arr[i]=(int*)malloc(size[i]*sizeof(int));
                    for (j=0;j<size[i];j++) {
                            arr[i][j]=rand()%100;
                sum+=arr[i][j];
            }
            #pragma omp critical
                ans+=sum;
            }
    }
    for(i=0;i<r;i++) {
        printf("Section - %2d ( %3d Items ) : ",i,size[i]);
        for(int j=0;j<size[i];j++)
            printf("%3d ",arr[i][j]);
        printf("\n");
    }
    printf("Total Amount : %d",ans);
}
```

## Consider a person named X on the earth, to find its accurate position on the globe, we require the value of pi.File

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
void main() {
    int num,i;
    printf("Enter the number of steps : ");
    scanf_s("%d",&num);
    time_t st,et;
    st=clock();
    double step=1.0/(double)num,pi=0.0;
    omp_set_num_threads(num);
    #pragma omp parallel for reduction(+:pi)
    for(i=0;i<num;i++) {
        double x=(i+0.5)*step;
        double local_pi=(4.0*step)/(1+x*x);
        pi+=local_pi;
    }
    et=clock();
    printf("Time Taken : %lf\n",(double)((double)(et-st)/CLOCKS_PER_SEC));
    printf("Value of Pi = %.16lf\n",pi);
}
```

## AtomicFile

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
void main() {
    int num, i;
    printf("Enter the number of steps : ");
    scanf_s("%d", &num);
    time_t st, et;
    st = clock();
    double step = 1.0 / (double)num, pi = 0.0;
    omp_set_num_threads(num);
    #pragma omp parallel for
    for (i = 0; i < num; i++) {
        double x = (i + 0.5) * step;
        double local_pi = (4.0 * step) / (1 + x * x);
        #pragma omp atomic
        pi += local_pi;
    }
    et = clock();
    printf("Time Taken : %lf\n", (double)((double)(et - st) /
CLOCKS_PER_SEC));
    printf("Value of Pi = %lf\n", pi);
}
```

## CriticalFile

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
void main() {
    int num,i;
    printf("Enter the number of steps : ");
    scanf_s("%d",&num);
    time_t st,et;
    st=clock();
    double step=1.0/(double)num,pi=0.0;
    omp_set_num_threads(num);
    #pragma omp parallel for
    for(i=0;i<num;i++) {
        double x=(i+0.5)*step;
        double local_pi=(4.0*step)/(1+x*x);
        #pragma omp critical
            pi+=local_pi;
    }
    et=clock();
    printf("Time Taken : %lf\n",(double)((double)(et-st)/CLOCKS_PER_SEC));
    printf("Value of Pi = %lf\n",pi);
}
```

## ParallelFile

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
void main() {
    unsigned long long int num,i;
    printf("Enter the number of steps : ");
    scanf_s("%llu",&num);
    time_t st,et;
    st=clock();
    double step=1.0/(double)num, pi=0.0;
    double *local_pi=(double*)malloc(num*sizeof(double));
    omp_set_num_threads(num);
    #pragma omp parallel for
    for(i=0;i<num;i++) {
        double x=(i+0.5)*step;
        local_pi[i]=(4.0*step)/(1+x*x);
    }
    for(i=0;i<num;i++)
        pi+=local_pi[i];
    et=clock();
    printf("Time Taken : %lf\n",(double)((double)(et-st)/CLOCKS_PER_SEC));
    printf("Value of Pi = %lf\n",pi);
}
```

## ReductionFile

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
void main() {
    int num,i;
    printf("Enter the number of steps : ");
    scanf_s("%d",&num);
    time_t st,et;
    st=clock();
    double step=1.0/(double)num,pi=0.0;
    omp_set_num_threads(num);
    #pragma omp parallel for reduction(+:pi)
    for(i=0;i<num;i++) {
        double x=(i+0.5)*step;
        double local_pi=(4.0*step)/(1+x*x);
        pi+=local_pi;
    }
    et=clock();
    printf("Time Taken : %lf\n",(double)((double)(et-st)/CLOCKS_PER_SEC));
    printf("Value of Pi = %lf\n",pi);
}
```

## Design, Develop and run a multithreaded program to generate and print Fibonacci series, one thread must generate the series upto number and other thread must print them. Ensure proper synchronization.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
int main() {
        int n, i;
        printf("Number of terms : ");
        scanf_s("%d",&n);
        int* a = (int*)malloc(n * sizeof(int));
        a[0] = 0;
        a[1] = 1;
        time_t st, et;
        st = clock();
        omp_set_num_threads(2);
        #pragma omp parallel
        {
                #pragma omp single
                {
                        printf("id of thread involved in the computation of
fibonacci numbers = %d\n", omp_get_thread_num());
                        for (i = 2; i < n; i++)
                                a[i] = a[i - 2] + a[i - 1];
                }
                #pragma omp single
                {
                        printf("id of thread involved in the displaying of
fibonacci numbers = %d\n", omp_get_thread_num());
                        printf("Fibonacci numbers : ");
                        for (i = 0; i < n; i++)
                                printf("%d ", a[i]);
                        printf("\n");
                }
        }
        et = clock();
        printf("Time Taken : %lfms\n", ((double)(et - st)*1000 /
CLOCKS_PER_SEC));
        return 0;
}
```

## CriticalFile

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
int main() {
        int n, i;
        printf("Number of terms : ");
        scanf_s("%d",&n);
        int* a = (int*)malloc(n * sizeof(int));
        a[0] = 0;
        a[1] = 1;
        time_t st, et;
        st = clock();
        omp_set_num_threads(2);
        #pragma omp parallel
        {
                #pragma omp critical
                if(omp_get_thread_num()==0)
                {
                        printf("id of thread involved in the computation of
fibonacci numbers = %d\n", omp_get_thread_num());
                        for (i = 2; i < n; i++)
                                a[i] = a[i - 2] + a[i - 1];
                }
                else if(omp_get_thread_num()==1)
                {
                        printf("id of thread involved in the displaying of
fibonacci numbers = %d\n", omp_get_thread_num());
                        printf("Fibonacci numbers : ");
                        for (i = 0; i < n; i++)
                                printf("%d ", a[i]);
                        printf("\n");
                }
        }
        et = clock();
        printf("Time Taken : %lfms\n", ((double)(et - st)*1000 /
CLOCKS_PER_SEC));
        return 0;
}
```

## SingleFile

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
int main() {
        int n, i;
        printf("Number of terms : ");
        scanf_s("%d",&n);
        int* a = (int*)malloc(n * sizeof(int));
        a[0] = 0;
        a[1] = 1;
        time_t st, et;
        st = clock();
        omp_set_num_threads(2);
        #pragma omp parallel
        {
                #pragma omp single
                {
                        printf("id of thread involved in the computation of
fibonacci numbers = %d\n", omp_get_thread_num());
                        for (i = 2; i < n; i++)
                                a[i] = a[i - 2] + a[i - 1];
                }
                #pragma omp single
                {
                        printf("id of thread involved in the displaying of
fibonacci numbers = %d\n", omp_get_thread_num());
                        printf("Fibonacci numbers : ");
                        for (i = 0; i < n; i++)
                                printf("%d ", a[i]);
                        printf("\n");
                }
        }
        et = clock();
        printf("Time Taken : %lfms\n", ((double)(et - st)*1000 /
CLOCKS_PER_SEC));
        return 0;
}
```

## University awards gold medal to the student who has scored highest. Write an OpenMP program to find the student with highest cgpa.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
int main() {
        int n, i;
        time_t st, et;
        st = clock();
        printf("Enter the number of students : ");
        scanf_s("%d", &n);
        double* arr = (double*)malloc(n * sizeof(double));
        double arr_max = 0;
        #pragma omp parallel for
        for (i = 0; i < n; i++) {
                srand(i);
                arr[i] = (double)(rand() % 10000)/10 ;
        }
        printf("CGPA of students : ");
        for (i = 0; i < n; i++)
                printf("%.2lf ", arr[i]);
        printf("\n");
        #pragma omp parallel for
        for (i = 0; i < n; i++) {
                #pragma omp critical
                if (arr_max < arr[i])
                        arr_max = arr[i];
        }
        et = clock();
        printf("Student with highest CGPA = %.2lf\n", arr_max);
        printf("Time Taken : %.2lfms\n", ((double)(et - st) * 1000 /
CLOCKS_PER_SEC));
}
```

## Multiply two square matrices (1000, 2000 or 3000 dimensions). Compare the performance of a squential and parallel algorithm using openMP.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
void main() {
        int n;
        printf("Enter the dimension of square matrices : ");
        scanf_s("%d", &n);
        int i = 0, j = 0, k = 0;
        int** arr1 = (int**)malloc(n * sizeof(int*));
        int** arr2 = (int**)malloc(n * sizeof(int*));
        int** res = (int**)malloc(n * sizeof(int*));
        omp_set_num_threads(64);
        #pragma omp parallel private(j)
        {
                #pragma omp for
                for (i = 0; i < n; i++) {
                        srand(i);
                        arr1[i] = (int*)malloc(n * sizeof(int));
                        arr2[i] = (int*)malloc(n * sizeof(int));
                        res[i] = (int*)malloc(n * sizeof(int));
                        for (j = 0; j < n; j++) {
                                arr1[i][j] = rand() % 100;
                                arr2[i][j] = rand() % 100;
                        }
                }
        }
        time_t st, et;
        st = clock();
        #pragma omp parallel private(j,k)
        {
                #pragma omp for
                for (i = 0; i < n; i++) {
                        for (j = 0; j < n; j++) {
                                res[i][j] = 0;
                                for (k = 0; k < n; k++)
                                        res[i][j] += arr1[i][k] * arr2[k][j];
                        }
                }
        }
        et = clock();
        printf("Time taken by parallel algorithm : %lf\n", (double)(et -
st) / CLOCKS_PER_SEC);
        st = clock();
        for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                        res[i][j] = 0;
                        for (k = 0; k < n; k++)
                                res[i][j] += arr1[i][k] * arr2[k][j];
                }
        }
        et = clock();
        printf("Time taken by Sequential algorithm : %lf\n", (double)(et -
st) / CLOCKS_PER_SEC);
}
```

Assume you have n robots which pick mangoes in a farm. Write a program to calculate the total number of mangoes picked by n robots parallelly using MPI.

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
int main(int argc, char** argv)
{
    int rank, numproc;
    int sum = 0;
    int total_sum = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    srand(rank);
    sum = rand() % 100;
    printf("Robot %d picked %d mangoes.\n", rank, sum);
    MPI_Reduce(&sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        printf("Total Mangoes picked by %d Robots = %d\n", numproc,
total_sum);
    MPI_Finalize();
}
```

- Design a program that implements application of MPI Collective CommunicationsFile

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
        int size, rank;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        float recvbuf, sendbuf[100];
        if (rank == 0) {
                int i;
                printf("Before Scatter : sendbuf of rank 0 : ");
                for (i = 0; i < size; i++) {
                        srand(i);
                        sendbuf[i] = (float)(rand()%1000)/10;
                        printf("%.1f ", sendbuf[i]);
                }
                printf("\nAfter Scatter :\n");
        }
        MPI_Scatter(sendbuf, 1, MPI_FLOAT, &recvbuf, 1, MPI_FLOAT, 0,
MPI_COMM_WORLD);
        printf("rank= %d Recvbuf: %.1f\n", rank, recvbuf);
        MPI_Finalize();
}
```

- Implement Cartesian Virtual Topology in MPI.File

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define SIZE 16
#define UP 0
#define DOWN 1
#define LEFT 2
#define RIGHT 3
int main(int argc, char* argv[])
{
        int numtasks, rank, source, dest, outbuf, i, tag = 1, inbuf[4] = {
MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL, }, nbrs[4],
dims[2] = { 4, 4 }, periods[2] = { 0, 0 }, reorder = 0, coords[2];
        MPI_Request reqs[8];
        MPI_Status stats[8];
        MPI_Comm cartcomm;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        if (numtasks == SIZE) {
                MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
&cartcomm);
                MPI_Comm_rank(cartcomm, &rank);
                MPI_Cart_coords(cartcomm, rank, 2, coords);
                MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
                MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);
                printf("rank= %d coords= %d %d neighbors(u,d,l,r)= %d %d %d
%d\n", rank, coords[0], coords[1], nbrs[UP], nbrs[DOWN], nbrs[LEFT],
nbrs[RIGHT]);
                outbuf = rank;
                for (i = 0; i < 4; i++) {
                        dest = nbrs[i];
                        source = nbrs[i];
                        MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
MPI_COMM_WORLD, &reqs[i]);
                        MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
MPI_COMM_WORLD, &reqs[i + 4]);
                }
                MPI_Waitall(8, reqs, stats);
                printf("rank= %d inbuf(u,d,l,r)= %d %d %d %d\n", rank,
inbuf[UP], inbuf[DOWN], inbuf[LEFT], inbuf[RIGHT]);
        }
        else
                printf("Must specify %d tasks. Terminating.\n", SIZE);
        MPI_Finalize();
}
```

- Design a MPI program to simulate the uses blocking send/receive routines and nonblocking send/receive routines.File

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
        int numtasks, rank, rc, count, next, prev, sz, inmsg;
        MPI_Status Stat;
        time_t st, et;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        sz = (numtasks / 2) * 2;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        st = clock();
        if (rank == 0) prev = sz - 1;
        else prev = rank - 1;
        if (rank == sz - 1) next = 0;
        else next = rank + 1;
        if (rank % 2 == 0 && rank < sz) {
                rc = MPI_Send(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
                rc = MPI_Recv(&inmsg, 1, MPI_INT, prev, 1, MPI_COMM_WORLD,
&Stat);
        }
        else if (rank % 2 == 1 && rank < sz) {
                rc = MPI_Recv(&inmsg, 1, MPI_INT, prev, 0, MPI_COMM_WORLD,
&Stat);
                rc = MPI_Send(&rank, 1, MPI_INT, next, 1, MPI_COMM_WORLD);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        et = clock();
        if(rank==0) printf("Time taken by Blocking send/receive : %lf\n",
(double)(et - st) / CLOCKS_PER_SEC);
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Request reqs[2];
        MPI_Status stats[2];
        st = clock();
        if (rank == numtasks - 1) next = 0;
        else next = rank + 1;
        if (rank == 0) prev = numtasks - 1;
        else prev = rank - 1;
        MPI_Irecv(&inmsg, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[0]);
        MPI_Isend(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[1]);
        MPI_Barrier(MPI_COMM_WORLD);
        et = clock();
        if (rank == 0) printf("Time taken by NonBlocking send/receive :
%lf\n", (double)(et - st) / CLOCKS_PER_SEC);
        MPI_Finalize();
}
```