Fibonacci Series One thread to generate the numbers and another thread to print them

```
#include<stdio.h>
#include<omp.h>
int main()
{
  int n,a[100],i;
  omp_set_num_threads(2);
  printf("enter the no of terms of fibonacci series which have to be generated\n");
  scanf("%d",&n);
  a[0]=0;
  a[1]=1;
  #pragma omp parallel
  {
            #pragma omp single
            for(i=2;i<n;i++)
            {
           a[i]=a[i-2]+a[i-1];
                      printf("id of thread involved in the computation of fib no %d
is=%d\n",i+1,omp_get_thread_num());
            }
        #pragma omp barrier
        #pragma omp single
            {
                      printf("the elements of fib series are\n");
                      for(i=0;i<n;i++)
                      printf("%d,id of the thread displaying this no is =
%d\n",a[i],omp_get_thread_num());
            }
  }
  return 0;
}
```

Array addition

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
int main()
{
	int n = 5;
	int n_per_thread;
	int i;
	int a[n];
	int b[n];
	int c[n];
	for(int i=0; i<n; i++)
	{
		a[i] = i;
		b[i] = i;
	}
	omp_set_num_threads(5);
	n_per_thread = n/5;
	#pragma omp parallel for shared(a,b,c) private(i) schedule(static, n_per_thread)
	for(i=0; i<n; i++)
	{
		c[i] = a[i]+b[i];
		printf(" Thread %d works on element%d\n", omp_get_thread_num(), i);
	}
	printf(" i\ta[i]\t+\tb[i]\t=\tc[i]\n");
	for(i=0; i<n; i++)
	{
		printf(" %d\t%d\t\t%d\t\t%d\n",i,a[i],b[i],c[i]);
	}
	return 0;
}
```

2D array addition

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
int main()
{
	int n = 4;
	int n_per_thread;
	int i,j;
	int a[n][n];
	int b[n][n];
	int c[n][n];
	for(i=0; i<n; i++)
	{
		for(j=0;j<n;j++)
		{
		a[i][j] = i;
		b[i][j] = i;
		}
	}
	omp_set_num_threads(4);
	n_per_thread = n/4;
	#pragma omp parallel for shared(a,b,c) private(i) schedule(static, n_per_thread)
	for(i=0; i<n; i++)
	{
		for(j=0;j<n;j++)
		{
		c[i][j] = a[i][j]+b[i][j];
		printf(" Thread %d works on element%d\n", omp_get_thread_num(), n*i+j);
		}
	}
	printf(" i\ta[i]\t+\tb[i]\t=\tc[i]\n");
	for(i=0; i<n; i++)
	{
		for(j=0;j<n;j++)
		{
			printf(" %d\t%d\t\t%d\t\t%d\n",n*i+j,a[i][j],b[i][j],c[i][j]);
		}
	}
	return 0;
}
```

Sum for odd position of numbers from 1 to 202 using MPI

Using MPI, the parallel exchange of information between processes is done using the 2 subroutines.
We have used MPI_Send, to send a message to another process and MPI_Receive, to receive a message from another process.

- int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

1) Adding the required libraries including mpi.h which is necessary to run the program.
2) Initialize the size of the array and the temporary arrays for slave process.
3) In the main, initialized variables for master process(pid=0) like elements_per_process =n/np meaning the size of array/number of processes.
4) Created the parallel process using MPI_Init to initialize the MPI part of the program.
5) Call function MPI_Comm_rank(MPI_COMM_WORLD, &pid)
6) Call function MPI_Comm_size(MPI_COMM_WORLD, &np)
7) These two will help to evaluate pid and number of processes which started.
8) In the master process(pid=0), distributed the portion of array to child processes to calculate the partial sums.
9) MPI_Send() is added to the remaining elements index which is equal to iterator X elements_per_process.
10) MPI_Recv() collected the partial sums from the processes in variable temp.
11) In the slave process, store the received array segment in local array a2[].
12) Calculate the partial sum.
13) Send the partial sum to the root process which adds the sums from the partial subarrays.
14) Cleaned up the MPI state using MPI_Finalize()
15) In the main, the odd elements are taken by i%2!=2 which means they aren't divisible by 2 and hence are odd. These odd elements are stored in array a[].

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char* argv[])
{
	int a2[101],a[101];
	//n is half as there is half odd numbers
	int n=101;
	int k=0;
	//save array as required
```

```
for(int i=1;i<=202;i++){
	if(i%2!=0)
	{
		printf("Array element is: %d\n", i);
		a[k]=i;
		k++;
	}
}
int pid, np,elements_per_process,n_elements_recieved;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &pid);
MPI_Comm_size(MPI_COMM_WORLD, &np);
// if it’s the MASTER PROCESS then its distributing the work to sub arrays
if (pid == 0)
{
int index, i;
elements_per_process = n / np;
// check if more than 1 processes are run
if (np > 1)
{
// distributes the sub arrays to calculate their partial sums
for (i = 1; i < np - 1; i++)
{
	index = i * elements_per_process;
	MPI_Send(&elements_per_process,1, MPI_INT, i, 0,MPI_COMM_WORLD);
      MPI_Send(&a[index],elements_per_process,MPI_INT,i,0,MPI_COMM_WORLD);
}
// LAST PROCESS will add the remaining elements
index = i * elements_per_process;
int elements_left = n - index;
MPI_Send(&elements_left,1, MPI_INT,i, 0,MPI_COMM_WORLD);
MPI_Send(&a[index],elements_left,MPI_INT, i, 0,MPI_COMM_WORLD);
}
// MASTER process will add its own array elements
int sum = 0;
for (i = 0; i < elements_per_process; i++)
sum += a[i];
// MASTER process collects collects partial sums from other processes
int tmp;
for (i = 1; i < np; i++) {
```

```
MPI_Recv(&tmp, 1, MPI_INT,MPI_ANY_SOURCE, 0,MPI_COMM_WORLD,&status);
int sender = status.MPI_SOURCE;
sum += tmp;
}
// Prints the final sum of array
printf("Sum of array is : %d\n", sum);
}
// SLAVE processes
else {
MPI_Recv(&n_elements_recieved, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,&status);
// Storing the received array segment in local array a2[]
MPI_Recv(&a2, n_elements_recieved, MPI_INT, 0, 0, MPI_COMM_WORLD,&status);
// Calculating the partial sum of slave processes
int partial_sum = 0;
for (int i = 0; i < n_elements_recieved; i++)
partial_sum += a2[i];
// Sending the partial sum to the master process
MPI_Send(&partial_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
// cleans up all MPI state before exit of process
MPI_Finalize();
return 0;
}
```