Department: *Computer Science & Engineering*	***Course Type:*** *Program Core*
Course Title: *HPC Lab*	***Course Code:*** *17CSL78*
L-T-P:*0-0-2*	***Credits****: 1*
Total Contact Hours: *36Hours*	***Duration of SEE:*** *3 Hours*
SEE Marks*: 50*	***CIE Marks:*** *50*

COURSE DESCRIPTION

The HPC lab is aimed at reinforcing the concepts of Parallel Programming Techniques applicable for the various strains of High-performance architectures: Multicore, Multiprocessor, Message processing based Distributed computing, and Heterogenous processing ensembles.

PREREQUISITES

- Knowledge of Advanced Computer Architectures
- Ability to design and analyse Numerical Processing algorithms, Vector Processing, Searching, Sorting and String functions.

COURSE OBJECTIVES

- Provide systematic and comprehensive treatment to the Highly Integrated development Environments for HPC program development and testing.
- Provide facility with the tools useful in performance analysis of HPC facilities.
- Introduce the concepts of Heterogenous Computing platforms: CPU + GPU architectures.
- Introduce the concepts of program development for Multi-core Shared memory architectures.
- Introduce the concepts of High-Performance Computing as a service on Cloud platforms (utilizing HP computing resources and storage made available through a Cloud platform).

COURSE CONTENTS:

Part -A

1.Matrix vector multiplication using OpenMP *PARALLEL* directive.

2. Sum of elements of one-dimensional real array A using OpenMP *PARALLEL DO* directive.

3.Compute the value of *PI* function by Numerical Integration using OpenMP *PARALLEL* section.

4. Using OpenMP, Design, develop and run a multi-threaded program to generate and print Fibonacci Series. One thread must generate the numbers up to the specified limit and another thread must print them. Ensure proper synchronization.

5. Recursive computation using OpenMP *CRITICAL* section.

do i = 2, N

$a(i) = (a(i-1) + a(i))/2$

end do

6. Largest element in a list of numbers using OpenMP *CRITICAL* section.

7. Write an OpenMP program to print Sum of Elements of Array using Reduction clause

8.To calculate the sum of given numbers in parallel using MPI

9. Design a program that implements MPI Collective Communications

10. Implement Cartesian Virtual Topology in MPI

11. Design a MPI program that uses blocking send/receive routines.

12. Design a MPI program that uses nonblocking send/receive routines.

13. Multiply two square matrices (1000,2000 or 3000dimensions). Compare the performance of a sequential and parallel algorithm using open MP.

Part – B

CUDA is a parallel computing platform and an API model that was developed by Nvidia. Using CUDA one can utilize the power of Nvidia GPUs to perform general computing tasks, such as multiplying matrices and performing other linear algebra operations, instead of just doing graphical calculations. Students write programs in CUDA and understand efficiency and power of parallelism.

ASSESSMENT METHODS

- **Experiment Write up + Execution + Viva - 15 Marks**
- **Lab Record Writing - 10 Marks**
- **Lab Internals Test - 15 Marks**
- **Surprise Test - 10 Marks**

Total = 50 M

- **Final examination will be conducted for 100 marks and evaluated for 50 Marks.**

COURSE OUTCOMES

Students will be able to

CO	Description
CO 1:	Design and implement high performance versions of standard single threaded algorithms
CO 2:	Demonstrate the architectural features in the GPU and MIC hardware accelerators
CO 3:	Design programs to extract maximum performance in a multicore, shared memory execution environment processor
CO 4:	Develop programs using OPENMP, MPI and CUDA
CO 5:	Design and deploy Parallel programs on Processor clusters, configuring clusters and cloud storage.

**Mapping of Course Outcomes (COs) to Program Outcomes (POs*)& PSO **															
Course Outcomes mapping to Program Outcomes													**PSOs**		
PO/CO	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**PSO1**	**PSO2**	**PSO3**
CO1	3	1											3		
CO2	3	3	1		3								3		
CO3	3	3	2		3								3		
CO4	3	3	2	3	3								3		
CO5	3	3	3		3								3		
Correlation Level	3	3	2	3	3								3		

PART-A

1.Matrix vector multiplication using OpenMP *PARALLEL* directive.

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

/* Main Program */

main()
{
        int             NoofRows, NoofCols, Vectorsize, i, j;
        float         **Matrix, *Vector, *Result, *Checkoutput;

        printf("Read the matrix size noofrows and columns and vectorsize\n");
        scanf_s("%d%d%d", &NoofRows, &NoofCols, &Vectorsize);

        if (NoofRows <= 0 || NoofCols <= 0 || Vectorsize <= 0) {
                printf("The Matrix and Vectorsize should be of positive sign\n");
                exit(1);
        }
        /* Checking For Matrix Vector Computation Necessary Condition */

        if (NoofCols != Vectorsize) {
                printf("Matrix Vector computation cannot be possible \n");
                exit(1);
        }
        /* Dynamic Memory Allocation  And Initialization Of Matrix Elements */

        Matrix = (float **) malloc(sizeof(float) * NoofRows);
        for (i = 0; i < NoofRows; i++) {
                Matrix[i] = (float *) malloc(sizeof(float) * NoofCols);
                for (j = 0; j < NoofCols; j++)
                        Matrix[i][j] = i + j;
        }

        /* Printing The Matrix */

        printf("The Matrix is \n");
        for (i = 0; i < NoofRows; i++) {
                for (j = 0; j < NoofCols; j++)
                        printf("%f \t", Matrix[i][j]);
                printf("\n");
        }

        printf("\n");

        /* Dynamic Memory Allocation */

        Vector = (float *) malloc(sizeof(float) * Vectorsize);

        /* vector Initialization */

        for (i = 0; i < Vectorsize; i++)
                Vector[i] = i;

        printf("\n");
```

```
        /* Printing The Vector Elements */

        printf("The Vector is \n");
        for (i = 0; i < Vectorsize; i++)
                printf("%f \t", Vector[i]);

        /* Dynamic Memory Allocation */

        Result = (float *) malloc(sizeof(float) * NoofRows);

        Checkoutput = (float *) malloc(sizeof(float) * NoofRows);

        for (i = 0; i < NoofRows; i = i + 1)
        {
        Result[i]=0;
        Checkoutput[i]=0;
        }

        /* OpenMP Parallel Directive */

#pragma omp parallel for private(j)
        for (i = 0; i < NoofRows; i = i + 1)
                for (j = 0; j < NoofCols; j = j + 1)
                        Result[i] = Result[i] + Matrix[i][j] * Vector[j];

        /* Serial Computation */

        for (i = 0; i < NoofRows; i = i + 1)
                for (j = 0; j < NoofCols; j = j + 1)
                        Checkoutput[i] = Checkoutput[i] + Matrix[i][j] *
Vector[j];

        for (i = 0; i < NoofRows; i = i + 1)
                if (Checkoutput[i] == Result[i])
                        continue;
                else {
                        printf("There is a difference from Serial and Parallel
Computation \n");
                        exit(1);
                }


        printf("\nThe Matrix Computation result is \n");
        for (i = 0; i < NoofRows; i++)
                printf("%f \n", Result[i]);

        /* Freeing The Memory Allocations */

        free(Vector);
        free(Result);
        free(Matrix);
        free(Checkoutput);

}
```

2. Sum of elements of one-dimensional real array A using OpenMP *PARALLEL DO* directive.

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

/* Main Program */

main()
{
        float          *Array, *Check, serial_sum, sum, partialsum;
        int             array_size, i;

        printf("Enter the size of the array\n");
        scanf_s("%d", &array_size);

        if (array_size <= 0) {
                printf("Array Size Should Be Of Positive Value ");
                exit(1);
        }
        /* Dynamic Memory Allocation */

        Array = (float *) malloc(sizeof(float) * array_size);
        Check = (float *) malloc(sizeof(float) * array_size);

        /* Array Elements Initialization */

        for (i = 0; i < array_size; i++) {
                Array[i] = i * 5;
                Check[i] = Array[i];
        }

        printf("The Array Elements Are \n");

        for (i = 0; i < array_size; i++)
                printf("Array[%d]=%f\n", i, Array[i]);

        sum = 0.0;
        partialsum = 0.0;

        /* OpenMP Parallel For Directive And Critical Section */

#pragma omp parallel for shared(sum)
        for (i = 0; i < array_size; i++) {
#pragma omp critical
                sum = sum + Array[i];

        }

        serial_sum = 0.0;

        /* Serial Calculation */
        for (i = 0; i < array_size; i++)
                serial_sum = serial_sum + Check[i];


        if (serial_sum == sum)
                printf("\nThe Serial And Parallel Sums Are Equal\n");
        else {
                printf("\nThe Serial And Parallel Sums Are UnEqual\n");
```

```
                exit(1);
        }

        /* Freeing Memory */
        free(Check);
        free(Array);

        printf("\nThe SumOfElements Of The Array Using OpenMP Directives Is %f\n",
sum);
        printf("\nThe SumOfElements Of The Array By Serial Calculation Is %f\n",
serial_sum);
}
```

3.Compute the value of *PI* function by Numerical Integration using OpenMP *PARALLEL* section.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
static long num_steps;
double step;
void main()
{
	int i; double x, pi, sum = 0.0;
	printf("Enter number of steps\n");
	scanf_s("%d", &num_steps);

	if (num_steps <= 0) {
		printf("Number of intervals should be positive integer\n");
		exit(1);
	}
	step = 1.0 / (double)num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
	for (i = 1; i <= num_steps; i++) {
		x = (i - 0.5) * step;
		sum = sum + 4.0 / (1.0 + x * x);
	}
	pi = sum / num_steps;
	printf("pi = %6.8f\n", pi);
}
```

4. Using OpenMP, Design, develop and run a multi-threaded program to generate and print Fibonacci Series. One thread must generate the numbers up to the specified limit and another thread must print them. Ensure proper synchronization.

```
#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;

}
```

OUTPUT

```
cc -fopenmp 11.c
./a.out
enter the no of terms of fibonacci series which have to be generated
5
id of thread involved in thes computation of fib no 3 is=0
id of thread involved in the computation of fib no 4 is=0
id of thread involved in the computation of fib no 5 is=0
the elements of fib series are
0,id of the thread displaying this no is =  1
1,id of the thread displaying this no is =  1
1,id of the thread displaying this no is =  1
2,id of the thread displaying this no is =  1
3,id of the thread displaying this no is =  1
```

5. Recursive computation using OpenMP *CRITICAL* section.

do $i = 2, N$

$a(i) = (a(i-1) + a(i))/2$

end do

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

/* Main Program */

main()
{
        int             i, N;
        float          *array, *check;

        /* Size Of An Array */

        printf("Enter the size \n");
        scanf("%d", &N);

        if (N <= 0) {
                printf("Array Size Should Be Of Postive Sign \n");
                exit(1);
        }
        /* Dynamic Memory Allocation */

        array = (float *) malloc(sizeof(float) * N);
        check = (float *) malloc(sizeof(float) * N);

        /* Initialization Of Array Elements  */

        for (i = 0; i < N; i++) {
                array[i] = i * 1;
                check[i] = i * 1;
        }

        /* The Input Array Is */
        printf("The Input Array Is\n");

        for (i = 0; i < N; i++)
                printf("%f\t", array[i]);


        /* OpenMP Parallel For Directive And Critical Section */

#pragma omp parallel for
        for (i = 1; i < N; i++) {
#pragma omp critical
                array[i] = (array[i - 1] + array[i]) / 2;
        }

        /* Serial Calculation */

        for (i = 1; i < N; i++)
```

```
                check[i] = (check[i - 1] + check[i]) / 2;

        /* Output Checking */

        for (i = 0; i < N; i++) {
                if (check[i] == array[i])
                        continue;
                else {
                        printf("There is a difference in the parallel and serial
calculation \n");
                        exit(1);
                }
        }

        /* The Final Output */

        printf("\nThe Array Calculation Is Same Using Serial And OpenMP
Directives\n");
        printf("The Output Array Is \n");
        for (i = 0; i < N; i++)
                printf("\n %f \t", array[i]);

printf("\n");
        /* Freeing The Memory */

        free(array);
        free(check);
}
```

6. Largest element in a list of numbers using OpenMP *CRITICAL* section.

```
#include <stdio.h>
#include <omp.h>
#define MAXIMUM 65536

/* Main Program */

main()
{
        int             *array, i, Noofelements, cur_max, current_value;

        printf("Enter the number of elements\n");
        scanf("%d", &Noofelements);

        if (Noofelements <= 0) {
                printf("The array elements cannot be stored\n");
                exit(1);
        }
        /* Dynamic Memory Allocation */

        array = (int *) malloc(sizeof(int) * Noofelements);

        /* Allocating Random Number Values To The Elements Of An Array */

        srand(MAXIMUM);
        for (i = 0; i < Noofelements; i++)
                array[i] = rand();

        if (Noofelements == 1) {
                printf("The Largest Number In The Array is %d", array[0]);
                exit(1);
        }
        /* OpenMP Parallel For Directive And Critical Section */

        cur_max = 0;
#pragma omp parallel for
        for (i = 0; i < Noofelements; i = i + 1) {
                if (array[i] > cur_max)
#pragma omp critical
                        if (array[i] > cur_max)
                                cur_max = array[i];
        }

        /* Serial Calculation */

        current_value = array[0];
        for (i = 1; i < Noofelements; i++)
                if (array[i] > current_value)
                        current_value = array[i];

        printf("The Input Array Elements Are \n");

        for (i = 0; i < Noofelements; i++)
                printf("\t%d", array[i]);

        printf("\n");
```

```
        /* Checking For Output Validity */

        if (current_value == cur_max)
                printf("\nThe Max Value Is Same From Serial And Parallel OpenMP
Directive\n");
        else {
                printf("\nThe Max Value Is Not Same In Serial And Parallel OpenMP
Directive\n");
                exit(1);
        }

        /* Freeing Allocated Memory */

        printf("\n");
        free(array);
        printf("\nThe Largest Number In The Given Array Is %d\n", cur_max);
}
```

7. Write an OpenMP program to print Sum of Elements of Array using Reduction clause

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

/* Main Program */

main()
{
        float          *array_A, sum, *checkarray, serialsum;
        int             arraysize, i, k, Noofthreads;

        printf("Enter the size of the array \n");
        scanf("%d", &arraysize);

        if (arraysize <= 0) {
                printf("Positive Number Required\n");
                exit(1);
        }
        /* Dynamic Memory Allocation */

        array_A = (float *) malloc(sizeof(float) * arraysize);
        checkarray = (float *) malloc(sizeof(float) * arraysize);

        for (i = 0; i < arraysize; i++) {
                array_A[i] = i + 5;
                checkarray[i] = array_A[i];
        }

        printf("\nThe input array is \n");
        for (i = 0; i < arraysize; i++)
                printf("%f \t", array_A[i]);


        sum = 0.0;

        /* OpenMP Parallel For With Reduction Clause */

#pragma omp parallel for reduction(+ : sum)
        for (i = 0; i < arraysize; i++)
                sum = sum + array_A[i];

        /* Serial Calculation */

        serialsum = 0.0;
        for (i = 0; i < arraysize; i++)
                serialsum = serialsum + array_A[i];

        /* Output Checking */

        if (serialsum != sum) {
                printf("\nThe calculation of array sum is different \n");
                exit(1);
        } else
                printf("\nThe calculation of array sum is same\n");

        /* Freeing Memory Which Was Allocated */
```

```
        free(checkarray);
        free(array_A);

        printf("The value of array sum using threads is %f\n", sum);
        printf("\nThe serial calculation of array is %f\n", serialsum);
}
```

8.To calculate the sum of given numbers in parallel using MPI

```
#include <stdio.h>
#include "mpi.h"

int main(int argc,char *argv[])
{
    int iproc;
    int MyRank, Numprocs, Root = 0;
    int value, sum = 0;
    int Source, Source_tag;
    int Destination, Destination_tag;
    MPI_Status status;


    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&Numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);

    if(MyRank == Root){

       for(iproc = 1 ; iproc < Numprocs ; iproc++){
           Source     = iproc;
           Source_tag = 0;

           MPI_Recv(&value, 1, MPI_INT, Source, Source_tag,
                     MPI_COMM_WORLD, &status);
           sum = sum + value;
       }
       printf("MyRank = %d, SUM = %d\n", MyRank, sum);
    }
    else{
        Destination     = 0;
        Destination_tag = 0;

        MPI_Send(&MyRank, 1, MPI_INT, Destination, Destination_tag,
                  MPI_COMM_WORLD);
    }

    MPI_Finalize();

}
```

9. Design a program that implements MPI Collective Communications

```
#include <stdio.h>
#include "mpi.h"

int main(int argc,char *argv[])
{
    int    sum = 0;
    int    MyRank, Numprocs, Root = 0;
    MPI_Status status;

    /*....MPI Initialisation....*/

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&Numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);

    /*....The REDUCE function of MPI....*/

    MPI_Reduce(&MyRank, &sum, 1, MPI_INT, MPI_SUM, Root, MPI_COMM_WORLD);

    if(MyRank == Root)
                printf("SUM = %d\n", sum);

    MPI_Finalize();
}
```

Or (scatter)

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4

int main (int argc, char *argv[])
{
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
              MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
```

```
          recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

10. Implement Cartesian Virtual Topology in MPI

```
#include "mpi.h"
 #include <stdio.h>
 #include <stdlib.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();
}
```

11. Design a MPI program that uses blocking send/receive routines.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  if (numtasks > 2)
    printf("Numtasks=%d. Only 2 needed. Ignoring extra...\n",numtasks);
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

if (rank < 2) {
  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
         rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
  }

MPI_Finalize();
}
```

12. Design a MPI program that uses nonblocking send/receive routines.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

MPI_Waitall(4, reqs, stats);
printf("Task %d communicated with tasks %d & %d\n",rank,prev,next);

MPI_Finalize();
}
```