| ***Department:*** *Computer Science & Engineering* | ***Course Type:*** *Program Core* |
| --- | --- |
| ***Course Title:*** *High Performance Computing Lab* | ***Course Code:*** *18CSL77* |
| ***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 Heterogeneous 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:



1. Given a nxn matrix A and a vector x of length n, their product y=A⋅x. Write a program to implement the multiplication using OpenMP PARALLEL directive.
2. Consider a Scenario where a person visits a supermarket for shopping. He purchases various items in different sections such as clothing, gaming, grocery, stationary. Write an open MP program to process his bill parallelly in each section and display the final amount to be paid. (sum of elements parallelly)
3. Consider a Person named X on the earth, to find his accurate position on the globe we require the value of Pi. Write a program to compute the value of pi function by Numerical Integration using OpenMP PARALLEL section.
4. Using OpenMP, Design and develop 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. University awards gold medals to the student who has scored highest CGPA. Write a program to find the student with highest CGPA in a list of numbers using OpenMP.
6. Design open MP program to multiply 2 square matrices and analyse the performance for the matrix dimension of 1000, 2000 and 3000.
7. 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.
8. Design a program that implements application of MPI Collective Communications.
9. Implement Cartesian Virtual Topology in MPI.
10. Design a MPI program to simulate the uses blocking send/receive routines and nonblocking send/receive routines.



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 the efficiency and power of parallelism.

# ASSESSMENT METHODS

| **HPC Lab** | | |
| --- | --- | --- |
| Lab Internals 1 | Open MP Programs | 15 Marks |
| Lab Internals 2 | MPI Programs | 10 Marks |
| Viva |  | 5 Marks |
| Record |  | 10 Marks |
| Observation and Conduction |  | 5 Marks |
| Programming Assignment (Quiz) |  | 5 Marks |
| **Total** | | **50 Marks** |

* + **Final examination will be conducted for 100 marks on Part A questions only and evaluated for 50 Marks.**



Students will be able to

| **CO** | **Description** | **Bloom’s Level** |
| --- | --- | --- |
| **CO 1:** | Design and implement high performance versions of standard single threaded algorithms | L5 |
| **CO 2:** | Demonstrate the architectural features in the GPU and MIC hardware  Accelerators | L3 |
| **CO 3:** | Design programs to extract maximum performance in a multicore, shared  memory execution environment processor | L4 |
| **CO 4:** | Develop programs using OPENMP, MPI and CUDA | L5 |
| **CO 5:** | Design and deploy Parallel programs on Processor clusters | L4 |

| **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 |  |  |
| **Correla**  **tion Level** | 3 | 3 | 2 | 3 | 3 |  |  |  |  |  |  |  | 3 |  |  |

**Programs**

1. **Given a nxn matrix A and a vector x of length n, their product y=A⋅x. 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]);

}

}

Output1:

Enter the size of square matrix : 5

Enter the size of vector : 5

Matrix \* Vector = Resultant Matrix

86 77 15 93 35 83 24900

36 11 68 67 29 72 18040

59 63 26 40 26 \* 90 = 17345

92 49 21 62 27 86 20600

30 62 23 67 35 82 17656

Output2:

Enter the size of square matrix : 4

Enter the size of vector : 2

Multiplication is not possible.

1. **Consider a Scenario where a person visits a supermarket for shopping. He purchases various items in different sections such as clothing, gaming, grocery, stationery. Write an open MP program to process his bill parallelly in each section and display the final amount to be paid. (sum of elements parallelly)**

#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);

}

Output:

Enter number of sections : 4

Section - 0 ( 3 Items ) : 86 77 15

Section - 1 ( 3 Items ) : 93 35 86

Section - 2 ( 10 Items ) : 92 49 21 62 27 90 59 63 26 40

Section - 3 ( 6 Items ) : 85 68 40 25 40 72

Total Amount : 1251

1. **Consider a Person named X on the earth, to find his accurate position on the globe we require the value of Pi. Write a program to compute the value of pi function by Numerical Integration using OpenMP PARALLEL section**.

#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);

}

Output1:

Enter the number of steps : 4

Time Taken : 0.000214

Value of Pi = 3.146801

Output2:

Enter the number of steps : 6

Time Taken : 0.000310

Value of Pi = 3.143907

1. **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<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;

}

**Output:** Number of terms : 25

id of thread involved in the computation of fibonacci numbers = 0

id of thread involved in the displaying of fibonacci numbers = 1

Fibonacci numbers : 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368

Time Taken : 0.226000ms

1. **University awards gold medals to the student who has scored highest CGPA. Write a program to find the student with highest CGPA in a list of numbers using OpenMP.**

#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));

}

Output:

Enter the number of students : 25

CGPA of students : 798.50 938.30 717.60 474.60 990.20 876.50 159.80 867.70 789.60 387.00 929.50 200.80 376.00 89.00 668.30 909.30 191.10 985.50 859.80 144.90 884.70 347.60 288.70 167.50 830.10

Student with highest CGPA = 990.20

Time Taken : 74.65ms

1. **Design open MP program to multiply 2 square matrices and analyse the performance for the matrix dimension of 1000, 2000 and 3000.**

#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);

}

Output:

Enter the dimension of square matrices : 100

Time taken by parallel algorithm : 0.014094

Time taken by Sequential algorithm : 0.013221

1. **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();

}

Output:

Robot 3 picked 46 mangoes.

Robot 1 picked 83 mangoes.

Robot 2 picked 90 mangoes.

Robot 0 picked 83 mangoes.

Total Mangoes picked by 4 Robots = 302

1. **Design a program that implements MPI Collective Communications**

#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();

}

Output:

Before Scatter : sendbuf of rank 0 : 38.3 38.3 29.0 74.6 30.1 67.5 54.1 67.7 89.6 91.5

After Scatter :

rank= 0 Recvbuf: 38.3

rank= 8 Recvbuf: 89.6

rank= 9 Recvbuf: 91.5

rank= 1 Recvbuf: 38.3

rank= 3 Recvbuf: 74.6

rank= 5 Recvbuf: 67.5

rank= 6 Recvbuf: 54.1

rank= 7 Recvbuf: 67.7

rank= 2 Recvbuf: 29.0

rank= 4 Recvbuf: 30.1

1. **Implement Cartesian Virtual Topology in MPI**

#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();

}

Output:

rank= 0 coords= 0 0 neighbors(u,d,l,r)= -1 4 -1 1

rank= 1 coords= 0 1 neighbors(u,d,l,r)= -1 5 0 2

rank= 2 coords= 0 2 neighbors(u,d,l,r)= -1 6 1 3

rank= 4 coords= 1 0 neighbors(u,d,l,r)= 0 8 -1 5

rank= 5 coords= 1 1 neighbors(u,d,l,r)= 1 9 4 6

rank= 6 coords= 1 2 neighbors(u,d,l,r)= 2 10 5 7

rank= 8 coords= 2 0 neighbors(u,d,l,r)= 4 12 -1 9

rank= 9 coords= 2 1 neighbors(u,d,l,r)= 5 13 8 10

rank= 13 coords= 3 1 neighbors(u,d,l,r)= 9 -1 12 14

rank= 3 coords= 0 3 neighbors(u,d,l,r)= -1 7 2 -1

rank= 7 coords= 1 3 neighbors(u,d,l,r)= 3 11 6 -1

rank= 10 coords= 2 2 neighbors(u,d,l,r)= 6 14 9 11

rank= 11 coords= 2 3 neighbors(u,d,l,r)= 7 15 10 -1

rank= 12 coords= 3 0 neighbors(u,d,l,r)= 8 -1 -1 13

rank= 14 coords= 3 2 neighbors(u,d,l,r)= 10 -1 13 15

rank= 15 coords= 3 3 neighbors(u,d,l,r)= 11 -1 14 -1

rank= 8 inbuf(u,d,l,r)= 4 12 -1 9

rank= 2 inbuf(u,d,l,r)= -1 6 1 3

rank= 15 inbuf(u,d,l,r)= 11 -1 14 -1

rank= 13 inbuf(u,d,l,r)= 9 -1 12 14

rank= 1 inbuf(u,d,l,r)= -1 5 0 2

rank= 5 inbuf(u,d,l,r)= 1 9 4 6

rank= 9 inbuf(u,d,l,r)= 5 13 8 10

rank= 0 inbuf(u,d,l,r)= -1 4 -1 1

rank= 4 inbuf(u,d,l,r)= 0 8 -1 5

rank= 11 inbuf(u,d,l,r)= 7 15 10 -1

rank= 3 inbuf(u,d,l,r)= -1 7 2 -1

rank= 6 inbuf(u,d,l,r)= 2 10 5 7

rank= 14 inbuf(u,d,l,r)= 10 -1 13 15

rank= 10 inbuf(u,d,l,r)= 6 14 9 11

rank= 7 inbuf(u,d,l,r)= 3 11 6 -1

rank= 12 inbuf(u,d,l,r)= 8 -1 -1 13

1. **Design an MPI program that uses blocking send/receive routines Design a MPI program that uses nonblocking send/receive routines.**

Blocking

#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();

}

Output:

Task 0: Received 1 char(s) from task 1 with tag 1

Task 1: Received 1 char(s) from task 0 with tag 1

Non Blocking

#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();

}

Output:

Task 0: Received 1 char(s) from task 1 with tag 1

Task 1: Received 1 char(s) from task 0 with tag 1