HPC Lab Programs

1) program - Matrix Multiplication

File: New project - visual C++ - console Application.

Project - properties

1. C/ C++ : Mullti- processor compilation - Yes / Mp
2. Code generation: Enable Parallel Code Generation: yes / Qpar
3. Language : Open Mp support: yes / openmp

CODE:

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

/* Main Program */

int 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);

}
```

Debug: Start Debugging

O/p:

Read the matrix size noofrows and columns and vectorsize
3 **4**
4 **The Matrix is**
0.000000 1.000000 2.000000 3.000000
1.000000 2.000000 3.000000 4.000000
2.000000 3.000000 4.000000 5.000000
The Vector is
0.000000 1.000000 2.000000 3.000000
The Matrix Computation result is
14.000000

20.000000
26.000000

2)Sum of elements

CODE:

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

/* Main Program */

int 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;

/* Serail 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);
}
```

Enter the size of the array **3**
The Array Elements Are
Array[0]=0.000000
Array[1]=5.000000
Array[2]=10.000000
The Serial And Parallel Sums Are Equal
The SumOfElements Of The Array Using OpenMP Directives Is 15.000000
The SumOfElements Of The Array By Serial Calculation Is 15.000000

3)Pi evaluation

CODE:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 2

static long steps = 1000000000;
double step;

int main(int argc, const char *argv[]) {

int i, j;
double x;
double pi, sum = 0.0;
double start, delta;

step = 1.0 / (double)steps;

// Compute parallel compute times for 1-MAX_THREADS
for (j = 1; j <= MAX_THREADS; j++) {

printf(" running on %d threads: ", j);

// This is the beginning of a single PI computation
omp_set_num_threads(j);

sum = 0.0;
double start = omp_get_wtime();


#pragma omp parallel for reduction(+:sum) private(x)
for (i = 0; i < steps; i++) {
x = (i + 0.5)*step;
sum += 4.0 / (1.0 + x * x);
}

// Out of the parallel region, finialize computation
pi = step * sum;
delta = omp_get_wtime() - start;
printf("PI = %.16g computed in %.4g seconds\n", pi, delta);

}


}
```

On RHS, check diagnostic tool, check process Memory and CPU

running on 1 threads: PI = 3.141592653589971 computed in 7.81 seconds
running on 2 threads: PI = 3.141592653589971 computed in 7.682 seconds

4)Fibonacci series

CODE:

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

int fib(int n)
{
if (n < 2) return n;
else return fib(n - 1) + fib(n - 2);
}

int main()
{
int fibnumber[100], i, j, n;
printf("Please Enter the series limit\n");
scanf_s("%d", &n);
#pragma omp parallel num_threads(2)
{
#pragma omp critical
if (omp_get_thread_num() == 0)
{
printf("There are %d threads\n", omp_get_num_threads());
printf("Thread %d generating numbers..\n", omp_get_thread_num());
for (i = 0; i < n; i++)
fibnumber[i] = fib(i);


printf("Thread %d Printing numbers..\n", omp_get_thread_num());
for (j = 0; j < n; j++)
printf("%d\t", fibnumber[j]);
}

}
return 0;
}
```

Please Enter the series limit
5
There are 1 threads

Thread 0 generating numbers..

5. Recursive Computation

```
// lab5.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

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

/* Main Program */

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

	/* Size Of An Array */

	printf("Enter the size \n");
	scanf_s("%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);
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
```

// 5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project

// 6. In the future, to open this project again, go to File > Open > Project and select the .sln file

Output:

```
Enter the size                                                    5
The Input Array Is
0.000000        1.000000        2.000000        3.000000        4.000000
The Array Calculation Is Same Using Serial And OpenMP Directives
The Output Array Is
0.000000

0.500000

 1.250000

2.125000

3.062500
```

6. Largest element in a list of numbers

```
// lab6.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <iostream>

#include <stdio.h>

#include <omp.h>

#define MAXIMUM 65536

/* Main Program */

int main()

{
```

```
	int* array, i, Noofelements, cur_max, current_value;

	printf("Enter the number of elements\n");
	scanf_s("%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);
```

```
}
// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln file
```

Output:

```
Enter the number of elements                                    6
The Input Array Elements Are
17443   1584    32475   22998   13732   26862
The Max Value Is Same From Serial And Parallel OpenMP Directive
The Largest Number In The Given Array Is 32475
```

7. To print sum of elements

```
// lab7.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <iostream>

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

```
/* Main Program */

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

	printf("Enter the size of the array \n");
	scanf_s("%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);
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
```

// Tips for Getting Started:

// 1. Use the Solution Explorer window to add/manage files

// 2. Use the Team Explorer window to connect to source control

// 3. Use the Output window to see build output and other messages

// 4. Use the Error List window to view errors

// 5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project

// 6. In the future, to open this project again, go to File > Open > Project and select the .sln file

Output:

```
Enter the size of the array                                                    6
The input array is
5.000000        6.000000        7.000000        8.000000        9.000000        10.000000
The calculation of array sum is same
The value of array sum using threads is 45.000000
The serial calculation of array is 45.000000
```

8. Please check the configuration

```
#include <stdio.h>
#include <mpi.h>
using namespace std;

int main(int argc, char** argv) {
        int mynode, totalnodes;
        int sum, startval, endval, accum;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD, &mynode);
        sum = 0;
        startval = 1000 * mynode / totalnodes + 1;
        endval = 1000 * (mynode + 1) / totalnodes;
        for (int i = startval;i <= endval;i = i + 1)
                sum = sum + i;
        if (mynode != 0)
                MPI_Send(&sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        else
                for (int j = 1;j < totalnodes;j = j + 1) {
                        MPI_Recv(&accum, 1, MPI_INT, j, 1, MPI_COMM_WORLD, &status);
                        sum = sum + accum;
                }
        if (mynode == 0)
                printf("The sum is %d\n", sum);
        MPI_Finalize();
}
```

Output: The sum is 500500

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

    /----------------/
   /* Initialize MPI */
  /----------------/

    MPI_Init(&argc, &argv);

    /-------------------------------------------------------/
   /* Get the size of the MPI_COMM_WORLD communicator group */
  /-------------------------------------------------------/

    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    if (numtasks == SIZE) {

        /-----------------------------------------------------------------/
       /* Make a new communicator to which 2-D Cartesian topology is attached */
      /-----------------------------------------------------------------/

        MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
```

```
  /----------------------------------------/
 /* Get my rank in the cartcomm communicator */
/----------------------------------------/

  MPI_Comm_rank(cartcomm, &rank);

  /------------------------------------------------------------------/
 /* Determine process coords in cartesian topology given rank in group */
/------------------------------------------------------------------/

  MPI_Cart_coords(cartcomm, rank, 2, coords);

  /------------------------------------------------------------------/
 /* Obtain the shifted source and destination ranks in both directions */
/------------------------------------------------------------------/
  MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
  MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

  outbuf = rank;

  for (i = 0; i < 4; i++) {
      dest = nbrs[i];
      source = nbrs[i];

      /-------------------------------------------/
     /* send messages to the four adjacent processes */
    /-------------------------------------------/

      MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
          MPI_COMM_WORLD, &reqs[i]);
```

```
            /------------------------------------------------/
            /* receive messages from the four adjacent processes */
           /------------------------------------------------/

            MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
                MPI_COMM_WORLD, &reqs[i + 4]);
        }

        /------------------------------------------------/
       /* Wait for all 8 communication tasks to complete */
      /------------------------------------------------/

        MPI_Waitall(8, reqs, stats);

        printf("rank = %2d coords = %2d%2d neighbors(u,d,l,r) = %2d %2d %2d %2d\n",
            rank, coords[0], coords[1], nbrs[UP], nbrs[DOWN], nbrs[LEFT], nbrs[RIGHT]);
        printf("rank = %2d                  inbuf(u,d,l,r) = %2d %2d %2d %2d\n",
            rank, inbuf[UP], inbuf[DOWN], inbuf[LEFT], inbuf[RIGHT]);
    }
    else
        printf("Must specify %d processors. Terminating.\n", SIZE);

    /--------------/
   /* Finalize MPI */
  /--------------/

    MPI_Finalize();

}
```

```
#include <iostream>
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define MASTER 0


int main(int argc, char* argv[])
{
   int  numtasks, taskid, len, partner, message;
   char hostname[MPI_MAX_PROCESSOR_NAME];
   MPI_Status status;

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

   /* need an even number of tasks  */
   if (numtasks % 2 != 0) {
      if (taskid == MASTER)
         printf("Quitting. Need an even number of tasks: numtasks=%d\n", numtasks);
   }

   else {
      if (taskid == MASTER)
         printf("MASTER: Number of MPI tasks is: %d\n", numtasks);

      MPI_Get_processor_name(hostname, &len);
      printf("Hello from task %d on %s!\n", taskid, hostname);

      /* determine partner and then send/receive with partner */
```

```
        if (taskid < numtasks / 2) {
            partner = numtasks / 2 + taskid;
            MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
            MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
        }
        else if (taskid >= numtasks / 2) {
            partner = taskid - numtasks / 2;
            MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
            MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
        }

        /* print partner info and exit*/
        printf("Task %d is partner with %d\n", taskid, message);
    }

    MPI_Finalize();

}



#include <iostream>

#include "mpi.h"
#include <math.h>
#include <stdio.h>
void main(int argc, char* argv[])
{
          int p, i, Iam, root;
          int counts[4] = { 1, 2, 3, 4 };
          int displs[4] = { 0, 1, 3, 6 };
          char x[10], y[10], a, alphabet;
          /----------------/
```

```
 /* initialize MPI */
/----------------/

        MPI_Init(&argc, &argv);

        /--------------------------/
  /* get the process ID number */
 /--------------------------/

        MPI_Comm_rank(MPI_COMM_WORLD, &Iam);

        /----------------------------------/
  /* get the size of the process group */
 /----------------------------------/

        MPI_Comm_size(MPI_COMM_WORLD, &p);
        root = 1;
        if (Iam == 0) {
                printf("  Function  Proc  Sendbuf             Recvbuf\n");
                printf("  --------  ----  -------             -------\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);

        for (i = 0; i < p; i++) {
                x[i] = ' ';
        }

        alphabet = 'a';
        /---------------------------------/
  /* MPI_Gather()                    */
 /---------------------------------/
```

```
        x[0] = alphabet + Iam;
        for (i = 0; i < p; i++) {
                y[i] = ' ';
        }
        MPI_Gather(x, 1, MPI_CHAR,          /* send buf,count,type */
                y, 1, MPI_CHAR,          /* recv buf,count,type */
                root,                    /* root (data origin)  */
                MPI_COMM_WORLD);         /* comm                */

        printf(" MPI_Gather    :  %d ", Iam);
        for (i = 0; i < p; i++) {
                printf("  %c", x[i]);
        }
        printf("     ");
        for (i = 0; i < p; i++) {
                printf("  %c", y[i]);
        }
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);
        /---------------------------------/
 /* MPI_Gatherv()                    */
/---------------------------------/

        for (i = 0; i < 10; i++) {
                x[i] = ' ';
                y[i] = ' ';
        }
        for (i = 0; i < counts[Iam]; i++) {
                x[i] = alphabet + displs[Iam] + i;
        }
```

```
        MPI_Gatherv(x, counts[Iam], MPI_CHAR, /* send buf,count,type  */
                y, counts,              /* recv buf,count array */
                displs, MPI_CHAR,       /* displacements,type   */
                root,                   /* root (data origin)  */
                MPI_COMM_WORLD);        /* comm                 */

        printf(" MPI_Gatherv   :  %d ", Iam);
        for (i = 0; i < p; i++) {
                printf("  %c", x[i]);
        }
        printf("     ");
        for (i = 0; i < 10; i++) {
                printf("  %c", y[i]);
        }
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);
        /---------------------------------/
 /* MPI_Allgather()                  */
/---------------------------------/

        x[0] = alphabet + Iam;

        for (i = 0; i < p; i++) {
                y[i] = ' ';
        }
        MPI_Allgather(x, 1, MPI_CHAR,     /* send buf,count,type */
                y, 1, MPI_CHAR,     /* recv buf,count,type */
                MPI_COMM_WORLD);  /* comm                 */

        printf(" MPI_Allgather :  %d ", Iam);
        for (i = 0; i < p; i++) {
```

```
            printf("  %c", x[i]);
        }
        printf("     ");
        for (i = 0; i < p; i++) {
            printf("  %c", y[i]);
        }
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);

        /---------------------------------/
 /* MPI_Allgatherv()                */
/---------------------------------/

        for (i = 0; i < 10; i++) {
            x[i] = ' ';
            y[i] = ' ';
        }
        for (i = 0; i < counts[Iam]; i++) {
            x[i] = alphabet + displs[Iam] + i;
        }

        MPI_Allgatherv(x, counts[Iam], MPI_CHAR, /* send buf,count,type  */
            y, counts,              /* recv buf,count array */
            displs, MPI_CHAR,       /* displacements,type   */
            MPI_COMM_WORLD);        /* comm                 */

        printf(" MPI_Allgatherv:  %d ", Iam);
        for (i = 0; i < p; i++) {
            printf("  %c", x[i]);
        }
        printf("     ");
```

```
        for (i = 0; i < 10; i++) {
                printf("  %c", y[i]);
        }
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);

        /---------------------------------/
 /* MPI_Scatter()                    */
/---------------------------------/

        for (i = 0; i < p; i++) {
                x[i] = alphabet + i + Iam * p;
                y[i] = ' ';
        }
        MPI_Scatter(x, 1, MPI_CHAR,      /* send buf,count,type */
                y, 1, MPI_CHAR,      /* recv buf,count,type */
                root,              /* root (data origin)  */
                MPI_COMM_WORLD);   /* comm                */

        printf(" MPI_Scatter   :  %d ", Iam);
        for (i = 0; i < p; i++) {
                printf("  %c", x[i]);
        }
        printf("     ");
        for (i = 0; i < p; i++) {
                printf("  %c", y[i]);
        }
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);
```

```
        /---------------------------------/
 /* MPI_Alltoall()                  */
/---------------------------------/

        for (i = 0; i < p;i++) {
                x[i] = alphabet + i + Iam * p;
                y[i] = ' ';
        }

        MPI_Alltoall(x, 1, MPI_CHAR,       /* send buf,count,type */
                y, 1, MPI_CHAR,       /* recv buf,count,type */
                MPI_COMM_WORLD);    /* comm,flag           */

        printf(" MPI_Alltoall  :  %d ", Iam);
        for (i = 0; i < p; i++) {
                printf("  %c", x[i]);
        }
        printf("     ");
        for (i = 0; i < p; i++) {
                printf("  %c", y[i]);
        }
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);

        /---------------------------------/
 /* MPI_Reduce()                    */
/---------------------------------/

        for (i = 0; i < p; i++) {
                x[i] = alphabet + i + Iam * p;
                y[i] = ' ';
```

```
        }
        MPI_Reduce(x, y,            /* send buf, recv buf  */
                p, MPI_CHAR,     /* count,type          */
                MPI_MAX,         /* operation           */
                root,            /* root (data origin)  */
                MPI_COMM_WORLD); /* comm                */

        printf(" MPI_Reduce MAX:  %d ", Iam);
        for (i = 0; i < p; i++) {
                printf("  %c", x[i]);
        }
        printf("     ");
        for (i = 0; i < p; i++) {
                printf("  %c", y[i]);
        }
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);

        /---------------------------------/
 /* MPI_Allreduce()                  */
/---------------------------------/

        for (i = 0; i < p; i++) {
                x[i] = alphabet + i + Iam * p;
                y[i] = ' ';
        }
        MPI_Allreduce(x, y,         /* send buf, recv buf  */
                p, MPI_CHAR,     /* count,type          */
                MPI_MAX,         /* operation           */
                MPI_COMM_WORLD); /* comm                */
```

```
        printf(" MPI_Allreduce :  %d ", Iam);
        for (i = 0; i < p; i++) {
                printf("  %c", x[i]);
        }
        printf("     ");
        for (i = 0; i < p; i++) {
                printf("  %c", y[i]);
        }
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);

        /---------------------------------/
 /* MPI_Bcast()                          */
/---------------------------------/

        a = ' ';
        for (i = 0; i < p; i++) {
                x[i] = ' ';
                y[i] = ' ';
        }
        if (Iam == root) {
                a = 'b';
                x[0] = a;
        }
        MPI_Bcast(&a, 1, MPI_CHAR,          /* buf,count,type */
                root, MPI_COMM_WORLD);   /* root,comm      */

        printf(" MPI_Bcast     :  %d ", Iam);
        for (i = 0; i < p; i++) {
                printf("  %c", x[i]);
        }
```

```
        printf("     ");
        printf("  %c", a);
        printf("\n");

        MPI_Barrier(MPI_COMM_WORLD);

        /--------------/
   /* Finalize MPI */
  /--------------/

        MPI_Finalize();
}


#include <iostream>
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER              0

int main(int argc, char* argv[])
{
    int  numtasks, taskid, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];
    int  partner, message;
    MPI_Status stats[2];
    MPI_Request reqs[2];

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

```
    /* need an even number of tasks  */
    if (numtasks % 2 != 0) {
       if (taskid == MASTER)
          printf("Quitting. Need an even number of tasks: numtasks=%d\n", numtasks);
    }

    else {
       MPI_Get_processor_name(hostname, &len);
       printf("Hello from task %d on %s!\n", taskid, hostname);
       if (taskid == MASTER)
          printf("MASTER: Number of MPI tasks is: %d\n", numtasks);

       /* determine partner and then send/receive with partner */
       if (taskid < numtasks / 2)
          partner = numtasks / 2 + taskid;
       else if (taskid >= numtasks / 2)
          partner = taskid - numtasks / 2;

       MPI_Irecv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &reqs[0]);
       MPI_Isend(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &reqs[1]);

       /* now block until requests are complete */
       MPI_Waitall(2, reqs, stats);

       /* print partner info and exit*/
       printf("Task %d is partner with %d\n", taskid, message);
    }

    MPI_Finalize();

}
```