**Program 1:**

#include<omp.h>

#include<stdio.h>

#include<stdlib.h>

void merge(int\* a, int l, int mid, int r)

{

int n1 = mid-l+1;

int n2 = r-mid;

int b[n1], c[n2];

int k = l;

for(inti = 0; i<n1; i++)

b[i] = a[k++];

for(inti = 0; i<n2; i++)

c[i] = a[k++];

k = l;

inti = 0, j = 0;

while(i<n1 & j<n2)

{

if(b[i]<c[j])

{

a[k++] = b[i++];

}

else

{

a[k++] = c[j++];

}

}

while(i<n1)

a[k++] = b[i++];

while(j<n2)

a[k++] = c[j++];

}

void mergesort(int\* a, int l, int r)

{

if(l<r)

{

int mid;

#pragma omp parallel sections

{

mid = (l+r)/2;

#pragma omp section

{

//printf("thread id = %d\t l=%d\t mid=%d\n",omp\_get\_thread\_num(),l,mid);

mergesort(a, l, mid);

}

#pragma omp section

{

//printf("thread id = %d\t r=%d\t mid+1=%d\n",omp\_get\_thread\_num(),r,mid+1);

mergesort(a, mid+1, r);

}

}

merge(a,l,mid,r);

}

}

int main()

{

omp\_set\_nested(1);

int start=1;

/\*a=(int\*)malloc(100\*sizeof(int));

for(inti = 0; i<100; i++)

a[i] = rand()%1000;

mergesort(a,0,99);\*/

printf("\n\nInput Size\t1\t2\t4\t8\t");

for(inti = 0; i<4; i++)

{

int size = start\*10;

start = size;

int a[size];

for(int j = 0; j<size; j++)

{

a[j] = rand()%100000;

}

printf("\n\n%d\t",size);

for(inti = 0; i<4; i++)

{

omp\_set\_num\_threads(2\*(i));

double t1 = omp\_get\_wtime();

mergesort(a,0,size-1);

double t2 = omp\_get\_wtime();

printf("%lf\t",t2-t1);

}

}

return 0;

}

**Program 2:**

#include <stdio.h>

#include <omp.h>

/\* Main Program \*/

main()

{

intNoofRows, NoofCols, Vectorsize, i, j;

/\*float \*\*Matrix, \*Vector, \*Result, \*Checkoutput;\*/

double \*\*Matrix, \*Vector, \*Result, \*Checkoutput;

printf("Read the matrix size noofrows and columns and vectorsize\n");

scanf("%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); \*/

Matrix = (double \*\*) malloc(sizeof(double) \* NoofRows);

for (i = 0; i<NoofRows; i++)

{

/\* Matrix[i] = (float \*) malloc(sizeof(float) \* NoofCols); \*/

Matrix[i] = (double \*) malloc(sizeof(double) \* 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("%lf \t", Matrix[i][j]);

printf("\n");

}

printf("\n");

/\* Dynamic Memory Allocation \*/

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

Vector = (double \*) malloc(sizeof(double) \* 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("%lf \t", Vector[i]);

/\* Dynamic Memory Allocation \*/

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

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

Result = (double \*) malloc(sizeof(double) \* NoofRows);

Checkoutput = (double \*) malloc(sizeof(double) \* NoofRows);

for (i = 0; i<NoofRows; i = i + 1)

{

Result[i]=0;

Checkoutput[i]=0;

}

/\* OpenMP Parallel Directive \*/

omp\_set\_num\_threads(32);

#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];

/\* Checking with the serial calculation \*/

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("%lf \n", Result[i]);

/\* Freeing The Memory Allocations \*/

free(Vector);

free(Result);

free(Matrix);

free(Checkoutput);

}

**Output:**

/PP\_Lab$ gcc -fopenmp mul\_matrixvector\_lab10.c

/PP\_Lab$ ./a.out mul\_matrixvector\_lab10

Read the matrix size noofrows and columns and vectorsize

3 3

3

**Program 3:**

#include<omp.h>  
#include<stdio.h>  
#include<stdlib.h>  
#include<math.h>  
  
int \*prime\_table ( intprime\_num )  
{  
    //printf("prime table by thread %d", omp\_get\_thread\_num());  
  inti;  
  int j;  
  int p;  
  int prime;  
  int \*primes;  
  primes = ( int \* ) malloc ( prime\_num \* sizeof ( int ) );  
  i = 2;  
  p = 0;  
  while ( p <prime\_num )  
  {  
    prime = 1;  
    for ( j = 2; j <i; j++ )  
    {  
      if ( ( i % j ) == 0 )  
      {  
        prime = 0;  
        break;  
      }  
    }        
    if ( prime )  
    {  
      primes[p] = i;  
      p = p + 1;  
    }  
    i = i + 1;  
  }  
  return primes;  
}  
double \*sine\_table ( intsine\_num )  
{  
    //printf("sine table by thread %d", omp\_get\_thread\_num());  
  double a;  
  inti;  
  int j;  
  double pi = 3.141592653589793;  
  double \*sines;  
  sines = ( double \* ) malloc ( sine\_num \* sizeof ( double ) );  
  for ( i = 0; i<sine\_num; i++ )  
  {  
    sines[i] = 0.0;  
    for ( j = 0; j <= i; j++ )  
    {  
      a = ( double ) ( j ) \* pi / ( double ) ( sine\_num - 1 );  
      sines[i] = sines[i] + sin ( a );  
    }  
  }  
  return sines;  
}  
int main()  
{  
    omp\_set\_nested(1);  
    int size=10;  
    printf("\n\nInput Size\t1\t2\t4\t8\t");  
    for(inti = 0; i<5; i++)  
    {  
        printf("\n\n%d\t",size);  
        for(int x = 0; x<4; x++)  
        {      
            double t1 = omp\_get\_wtime();  
            #pragma omp parallel sections  
            {  
                omp\_set\_num\_threads(2\*x);  
                #pragma omp section  
                {  
                    int\* a = (int\*)malloc(size\*sizeof(int));  
                    a = prime\_table(size);  
                    /\*for(int y=0; y<size; y++)  
                    {  
                        printf("%d\n",a[y]);  
                    }\*/  
                }  
                #pragma omp section  
                {  
                   double\* b = (double\*)malloc(size\*sizeof(double));  
                    b = sine\_table(size);  
                    for(int z=0; z<size; z++)  
                    {  
                        printf("%lf\n",b[z]);  
                    }  
                }  
            }  
            double t2 = omp\_get\_wtime();  
            printf("%lf\t",t2-t1);  
        }  
        size = size\*10;      
    }  
    return 0;  
}

**Program 4:**

#include <stdio.h>

#include <malloc.h>

#include <omp.h>

long long factorial(long n)

{

long long i,out;

out = 1;

for (i=1; i<n+1; i++) out \*= i;

return(out);

}

int main(int argc, char \*\*argv)

{

int i,j,threads;

long long \*x;

long long n=12;

/\* Set number of threads equal to argv[1] if present \*/

if (argc > 1)

{

threads = atoi(argv[1]);

if (omp\_get\_dynamic())

{

omp\_set\_dynamic(0);

printf("called omp\_set\_dynamic(0)\n");

}

omp\_set\_num\_threads(threads);

}

printf("%d threads\n",omp\_get\_max\_threads());

x = (long long \*) malloc(n \* sizeof(long));

for (i=0;i<n;i++) x[i]=factorial(i);

j=0;

/\* Is the output the same if the following line is commented out? \*/

#pragma omp parallel for firstprivate(x,j)

for (i=1; i<n; i++)

{

j += i;

x[i] = j\*x[i-1];

}

for (i=0; i<n; i++)

printf("factorial(%2d)=%14lld x[%2d]=%14lld\n",i,factorial(i),i,x[i]);

return 0;

}

**Output:**

/PP\_Lab$ gcc -fopenmp factorial\_lab13.c

/PP\_Lab$ ./a.out factorial\_lab13

1 threads

factorial( 0)= 1 x[ 0]= 1

factorial( 1)= 1 x[ 1]= 1

factorial( 2)= 2 x[ 2]= 3

factorial( 3)= 6 x[ 3]= 18

factorial( 4)= 24 x[ 4]= 180

factorial( 5)= 120 x[ 5]= 2700

factorial( 6)= 720 x[ 6]= 56700

factorial( 7)= 5040 x[ 7]= 1587600

factorial( 8)= 40320 x[ 8]= 57153600

factorial( 9)= 362880 x[ 9]= 2571912000

factorial(10)= 3628800 x[10]= 141455160000

factorial(11)= 39916800 x[11]= 9336040560000

**Program 5:**

#include <iostream>

#include <cstdlib> // or <stdlib.h> rand, srand

#include <ctime> // or <time.h> time

#include <omp.h>

#include <math.h>

#define K 4

using namespace std;

intnum\_threads;

longnum\_points;

long\*\* points; // 2D array points[x][0] -> point location, points[x][1] -> distance from cluster mean

int cluster[K][2] = {

{75, 25}, {25, 25}, {25, 75}, {75, 75}

};

longcluster\_count[K];

voidpopulate\_points() {

// Dynamically allocate points[num\_points][2] 2D array

points = new long\*[num\_points];

for (long i=0; i<num\_points; i++)

points[i] = new long[2];

// Fill random points (0 to 100)

srand(time(NULL));

for (long i=0; i<num\_points; i++) {

points[i][0] = rand() % 100;

points[i][1] = rand() % 100;

}

// Initialize cluster\_count

for (inti=0; i<K; i++) {

cluster\_count[i] = 0;

}

}

doubleget\_distance(int x1, int y1, int x2, int y2) {

int dx = x2-x1, dy = y2-y1;

return (double)sqrt(dx\*dx + dy\*dy);

}

voidclassify\_points() {

#pragma omp parallel for num\_threads(num\_threads)

for (long i=0; i<num\_points; i++) {

doublemin\_dist = 1000, cur\_dist = 1;

intcluster\_index = -1;

for (int j=0; j<K; j++) {

cur\_dist = get\_distance(

points[i][0], points[i][1],

cluster[j][0], cluster[j][1]

);

if (cur\_dist<min\_dist) {

min\_dist = cur\_dist;

cluster\_index = j;

}

}

cluster\_count[cluster\_index]++;

}

}

int main(intargc, char\* argv[]) {

num\_points = atol(argv[1]);

num\_threads = atoi(argv[2]);

populate\_points();

double t1 = omp\_get\_wtime();

classify\_points();

double t2 = omp\_get\_wtime();

double t = (t2 - t1) \* 1000;

cout<< "Time Taken: " << t << "ms" <<endl;

}

**Program 6:**

#include <stdio.h>

#include <gd.h>

#include <string.h>

#include <omp.h>

int main(intargc, char \*argv[]) {

if (argc< 4) {

printf("Usage: ./negative input.png output.png num\_threads\n");

return 1;

}

char \*input\_file = argv[1];

char \*output\_file = argv[2];

intnum\_threads = atoi(argv[3]);

intcolor, x, y, i;

int red, green, blue;

FILE \*fp;

if((fp = fopen(input\_file, "r")) == NULL) {

printf("Error opening file %s\n", input\_file);

return 1;

}

gdImagePtrimg = gdImageCreateFromPng(fp);

int width = gdImageSX(img);

int height = gdImageSY(img);

double t1 = omp\_get\_wtime();

#pragma omp parallel for private(y, color, red, green, blue) num\_threads(num\_threads)

for(x=0; x<width; x++) {

for(y=0; y<height; y++) {

color = x + 0;

color = gdImageGetPixel(img, x, y);

red = 255 - gdImageRed(img, color);

green = 255 - gdImageGreen(img, color);

blue = 255 - gdImageBlue(img, color);

color = gdImageColorAllocate(img, red, green, blue);

gdImageSetPixel(img, x, y, color);

}

}

double t2 = omp\_get\_wtime();

if((fp=fopen(output\_file, "w")) == NULL) {

printf("Error opening output file %s\n", output\_file);

return 1;

}

gdImagePng(img, fp);

gdImageDestroy(img);

fclose(fp);

printf("File Size: %dx%d\n", width, height);

printf("Time Taken: %.3lfms\n",(t2 - t1) \* 1000);

return 0;

}

Using Critical Section

#include <stdio.h>

#include <gd.h>

#include <string.h>

#include <omp.h>

int main(intargc, char \*argv[]) {

if (argc< 4) {

printf("Usage: ./negative input.png output.png num\_threads\n");

return 1;

}

char \*input\_file = argv[1];

char \*output\_file = argv[2];

intnum\_threads = atoi(argv[3]);

intcolor, x, y, i;

int red, green, blue;

FILE \*fp;

if((fp = fopen(input\_file, "r")) == NULL) {

printf("Error opening file %s\n", input\_file);

return 1;

}

gdImagePtrimg = gdImageCreateFromPng(fp);

int width = gdImageSX(img);

int height = gdImageSY(img);

double t1 = omp\_get\_wtime();

#pragma omp parallel for private(y, color, red, green, blue) num\_threads(num\_threads)

for(x=0; x<width; x++) {

#pragma omp critical

{

for(y=0; y<height; y++) {

color = x + 0;

color = gdImageGetPixel(img, x, y);

red = 255 - gdImageRed(img, color);

green = 255 - gdImageGreen(img, color);

blue = 255 - gdImageBlue(img, color);

color = gdImageColorAllocate(img, red, green, blue);

gdImageSetPixel(img, x, y, color);

}

}

}

double t2 = omp\_get\_wtime();

if((fp=fopen(output\_file, "w")) == NULL) {

printf("Error opening output file %s\n", output\_file);

return 1;

}

gdImagePng(img, fp);

gdImageDestroy(img);

fclose(fp);

printf("File Size: %dx%d\n", width, height);

printf("Time Taken: %.3lfms\n",(t2 - t1) \* 1000);

return 0;

}

**Program 7**

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

// function declaration

void sighup();

void sigint();

void sigquit();

// driver code

void main()

{

int pid;

/\* get child process \*/

if ((pid = fork()) < 0) {

perror("fork");

exit(1);

}

if (pid == 0) { /\* child \*/

signal(SIGHUP, sighup);

signal(SIGINT, sigint);

signal(SIGQUIT, sigquit);

for (;;)

; /\* loop for ever \*/

}

else /\* parent \*/

{ /\* pid hold id of child \*/

printf("\nPARENT: sending SIGHUP\n\n");

kill(pid, SIGHUP);

sleep(3); /\* pause for 3 secs \*/

printf("\nPARENT: sending SIGINT\n\n");

kill(pid, SIGINT);

sleep(3); /\* pause for 3 secs \*/

printf("\nPARENT: sending SIGQUIT\n\n");

kill(pid, SIGQUIT);

sleep(3);

}

}

// sighup() function definition

void sighup()

{

signal(SIGHUP, sighup); /\* reset signal \*/

printf("CHILD: I have received a SIGHUP\n");

}

// sigint() function definition

void sigint()

{

signal(SIGINT, sigint); /\* reset signal \*/

printf("CHILD: I have received a SIGINT\n");

}

// sigquit() function definition

void sigquit()

{

printf("My DADDY has Killed me!!!\n");

exit(0);

}

**Program 8:**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

/\* Define length of dot product vectors \*/

#define VECLEN 100

int main (int argc, char\* argv[])

{

int i,myid, numprocs, len=VECLEN;

double \*a, \*b;

double mysum, allsum;

/\* MPI Initialization \*/

MPI\_Init (&argc, &argv);

MPI\_Comm\_size (MPI\_COMM\_WORLD, &numprocs);

MPI\_Comm\_rank (MPI\_COMM\_WORLD, &myid);

/\*

Each MPI task performs the dot product, obtains its partial sum, and then calls

MPI\_Reduce to obtain the global sum.

\*/

if (myid == 0)

printf("Starting omp\_dotprod\_mpi. Using %d tasks...\n",numprocs);

/\* Assign storage for dot product vectors \*/

a = (double\*) malloc (len\*sizeof(double));

b = (double\*) malloc (len\*sizeof(double));

/\* Initialize dot product vectors \*/

for (i=0; i<len; i++) {

a[i]=1.0;

b[i]=a[i];

}

/\* Perform the dot product \*/

mysum = 0.0;

for (i=0; i<len; i++)

{

mysum += a[i] \* b[i];

}

printf("Task %d partial sum = %f\n",myid, mysum);

/\* After the dot product, perform a summation of results on each node \*/

MPI\_Reduce (&mysum, &allsum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (myid == 0)

printf ("Done. MPI version: global sum = %f \n", allsum);

free (a);

free (b);

MPI\_Finalize();

}

**Program 9:**

#include <stdio.h>

#include <stdlib.h>

#include "mpi.h"

#include <math.h>

#define SEED 35791246

int main(int argc, char\* argv[])

{

    long niter = 1000000;

    int myid;                       //holds process's rank id

    double x,y;                     //x,y value for the random coordinate

    int i, count=0;                 //Count holds all the number of how many good coordinates

    double z;                       //Used to check if x^2+y^2<=1

    double pi;                      //holds approx value of pi

    int nodenum;

    MPI\_Init(&argc, &argv);                 //Start MPI

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);           //get rank of node's process

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &nodenum);

    int recieved[nodenum];

    long recvniter[nodenum];

    srand(SEED+myid);                       //Give rand() a seed value. Needs to be different on each node

    if(myid != 0)

    {

        for (i=0; i<niter; ++i)                  //main loop

        {

            x= ((double)rand())/RAND\_MAX;           //gets a random x coordinate

            y =((double)rand())/RAND\_MAX;           //gets a random y coordinate

            z = sqrt(x\*x+y\*y);                  //Checks to see if number in inside unit circle

            if (z<=1)

            {

                count++;                //if it is, consider it a valid random point

            }

        }

        for(i=0; i<nodenum; ++i)

        {

            MPI\_Send(&count,

                                 1,

                                 MPI\_INT,

                                 0,

                                 1,

                                 MPI\_COMM\_WORLD);

            MPI\_Send(&niter,

                                 1,

                                 MPI\_LONG,

                                 0,

                                 2,

                                 MPI\_COMM\_WORLD);

        }

    }

    else if (myid == 0)

    {

        for(i=0; i<nodenum; ++i)

        {

            MPI\_Recv(&recieved[i],

                                 nodenum,

                                 MPI\_INT,

                                 MPI\_ANY\_SOURCE,

                                 1,

                                 MPI\_COMM\_WORLD,

                                 MPI\_STATUS\_IGNORE);

            MPI\_Recv(&recvniter[i],

                                 nodenum,

                                 MPI\_LONG,

                                 MPI\_ANY\_SOURCE,

                                 2,

                                 MPI\_COMM\_WORLD,

                                 MPI\_STATUS\_IGNORE);

        }

    }

    if (myid == 0)                      //if root process

    {

        int finalcount = 0;

        long finalniter = 0;

        for(i = 0; i<nodenum; ++i)

        {

            finalcount += recieved[i];

            finalniter += recvniter[i];

        }

        pi = ((double)finalcount/(double)finalniter)\*4.0;               //p = 4(m/n)

        printf("Pi: %f\n", pi);             //Print the calculated value of pi

    }

    MPI\_Finalize();                     //Close the MPI instance

    return 0;

**Program 10:**

# include <stdlib.h>

# include <stdio.h>

# include <time.h>

# include "mpi.h"

int main ( int argc, char \*argv[] );

void timestamp ( );

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main ( int argc, char \*argv[] )

{

MPI\_Comm even\_comm\_id;

MPI\_Group even\_group\_id;

int even\_id;

int even\_id\_sum;

int even\_p;

int \*even\_rank;

int i;

int id;

int ierr;

int j;

MPI\_Comm odd\_comm\_id;

MPI\_Group odd\_group\_id;

int odd\_id;

int odd\_id\_sum;

int odd\_p;

int \*odd\_rank;

int p;

MPI\_Group world\_group\_id;

/\*

Initialize MPI.

\*/

ierr = MPI\_Init ( &argc, &argv );

if ( ierr != 0 )

{

printf ( "\n" );

printf ( "COMMUNICATOR\_MPI: Fatal error!\n" );

printf ( " MPI\_Init returned nonzero error flag\n" );

exit ( 1 );

}

/\*

Get the number of processes.

\*/

ierr = MPI\_Comm\_size ( MPI\_COMM\_WORLD, &p );

/\*

Get the individual process ID.

\*/

ierr = MPI\_Comm\_rank ( MPI\_COMM\_WORLD, &id );

/\*

Process 0 prints an introductory message.

\*/

if ( id == 0 )

{

timestamp ( );

printf ( "\n" );

printf ( "COMMUNICATOR\_MPI - Master process:\n" );

printf ( " C/MPI version\n" );

printf ( " An MPI example program.\n" );

printf ( "\n" );

printf ( " The number of processes is %d.\n", p );

printf ( "\n" );

}

/\*

Every process prints a hello.

\*/

printf ( " Process %d says 'Hello, world!'\n", id );

/\*

Get a group identifier for MPI\_COMM\_WORLD.

\*/

MPI\_Comm\_group ( MPI\_COMM\_WORLD, &world\_group\_id );

/\*

List the even processes, and create their group.

\*/

even\_p = ( p + 1 ) / 2;

even\_rank = ( int \* ) malloc ( even\_p \* sizeof ( int ) );

j = 0;

for ( i = 0; i < p; i = i + 2 )

{

even\_rank[j] = i;

j = j + 1;

}

MPI\_Group\_incl ( world\_group\_id, even\_p, even\_rank, &even\_group\_id );

MPI\_Comm\_create ( MPI\_COMM\_WORLD, even\_group\_id, &even\_comm\_id );

/\*

List the odd processes, and create their group.

\*/

odd\_p = p / 2;

odd\_rank = ( int \* ) malloc ( odd\_p \* sizeof ( int ) );

j = 0;

for ( i = 1; i < p; i = i + 2 )

{

odd\_rank[j] = i;

j = j + 1;

}

MPI\_Group\_incl ( world\_group\_id, odd\_p, odd\_rank, &odd\_group\_id );

MPI\_Comm\_create ( MPI\_COMM\_WORLD, odd\_group\_id, &odd\_comm\_id );

/\*

Try to get ID of each process in both groups.

If a process is not in a communicator, set its ID to -1.

\*/

if ( id % 2 == 0 )

{

ierr = MPI\_Comm\_rank ( even\_comm\_id, &even\_id );

odd\_id = -1;

}

else

{

ierr = MPI\_Comm\_rank ( odd\_comm\_id, &odd\_id );

even\_id = -1;

}

/\*

Use MPI\_Reduce to sum the global ID of each process in the even group.

Assuming 4 processes: EVEN\_SUM = 0 + 2 = 2;

\*/

if ( even\_id != -1 )

{

MPI\_Reduce ( &id, &even\_id\_sum, 1, MPI\_INT, MPI\_SUM, 0, even\_comm\_id );

}

if ( even\_id == 0 )

{

printf ( " Number of processes in even communicator = %d\n", even\_p );

printf ( " Sum of global ID's in even communicator = %d\n", even\_id\_sum );

}

/\*

Use MPI\_Reduce to sum the global ID of each process in the odd group.

Assuming 4 processes: ODD\_SUM = 1 + 3 = 4;

\*/

if ( odd\_id != -1 )

{

MPI\_Reduce ( &id, &odd\_id\_sum, 1, MPI\_INT, MPI\_SUM, 0, odd\_comm\_id );

}

if ( odd\_id == 0 )

{

printf ( " Number of processes in odd communicator = %d\n", odd\_p );

printf ( " Sum of global ID's in odd communicator = %d\n", odd\_id\_sum );

}

/\*

Terminate MPI.

\*/

ierr = MPI\_Finalize ( );

/\*

Terminate

\*/

if ( id == 0 )

{

printf ( "\n" );

printf ( "COMMUNICATOR\_MPI:\n" );

printf ( " Normal end of execution.\n" );

printf ( "\n" );

timestamp ( );

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void timestamp ( void )

{

# define TIME\_SIZE 40

static char time\_buffer[TIME\_SIZE];

const struct tm \*tm;

time\_t now;

now = time ( NULL );

tm = localtime ( &now );

strftime ( time\_buffer, TIME\_SIZE, "%d %B %Y %I:%M:%S %p", tm );

fprintf ( stdout, "%s\n", time\_buffer );

return;

# undef TIME\_SIZE

}

Output:

COMMUNICATOR\_MPI - Master process:

C/MPI version

An MPI example program.

The number of processes is 4.

Process 0 says 'Hello, world!'

Process 1 says 'Hello, world!'

Process 2 says 'Hello, world!'

Number of processes in even communicator = 2

Sum of global ID's in even communicator = 2

Number of processes in odd communicator = 2

Sum of global ID's in odd communicator = 4

COMMUNICATOR\_MPI:

Normal end of execution.

**Program 11:**

|  |
| --- |
| #include <stdio.h> |
|  |

|  |
| --- |
| #include "mpi.h" |
|  |

|  |
| --- |
| #define N 4 /\* number of rows and columns in matrix \*/ |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| MPI\_Status status; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| double a[N][N],b[N][N],c[N][N]; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| main(int argc, char \*\*argv) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| int numtasks,taskid,numworkers,source,dest,rows,offset,i,j,k; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| struct timeval start, stop; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| MPI\_Init(&argc, &argv); |
|  |

|  |
| --- |
| MPI\_Comm\_rank(MPI\_COMM\_WORLD, &taskid); |
|  |

|  |
| --- |
| MPI\_Comm\_size(MPI\_COMM\_WORLD, &numtasks); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| numworkers = numtasks-1; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| /\*---------------------------- master ----------------------------\*/ |
|  |

|  |
| --- |
| if (taskid == 0) { |
|  |

|  |
| --- |
| for (i=0; i<N; i++) { |
|  |

|  |
| --- |
| for (j=0; j<N; j++) { |
|  |

|  |
| --- |
| a[i][j]= 1.0; |
|  |

|  |
| --- |
| b[i][j]= 2.0; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| gettimeofday(&start, 0); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| /\* send matrix data to the worker tasks \*/ |
|  |

|  |
| --- |
| rows = N/numworkers; |
|  |

|  |
| --- |
| offset = 0; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| for (dest=1; dest<=numworkers; dest++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| MPI\_Send(&offset, 1, MPI\_INT, dest, 1, MPI\_COMM\_WORLD); |
|  |

|  |
| --- |
| MPI\_Send(&rows, 1, MPI\_INT, dest, 1, MPI\_COMM\_WORLD); |
|  |

|  |
| --- |
| MPI\_Send(&a[offset][0], rows\*N, MPI\_DOUBLE,dest,1, MPI\_COMM\_WORLD); |
|  |

|  |
| --- |
| MPI\_Send(&b, N\*N, MPI\_DOUBLE, dest, 1, MPI\_COMM\_WORLD); |
|  |

|  |
| --- |
| offset = offset + rows; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| /\* wait for results from all worker tasks \*/ |
|  |

|  |
| --- |
| for (i=1; i<=numworkers; i++) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| source = i; |
|  |

|  |
| --- |
| MPI\_Recv(&offset, 1, MPI\_INT, source, 2, MPI\_COMM\_WORLD, &status); |
|  |

|  |
| --- |
| MPI\_Recv(&rows, 1, MPI\_INT, source, 2, MPI\_COMM\_WORLD, &status); |
|  |

|  |
| --- |
| MPI\_Recv(&c[offset][0], rows\*N, MPI\_DOUBLE, source, 2, MPI\_COMM\_WORLD, &status); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| gettimeofday(&stop, 0); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| printf("Here is the result matrix:\n"); |
|  |

|  |
| --- |
| for (i=0; i<N; i++) { |
|  |

|  |
| --- |
| for (j=0; j<N; j++) |
|  |

|  |
| --- |
| printf("%6.2f ", c[i][j]); |
|  |

|  |
| --- |
| printf ("\n"); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| fprintf(stdout,"Time = %.6f\n\n", |
|  |

|  |
| --- |
| (stop.tv\_sec+stop.tv\_usec\*1e-6)-(start.tv\_sec+start.tv\_usec\*1e-6)); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| /\*---------------------------- worker----------------------------\*/ |
|  |

|  |
| --- |
| if (taskid > 0) { |
|  |

|  |
| --- |
| source = 0; |
|  |

|  |
| --- |
| MPI\_Recv(&offset, 1, MPI\_INT, source, 1, MPI\_COMM\_WORLD, &status); |
|  |

|  |
| --- |
| MPI\_Recv(&rows, 1, MPI\_INT, source, 1, MPI\_COMM\_WORLD, &status); |
|  |

|  |
| --- |
| MPI\_Recv(&a, rows\*N, MPI\_DOUBLE, source, 1, MPI\_COMM\_WORLD, &status); |
|  |

|  |
| --- |
| MPI\_Recv(&b, N\*N, MPI\_DOUBLE, source, 1, MPI\_COMM\_WORLD, &status); |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| /\* Matrix multiplication \*/ |
|  |

|  |
| --- |
| for (k=0; k<N; k++) |
|  |

|  |
| --- |
| for (i=0; i<rows; i++) { |
|  |

|  |
| --- |
| c[i][k] = 0.0; |
|  |

|  |
| --- |
| for (j=0; j<N; j++) |
|  |

|  |
| --- |
| c[i][k] = c[i][k] + a[i][j] \* b[j][k]; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| MPI\_Send(&offset, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD); |
|  |

|  |
| --- |
| MPI\_Send(&rows, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD); |
|  |

|  |
| --- |
| MPI\_Send(&c, rows\*N, MPI\_DOUBLE, 0, 2, MPI\_COMM\_WORLD); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| MPI\_Finalize(); |
|  |

}

**Program 12:**

# include <stdlib.h>

# include <stdio.h>

# include <time.h>

# include "mpi.h"

int main ( int argc, char \*argv[] );

int search ( int a, int b, int c, int id, int p );

int f ( int i );

void timestamp ( );

int main ( int argc, char \*argv[] )

{

int a;

int b;

int c;

int i4\_huge = 2147483647;

int id;

int j;

int p;

double wtime;

/\*

Initialize MPI.

\*/

MPI\_Init ( &argc, &argv );

/\*

Get this processor's ID.

\*/

MPI\_Comm\_rank ( MPI\_COMM\_WORLD, &id );

/\*

Get the number of processes.

\*/

MPI\_Comm\_size ( MPI\_COMM\_WORLD, &p );

a = 1;

b = i4\_huge;

c = 45;

if ( id == 0 )

{

timestamp ( );

printf ( "\n" );

printf ( "SEARCH\_MPI:\n" );

printf ( " C/MPI version\n" );

printf ( " Search the integers from A to B\n" );

printf ( " for a value J such that F(J) = C.\n" );

printf ( "\n" );

printf ( " A = %d\n", a );

printf ( " B = %d\n", b );

printf ( " C = %d\n", c );

}

wtime = MPI\_Wtime ( );

j = search ( a, b, c, id, p );

wtime = MPI\_Wtime ( ) - wtime;

if ( j != -1 )

{

printf ( "\n" );

printf ( " Process %d found J = %d\n", id, j );

printf ( " Verify F(J) = %d\n", f ( j ) );

}

if ( id == 0 )

{

printf ( " Elapsed wallclock time is %g\n", wtime );

}

/\*

Terminate MPI.

\*/

MPI\_Finalize ( );

/\*

Terminate.

\*/

if ( id == 0 )

{

printf ( "\n" );

printf ( "SEARCH\_MPI:\n" );

printf ( " Normal end of execution.\n" );

printf ( "\n" );

timestamp ( );

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int search ( int a, int b, int c, int id, int p )

{

int fi;

int i;

int j;

j = -1;

/\*

i = i + p can take us "over top" so that i becomes negative!

So we have to be more careful here!

\*/

for ( i = a + id; 0 < i && i <= b; i = i + p )

{

fi = f ( i );

if ( fi == c )

{

j = i;

break;

}

}

return j;

}

int f ( int i )

{

int i4\_huge = 2147483647;

int j;

int k;

int value;

value = i;

for ( j = 1; j <= 5; j++ )

{

k = value / 127773;

value = 16807 \* ( value - k \* 127773 ) - k \* 2836;

if ( value <= 0 )

{

value = value + i4\_huge;

}

}

return value;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void timestamp ( )

{

# define TIME\_SIZE 40

static char time\_buffer[TIME\_SIZE];

const struct tm \*tm;

time\_t now;

now = time ( NULL );

tm = localtime ( &now );

strftime ( time\_buffer, TIME\_SIZE, "%d %B %Y %I:%M:%S %p", tm );

printf ( "%s\n", time\_buffer );

return;

# undef TIME\_SIZE

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double cpu\_time ( void )

{

double value;

value = ( double ) clock ( )

/ ( double ) CLOCKS\_PER\_SEC;

return value;

}

Output:

SEARCH\_MPI:

C/MPI version

Search the integers from A to B

for a value J such that F(J) = C.

A = 1

B = 2147483647

C = 45

Process 4 found J = 1674924981

Verify F(J) = 45

Elapsed wallclock time is 12.1726