- MPI has facilities for both blocking and non-blocking sending and receiving of messages.
- In blocking send and receive, a *send* or a *receive* does not return until it is complete at the other end. This is good since extra synchronization is not required. However, deadlock may result in incorrect code.
- In non-blocking send and recieve, the sending process may start its computation immediately after sending a message, there is no need to wait for its 'correct' completion. Similarly, a receiving process need not block due to waiting for a message.
- However, extra synchronization is required for non-blocking send and receive.

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

```
example:

#define TAG_PI 100

double pi = 3.1415926535;

MPI_Send(&pi, 1, MPI_DOUBLE, 0, TAG_PI, MPI_COMM_WORLD);
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Status *status);

example:

double num;
MPI_Status status;

MPI_Recv(&num, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
```

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request);


int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
           int source, int tag, MPI_Comm comm,
              MPI_Request *request);

'request' identifies a process within the overall process group.
```

Non blocking send/receive completion (synchronization):

```
int MPI_Wait(MPI_Request *request, MPI_STatus *status);

int MPI_Waitall(int count, MPI_Request *array_of_requests,
                MPI_Status *array_of_statuses);
```

- An example of blocking send and receive. The purpose of this example is to pair MPI processes.
- Each process will select a partner and will exchange messages with that partner.

```
#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 status;
MPI_Ini
MPI_Com
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Get_processor_name(hostn
printf ("Hello from task %d on %s!\n", tas
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;
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();

}
```

- The same example, but this time with non-blocking send and receive.

```
#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_Ini
MPI_Com
MPI_Comm_rank(MPI_COMM_WORLD
MPI_Get_processor_name(hostn
```

```
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();

}
```

- The purpose of this example is to illustrate data distribution among processes.
- The master process divides an array and distributes it among the slaves. The master and slaves do different computations on the array elements and finally the master gets back the results.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  ARRAYSIZE 16000000
#define  MASTER 0
float  data[ARRAYSIZE];
int main (i
{
int    numtask
       tag2, source, chunksize;
float mysum, sum;
float update(int myoffset, int chunk
MPI_Status status;
/***** Initializations *****/
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks % 4 != 0) {
   printf("Quitting. Number of MPI tasks must be divisible
                               by 4.\n");
   MPI_Abort(MPI_COMM_WORLD, rc);
   exit(0);
   }
```

```
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("MPI task %d has started...\n", taskid);
chunksize = (ARRAYSIZE / numtasks);
tag2 = 1;
tag1 = 2;

/***** Master task only ******/
if (taskid == MASTER){

  /* Initialize the array */
  sum = 0;
  for(i=0; i<ARRAYSIZE; i++) {
    data[i] = i * 1.0;
    sum = sum + data[i];
    }
  print
```

```
/* Send each task its portion of the array -
                  master keeps 1st part */
  offset = chunksize;
  for (dest=1; dest<numtasks; dest++) {
    MPI_Send(&offset, 1, MPI_INT, dest,
                        tag1, MPI_COMM_WORLD);
    MPI_Send(&data[offset], chunksize, MPI_FLOAT,
                    dest, tag2, MPI_COMM_WORLD);
    printf("Sent %d elements to task %d offset= %d\n",
                         chunksize,dest,offset);
    offset = offset + chunksize;
    }
```

```
  /* Master does its part of the work */
  offse
  mysum = u
```

```
/* Wait to receive results from each task */
  for (i=1; i<numtasks; i++) {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, tag1,
              MPI_COMM_WORLD, &status);
    MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2,
      MPI_COMM_WORLD, &status);
    }

  /* Get final sum and print sample results */
  MPI_Reduce(&mysum, &sum, 1, MPI_FLOAT, MPI_SUM,
                     MASTER, MPI_COMM_WORLD);
  printf("Sample results: \n");
  offset = 0;
  for (i=
    for (
      printf("  %e",data[offset+
    printf("\n");
    offset = offset + chunksize;
    }
  printf("*** Final sum= %e ***\n",sum);

  }  /* end of master section */
```

```
/***** Non-master tasks only *****/

if (taskid > MASTER) {

  /* Receive my portion of array from the master task */
  source = MASTER;
  MPI_Recv(&offset, 1, MPI_INT, source, tag1,
                         MPI_COMM_WORLD, &status);
  MPI_Recv(&data[offset], chunksize,
                          MPI_FLOAT, source, tag2,
                         MPI_COMM_WORLD, &status);

  mysum = update(offset, chunksize, taskid);

  /* Send m
  dest = MA
  MPI_Send(&offset, 1, MPI_INT, de
                        MPI_
  MPI_Send(&data[offset], chun
                     MASTER, tag2, MPI_COMM_WORLD);

  MPI_Reduce(&mysum, &sum, 1, MPI_FLOAT, MPI_SUM,
                           MASTER, MPI_COMM_WORLD);

  } /* end of non-master */
MPI_Finalize();
}   /* end of main */
```

```
float update(int myoffset, int chunk, int myid) {
  int i;
  float mysum;
  /* Perform addition to each of my array elements
             and keep my sum */
  mysum = 0;
  for(i=myoffset; i < myoffset + chunk; i++) {
    data[i] = data[i] + i * 1.0;
    mysum = mysum + data[i];
    }
  printf("Task %d mysum = %e\n",myid,mysum);
  return(mysum);
  }
```

- Matrix multiplication : A.B = C; each row of A is multiplied by a column of B element by element and summed (dot product).

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

#define NRA 62                        /* number of rows in matrix A */
#define NCA 15                        /* number of columns in matrix A */
#define NCB 7                         /* number of columns in matrix B */
#define MASTER 0                      /* taskid of first task */
#define FROM_MASTER 1                 /* setting a message type */
#define FROM_WORKER 2                 /* setting a message type */

int main (i
{
int numta
taskid,     /* a task identifier */
numworkers, /* number of worker tasks */
source,     /* task id of message source */
dest,      /* task id of message destination */
mtype,    /* message type */
rows,     /* rows of matrix A sent to each worker */
averow, extra, offset, /* used to determine rows sent
                    to each worker */
i, j, k, rc;           /* misc */
double a[NRA][NCA],           /* matrix A to be multiplied */
b[NCA][NCB],           /* matrix B to be multiplied */
c[NRA][NCB];           /* result matrix C */
MPI_Status status;
```

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
if (numtasks < 2 ) {
  printf("Need at least two MPI tasks. Quitting...\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  exit(1);
  }
numworkers = numtasks-1;
```

```
/*************************** master task ************************
   if (taskid == MASTER)
   {
      printf("mpi_mm has started with %d tasks.\n",numtasks);
      printf("Initializing arrays...\n");
      for (i=0; i<NRA; i++)
         for (j=0; j<NCA; j++)
            a[i][j]= i+j;
      for (i=0; i<NCA; i++)
         for (j=0; j<NCB; j++)
            b[i][j]= i*j;

      /* Send matrix data to the worker tasks */
      averow = NRA/numworkers;
      extra = NRA%numworkers;
      o
      m
```

```
for (dest=1; dest<=numworkers; dest++)
      {
         rows = (dest <= extra) ? averow+1 : averow;
         printf("Sending %d rows to task %d offset=%d\n",
                                   rows,dest,offset);
         MPI_Send(&offset, 1, MPI_INT, dest,
                               mtype, MPI_COMM_WORLD);
         MPI_Send(&rows, 1, MPI_INT, dest,
                            mtype, MPI_COMM_WORLD);
         MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE,
                        dest, mtype, MPI_COMM_WORLD);
         MPI_Send(&b, NCA*NCB, MPI_DOUBLE,
                          dest, mtype, MPI_COMM_WORLD);
         offset = offset + rows;
      }

      /
      mtype = FROM_WORKER;
      for (i=1; i<=numworkers; i++)
      {
         source = i;
         MPI_Recv(&offset, 1, MPI_INT, source, mtype,
                            MPI_COMM_WORLD, &status);
         MPI_Recv(&rows, 1, MPI_INT, source, mtype,
                           MPI_COMM_WORLD, &status);
         MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE,
                     source, mtype, MPI_COMM_WORLD, &status);
         printf("Received results from task %d\n",source);
      }
```

```
/* Print results */
      printf("**********************************************
                ********\n");
      printf("Result Matrix:\n");
      for (i=0; i<NRA; i++)
      {
         printf("\n");
         for (j=0; j<NCB; j++)
            printf("%6.2f   ", c[i][j]);
      }
      printf("\n*************************************
                          ***************\n");
      printf ("Done.\n");
   }
```

```
/**************************** worker task *************
                   ***********************/
   if (taskid > MASTER)
   {
      mtype = FROM_MASTER;
      MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype,
                                  MPI_COMM_WORLD, &status);
      MPI_Recv(&rows, 1, MPI_INT, MASTER,
                                 mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER,
                                 mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype,
                                    MPI_COMM_WORLD, &status);
```

```
      for (k=0; k<NCB; k++)
```

```
            c[i][k] = 0.0;
            for (j=0; j<NCA; j++)
               c[i][k] = c[i][k] + a
```

```
         }
      mtype = FROM_WORKER;
      MPI_Send(&offset, 1, MPI_INT, MASTER, mtype,
                                  MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, MASTER, mtype,
                                  MPI_COMM_WORLD);
      MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER,
                                  mtype, MPI_COMM_WORLD);
   }
   MPI_Finalize();
```

```
}
```

- Monte Carlo estimation of Pi using the dartboard computation.
- Consider a 1×1 square. Its area is 1. Now consider a circle inscribed in this square. Its radius is $\frac{1}{2}$. Hence the area of the circle is $\pi \frac{1}{2}^2$.
- Generate points with both coordinates in the interval $[-1, 1]$. These are the darts. Count how many of those fall inside the circle (a simple comparison).
- Suppose you throw k darts. You will get a good estimate of π by counting the number of darts inside the circle, dividing that by k and multiplying by 4.
- The more darts you throw, the better your estimate would be.

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

void srandom (unsigned seed);
double dboard (int darts);
#define DARTS 50000     /* number of throws at dartboard */
#define ROUNDS 100      /* number of times "darts" is iterated */
#define MASTER 0        /* task ID of master task */

int main (int argc, char *argv[])
{
double homepi,  /* value of pi calculated by current task */
pi,     /* average of pi after "darts" is thrown */
avepi,  /* average pi value for all iterations */
pirecv, /
pisum;  /* su
int taskid, /* task ID - also used as seed numbe
numtasks, /* number of tasks */
source,   /* source of incoming message */
mtype,    /* message type */
rc,       /* return code */
i, n;
MPI_Status status;
```

```
* Obtain number of tasks and task ID */
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("MPI task %d has started...\n", taskid);

/* Set seed for random number generator equal to task ID */
srandom (taskid);

avepi = 0;
for (i = 0; i < ROUNDS; i++){
   /* All tasks calculate pi using dartboard algorithm */
   homepi = dboard(DARTS);
```

```
/* Workers send homepi to master */
   /* - Message type will be set to the iteration count */
   if (taskid != MASTER) {
      mtype = i;
      rc = MPI_Send(&homepi, 1, MPI_DOUBLE,
                    MASTER, mtype, MPI_COMM_WORLD);
      if (rc != MPI_SUCCESS)
       printf("%d: Send failure on round %d\n", taskid, mtype);
      }
   else
      {
      /* Master receives messages from all workers */
      /*Message type will be set to the iteration count */
      /*Message source will be set to the wildcard DONTCARE: */
      /*a message can be received from any task, as long as the */
      /
      / [illegible]                                    layed *
      /*if a problem occurred */
```

```
mtype = i;
      pisum = 0;
      for (n = 1; n < numtasks; n++) {
         rc = MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                        mtype, MPI_COMM_WORLD, &status);
     if (rc != MPI_SUCCESS)
       printf("%d: Receive failure on round %d\n", taskid, mtype);
         /* keep running total of pi */
         pisum = pisum + pirecv;
         }
     /* Master calculates the average value of pi
                  for this iteration */
      pi = (pisum + homepi)/numtasks;
      /* Master calculates the average value of
                      pi over all iterations */
      a
      p
            pi = %10.8f\n", (DARTS * (
      }
   }
```

```
if (taskid == MASTER)
   printf ("\nReal value of PI: 3.1415926535897 \n");

MPI_Finalize();
return 0;
}

/****************************************************************
* subroutine dboard
* DESCRIPTION:
*   Used in pi calculation example codes.
*   See mpi_pi_send.c and mpi_pi_reduce.c
*   Throw darts at board.  Done by generating random numbers
*   between 0 and 1 and converting them to values for x and y
*   coordinates and then testing to see if they "land" in
*   the circle.   I
*   specified n
*   of pi is returned as the value of this function, d
*
*   Explanation of constants and variables u
*   darts        = number of throws at dartboard
*   score        = number of darts that hit circle
*   n            = index variable
*   r            = random number scaled between 0 and 1
*   x_coord      = x coordinate, between -1 and 1
*   x_sqr        = square of x coordinate
*   y_coord      = y coordinate, between -1 and 1
*   y_sqr        = square of y coordinate
*   pi           = computed value of pi
****************************************************************
```

```
double dboard(int darts)
{
#define sqr(x) ((x)*(x))
long random(void);
double x_coord, y_coord, pi, r;
int score, n;
unsigned int cconst;  /* must be 4-bytes in size */
/*****************************************************************
 * The cconst variable must be 4 bytes. We check this and bail if
 * not the right size
 *****************************************************************
if (sizeof(cconst) != 4) {
   printf("Wrong data size for cconst variable in dboard routine!\
   printf("See comments in source file. Quitting.\n");
   exit(1);
   }
   /* 2 bit s
      random number between 0 and 1 */
   cconst = 2 << (31 - 1);
   score = 0;
```

```
/* "throw darts at board" */
   for (n = 1; n <= darts; n++)  {
      /* generate random numbers for x and y coordinates */
      r = (double)random()/cconst;
      x_coord = (2.0 * r) - 1.0;
      r = (double)random()/cconst;
      y_coord = (2.0 * r) - 1.0;

      /* if dart lands in circle, increment score */
      if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)
           score++;
      }

/* calculate pi */
pi = 4.0 * (double)score/(double)darts;
return(
}
```

- Generating prime numbers and counting them.
- Except the number 2, all primes are odd numbers. Hence, there are two possibilities. Each process can be allocated a block of odd integers, or each process can be allocated a *stride*, that is a sequence of odd integers.
- The second method is used here. The example also illustrates a simple way of timing MPI programs.

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

```
#define LIMIT     2500000     /* Increase this to find more primes
#define F         0           /* Rank of first ta
```

```
int isprime(int n) {
int i,squareroot;
if (n>10) {
   squareroot = (int) sqrt(n);
   for (i=3; i<=squareroot; i=i+2)
      if ((n%i)==0)
         return 0;
   return 1;
   }
/* Assume first four primes are counted elsewhere. Forget everythi
else
   return 0;
}
```

```
int main (int argc, char *argv[])
{
int   ntasks,               /* total number of tasks in partitiion
      rank,                 /* task identifier */
      n,                    /* loop variable */
      pc,                   /* prime counter */
      pcsum,                /* number of primes found by all tasks
      foundone,             /* most recent prime found */
      maxprime,             /* largest prime found */
      mystart,              /* where to start calculating */
      stride;               /* calculate every nth number */

double start_time,end_time;
```

```
MPI_Init(&argc,&argv);
MPI_Com
MPI_Com
```

```
if (((ntasks%2) !=0) || ((LIMIT%ntas
   printf("Sorry - this exercise req                          sks.
   printf("evenly divisible into %
```

```
   MPI_Finalize();
   exit(0);
   }

start_time = MPI_Wtime();   /* Initialize start time */
mystart = (rank*2)+1;       /* Find my starting point - must be od
stride = ntasks*2;          /* Determine stride, skipping even num
pc=0;                       /* Initialize prime counter */
foundone = 0;               /* Initialize */
```

```
/******************** task with rank 0 does this part ***********
if (rank == FIRST) {
   printf("Using %d tasks to scan %d numbers\n",ntasks,LIMIT);
   pc = 4;                    /* Assume first four primes are counte
   for (n=mystart; n<=LIMIT; n=n+stride) {
      if (isprime(n)) {
         pc++;
         foundone = n;
         /***** Optional: print each prime as it is found
         printf("%d\n",foundone);
         *****/
         }
      }
   MPI_Reduce(&pc,&pcsum,1,MPI_INT,MPI_SUM,

   MPI_                                    NT,
           MPI_MAX,FIRST,MPI
   end_time=MPI_Wtime();
   printf("Done. Largest prime is %d
                Total primes %d\n",maxprime,pcsum);
   printf("Wallclock time elapsed: %.2lf seconds\n",
                  end_time-start_time);
   }
/******************** all other tasks do this part ***************
if (rank > FIRST) {
   for (n=mystart; n<=LIMIT; n=n+stride) {
      if (isprime(n)) {
         pc++;
         foundone = n;
         /***** Optional: print each prime as it is found
         printf("%d\n",foundone);
         *****/
         }
      }
   MPI_Reduce(&pc,&pcsum,1,MPI_INT,MPI_SUM,FIRST,
              MPI_COMM_WORLD);
   MPI_Reduce(&foundone,&maxprime,1,MPI_INT,
              MPI_MAX,FIRST,MPI_COMM_WORLD);
   }

MPI_Finalize();
}
```