1 Performance tuning MPI 1

1. Determining delivered memory performance; the following program

   - uses *MPI_Wtime* to benchmark the performance of the system **memcpy** routine on your system.
   - generates a table for 1, 2, 4, 8, ..., 524288 integers showing the number of bytes, time to send, and the rate in Megabytes per second.
   - uses unaligned data items; that is, make sure that the low-order bits of the source and destination addresses are different. Also, ensure that the source and destination are "well separated" in memory.
   - performs enough memcpy operations to take a good fraction of a second; the sample solution does 100000/size iterations for size integers. It also repeats the test 10 times and reports the best time.

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

int main( argc, argv )
int argc;
char **argv;
{
    double t1, t2;
    double tmin;
    int    size, *in_data, *out_data;
    int    j, nloop, k;

    MPI_Init( &argc, &argv );

    printf( "Size (bytes) Time (sec)\tRate (MB/sec)\n" );
    for (size = 1; size < 1000000; size *= 2 ) {
        in_data = (int *)malloc( size * sizeof(int) );
        out_data = (int *)malloc( size * sizeof(int) );
        if (!in_data || !out_data) {
            fprintf( stderr, "Failed to allocate space for %d
    ints\n", size );
            break;
        }
```

```
        tmin = 1000.0;
        nloop = 100000/size;
        if (nloop == 0) nloop = 1;
        for (k=0; k < 10; k++) {
            t1 = MPI_Wtime();
            for (j=0; j<nloop; j++)
             memcpy( out_data, in_data, size * sizeof(int) );
            t2 = (MPI_Wtime() - t1) / nloop;

            if (t2 < tmin) tmin = t2;
        }
        printf( "%d\t%f\t%f\n", size * sizeof(int), tmin,
                1.0e-6*size*sizeof(int)/tmin );
        free( in_data );
        free( out_data );
    }

    MPI_Finalize( );
    return 0;
}
```

Execute as

```
mpicc -o memcpy memcpy.c
mpirun -np 1 memcpy
```

2. Determining delivered memory performance with unaligned data; the following program

   - uses *MPI_Wtime* to benchmark the performance of the system **memcpyunal** routine on your system.
   - generates a table for 1, 2, 4, 8, ..., 524288 integers showing the number of bytes, time to send, and the rate in Megabytes per second.
   - performs enough *memcpy* operations to take a good fraction of a second; the sample solution does 100000/size iterations for size integers. It also repeats the test 10 times and reports the best time.

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

```
#include "mpi.h"

#define PAD 4096

int main( argc, argv )
int argc;
char **argv;
{
    double t1, t2;
    double tmin;
    int    size;
    char   *in_data, *out_data;
    char   *in_p, *out_p;
    int    j, nloop, k;

    MPI_Init( &argc, &argv );

    printf( "Size (bytes) Time (sec)\tRate (MB/sec)\n" );
    for (size = 1; size < 1000000; size *= 2 ) {
        in_data = in_p = (char *)malloc( size * sizeof(int) + PAD );
        out_data = out_p = (char *)malloc( size * sizeof(int) + PAD );
        if (!in_data || !out_data) {
            fprintf( stderr, "Failed to allocate space for %d
    ints\n", size );
            break;
        }

        /* make out_p, in_p unaligned */
        out_p += 7;
        in_p  += 10;
        if ((((long)out_p) & 0x3) == (((long)in_p) & 0x3)) {
            out_p += 3;
        }
        tmin = 1000.0;
        nloop = 100000/size;
        if (nloop == 0) nloop = 1;
        for (k=0; k < 10; k++) {
            t1 = MPI_Wtime();
            for (j=0; j<nloop; j++)
                memcpy( out_p, in_p, size * sizeof(int) );
            t2 = (MPI_Wtime() - t1) / nloop;

            if (t2 < tmin) tmin = t2;
```

```
        }
        printf( "%d\t%f\t%f\n", size * sizeof(int), tmin,
                1.0e-6*size*sizeof(int)/tmin );
        free( in_data );
        free( out_data );
    }

    MPI_Finalize( );
    return 0;
}
```

Execute as

```
mpicc -o memcpyunal memcpyunal.c
mpirun -np 1 memcpyunal
```

3. Benchmarking point to point performance; the following program
   - measures the time it takes to send 1, 2, 4, ..., 1M C doubles from one processor to another using *MPI_Send* and *MPI_Recv*. Use the same techniques as in the **memcpy** assignment to average out variations and overhead in *MPI_Wtime*.
   - rints the size, time, and rate in MB/sec for each test.
   - makes sure that both sender and reciever are ready when you begin the test (*MPI_Sendrecv*).

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

#define NUMBER_OF_TESTS 10

int main( argc, argv )
int argc;
char **argv;
{
    double       *buf;
    int          rank;
    int          n;
    double       t1, t2, tmin;
    int          i, j, k, nloop;
    MPI_Status   status;
```

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

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if (rank == 0)
        printf( "Kind\t\tn\ttime (sec)\tRate (MB/sec)\n" );

    for (n=1; n<1100000; n*=2) {
        if (n == 0) nloop = 1000;
        else        nloop  = 1000/n;
        if (nloop < 1) nloop = 1;

        buf = (double *) malloc( n * sizeof(double) );
        if (!buf) {
            fprintf( stderr,
      "Could not allocate send/recv buffer of size %d\n", n );
            MPI_Abort( MPI_COMM_WORLD, 1 );
        }
        tmin = 1000;
        for (k=0; k<NUMBER_OF_TESTS; k++) {
            if (rank == 0) {
                /* Make sure both processes are ready */
MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD,
                              &status );
                t1 = MPI_Wtime();
                for (j=0; j<nloop; j++) {
MPI_Send( buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD );
MPI_Recv( buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD,
                              &status );
                }
                t2 = (MPI_Wtime() - t1) / nloop;
                if (t2 < tmin) tmin = t2;
            }
            else if (rank == 1) {
                /* Make sure both processes are ready */
MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD,
                              &status );
                for (j=0; j<nloop; j++) {
MPI_Recv( buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD,
                              &status );
MPI_Send( buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD );
```

```
                }
            }
        }
        /* Convert to half the round-trip time */
        tmin = tmin / 2.0;
        if (rank == 0) {
            double rate;
            if (tmin > 0) rate = n * sizeof(double) * 1.0e-6 /tmin;
            else          rate = 0.0;
            printf( "Send/Recv\t%d\t%f\t%f\n", n, tmin, rate );
        }
        free( buf );
    }

    MPI_Finalize( );
    return 0;
}
```

Execute as

```
mpicc -o pingpong -O pingpong.c
mpirun -np 2 pingpong
```

4. Benchmarking point to point performance with MPI_Ssend; modify the program in the previous item to use *MPI_Ssend* instead of *MPI_Send*. Analyse and compare the output. What does *MPI_Ssend* function?

5. Using synchronous send; the following program

   - consists of one sender process and one receiver.
   - The sender process sends a message containing its identifier to the receiver. This receives the message and sends it back.
   - Both processes use synchronous send operations (*MPI_Ssend*).

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[]) {
  int x, y, np, me;
  int tag = 42;
  MPI_Status  status;
  MPI_Init(&argc, &argv);               /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &np);   /* Get number of processes */
```

```
  MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* Get own identifier */
  x = me;
  if (me == 0) {    /* Process 0 does this */
    printf("Sending to process 1\n");
MPI_Ssend(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);  /* Synchronous send */
    printf("Receiving from process 1\n");
    MPI_Recv (&y, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
printf("Process %d received a message containing value %d\n", me, y);
  } else {         /* Process 1 does this */
/* Since we use synchronous send, we have to do the receive-operation */
/* first, otherwise we will get a deadlock */
    MPI_Recv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
MPI_Ssend (&x, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);  /* Synchronous send */
  }
  MPI_Finalize();
}
```

Execute as

```
mpicc -o send-recv3 send-recv3.c
mpirun -machinefile hostfile -np 2 send-recv3
```

6. Write a program to add n numbers

    - a sequential code; necessary code segment for time analysis

      ```
      #include <sys/resource.h>
      long int who;
      struct rusage ru;
      double tsec;
      who=0;
      getrusage(who,&ru);
      tsec=(ru.ru_utime.tv_sec + 1.e-6*ru.ru_utime.tv_usec);
      tsec+=(ru.ru_stime.tv_sec + 1.e-6*ru.ru_stime.tv_usec);
      ```

      or find a better one!

    - a parallel code with *send* and *recieve*;
    - a parallel code by *broadcasting*;
    - make a time analysis with *MPI_Wtime* while increasing n.