1 Performance tuning MPI 2

1. Non-blocking send and receive; the following program

   - consists of one sender process and up to 7 receiver processes.
   - The sender process sends a message containing its identifier to all the other processes. These receive the message and replies with a message containing their own identifier.
   - Both processes use non-blocking send and receive operations (MPI_Isend and MPI_Irecv, and MPI_Waitall).

```
#include <stdio.h>
#include "mpi.h"
#define MAXPROC 8    /* Max number of procsses */
main(int argc, char* argv[]) {
  int i, x, np, me;
  int tag = 42;
  MPI_Status status[MAXPROC];
  /* Request objects for non-blocking send and receive */
  MPI_Request send_req[MAXPROC], recv_req[MAXPROC];
  int y[MAXPROC];  /* Array to receive values in */
  MPI_Init(&argc, &argv);                /* Initialize */
  MPI_Comm_size(MPI_COMM_WORLD, &np);    /* Get nr of processes */
  MPI_Comm_rank(MPI_COMM_WORLD, &me);    /* Get own identifier */
  x = me;   /* This is the value we send, the process id */
  if (me == 0) {    /* Process 0 does this */
/* First check that we have at least 2 and at most MAXPROC processes */
    if (np<2 || np>MAXPROC) {
printf("You have to use at lest 2 and at most %d processes\n", MAXPROC);
      MPI_Finalize();
      exit(0);
    }
printf("Process %d sending to all other processes\n",me);
/* Send a message containing the process id to all other processes */
    for (i=1; i<np; i++) {
      MPI_Isend(&x, 1, MPI_INT, i, tag, MPI_COMM_WORLD, &send_req[i]);
    }
/* While the messages are delivered, we could do computations here */
/* Wait until all messages have been sent */
/* Note that we use requests and statuses starting from position 1 */
    MPI_Waitall(np-1, &send_req[1], &status[1]);
    printf("Process %d receiving from all other processes\n", me);
```

```
/* Receive a message from all other processes */
    for (i=1; i<np; i++) {
MPI_Irecv (&y[i], 1, MPI_INT, i, tag, MPI_COMM_WORLD, &recv_req[i]);
    }
/* While the messages are delivered, we could do computations here */
/* Wait until all messages have been received */
/* Requests and statuses start from position 1 */
    MPI_Waitall(np-1, &recv_req[1], &status[1]);
/* Print out one line for each message we received */
    for (i=1; i<np; i++) {
printf("Process %d received message from process %d\n", me, y[i]);
    }
    printf("Process %d ready\n", me);
  } else { /* all other processes do this */
    /* Check sanity of the user */
    if (np<2 || np>MAXPROC) {
      MPI_Finalize();
      exit(0);
    }
    MPI_Irecv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &recv_req[0]);
    MPI_Wait(&recv_req[0], &status[0]);

    MPI_Isend (&x, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &send_req[0]);
    /* Lots of computations here */
    MPI_Wait(&send_req[0], &status[0]);
  }
  MPI_Finalize();
  exit(0);
}
```

Execute as

```
mpicc send-recv6.c -o send-recv6
mpirun -machinefile hostfile -np 4 send-recv6
```

- Use *MPI_Wtime* to benchmark the performance.
- Rewrite the code such taht both processes use blocking send and receive operations (MPI_send and MPI_recv). Use *MPI_Wtime* to benchmark the performance.

2. 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.