README for exercise set 11 in IN3200/IN4200

Exercise 1) Ping Pong

The test described in the exercise is often called the ping pong test, and is one way to estimate the overhead of point to point communication.

The actual back and forth message passing is not difficult (see the provided source code), but to use this to get a good estimate of the overhead is worse. In this implementation we track the time used over several send and receives for different amounts of data in an interval. We can then plot the time used as a function of message size and do a linear regression to find the intercept of that curve. This intercept is our estimate of the latency/overhead of the communication.

Gnuplot

After the data file is generated you can do the linear regression any way you want. Here I'm going to show some basics in gnuplot. Gnuplot is available for Linux, mac and windows. but there are differences in how to install and run the program. Here I'll focus on Ubuntu.

gnuplot can be installed with:

```
sudo apt-get install gnuplot
```

With gnuplot installed the cmd `gnuplot` opens the plotting console. To plot and fit a linear model we can use the following commands:

```
gnuplot> plot "w11_e1.dat" using 1:2 with lines
gnuplot> f(x)  = m*x + b
gnuplot> fit f(x) "w11_e1.dat" using 1:2 via m,b
```

The first line opens a plot of the graph. The second and third fits a linear model ($f(x) = mx + b$)

There is a lot of output to the console, but we will only focus on the part that looks something like this:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
m               = 0.000354591      +/- 9.637e-07    (0.2718%)
b               = 0.191219         +/- 0.05524      (28.89%)
```

In the program the time is recorded in milliseconds. And the intercept term is 0.19 meaning a latency/overhead of 0.19ms.

Of course this is only valid if the underlying data is linear. If the plot shows a graph that is not linear then this is not valid for that range of message sizes.

Exercise 2) Trapezoidal

They way the loop is programmed, each iteration depends on the iterations before it due to the update of x. A simple fix is to compute the value of x using the iteration variable i.

```
for (i=1; i<n; i++) {
    x = i*h; // Compute x without any dependency.
    result += exp(5.0*x)+sin(x)-x*x;
}
```

This loop is trivially parallelizable. Each process gets it's share of the loop. In this implementation the last few ranks gets an extra iteration each. Their interval is calculated locally on each process to avoid communication.

```
rest = n%nprocs;
low_rank = nprocs - rest; // rank >= low_rank gets an extra iteration.
if (myrank >= low_rank) {
    displacement = myrank - low_rank;
} else {
    displacement = 0;
}
iters = n/nprocs + (myrank >= low_rank ? 1:0);

i_start = myrank*(n/nprocs) + displacement;
i_stop = i_start + iters;
```

The source code for this exercise tweaks this algorithm slightly to make it work with the limits in the for loop.

With the iterations divided between each process all we need to do is a reduction at the end.

```
MPI_Reduce(&local_result,
           &result,
           1,
           MPI_DOUBLE,
           MPI_SUM,
           0,
           MPI_COMM_WORLD)
```

Exercise 3) Wave Equation

The key thing to note in this exercise is the use of "ghost" points for each process' `u[0]` and `u[M+1]`, the values of which are retrieved from its neighbouring processes. Below is one solution.

```c
#include <malloc.h>
#include <math.h>
#include <mpi.h>
#include <stdlib.h>

//#define MPI_Send(a, b, c, d, e, f) MPI_Send(a, b, c, d, e, f); \
    if(rank < 2) printf("Rank %d sent to %d value %g\n", rank, d, *a)
//#define MPI_Recv(a, b, c, d, e, f, g) MPI_Recv(a, b, c, d, e, f, g); \
    if(rank < 2) printf("Rank %d recd from %d value %g\n", rank, d, *a)

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

    MPI_Init(&argc, &argv);
    int rank, size;
    int root = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Read resolution and time steps.
    int fullM, M, T;
    if (rank == root) {
        if (argc < 2) {
            fullM = 42;
        } else {
            fullM = atoi(argv[1]);
        }
        if (argc < 3) {
            T = 42;
        } else {
            T = atoi(argv[2]);
        }
    }
    MPI_Bcast(&fullM, 1, MPI_INT, root, MPI_COMM_WORLD);
    MPI_Bcast(&T, 1, MPI_INT, root, MPI_COMM_WORLD);

    // Find widths and displacements
    int *width = malloc(size * sizeof *width);
    int *displs = malloc(size * sizeof *displs);
    displs[0] = 0;
    for(int i = 1; i <= size; i++) {
```

```
    int curdis = i * fullM / size;
    if(i < size) // Don't go out of bounds
        displs[i] = curdis;
    width[i - 1] = curdis - displs[i - 1];
    if(rank == i-1)
        printf("Rank %d:\tWidth %d,\tOffset %d\n", rank, width[i-1], displs[i-1]);
}

// Local width
M = width[rank];

// Initial conditions
double x, dx = 1.0/(fullM+1);
double t, dt = 1. / T;
double *tmp;
int i;
double *fullum;
double *fullu;
if(rank == root) {
    fullum = malloc(sizeof *fullum * (fullM+2));
    fullu = malloc(sizeof *fullu * (fullM+2));
    for (i=0; i<fullM+2; i++) {
        x = i*dx;
        fullum[i] = x > .5 ? 1. : -1.; // sin(2.0*M_PI*x);
    }
    for (i=1; i<=fullM; i++)
        fullu[i] = fullum[i] + 0.5*(fullum[i-1]-2*fullum[i]+fullum[i+1]);
    fullu[0] = fullu[fullM+1] = 0.0;
}

/* allocating three 1D arrays um, u, up of length M+2 */
/* ... */
double *um = malloc(sizeof *um * (M+2));
double *u = malloc(sizeof *u * (M+2));
double *up = malloc(sizeof *up * (M+2));

// Distribute values. Remember offset 1.
MPI_Scatterv(fullum + 1, width, displs, MPI_DOUBLE,
        um + 1, M, MPI_DOUBLE, root, MPI_COMM_WORLD);
MPI_Scatterv(fullu + 1, width, displs, MPI_DOUBLE,
        u + 1, M, MPI_DOUBLE, root, MPI_COMM_WORLD);

// Initialise edges for nodes not passing them.
if(!rank) {
    um[0] = 0.0;
    u[0] = 0.0;
    up[0] = 0.0;
}
if(rank == size - 1) {
    um[M+1] = 0.0;
    u[M+1] = 0.0;
    up[M+1] = 0.0;
}

// Main loop

t = dt;
while (t<1.0) {
    t += dt;

    // Send/recv neighbouring values ("ghost" points).
    double lbuf, rbuf;
    MPI_Status s;

    // Odd nodes send first.
    if(rank % 2) {
        // Send left. Odd nodes can always do this.
        lbuf = u[1];
        MPI_Send(&lbuf, 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD);
        // Do not send to right if rightmost node.
        if(rank != size - 1) {
            rbuf = u[M];
            MPI_Send(&rbuf, 1, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
        }
        // Receive left.
        MPI_Recv(&lbuf, 1, MPI_DOUBLE, rank - 1, MPI_ANY_TAG,
                MPI_COMM_WORLD, &s);
        u[0] = lbuf;
        // Receive right.
        if(rank != size - 1) {
            MPI_Recv(&rbuf, 1, MPI_DOUBLE, rank + 1, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &s);
            u[M+1] = rbuf;
        }
    }

    else { // Even nodes
        // Receive right.
        if(rank != size - 1) {
            MPI_Recv(&rbuf, 1, MPI_DOUBLE, rank + 1, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &s);
            u[M+1] = rbuf;
        }
        // Receive left, unless leftmost node.
        if(rank) {
            MPI_Recv(&lbuf, 1, MPI_DOUBLE, rank - 1, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &s);
            u[0] = lbuf;
        }
        // Send right.
        if(rank != size - 1) {
            rbuf = u[M];
            MPI_Send(&rbuf, 1, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
        }
        // Send left.
        if(rank) {
            lbuf = u[1];
            MPI_Send(&lbuf, 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD);
        }
    }

    // Update.
    for (i=1; i<=M; i++)
        up[i] = (um[i]+u[i-1]+u[i+1]) / 3;

    /* shuffle the three arrays */
    tmp = um;
    um = u;
    u = up;
    up = tmp;
}

// Gather values. u has the final ones after shuffling.
MPI_Gatherv(u + 1, M, MPI_DOUBLE,
        fullu + 1, width, displs, MPI_DOUBLE,
        root, MPI_COMM_WORLD);

// Clean up.
free(um);
free(u);
free(up);
free(width);
free(displs);

// Output for plotting
if(rank == root) {
    FILE *fp = fopen("w11_e3.dat", "w");
    for (size_t i = 0; i < fullM + 2; i++) {
        fprintf(fp, "%.17g, %.17g\n", (double)i/(fullM+1),
                fullu[i]);
    }
    fclose(fp);
}

// Clean up.
if(rank == root) {
    free(fullum);
    free(fullu);
}
MPI_Finalize();
}
```