README for exercise set 10 in IN3200/IN4200

MPI

In the exercises this week (and later), we will be using MPI to create concurrency in our programs. MPI (Message Passing Interface) is built around the concept of passing messages (duh), and creating communicators with different topologies. This first week we will not be thinking to hard about the topology of our MPI communicators, but rather get to know some of the different communication options we have available.

Initiating MPI

All MPI programs need to be initialized and resolved using some special MPI invocations. This is code that should be added in every "main" source code.

```c
#include <mpi.h>

int main(int argc, char const *argv[]) {
    MPI_Init(&argc, &argv);

    // Find the number of processors.
    int num_procs;
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    // Find the rank of this instance.
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Do something in parallel.

    MPI_Finalize();
    return 0;
}
```

Strictly speaking it is not needed to find `num_procs` and `my_rank`, but they are frequently used in communication operations.

Note: `MPI_COMM_WORLD` is an object set up in the `MPI_Init` function. This object is called a communicator and is used in all communication operations. It is possible to define your own communicators, but this week we will keep it simple.

Exercise 1) Hello World

This exercise is a first step in to the world of message passing. Now that we are casting away the assumption of a shared memory space, we have to specify every communication between processes explicitly.

Setting up the serial code that creates the message is rather easy.

```c
#include <stdio.h>

int main(int argc, char const *argv[]) {

    // Find my_rank.

    char msg[256]; // Intentionally way to long.

    sprintf(msg, "Hello World from process %d", my_rank);
    int msglen = strlen(msg) + 1;  // The +1 includes the terminating char.

    // Send the message to processes 0, or receive and print if my_rank == 0.
    return 0;
}
```

For the communication we will use:

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

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

`buf` is a pointer telling the process where to start reading/writing the message. `MPI_Datatype` is `MPI_CHAR` in this case. `dest` and `source` is the rank of the destination and source process. This is used to make sure the connection is between the right processes. `tag` is a message identifier. In this program we will just set it to 0 and not worry about it. `MPI_Status` is an object used to figure out if the message was sent successfully, how many words was transmitted and so on.

In the source code I'm using two other functions, `MPI_Probe` and `MPI_Get_count`. These are used to figure out the length of the message being sent. In this simple program they are not necessary. But they might be useful in the future.

Compilation

To compile MPI programs it is simplest to use a wrapper around your C compiler. Most (all?) MPI distributions come with this wrapper, and it is usually called as `mpicc`. See the Makefile for details.

Running a MPI Program

To run a MPI program we need to use a special command that sets up all the processes for us. This command looks something like this on Linux:

```
$ mpirun -n 2 ./myprog.exe arg1 arg2 ...
```

Or this if on Windows:

```
$ mpiexec -n 2  myprog.exe arg1 arg2 ...
```

Exercise 2) Reduce

`MPI_Reduce` is made for the purpose of "reducing" multiple values down to one, and should be the tool of choice in other programs for this tasks. But to get a feel for how the other operations work we have an implementation of all the options in the source code.

Gather

A gather is a collective communication operation that involves all the processes in a MPI communicator. These collective communication operations make our life much simpler.

```
int MPI_Gather(const void *sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf,          // Matters only at root.
               int recvcount,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm)
```

In this program the sendbuf is just the address of the local sum, and the `sendcount` is 1. `recvbuf` must we an array with space to hold all of the gathered values. and `recvcount` is the number of processes. `root` is the rank of the process we want to send all the values to.

Reduce

Reduce also involves all the processes in a communicator. The difference to gather is that the values that are communicated are combined in some way to a new result at the root process. This is a very common problem in distributed computing, and recognizing that you have a reduction problem and using the right invocations can save you allot of time and effort.

```
int MPI_Reduce(const void *sendbuf,
               void *recvbuf,
               int count,
               MPI_Datatype datatype,
               MPI_Op op,
               int root,
               MPI_Comm comm)
```

Using a reduce the `recvbuf` only need to hold one value, and `count` is therefore set to 1. We can reduce arrays to, and in that case `count` should be the length of the array.

`MPI_Op op` is worth a mention. There are several reduction operations predefined in MPI. Some of the most important are: - `MPI_MAX` - `MPI_MIN` - `MPI_SUM` - `MPI_PROD`

In this exercise we will of course use `MPI_SUM`. It's possible to define your own, but seldom needed.

Notice in the source code how easy this problem is when you are using the right tool(s), `MPI_Reduce` in this case.

Exercise 3) Matrix-Vector Multiplication

In this exercise we have to implement proper communication and work sharing logic. Earlier in the course you have implemented several different work decomposition schemes, and here we are using just a basic row decomposition. What is new is the explicit communication of the values in the matrix A and the vector x.

Broadcast

All the processes have to have a complete copy of x to do their local computations, so we can use a broadcast to send x.

```
int MPI_Bcast(void *buffer,
              int count,
              MPI_Datatype datatype,
              int root,
              MPI_Comm comm)
```

Note that all processes in the MPI communicator must call `MPI_Bcast` or the program will block forever. The buffer is the space where the message is saved on all processes, except on the root process, where the memory of the buffer is read and sent instead.

Scatter

Each process gets its own part of the matrix A. This operation is called a "scatter". There are two functions in MPI that are used to scatter data. `MPI_Scatter` and `MPI_Scaterv`, notice the appended "v". `MPI_Scatter` assumes that each process gets the same amount of data. This simplifies the message passing allot, but is often to restrictive an assumption. In this exercise we can not assume that N is divisible by the number of processors, so the processes might get different number of elements of A to work with.

```
int MPI_Scatterv(const void *sendbuf,
                 const int *sendcounts,
                 const int *displs,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm)
```

To use this function we have to construct the arrays `sendcounts` and `displs`. The first is an array that contains the number of elements to send to the different processes. So if process 0 should get 5 elements, and process 1 should get 10, then the first two entries in `sendcounts` should be

```
sendcounts[0] = 5;  // Number of elements to send to process 0.
sendcounts[1] = 10; // NUmber of elements to send to process 1.
```

The index in `sendcounts` should match the processor rank. `displs` is short for displacement, and tells MPI where to start reading the buffer when sending to each process. In our example process 0 should get the first 5 elements, so `displs[0] = 0;`, we start reading from the start when sending to process 0. `displs[1] = 4;`, we start reading from the fifth element when sending to process 1.

Gather

Gather is the “inverse” of scatter, and also come in two variant. Since each process here have different number of elements of the results vector we must use the “v” variant.

```
int MPI_Gatherv(const void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                const int *recvcounts,
                const int *displs,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm)
```

This is very similar to the `MPI_Scatterv` function, but now `recvcounts` is an array telling how many elements that each process wants to send to root.

Note: In this exercise it would be easier to use something called a MPI type, specifically `MPI_Type_contiguous`. See exercise 4 for an explanation of MPI types, they will become very important later in the course.

Exercise 4) Column Decomposition

The problem of communicating subsets of arrays that are not contiguous in memory is quite common, so fortunately MPI have built in tools to deal with these cases.

MPI Vectors

We will define a new MPI vector type that represents a column in A. Then we can send entire columns without needing to think of the strides, block lengths and so on. These types are all local to the processes. Each process must define all the types it’s going to use. It’s also possible to send in one type and receive in another. To create a new usable type we must define and commit the new type.

```
int MPI_Type_vector(int count,
                    int blocklength,
                    int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype * newtype)

int MPI_Type_commit(MPI_Datatype *datatype)
```

`count` is the number of blocks. `blocklength` is the number of elements in each block. `stride` is the distance between the start of each block. `oldtype` is the type of each element in each block (typically `MPI_DOUBLE`). `newtype` is the new type we are defining.

Each column contains N blocks of 1 element, with a stride of N between them in contiguous memory. So to define a column type we can do the following:

```
MPI_Datatype col_temp;
MPI_Type_vector(N, 1, N, MPI_DOUBLE, &col_temp);
```

There is one problem however. Every type in MPI have an additional parameter called “extent”. This parameter tells MPI how closely it can stack several of these types together. For regular double precision floats, the extent is 8 bytes. If the first double starts at position 0, then the second starts at 8, third at 16 etc. For our type however, the extent is automatically set to `((count - 1)*stride + blocklenght)*sizeof(Type)`, meaning, the range from the first element in the first block to the last element in the final block. Ex: With a `count` of 4, `blocklenght` of 1, and stride of 3 we get

```
   Stride
|-----------|
¦-----------------------------------------------¦
¦ x |   |   | x |   |   | x |   |   | x | o |   ¦
¦-----------------------------------------------¦
|---------------------------------------| ^
   Extent                               First element of the next vector
```

But what we want in this case is something more like this:

```
|-----------|
    |-----------|
¦-----------------------------------------------¦
¦ x | o |   | x | o |   | x | o |   | x | o |   ¦
¦-----------------------------------------------¦
|---|
   Extent
```

To achieve this we have to use another MPI function.

```
int MPI_Type_create_resized(MPI_Datatype oldtype,
                            MPI_Aint lb,
                            MPI_Aint extent,
                            MPI_Datatype *newtype)
```

`lb` is the “lower bound” and can be set to 0, unless you have something very specific in mind. Now we create a new type from the other we defined.

```
MPI_Datatype col_vec;
MPI_Type_create_resized(col_temp, 0, blocklenght*sizeof(double), &col_vec);
```

To use a type we must commit it.

```
MPI_Type_commit(&col_vec);
```

Now we can use this new type when distributing the matrix A.

Reduction

In this version of our program all the processes compute part of all of the elements in y. To get the correct result we must therefore do a reduction instead of a gather.

The structure of the program is very similar to the one in ex. 3 but there are some details that differ. I recommend going through it. Make a special note of how we define a column receive type with a stride that matches the number of columns that the process gets in the scatter.