Examining the MPI max.c Program

This tutorial will walk through this program in hopes of achieving a basic understanding of how a program with Message Passing Interface (MPI) works. It will assume that there is a basic understanding of programming already achieved, and will mainly focus on how MPI and parallelization adds to the programming techniques.

In general, this program finds the maximum value in an array of integers. This could be done with a sequential algorithm rather easily, but a parallel algorithm would seek to make the process faster. In a parallel approach, the array will be broken into equal sections (or "chunks") that are then distributed to the different processors. Each processor can individually find the maximum of their section. When this is finished, these local maximums can be compared to find a global maximum, that is, the maximum of the entire original array.

Now, to study the actual code:

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

The source code begins as all C programs do: with header files. These two are `<stdio.h>`, standard input/output, and `"mpi.h"`, which is our Message Passing Interface library.

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

Next, the main function is called, again with standard C programming form.

```
        int numprocs, rank, chunk_size, i;
        int max, mymax,rem;
        int array[800];
        MPI_Status status;
```

Here, we have the variable declarations. Everything should look similar here, except for the `MPI_Status`. This `status` variable is used when receiving information from other processors. As for the rest of the variables, a brief description of their role in the program would be beneficial to understanding.

- `numprocs` holds the number of parallel processors being used
- `rank` is the special ID assigned to each processor. For example, the first processor could have rank 0, while the second could have rank 1, and so on.
- `chunk_size` refers to the size of the section of the array that is distributed to each array (see above)
- `i` is merely a running index for the for-loops in the program
- `max` is the maximum of the entire array, the *global* maximum
- `mymax` is the maximum of each chunk, or the *local* maximum. Each processor will have its own `mymax`

- `rem` is the remainder. In the case that the array size is not evenly divided by the number of processors, `rem` will be used to as evenly as possible divide up the extras amongst the processors.
- `array[]` is, obviously, the array of values. The size of this array could really be anything, but in this example it is 800.

```
MPI_Init( &argc,&argv);
MPI_Comm_rank( MPI_COMM_WORLD, &rank);
MPI_Comm_size( MPI_COMM_WORLD, &numprocs);
```

The next piece of code is specific to MPI. `MPI_Init` is how all MPI programs must start. This causes all parallel processes to exist after it is called. `MPI_Comm_rank` assigns the processor a unique rank not exceeding the size of `MPI_COMM_WORLD`, which is the number of total processors. `MPI_Comm_size` is similar; it assigns `numprocs` the value of `MPI_COMM_WORLD`.

```
printf("Hello from process %d of %d \n",rank,numprocs);
```

This print statement is a roll-call of sorts. It has all the processors report in with their rank number out of the total.

```
chunk_size = 800/numprocs;
rem = 800%numprocs;
```

This initializes the variables that concern the distribution of the array. As mentioned before, the `chunk_size` will be the total size of the array divided by the number of processors, and `rem` will be whatever is left over.

```
if (rank == 0) {
/* Initialize Array */
        printf("REM %d \n",rem);
        for(i=0;i<800;i++) {
                array[i] = i;
        }
/* Distribute Array */
        for(i=1;i<numprocs;i++) {
                if(i<rem ) {
                MPI_Send(&array[i*chunk_size],chunk_size+1, MPI_INT, i, 1, MPI_COMM_WORLD);
                } else {
                MPI_Send(&array[i*chunk_size],chunk_size, MPI_INT, i, 1, MPI_COMM_WORLD);
                }
        }
}
else {
        MPI_Recv(array, chunk_size, MPI_INT, 0,1,MPI_COMM_WORLD,&status);
}
```

This won't all be explained immediately, but the structure of this if/else-statement is something to understand in its entirety. This statement says which processors do which tasks. Notice that the if-statement is only true when `rank` is 0, or—more precisely—only Processor 0 will perform the tasks in the if-statement. The rest of the processors will perform the statements within the else block.

```
/* Initialize Array */
        printf("REM %d \n",rem);
        for(i=0;i<800;i++) {
                array[i] = i;
        }
```

Study first the tasks of Processor 0 (those statements in the if block). As the comment suggests, this first task performed by Processor 0 is the initialization of the array. The print statement merely displays what the remainder is after the chunk size is determined. The for-loop initializes the array values to their respective indexes. In an actual situation, the array would probably be populated by some outside source of numbers (since it's rather obvious that in a list of 0-799, 799 is the maximum), but it is for that reason exactly that this is a good way to populate the array. This way, it is 100% certain whether or not the algorithm is working correctly.

```
/* Distribute Array */
        for(i=1;i<numprocs;i++) {
                if(i<rem ) {
                MPI_Send(&array[i*chunk_size],chunk_size+1, MPI_INT, i, 1, MPI_COMM_WORLD);
                } else {
                MPI_Send(&array[i*chunk_size],chunk_size, MPI_INT, i, 1, MPI_COMM_WORLD);
                }
        }
```

The next section is a little more involved. Here, the algorithm needs to distribute chunks of the complete array to the separate processors, but because it needs to consider the remainder, some chunks need to be one larger than others. The entire distribution is enclosed in a for-loop. There will be as many distributions as there are processors *minus one*. (Note here that Processor 0 can keep a chunk for itself. There's no reason it can't do work too!) The if-statement says that while there are still undistributed remainders, the chunk size should be bigger. Note the difference between the two send statements:

```
if(i<rem ) {
MPI_Send(&array[i*chunk_size],chunk_size+1, MPI_INT, i, 1, MPI_COMM_WORLD);
} else {
MPI_Send(&array[i*chunk_size],chunk_size, MPI_INT, i, 1, MPI_COMM_WORLD);
}
```

The only thing left to understand is the syntax of the `MPI_Send` function. It has a pointer to the position of the array that it should begin sending from (`&array[i*chunk_size]`); defines how much of the array it should send from that point (`chunk_size`); says what data type the parameters are, in this case integer (`MPI_INT`); which processor to send it to (`i`); the "tag", which must be the same for all sends and receives in the program (`1`); and, finally, ends with the size of the MPI world (`MPI_COMM_WORLD`).

```
else {
        MPI_Recv(array, chunk_size, MPI_INT, 0,1,MPI_COMM_WORLD,&status);
}
```

The algorithm now addresses what the rest of the processors do. Because of the `MPI_Recv` function call, they've been waiting to receive information the entire time Processor 0 was initializing and dividing up the array. (This idle time where not all the processors are working in parallel is often referred to as *overhead time*.) The `MPI_Recv` function has much of the same syntax as `MPI_Send`, but with key differences. It is waiting for an `array` of size `chunk_size`, of data type `MPI_INT`, but notice how it isn't concerned with the pointer and remainder size like `MPI_Send` was. It just knows that it's waiting for an array of type integer. The next parameter says where it expects to receive from, in this case, `0`. This is followed by the "tag" `1` and the size of the MPI world, `MPI_COMM_WORLD`. It ends with a pointer to the status that was discussed at the beginning of this tutorial.

```
    /*Each processor has a chunk, now find local max */
    mymax = array[0];
    for(i=1;i<chunk_size;i++) {
                    if(mymax<array[i]) {
                            mymax = array[i];
                    }
         }
```
At this point, the algorithm assumes a rather familiar appearance. This is a very normal way of looping through an array to find a maximum, using a temporary maximum and updating it every time a larger value is found. The only parallel note to understand here is that every processor is doing this for its own chunk at the same time.

(The following print statements were omitted because they weren't essential to finding the maximum. All they did was print the beginning and ending values of each chunk, and then printed the local maximums of each. Every processor did this for its own chunk.)

```
   /*Send local_max back to master */
        if (rank == 0) {
      max = mymax; //Store rank 0 local maximum
                for(i=1;i<numprocs;i++) {
                        MPI_Recv(&mymax,1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,&status);
                        if(max<mymax) max = mymax;
                }
                printf("The Max is: %d",max);
        }
        else {
                MPI_Send(&mymax, 1, MPI_INT, 0,1,MPI_COMM_WORLD);
        }
```
This if/else structure should look familiar, since it's the same approach that the algorithm took to sending out the chunks. Here it's the exact opposite: Processor 0 stores its own local maximum and receives all the local maximums from the other processors, and the other processors are doing the sending.

```
        if (rank == 0) {
      max = mymax; //Store rank 0 local maximum
                for(i=1;i<numprocs;i++) {
                        MPI_Recv(&mymax,1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,&status);
                        if(max<mymax) max = mymax;
                }
                printf("The Max is: %d",max);
        }
```
Again looking at the algorithm in sections, Processor 0 has its own set of tasks that are unique from those of the other processors. First, it stores its local maximum as the global maximum (see the comment). Then, it does a similar looping through values to see if any other processor's local maximum is bigger than the current maximum. If so, the global maximum is updated. Finally, it prints out the global maximum.

```
                        MPI_Recv(&mymax,1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,&status);
```
The `MPI_Recv` statement, replicated above, is similar to the previous receive statements. It begins with a pointer to `mymax`, which means it is storing the incoming value in `mymax`, which is of size `1` and of type integer (`MPI_INT`). A new parameter here is `MPI_ANY_SOURCE`, which means that it is receiving from any sender available. The following three parameters are again the "tag", `1`, `MPI_COMM_WORLD`, and a pointer to `status`.

```
        else {
                MPI_Send(&mymax, 1, MPI_INT, 0,1,MPI_COMM_WORLD);
        }
```

The rest of the processors are using `MPI_Send` to send their `mymax`, which is of size `1` and data type integer (`MPI_INT`) to processor `0`. That's then followed by the "tag", `1`, and `MPI_COMM_WORLD`.

```
        MPI_Finalize();
        return 0;
 }
```

The main function is ended with a call to `MPI_FINALIZE`, which ends all the processes (much like `MPI_INIT` started them), and a `return` statement that is analogous to any C program.