CS402: Seminar 5
MPI

1 MPI

The Message Passing Interface, MPI, is a specification for a number of methods and some data types that support a particular model of parallel programming. All data is shared using explicit messages, that is to say we don't have any shared memory space. Each processor can send to any other processor, and messages must be explicitly received. If we want to synchronise our processes, we must use a barrier. Some MPI operations will have the side of effect of creating a barrier.

1.1 Program 1: `helloWorldMPI.c`

The first program we will look at is the ubiquitous *"Hello, world!"* program, however, this is an MPI version of the program. This program will have each process print out a message that includes its rank. You can find this program as `helloWorldMPI.c`. We can compile this program using the following command:

```
mpicc helloWorldMPI.c -o helloWorld
```

The `mpicc` compiler is just a wrapper around a C compiler, which in our case is the `gcc` compiler that you have been using in previous worksheets. This will create the executable `helloWorld`, which we can run using the following command:

```
mpirun -n <num_procs> helloWorld
```

where `<num_procs>` is the number of processes you want. The output of the command should look something like this:

```
$ mpirun -n 4 ./hello
Hello world, from process 1 of 4
Hello world, from process 3 of 4
Hello world, from process 0 of 4
Hello world, from process 2 of 4
```

Notice that the output may be out of order. Since the processes are all executing simultaneously and independently, we can't guarantee which process will print its output to the screen first. We can now look at the interesting parts of this program.

- `MPI_Init(&argc, &argv)` – this sets up the MPI run time environment, ensuring all our processes can use any of the MPI methods.

- `MPI_Comm_size(MPI_COMM_WORLD, &numprocs)` – this method returns the number of processes in the given communicator.[1] In this case we give the *world* communicator, `MPI_COMM_WORLD`, which contains all the processes. The result is placed in the second argument, so you need to pass in a pointer here.
- `MPI_Comm_rank(MPI_COMM_WORLD, &id)` – this finds the rank of the current process in the given communicator, storing it in the second argument. Note that the ranks will start at 0, so if there are n processes, there will be ranks 0 to $n-1$.
- `MPI_Finalize()` – this signals the end of the MPI part of the program. Every process **must** call this function, and you can't use any MPI functions after it has been called.

Congratulations, you have compiled and run a basic MPI program. If you have any questions at this point, make sure you ask one of the tutors for help.

2 Basic Message Passing

We will now see the methods used to actually send data. The simple program `message_passing.c` will send a message from one process to another.

The `MPI_Send` method, as you might expect, is used to send a message from one process to another. The arguments we have passed to this function are:

- `myarray` – the address of the data you want to send.
- `3` – the number of elements of data you want to send.
- `MPI_INT` – the type of the data.
- `1` – the rank of the process you are sending to.
- `tag` – the tag of the message; this is a way to identify the message.
- `MPI_COMM_WORLD` – the communicator to use.

The `MPI_Recv` method receives messages. The arguments are similar to `MPI_Send`, but with an added argument, `&status`, which will hold the status of the call to receive.

Hopefully you can see how MPI treats processes. Each MPI process simply runs the same program, executing exactly the same code path as the others. However, when you make calls to any MPI functions, the different processes will get different results.

Task 1 As a quick C refresher, and to check the message passing is working, add some code to print out the array on the sending and then receiving processors. As you have seen over the past few weeks, print statements can be very helpful when debugging your code!

[1] A communicator is a named group of processes

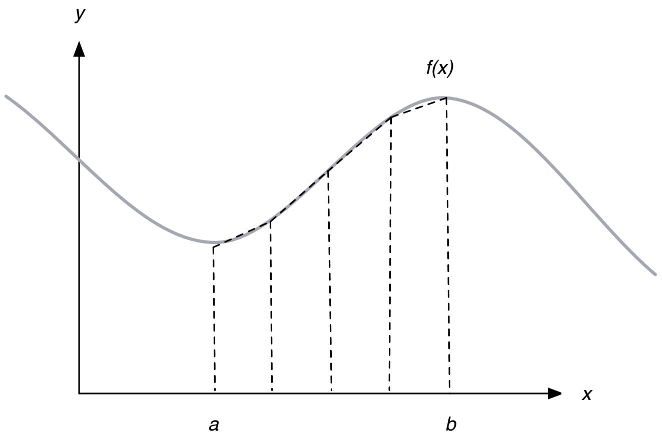

Figure 1: The trapezoidal approximation of the integral (area under the curve).

2.1 Message Passing: Estimating the Integral of a Curve

In this section, we will look at using MPI to parallelize a simple mathematical problem: estimating the integral of a curve. The integral of a curve is simply the area underneath it. In order to calculate this area, we can use a simple approximation called the trapezoidal rule. Imagine splitting the area under the curve in to a number of equal-width trapezoids, as in Figure 1, we can then add up all these areas in order to approximate the integral of the curve. Mathematically, the area of the ith trapezoid can be expressed as follows:

$$\frac{1}{2}h(f(x_i) + f(x_{i+1}))$$

where h is the width of the base, and $f(x)$ gives us the height of the curve at point x. Hopefully you can see the similarities between this formula, and the discretisation used in the `deqn` program we have been studying. Summing up all these areas in a serial program would look a little bit like this (*note:* this is not a complete program):

```
// Set up the program here

h = (b - a)/n;
integral = (f(a) + f(b))/2.0;

x = a;

for( i = 1; i <= n-1; i++) {
    x = x + h;
    integral = integral + f(x);
}

integral = integral*h;

float f(float x) {
    // Return the value of the function f at point x
    // e.g. x^2
    return x * x;
}
```

If the maths looks a little bit off, it's because the sum of the areas has been simplified, giving the following equation.

$$h(f(x_0)/2 + f(x_n)/2 + f(x_1) + f(x_2) + \ldots + f(x_{n-1}))$$

This is the formula that the above program is implementing.

Task 2 Complete the serial version of the application. Check it works, and then try timing it using the `time` command, like this

```
time ./trapezoid
```

Make a note of the time this takes.

2.1.1 Parallel Integral Estimation

Hopefully you can see how we might parallelize this program. Each of the areas can be calculated separately from all the others. So we can give each process a portion of the curve to approximate. For example, one process could work on part A of Figure 2 and the other process could work on part B. We can then add up these two estimates to end up with the total area under the curve.

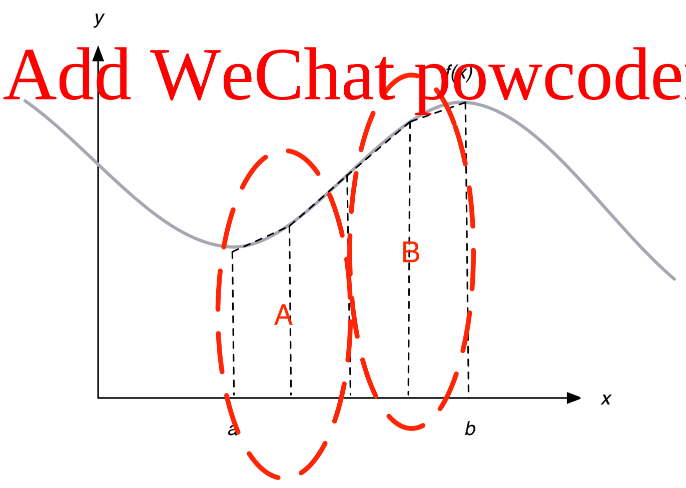

Figure 2: Parallelizing the integral approximation.

In order to make sure each process only calculates the area for its section of the curve, we must make sure each process knows the values that define its upper and lower ends. We can then calculate this partial area, before sending the result back to one of the processes to calculate the total. This is what the code in `trap.c` does.

Task 3 Compile the code found in `trap.c`. Try running it, making sure you use the `mpirun` command like in Task 1. Like Task 2, try timing it with different numbers of processes. Does the time taken change?

Feel free to add some print statements to check your understanding of the program. Try changing the number of trapezoids used, and watch the accuracy of the estimate increase!

2.2 Collectives and Integral Estimation

The pattern used in the integral estimation program is very common: share out the data, do some work, then collect all the data back onto one of the processes. Conveniently, the MPI standard includes some *collective* operations, which can handle these kinds of situations.

The collective operation we can use to sum up the partial integrals is `MPI_Reduce`. This method will have each process send to one of the other processes (typically the root), and on the receiving process, the messages are combined using some operation, for example, a sum. The `MPI_Reduce` function has the following definition

```
int MPI_Reduce(
    void* operand,          /* data to send */
    void* result,           /* where to store the result */
    int count,              /* the number of data elements */
    MPI_Datatype type,      /* the type of the data */
    MPI_Op operator,        /* the operator to apply to the data */
    int root,               /* the process to collect the data on */
    MPI_Comm comm)          /* the communicator to use */
```

A typical call to `MPI_Reduce` will look something like this

```
MPI_Reduce(&local_data, &result, 1, MPI_INT, MPI_SUM, 0,
    MPI_COMM_WORLD);
```

This call would sum up the values of `local_data` from all the processes, storing it in the `result` variable on the root. A few things to note: you can't use the same variable for the operand and the result, and *all* processes must call this function, in exactly the same way.

The `MPI_Op` argument is the operation that can be applied to the data, a few of the most useful are:

- `MPI_SUM` – Adds up the elements.
- `MPI_PROD` – Multiplies the elements.
- `MPI_MAX` – The maximum of all the elements.
- `MPI_MIN` – The minimum of all the elements.

Task 4 Now that you have seen the `MPI_Reduce` collective, your task is to replace the for loop in the parallel trapezoidal integral program with a single collective call. You can replace all the code between `/* BEGIN SUM */` and `/* END SUM */` with this single call.