Parallelisation in Magma.

VUB Magma workshop.

Andrew Darlington

Tuesday 17th March 2026

Plan

1. What is parallelisation?
2. First steps to parallelising Magma code.
3. A few more functions and techniques.
4. Demonstration.

What is parallelisation?

What is parallelisation?

The first time we write code, it is usually written linearly , with possibly multiple modules:

```
> MyFirstFunction := function();
> return "Hello World!";
> end function;
>
> MySecondFunction := function(n);
> for i in [1..n] do
> print MyFirstFunction();
> end for;
> return "done!";
> end function;
```

This will run on a **single** processor.

What is parallelisation?

Suppose now we have the following modified functions:

```
> MyFirstFunction := function(t);
> System("sleep " cat IntegerToString(t));
> return "Hello World!";
> end function;
>
> MySecondFunction := function(n);
> for i in [1..n] do
> print MyFirstFunction(i);
> end for;
> return "done!";
> end function;
```

Then `MySecondFunction(n)` will take $n(n+1)/2 \sim O(n^2)$ seconds to complete...

What is parallelisation?

... But if we now have a number of ‘workers’ (these will be our processors or ‘threads’) and assigned each of them a particular instance of `MyFirstFunction(i)` to work on, then the actual time taken can be vastly reduced...

What is parallelisation?

- If we have n workers, then we can reduce the time to just n seconds, even though the cumulative total time of the workers is still $n(n+1)/2$.
- Even if we have fewer than n workers, whenever a given worker has finished their task, and there are more tasks to complete, we can assign one of the uncompleted tasks to the now free worker. Then `MySecondFunction(n)` should still take less than $n(n+1)/2$ seconds to complete.

This is the idea behind *parallelisation*.

First steps to implementation.

Parallelisation in Magma how?

Parallelisation in Magma can be accomplished by use of its **manager-worker** model.

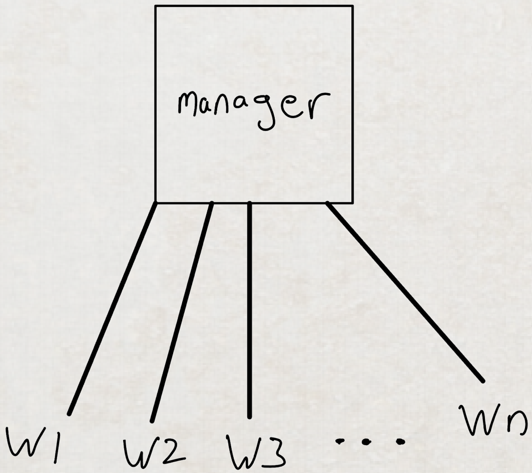
The manager

The **manager** is the part of the code which assigns tasks to the workers, and collects and deals with their output after they have finished. It will typically contain the main function, and will provide the general structure of the code.

The worker(s)

The **workers** receive tasks from the manager. They will work on their task until it is completed, return the result, and then tell the manager that they are ready to be assigned another task, until all tasks have been completed.

Magma's manager-worker model



Connecting manager with workers

The manager and workers first need to know how to communicate with each other. This is achieved with the `Socket` function:

```
Socket:(LocalHost:="host_name",LocalPort:=port);
```

This is like choosing a channel for two radio devices to talk to each other.

- The **host name** is the name of the machine (or network address) you're working on.
- The **(local) port** is a number which designates the specific port for communication. This number should be less than 65,536.

Manager delegating jobs

We now need a function which allows the manager to send off tasks to the workers. This is achieved with

```
DistributedManager(socket,tasks);
```

- The ‘socket’ is what we defined on the previous slide.
- The ‘tasks’ are a sequence, where each term is the input data for a particular worker. If, for example, `tasks:=[n1,n2,n3]`, then the first worker gets input ‘`n1`’, the second worker gets ‘`n2`’, and the third ‘`n3`’.

Workers accepting jobs

The corresponding function for the workers is

```
DistributedWorker(host,port,main_function);
```

This sits **at the bottom** of the worker file, and tells Magma where to send the task data the worker has received.

However, this still is not quite enough! Let's see why...

Starting up workers

To start up a worker, we type

```
magma worker.m
```

into a terminal window. Each worker we start up represents one thread/processor of our machine. Type

```
magma worker.m > worker.log
```

if you want to also create a log file (called ”worker.log”) for the worker.

Starting up workers

To start up multiple workers, we'll need to type

```
for i in {1..n} do
magma worker.m > worker-$i.log &
done
```

where *n* is the number of workers we want to start, each with its own log file, "worker-i.log".

But this is still very laborious — we would like to run these automatically!

Starting up workers within Magma

We can run terminal-based commands within Magma by using the `System` function. This takes a string and executes it as a terminal command:

```
> for i in [1..n] do
> command := "magma worker.m > worker-" cat
    IntegerToString(i) cat ".log" &;
> System(command);
> end for;
> Results := DistributedManager(socket,tasks);
```

Multiple types of workers

Everything we have talked about so far can be extended to parallelise multiple parts of your code.

```
> for i in [1..n] do System("magma workerA.m > workerA-"
    cat IntegerToString(i) cat ".log 2 &";
> end for;
> ResultsA := DistributedManager(socketA,tasksA);
> blah;
> for i in [1..n] do System("magma workerB.m > workerB-"
    cat IntegerToString(i) cat ".log 2 &";
> end for;
> ResultsB := DistributedManager(socketB,tasksB);
```

A few more functions.

Time commands

When running linear code, getting the time it takes for a function to execute is straightforward:

```
> time function(params);
output
of
function
Time: x.xx
```

But if we try this for parallel code, we don't get accurate information...

Time commands

Instead, we must collect the time values from several locations, and then display it in a helpful way.

- Every worker log file finishes by displaying the time taken by that specific worker.
- We may also want to take the time taken by just the manager into account.
- Neither of these are the time elapsed ‘in the real world’.

Time commands

We use shell commands again! We can collect the total worker times with the following:

```
> t := Pipe("grep -h 'Total time:' worker*-*.log | awk
    '{sum += $3} END {print sum}'", "");
> worker_time := eval StripWhiteSpace(t);
```

Time commands

The manager time is collected using the function `Cputime()`:

```
> main_function := function(params);
> tt := Cputime();
> blah;
> manager_time := Cputime(tt);
> fprintf "time file", "\n%o %.2o %o\n", "CPU time for
    manager is", manager_time, "seconds";
> return blah;
> end;
```

Note that `worker_time` and `manager_time` can be summed to obtain the total CPU time. It is possible to get the 'real time' elapsed using the `Time()` command, but it's very fiddly and not usually done.

Occupied sockets and deleting jobs

Sometimes your code might throw an error while running, or you might kill it in the middle of something. When running code in parallel, this can be problematic...

To clear the jobs and to make the socket useable again, we need to run the following command in the terminal:

```
pgrep -f 'magma' | xargs -r kill -9 2 > /dev/null ||
    true
```

Demonstration

Questions?