Parallel luau

Parallel Luau is a way of achieving real multithreading in roblox using the Actor instance.

> **What is Multithreading**
>
> Multithreading is a way of running multiple threads at the same time. Base Lua does not have this feature and when you use task.spawn or coroutines it is not really multithreading. The function is still only running on a single thread but Lua makes it so it looks like it is running at the same time

Actors

Actors is an Instance that allows scripts under that Actor to run in Parallel.

> **Actors and Modules**
>
> The Memory under Actors will not be the same so when you require a module a different table will be returned (if your module returns a table) than in the main Thread. This means you can't use modules to transfer data between Actors.

```
--Script1
local module = require(...)
module.X = 1
print(module.X) --> 1

--Script2
local module = require(...)
task.wait(1)
print(module.X) --> 1

--Script Under Actor
local module = require(...)
task.wait(1)
print(module.X) --> nil
```

Communicating Between Threads

There are currently 4 ways to efficiently communicate between threads: Actor Messaging API, Shared Tables, Bindables, or Direct Data Model Communication.

Actor Messaging API

The Actor Messaging API allows a script to receive data from other actors or the main thread. It currently consists of 3 methods: SendMessage(), BindToMessage(), and BindToMessageParallel().

SendMessage

Used to Send Messages between threads. The first argument would be the key while the rest will be the data to send

```
local Actor = ...
workerActor:SendMessage("myKey", "hello world")
```

BindToMessage(Parallel)

Used to receive data. If BindToMessageParallel was used then run the code in Parallel. The first Parameter is the key while the second Parameter is a callback.

```
local Actor = script:GetActor()
Actor:BindToMessage("myKey", function(data)
    print(data)
end)

Actor:BindToMessageParallel("myKey", function(data)
    print(data)
end)
```

Shared Tables

Shared Table is like a normal table except its contents are shared between threads. This means that if you update a value in one thread, it will also update for other threads.

```lua
local SharedTableRegistry = game:GetService("SharedTableRegistry")
local SharedTable = SharedTableRegistry:GetSharedTable("mySharedTable")
--//Parallel Thread
SharedTable.X = 1
--//Another Parallel Thread
print(SharedTable.X) --> 1
```

> **Thread Safety**
>
> When using SharedTable you could have race conditions that can cause unwanted behaviors. If you want to know more about Thread Safety you can scroll down

Bindables

One of the best ways right now (9/29/2023) to send data between threads is using Bindables as they are almost 4x more efficient than SharedTables.

```lua
--//main Thread
local Actor = ...
local BindableFunction = Actor.Bindable
-- assuming you have an event/function parented to the Actor

local data = BindableFunction:Invoke("DoSomething")


--//Actor Script
local Actor = script:GetActor()
local BindableFunction = Actor.Bindable
BindableFunction.OnInvoke(data)
    task.desynchronized() -- runs in parallel
    return doSomething(data)
end
```

> **Tables and Bindables**
>
> When Sending/Returning Tables, avoid large dictionaries if you can. If possible try to convert dictionaries into arrays. Another thing is if your array is going to consist of Strings that are

showing up more than once you could send/return two tables, the data table, and a key table.

```
--//before
local data = {
    [1] = "String1",
    [2] = "String1",
    [3] = "String3",
    [4] = "String2",
    [5] = "String2",
}
return data
----------------------------------------
--//after
local key = {
   [1] = "String1",
   [2] = "String2",
   [3] = "String3",
}
local data = {
    [1] = 1,
    [2] = 1,
    [3] = 3",
    [4] = 2,
    [5] = 2,
}
--to get the value of an index you can do key[valueAtIndex]
return data,key
--or
return {data,key}
```

This is much more efficient then as it's faster to Serialize numbers than it is for strings. And the size of the data being sent over is lower as well. (this can also apply to remotes)

Direct Data Model Communication

This type of communication uses the Instances under the game and modifying/reading their properties. However, to maintain thread-safety roblox has systems that disallow threads that are desynchronized to read/write to the Instances.

Thread Safety

Thread Safety is the avoidance of race conditions, Which happens when multiple threads try to read and write to a shared resource causing unwanted behaviors. When using Parallel lua with roblox's DataModel. Roblox already has Thread Safety implemented into its Instances. Roblox has 4 Safety Levels: Unsafe, Read Parallel, Local Safe, and Safe. You can tell if a function/property has one of these tags by looking at the Roblox's API. If it has no tags then it defaults to UnSafe.

> **Safety Tags**
>
> **UnSafe**
>
> Functions cannot be called and Properties cannot be read or written (modified) to
>
> **Read Parallel**
>
> Properties can be read but not written to
>
> **Local Safe**
>
> If the instance was created was created under an actor then that actor can Read and Write to. Other actors can Read but not write. Functions can be called only in that actor.
>
> **Safe**
>
> Functions can be called and Properties can be read and written to

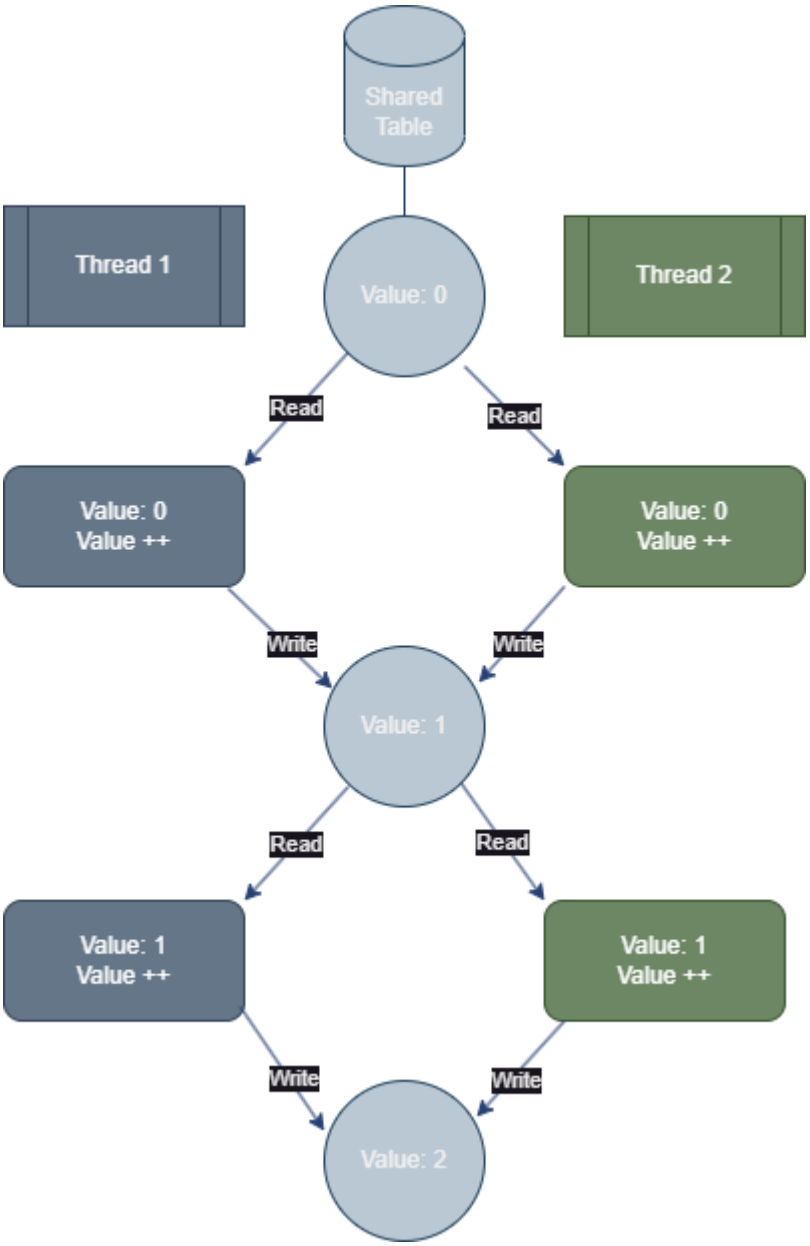
Race Conditions

Race Conditions are when a thread reads or writes data when another thread is not finished computing that data, causing data to overlap or create unexpected behaviors.

```lua
--Example
local SharedTable = ...
SharedTable.Value = 0
--Let's say we are trying to add to a key called Value in the SharedTable
--Each Thread will add 100 to the Value Key

--//Parallel Thread
for i = 1,100 do
    SharedTable.Value = SharedTable.Value + 1
end

--//Another Parallel Thread
for i = 1,100 do
    SharedTable.Value = SharedTable.Value + 1
end

--//Outside of Parallel (After Parallel Threads ran)
print(SharedTable.Value)
--> This won't always print 200 as both Parallel are writing to The Value
--at the same time which can overlap
```

More Explanation

The reason why it's happening can be shown in this diagram.



If you look at Value, each time a thread reads and writes it is reading and writing at the same time (this will not always happen). So it means that if both Threads run a for loop increase Value by 100, the Value will not be 200 most of the time.

Avoiding Race Conditions

When working with the DataModel roblox already implemented Thread Safety as shown above. Otherwise what you can do is combine the data in a Synchronized state so there is a lesser chance of data overlapping with each other or if you are working with a shared table use the functions increment() or update() to update data as it does an atomic update to the values.

Using Microprofiler

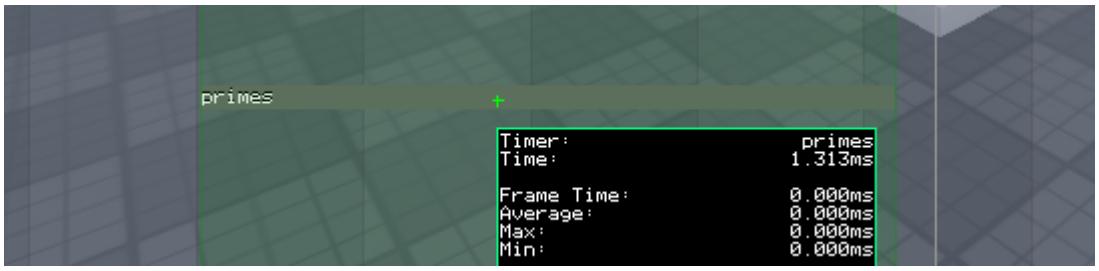
To open microprofiler on the client press ctrl-shift-f6. To pause press ctrl-P, If it opens an object viewer press ctrl-shift-f6 twice to reopen it and press ctrl-P again.

Using the Debug Library

You can use profilebegin to help find how long your tasks takes.

```
local function is_prime(n)
    if n <= 1 then
        return false
    end
    for i = 2, math.sqrt(n) do
        if n % i == 0 then
            return false
        end
    end
    return true
end
local function calculatePrimesFrom(from,to)
    debug.profilebegin("primes")
    local primes = {}
    for i = from,to do
        table.insert(primes,is_prime(i))
    end
    debug.profileend()
    return primes
end
calculatePrimesFrom(0,10000)
```

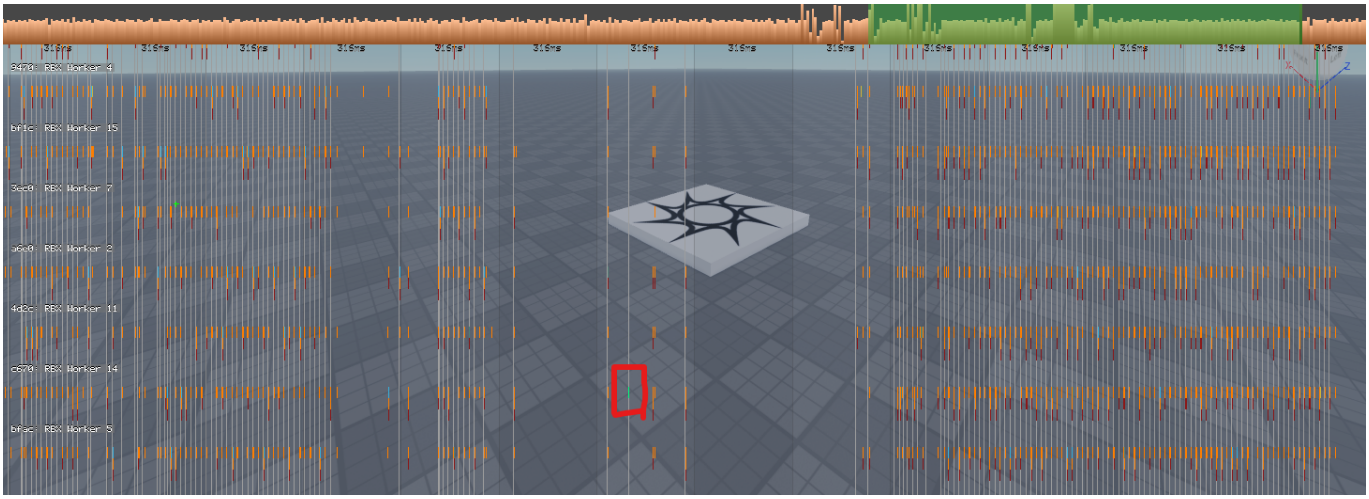
This will allow you to see how long primes takes to calculate.



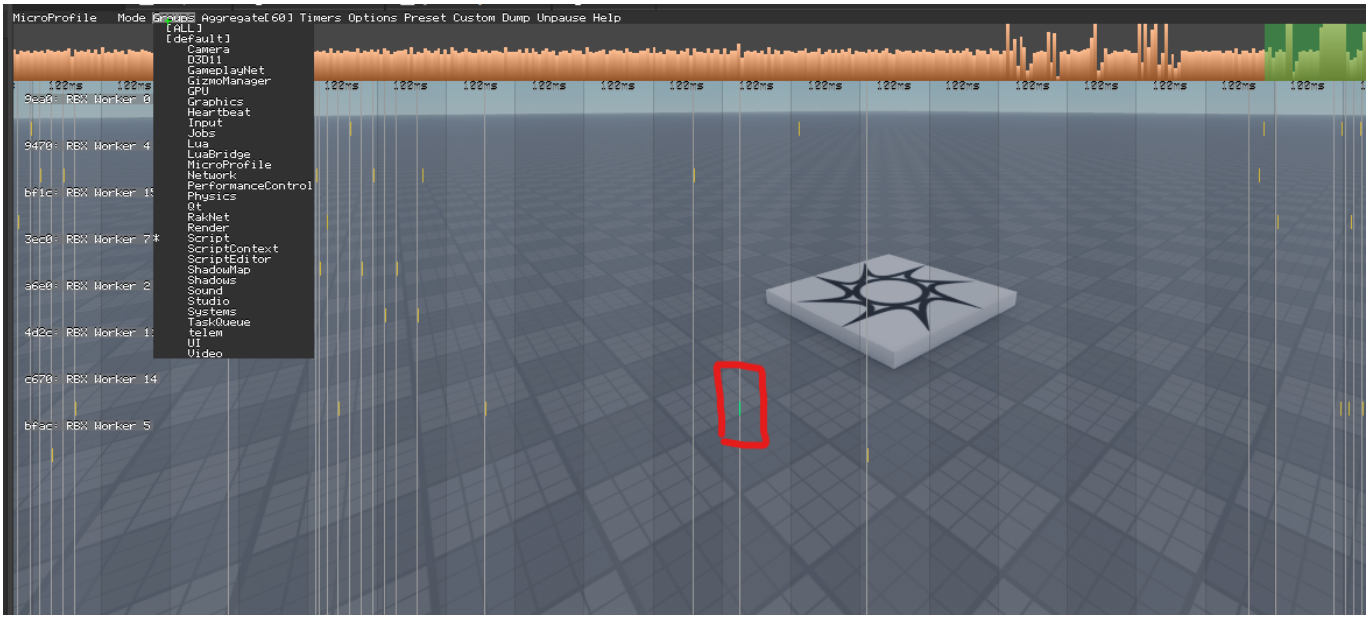
which you can see here takes 1.313 ms

How to look for profiles

Profiles will have a unique color depending on the name, so each profile will have the each color each time. To look for a profile look for colored boxes that stand out more.



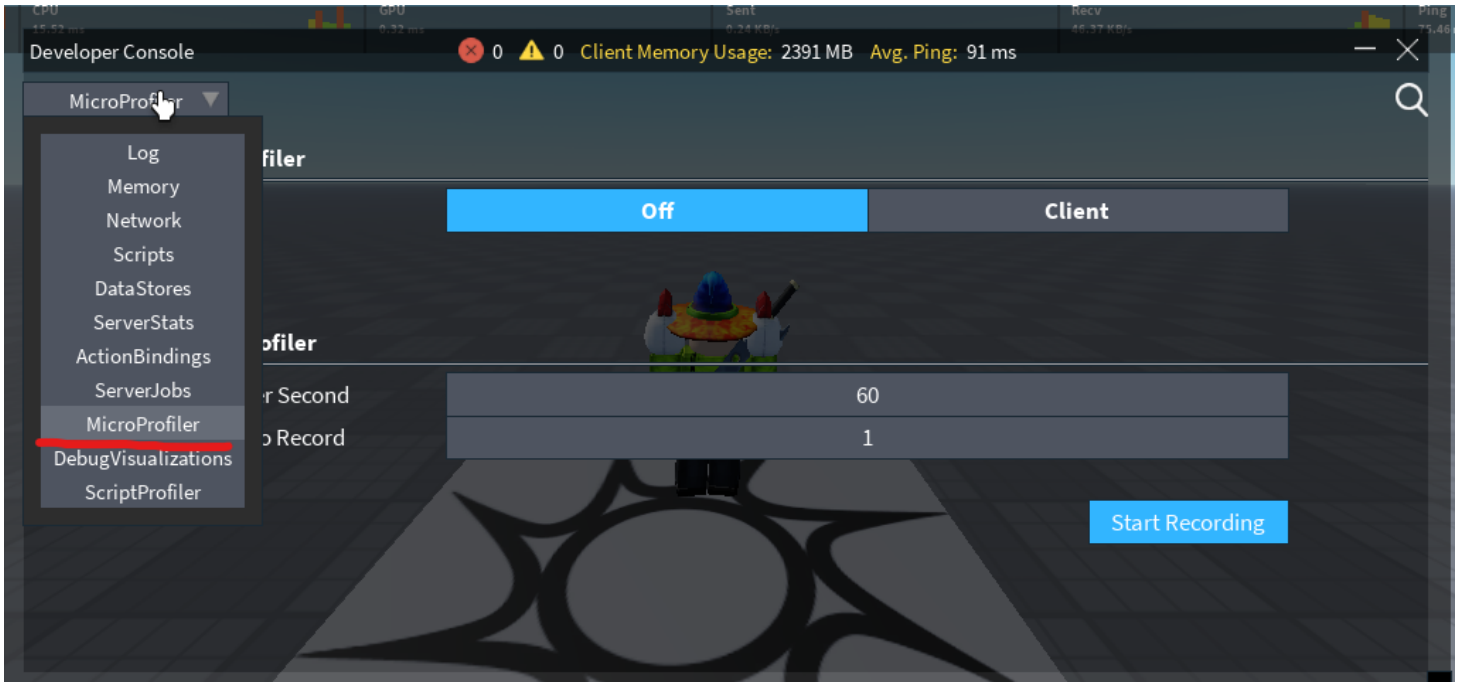
But if your profiles last very short you can sort them by going to `Groups` and disabling [ALL] and enabling just Script and it will make it easier to spot



Some other groups I recommend enabling are `Lua` and `TaskQueue`. Lua will display the Scripts while TaskQueue will display stuff like the Sleep timer.

Accessing MicroProfiler for Server

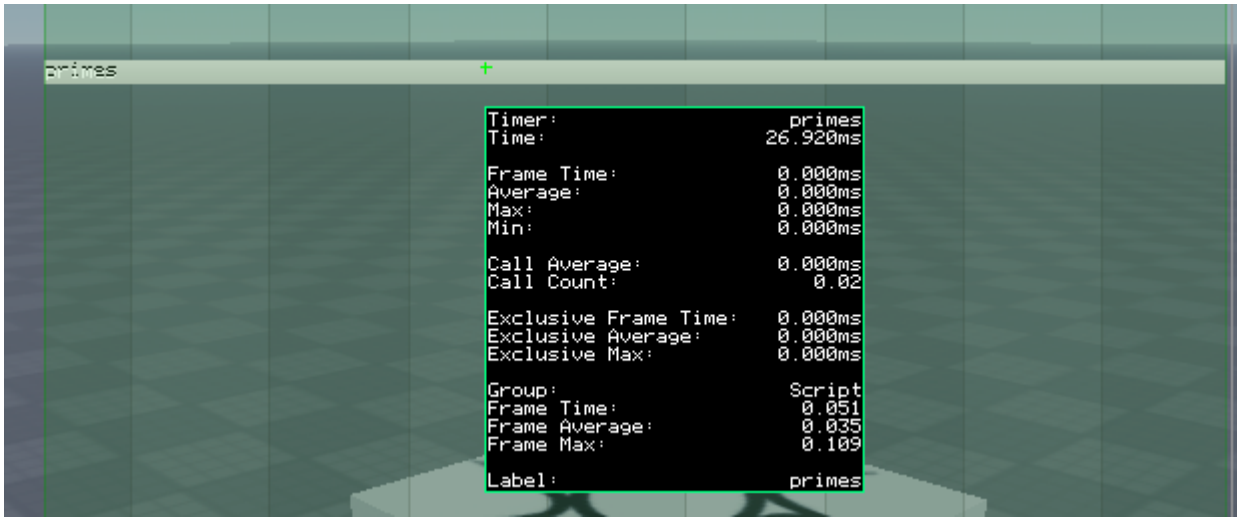
To see the microprofiler of a server you would need to go to Developer Console (f9) and go to the `MicroProfiler` tab.



After that, I recommend setting `Frames Per Second` to 60 and `Seconds to Record` to 4 (maximum time). And to Record press `Start Recording`. After it is done Recording it will display a path in which it is saved. Follow that and you can view the data in a browser.

Why multithread

Multithreading can help increase performance by a lot. For example, if we try to calculate the primes from 0-100000 it will take the server



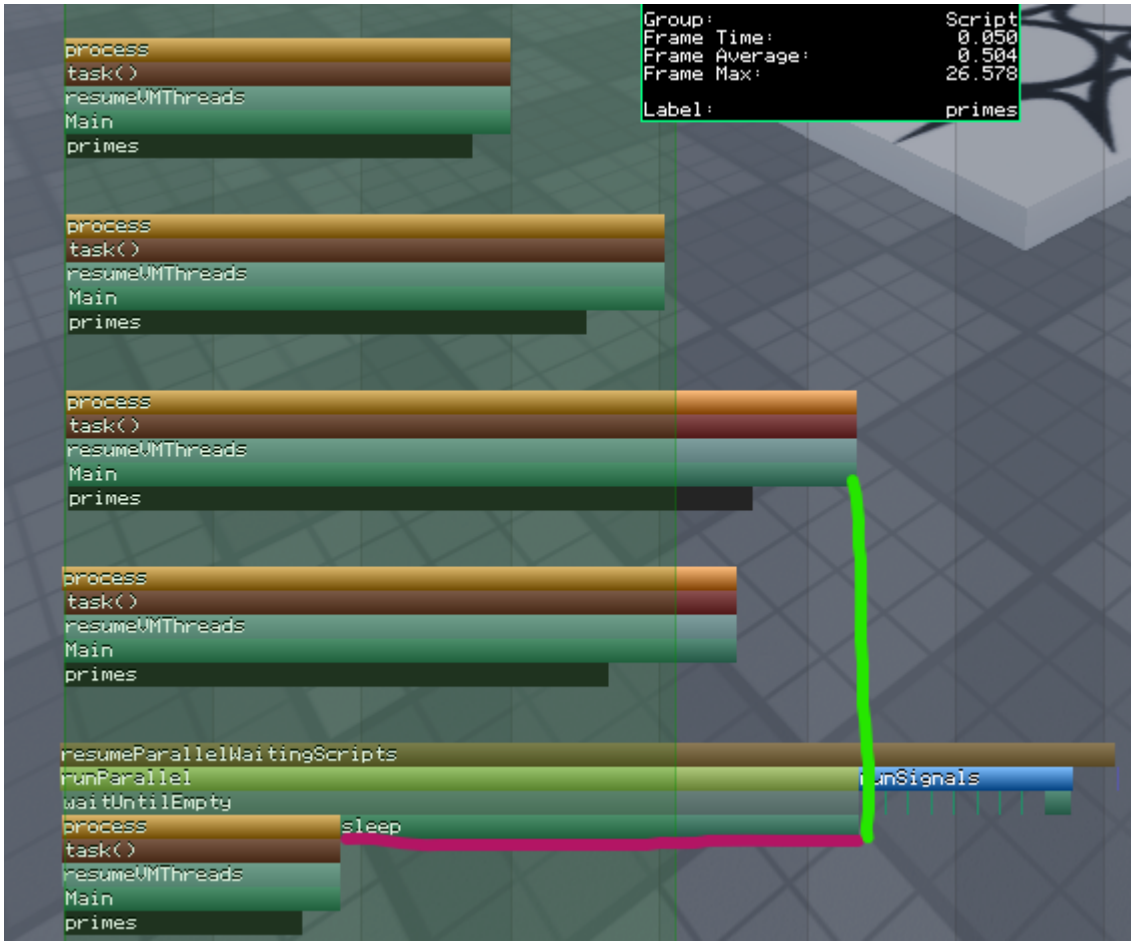
The tasks takes 26.920ms to calculate which is going to cause performance issues as each frame lasts 1/60 seconds which is ~16ms and since the task takes over 16ms the server will lag for 10ms.

But if we split the tasks into 8 separate tasks

each task only takes about 4ms and since they are running in parallel to each other calculating 100000 primes only takes 4ms in total.

Utilizing Parallel Luau properly

When using Parallel Luau it is recommended to separate tasks into smaller tasks. So let's say you have a task that takes 5ms to compute on a single thread. What you can do is split the tasks into 5 threads, Each chunk taking ~1 ms to compute, Saving 4 ms. Also when using Parallel Luau avoid yielding threads with wait or coroutine. Because when you use task.wait it will bring it out of parallel and avoid long tasks as will cause the main Thread (not to be confused with the Main script in the Image) will display the `sleep` timer because one of the tasks that is in parallel is not finished forcing it to wait.

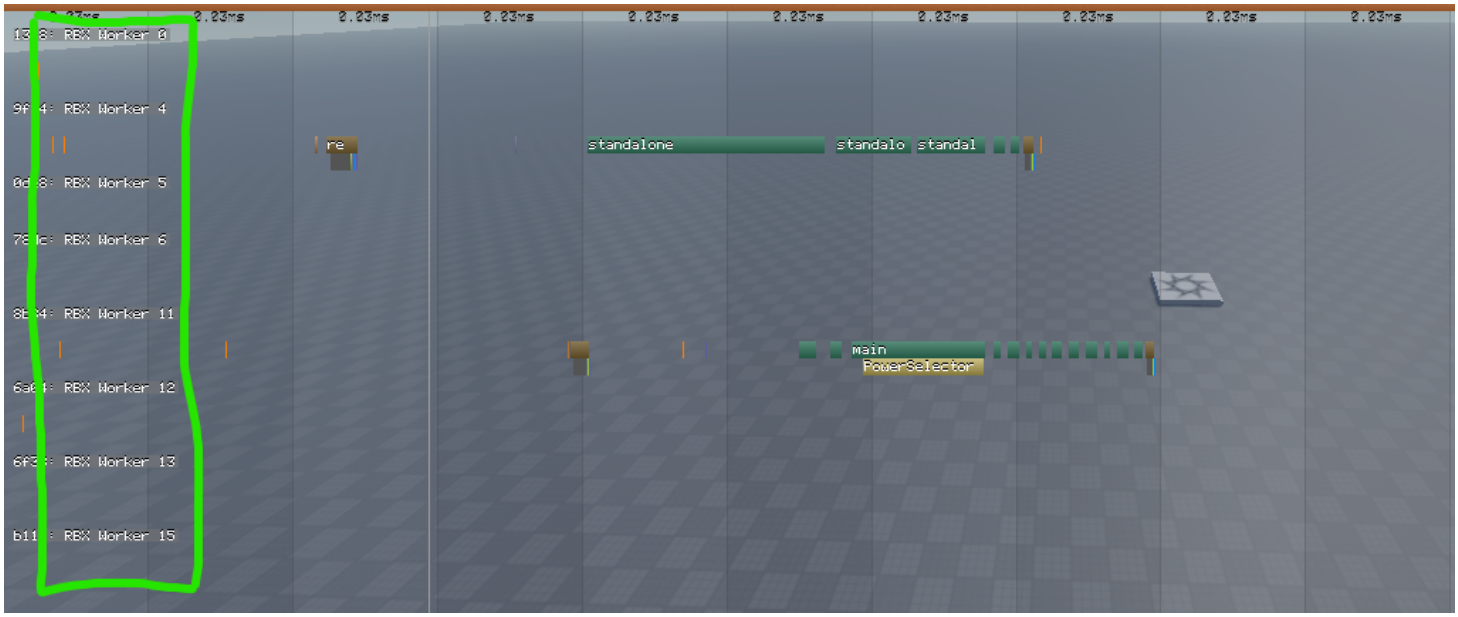


What we could have done better here was to split the tasks into smaller tasks and use more Actors.

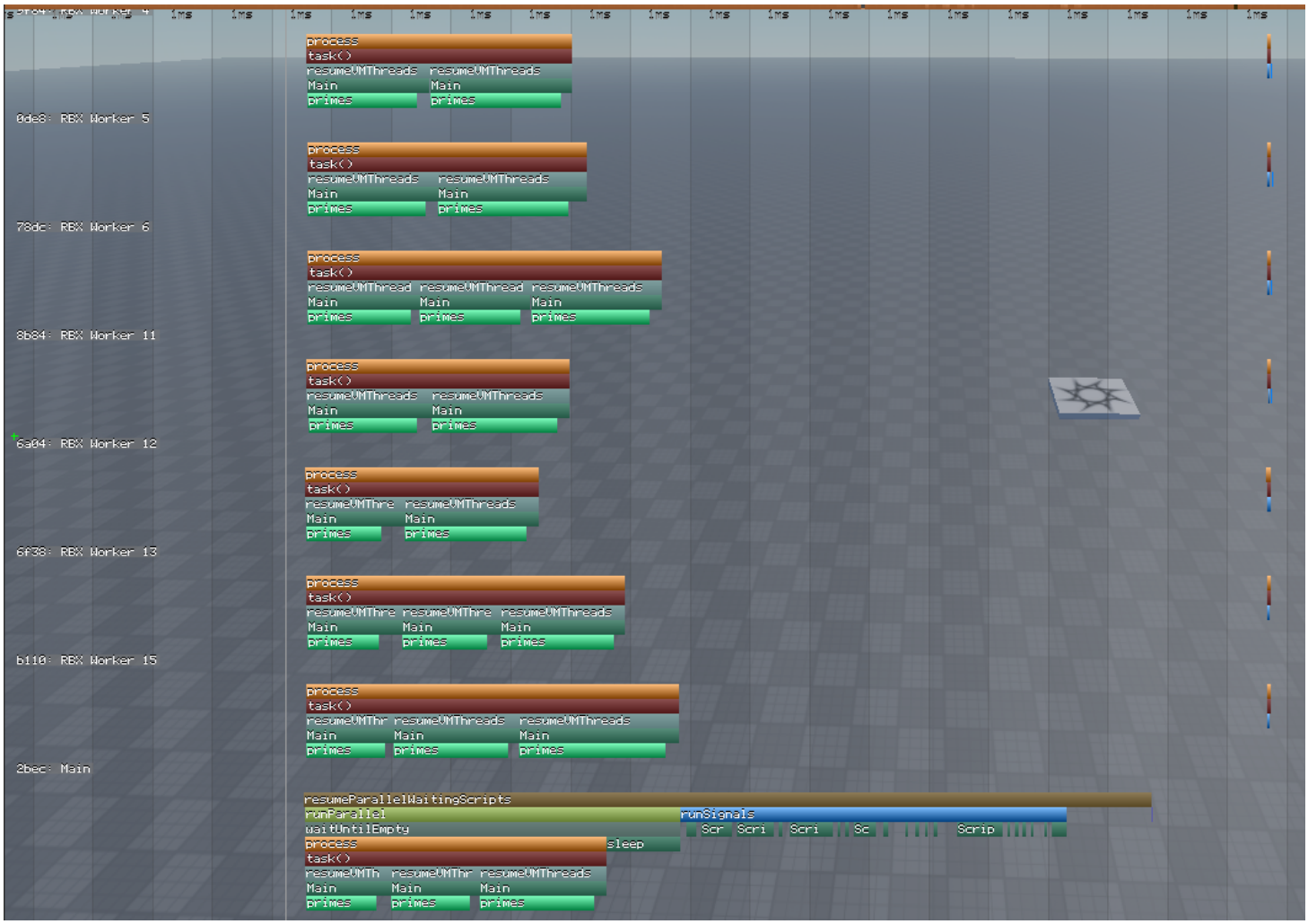
> **Splitting too much**
>
> When using Parallel luau try not to split the tasks to into very small sections. If a simple task takes 1 ms on a single thread and you try to split it into 2 separate threads you could see a .4 ms decrease in time but if you split it into 4 threads then you could see it take longer than on a single thread as other factors can cause delay such as going into parallel and sending data across.

How many actors should you use

On a roblox server, the amount of workers is determined by the maximum player count (source). While on the client it depends on the client's device. Workers are just how many threads can be utilized. Every time you run a parallel task the task will go to one of these workers. Roblox will try to balance which Worker should a task go to.



So when determining the number of Actors you want to use it is usually recommended to use more Actors than workers as roblox will balance the tasks between workers.



but avoid using too many or else other problems such as memory will show up.

Conclusion

Parallel Luau is not an easy thing to explain and understand so don't worry if you don't understand. If you want to see the code I used for the Images above, make a copy of this place. If you want to learn more about Parallel Luau check out here.