Parallel and Distributed Computing

Alberto Paoluzzi – Lecture 08 Parallel programming in Julia

Wed 23-03-2022

1. Parallel programming
2. Asynchronous programming (concurrent-computing)
3. Multithreading
4. Distributed computing

Section 1

Parallel programming

Parallel programming

Julia provides flexibility in in parallel programming, with solutions for

- asynchronous computing,

Parallel programming

Julia provides flexibility in in parallel programming, with solutions for

- asynchronous computing,
- multithreading,

Parallel programming

Julia provides flexibility in in parallel programming, with solutions for

- asynchronous computing,
- multithreading,
- distributed computing

Parallel programming

Julia provides flexibility in in parallel programming, with solutions for

- asynchronous computing,
- multithreading,
- distributed computing

Parallel programming

Julia provides flexibility in in parallel programming, with solutions for

- asynchronous computing,
- multithreading,
- distributed computing

and also for

- GPU computing.

Asynchronous programming:

A programming paradigm where parts of code are broken into small independent and parallel tasks.

The tasks can be synchronized on specific conditions and can communicate over channels.

Asynchronous programming:

A programming paradigm where parts of code are broken into small independent and parallel tasks.

The tasks can be synchronized on specific conditions and can communicate over channels.

synchronous operations

In tasks are performed one at a time and only when one is completed, the following is unblocked.
In other words, you need to wait for a task to finish to move to the next one.

asynchronous operations

on the other hand, you can move to another task before the previous one finishes

Asynchronous programming example

When a program needs to interact with the outside world, for example communicating with another machine over the internet, operations in the program may need to happen in an unpredictable order.

Asynchronous programming example

When a program needs to interact with the outside world, for example communicating with another machine over the internet, operations in the program may need to happen in an unpredictable order.

Say your program needs to download a file.

- We would like to initiate the download operation, perform other operations while we wait for it to complete, and then resume the code that needs the downloaded file when it is available.

Asynchronous programming example

When a program needs to interact with the outside world, for example communicating with another machine over the internet, operations in the program may need to happen in an unpredictable order.

Say your program needs to download a file.

- We would like to initiate the download operation, perform other operations while we wait for it to complete, and then resume the code that needs the downloaded file when it is available.
- This sort of scenario falls in the domain of asynchronous programming, sometimes also referred to as concurrent programming (since, conceptually, multiple things are happening at once).

Asynchronous programming: (concurrent-computing)

To address these scenarios, Julia provides Tasks (also known by several other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations).

When a piece of computing work (in practice, executing a particular function) is designated as a Task, it becomes possible to interrupt it by switching to another Task.

The original Task can later be resumed, at which point it will pick up right where it left off.

This may seem similar to a function call. However there are two key differences:

1. switching tasks does not use any space, so any number of task switches can occur without consuming the call stack.

Asynchronous programming: (concurrent-computing)

To address these scenarios, Julia provides Tasks (also known by several other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations).

When a piece of computing work (in practice, executing a particular function) is designated as a Task, it becomes possible to interrupt it by switching to another Task.

The original Task can later be resumed, at which point it will pick up right where it left off.

This may seem similar to a function call. However there are two key differences:

1. switching tasks does not use any space, so any number of task switches can occur without consuming the call stack.
2. switching among tasks can occur in any order, unlike function calls, where the called function must finish executing before control returns to the calling function.

Multithreading: (multi-processing)

Julia by default runs as a single-threaded application.

However, it can be made to run with a number of threads where the operating system supports it.

(multiple cores)

Julia provides APIs where applications can take advantage from such threads available to them.

Distributed computing:

Goes beyond the realm of a single process.

Two independent processes that cannot normally share any resources among themselves can communicate over a message passing interface designed specifically for Julia.

Packages like `MPI.jl`.

see https://juliaparallel.github.io/MPI.jl/stable/

Julia supports industry standard MPI protocols as well.

Section 2

Asynchronous programming (concurrent-computing)

Asynchronous programming (concurrent-computing)

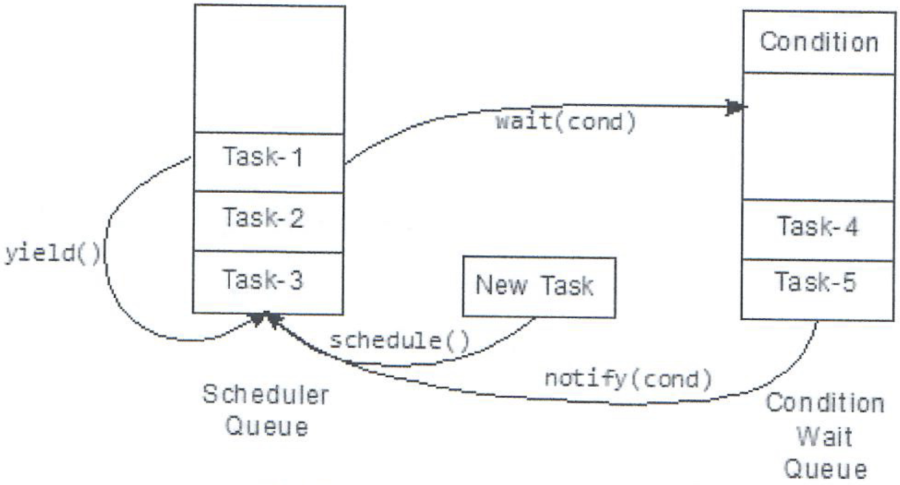

Figure 9.1: Task state transition based on various actions

Tasks:

Julia's asynchronous programming model can be understood from the preceding figure:

1. A new task created is added to the scheduler's runnable task queue by the `schedule` method.

Tasks:

Julia's asynchronous programming model can be understood from the preceding figure:

1. A new task created is added to the scheduler's runnable task queue by the `schedule` method.
2. When a task is executing it can call `yield` and relinquish the control to the scheduler to schedule the next runnable task.

Tasks:

Julia's asynchronous programming model can be understood from the preceding figure:

1. A new task created is added to the scheduler's runnable task queue by the `schedule` method.
2. When a task is executing it can call `yield` and relinquish the control to the scheduler to schedule the next runnable task.
3. A task can wait on a condition and move to the condition's wait queue.

Tasks:

Julia's asynchronous programming model can be understood from the preceding figure:

1. A new task created is added to the scheduler's runnable task queue by the `schedule` method.
2. When a task is executing it can call `yield` and relinquish the control to the scheduler to schedule the next runnable task.
3. A task can wait on a condition and move to the condition's wait queue.
4. When the condition is met, `notify` can be called on the condition and that will move `all` the tasks assigned to the condition to the runnable queue of the scheduler.

Tasks are not functions

They can be switched easily as there is no need to recreate a stack frames as is needed to be carried out for functions.

How the scheduler decides on the task to pick up is dependent on the architecture of the Julia runtime.

For example, a scheduler with the availability of multiple thread execution may be able to pick up multiple tasks for execution.

Similarly, the condition is another system dependent entity.

For example, the condition can be a file being locked, a device being busy or a channel being full or empty.

Tasks:

A Task object can be constructed by passing a function with no parameters as a parameter to its constructor.

The Task object thus created has not run yet and will be put on the scheduler queue when the schedule function is called on it.

```
julia> t = Task() do
           sleep(10)
           println("done")
       end


julia> schedule(t)
```

The prompt returns immediately on the call `schedule(t)` as the call only places the task on the queue. The `println` is carried out after 10s.

```
julia> t = Task() do
           sleep(10)
```

Tasks:

A call to `wait` will ensure the execution shall wait till the task is completed.

`@task x` is a common way to take any arbitrary code and make a Task object out of it by Task(()->x).

```
t = @task begin
    sleep(10)
    println("done")
end
```

An asynchronous task is one where a `Task` is followed by a call to `schedule()` and often represented by macro `@async`.

If a `wait` is called on the task then the call would wait for the task to complete. Such system will be called a synchronous task.

A macro `@sync` can address such a need.

Channels 1/3

Channels are shared memory queues that can help design producer and consmner kinds of data interchange between tasks.

A channel can have a fixed number of data elements of a specific type or of type `Any` if no specific type is specified.

```
julia> c = Channel(8)
Channel{Any}(8) (empty)
```

You can also create a channel with a specific type of data in the queue like an `Int`.

```
julia> c = Channel{Int}(8)
Channel{Int64}(8) (empty)
```

Channels 2/3

Let us use an asynchronous producer to write to the channel.

```
@async begin
    for i = 1:6
        println("Adding $i to channel")
        put!(c, i)
    end
end
```

An asynchronous consumer reads from channel as well.

```
@async begin
        for i = 1:6
            v = take!(c)
            println("Removing value: $v")
        end
    end
```

Channels 3/3

Similarly, as soon as the channel gets emptied the consumer waits.

```
Adding 1 to channel
Adding 2 to channel
Adding 3 to channel
Adding 4 to channel
Removing value: 1
Removing value: 2
Removing value: 3
Removing value: 4
Adding 5 to channel
Adding 6 to channel
Removing value: 5
Removing value: 6
```

Finally, channels can be closed with a `close` call.

Once the channel is closed, no further communication can take place trough them.

Generators & iterators in Python

Python generators are a simple way of creating iterators.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Generators & iterators in Python

Python generators are a simple way of creating iterators.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

- Iterators are the objects that use the `next()` method to get the next value of the sequence.

https://www.codingninjas.com/blog/2021/09/06/iterators-and-generators-in-python/

Generators & iterators in Python

Python generators are a simple way of creating iterators.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

- Iterators are the objects that use the `next()` method to get the next value of the sequence.
- A generator is a function that produces or yields a sequence of values using a `yield` statement.

https://www.codingninjas.com/blog/2021/09/06/iterators-and-generators-in-python/

Generators & iterators in Python

Python generators are a simple way of creating iterators.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

- Iterators are the objects that use the `next()` method to get the next value of the sequence.
- A generator is a function that produces or yields a sequence of values using a `yield` statement.
- Classes are used to implement the iterators

https://www.codingninjas.com/blog/2021/09/06/iterators-and-generators-in-python/

Julia generators & iterators using tasks and channels

Amazing web article (blog)

generators-and-iterators-in-julia-and-python

Julia generators & iterators using tasks and channels

Amazing web article (blog)

generators-and-iterators-in-julia-and-python

Julia's Iteration utilities

Dalla documentazione ufficiale:
https://docs.julialang.org/en/v1/base/iterators/

Locks

Locks are synchronization objects used to synchronize tasks.

If a resource can be accessed by more than one task, it may be best managed under a lock to avoid the state be modified when they are being accessed.

The basic premise of a `lock` is a condition. When the condition is notified then the tasks waiting on it are released.

Locks

Locks are synchronization objects used to synchronize tasks.

If a resource can be accessed by more than one task, it may be best managed under a lock to avoid the state be modified when they are being accessed.

The basic premise of a `lock` is a condition. When the condition is notified then the tasks waiting on it are released.

Typical lock code looks like the following:

```
lock(c)
try
    while !condition_to_wait
        wait(c)
    end
finally
    unlock(c)
end
```

Locks

A smarter way to address this scenario would be to use the `lock ... do` syntax with the `lock(f, cond)` function that automatically includes the `unlock()` call.

```
lock(c) do
    # Access the resource when the lock is active
    work_on_resource()
end
```

Locks are particularly crucial for multi threading scenarios where there is a possibility of simultaneous access of resources by tasks.

Section 3

Multithreading

Multithreading

Julia process starts with a single thread when launched.

However, it can be launched with multiple threads by using the command

`julia -t <no_of_threads>` or
`julia --threads <no_of_threads>`

The same can be achieved by setting the environment variable `JULIA_NUM_THREADS` to the required number of threads.

By starting Julia with `julia -t 4`, the following code provides the number of active threads in the Julia environment.

```
julia> Threads.nthreads()
4
```

Threads.@spawn

`@spawn` will use any available thread and run a task on it.

```
julia> Threads.@spawn for i=1:100
           sleep(1)
           println("Step: $i")
       end
Task (runnable) @0x000000016a73990

julia> Step: 1
Step: 2
Step: 3
Step: 4
...
Step: 100
```

Threads.@threads

The values are printed serially as the task is running on a single thread of the 4 currently available.

If you will like to parallelize the for loop, you could use the Threads.@threads macro.

As can be seen, the printing is now in parallel. One could clearly see the for loop has been split into 4 different threads of execution.

```
julia> Threads.@threads for i=1:100
           sleep(i)
           println("Step: $i")
       end
Step: 26
Step: 51
```

Threads.@threads

You can launch Julia as shown: $ JULIA_NUM_THREADS=4
If you are using IJulia, you could follow the following steps.

```
ENV["JULIA_NUM_THREADS"] = 4
using IJulia
notebook()
```

```
Step: 1
Step: 76
Step: 77
Step: 52
Step: 2
Step: 27
Step: 100
Step: 25
Step: 50
Step: 75
```

Threads

`Conditions()` are thread-safe condition objects and must be used for thread safety.

The second issue is low-level atomic operations.

Some of the LLVM defined low-level atomic operations are supported by Julia.

Lastly, finalizers and garbage collection may get affected with multithreaded programs.

We will suggest the user reviews the manuals on multithreading for a better understanding of the topic:

Threads

`Conditions()` are thread-safe condition objects and must be used for thread safety.

The second issue is low-level atomic operations.

Some of the LLVM defined low-level atomic operations are supported by Julia.

Lastly, finalizers and garbage collection may get affected with multithreaded programs.

We will suggest the user reviews the manuals on multithreading for a better understanding of the topic:

Multi-Threading, Julia Documentation Manuals:

https://docs.julialang.org/en/v1/manual/multi-threading/#man-multithreading

Section 4

Distributed computing

Distributed computing

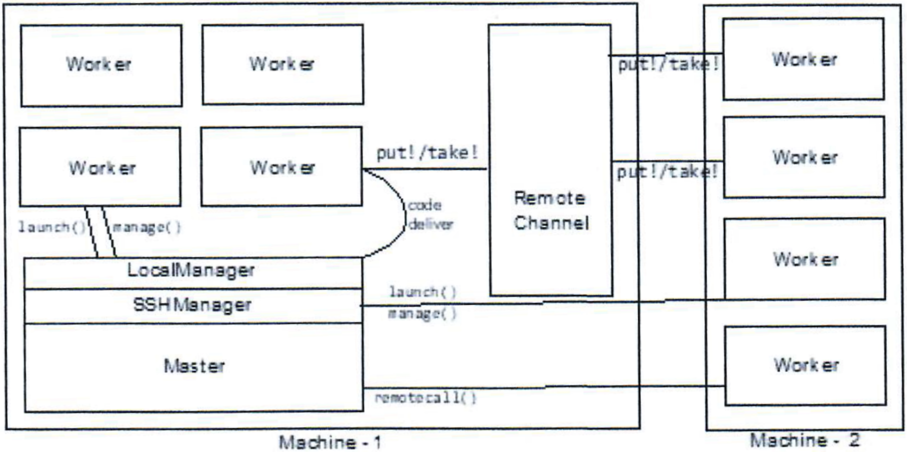

Figure 9.2: Distributed computing architecture in Julia

Distributed computing architecture in Julia

The distributed architecture in Julia has all the similar models discussed n before, related to asynchronous programming as well as multithreading programming.

However, there are certain additional complexities that are due to the distributed architecture.

Distributed computing architecture in Julia

The distributed architecture in Julia has all the similar models discussed n before, related to asynchronous programming as well as multithreading programming.

However, there are certain additional complexities that are due to the distributed architecture.

1. A master process governs the computing infrastructure management.

Distributed computing architecture in Julia

The distributed architecture in Julia has all the similar models discussed n before, related to asynchronous programming as well as multithreading programming.

However, there are certain additional complexities that are due to the distributed architecture.

1. A master process governs the computing infrastructure management.
2. It utilizes a `LocalManager` to launch and manage Julia worker processes where the local load can be executed in the same machine.

Distributed computing architecture in Julia

The distributed architecture in Julia has all the similar models discussed n before, related to asynchronous programming as well as multithreading programming.

However, there are certain additional complexities that are due to the distributed architecture.

1. A master process governs the computing infrastructure management.
2. It utilizes a `LocalManager` to launch and manage Julia worker processes where the local load can be executed in the same machine.
3. There is also a `SSHManager` that can connect to a remote host over an SSL connection and communicate all remote actions to be carried out.

Distributed computing architecture in Julia

The distributed architecture in Julia has all the similar models discussed n before, related to asynchronous programming as well as multithreading programming.

However, there are certain additional complexities that are due to the distributed architecture.

1. A master process governs the computing infrastructure management.
2. It utilizes a `LocalManager` to launch and manage Julia worker processes where the local load can be executed in the same machine.
3. There is also a `SSHManager` that can connect to a remote host over an SSL connection and communicate all remote actions to be carried out.
4. The connection management is kept outside of the architecture and it is presumed all needed security restrictions and protocols are handled by the network layer.

Distributed computing architecture in Julia

The distributed architecture in Julia has all the similar models discussed n before, related to asynchronous programming as well as multithreading programming.

However, there are certain additional complexities that are due to the distributed architecture.

1. A master process governs the computing infrastructure management.
2. It utilizes a `LocalManager` to launch and manage Julia worker processes where the local load can be executed in the same machine.
3. There is also a `SSHManager` that can connect to a remote host over an SSL connection and communicate all remote actions to be carried out.
4. The connection management is kept outside of the architecture and it is presumed all needed security restrictions and protocols are handled by the network layer.
5. A remote worker can be launched by the master process and a new id will be assigned to the worker process. `Id 1` is reserved for the master process.

Distributed computing architecture in Julia

1. Any code to be lanched by a worker is launched by a `remotecall`.

Distributed computing architecture in Julia

1. Any code to be lanched by a worker is launched by a `remotecall`.
2. The `remotecall` may return an object of type `Future` that can be waited on by a `fetch` call to get the output of the remote call.

Distributed computing architecture in Julia

1. Any code to be lanched by a worker is launched by a `remotecall`.
2. The `remotecall` may return an object of type `Future` that can be waited on by a `fetch` call to get the output of the remote call.
3. The other mode of data exchange is a `RemoteChannel` that can be set up in one of the processes while other worker processes can `put!()` data in it or `take!()` data from it.

Distributed computing architecture in Julia

1. Any code to be lanched by a worker is launched by a `remotecall`.
2. The `remotecall` may return an object of type `Future` that can be waited on by a `fetch` call to get the output of the remote call.
3. The other mode of data exchange is a `RemoteChannel` that can be set up in one of the processes while other worker processes can `put!()` data in it or `take!()` data from it.
4. Just like local `Channel` objects they provide an easy way to program publish and subscribe models.

Distributed computing architecture in Julia

1. Any code to be lanched by a worker is launched by a `remotecall`.
2. The `remotecall` may return an object of type `Future` that can be waited on by a `fetch` call to get the output of the remote call.
3. The other mode of data exchange is a `RemoteChannel` that can be set up in one of the processes while other worker processes can `put!()` data in it or `take!()` data from it.
4. Just like local `Channel` objects they provide an easy way to program publish and subscribe models.
5. Many remote connections may not need to exactly know the specifics of the machine where the workload needs to be run.

Distributed computing architecture in Julia

1. Any code to be lanched by a worker is launched by a `remotecall`.
2. The `remotecall` may return an object of type `Future` that can be waited on by a `fetch` call to get the output of the remote call.
3. The other mode of data exchange is a `RemoteChannel` that can be set up in one of the processes while other worker processes can `put!()` data in it or `take!()` data from it.
4. Just like local `Channel` objects they provide an easy way to program publish and subscribe models.
5. Many remote connections may not need to exactly know the specifics of the machine where the workload needs to be run.
6. They use macros like `@spawnat` with `Any` as parameter such that the workload can be launched at any of the machines.

Distributed computing architecture in Julia

1. Any code to be lanched by a worker is launched by a `remotecall`.
2. The `remotecall` may return an object of type `Future` that can be waited on by a `fetch` call to get the output of the remote call.
3. The other mode of data exchange is a `RemoteChannel` that can be set up in one of the processes while other worker processes can `put!()` data in it or `take!()` data from it.
4. Just like local `Channel` objects they provide an easy way to program publish and subscribe models.
5. Many remote connections may not need to exactly know the specifics of the machine where the workload needs to be run.
6. They use macros like `@spawnat` with `Any` as parameter such that the workload can be launched at any of the machines.
7. The `Future` object thus returned will be able to tunnel* through the proper SSH connection to communicate to the correct worker in another physical machine.

Distributed computing architecture in Julia

1. Similarly, the code to be run by all the worker processes must be available to them.

Distributed computing architecture in Julia

1. Similarly, the code to be run by all the worker processes must be available to them.
2. One option may be to deploy it as a package such that the package is available in all the processes.

Distributed computing architecture in Julia

1. Similarly, the code to be run by all the worker processes must be available to them.
2. One option may be to deploy it as a package such that the package is available in all the processes.
3. But the package needs to be selected by `using` in each individual processes separately.

Distributed computing architecture in Julia

1. Similarly, the code to be run by all the worker processes must be available to them.
2. One option may be to deploy it as a package such that the package is available in all the processes.
3. But the package needs to be selected by `using` in each individual processes separately.

Distributed computing architecture in Julia

1. Similarly, the code to be run by all the worker processes must be available to them.
2. One option may be to deploy it as a package such that the package is available in all the processes.
3. But the package needs to be selected by `using` in each individual processes separately.

Multiprocessing and Distributed Computing

However, interested readers are advised to refer to the Julia documentation manual on Multiprocessing and Distributed Computing https://docs.julialang.org/en/v1/manual/distributed-computing/