Guide to Parallel Computing with Julia

Jacob Vaverka

There exists several parallel programming paradigms that unlock the potential of modern computing in different ways. The Julia Programming language provides builtin support for these paradigms, and this article will introduce the concepts, benefits and syntax of the following paradigms:

1. Asynchronous tasks or coroutines,
2. Multi-threading,
3. Distributed computing,
4. GPU Computing.

TERMINOLOGY

- **task:** a unit of work or process to be executed
- **thread:** sequences of instructions that can be executed by a CPU core
- **core:** individual processing unit within a CPU that can perform tasks independently
- **machine:** individual computer with its own hardware resources

---

Asynchronous Tasks or Coroutines

Typically, tasks are performed in sequence or synchronously on a thread. If the thread is running on a single core machine, then this task can be blocking which means the program must finish executing the thread before moving on to other tasks.

Asynchronous tasks, also known as coroutines, targets maximum efficiency within a single thread by splitting tasks into multiple threads and allowing the core to quickly switch between the threads. This means tasks can start and stop without unnecessarily tying up system resources. In other words, coroutines enable non-blocking execution.

Coroutines prove useful when tasks involve event handling, producer-consumer processes, or waiting for I/O operations such as network requests or file operations. For example, say some data transformation takes a long time to complete. This process is approximated by the following function.

```
function long_process()
    sleep(3)
    return 42
end
```

Line 2
simulate the long running transformation

Line 3
return some result

```
long_process (generic function with 1 method)
```

This becomes expensive when there is a large quantity of data to process. For example, the following code block takes a little over 9 seconds to execute because 3 functions are called sequentially and each takes roughly 3 seconds.

```
@elapsed begin
    p1 = long_process()
    p2 = long_process()
    p3 = long_process()
    (p1, p2, p3)
end
```

Lines 2-4
simulate the long running transformation over more data

Line 5
return some result

```
9.004128476
```

The code block above represents typical execution, where actions occur sequentially and result in a sum of the total execution time. Thankfully, this can be reduced to just the longest running process. We acheive this by using the Julia coroutine, Task.

```
@elapsed begin
    t1 = Task(long_process); schedule(t1)
    t2 = Task(long_process); schedule(t2)
    t3 = Task(long_process); schedule(t3)
    (fetch(t1), fetch(t2), fetch(t3))
end
```

Lines 2-4

start the long_process as a coroutine and schedule it to run when resources are available

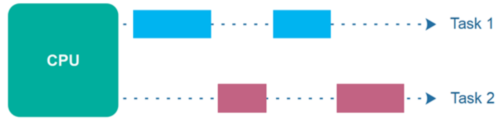
Line 5

retrieve the results from each coroutine once they complete and return the results

```
3.011052091
```

The same result was achieved in just 3 seconds. What happened? Each long_process was started as a separate unit of work, or task. This separation allows the CPU to switch between tasks during execution and make progress on more than one task at a time.

Not only does this approach help make efficient use of available resources but can also be a powerful abstraction to decompose complex problems into more manageable and easy to reason about units of work.



Concurrent

ⓘ Alternative Form: Task Macro

If preferred, the @task macro can be used to acheive the same results.

```
@elapsed begin
    t1 = @task long_process(); schedule(t1)
    t2 = @task long_process(); schedule(t2)
    t3 = @task long_process(); schedule(t3)
    (fetch(t1), fetch(t2), fetch(t3))
end
```

Multi-threading

So far every task has occurred on the same thread - how is this parallel computing? True enough, strictly speaking concurrency is not parallelism[1]. Asynchronous tasks can be very beneficial to your program, but sometimes true parallelism is required. Thankfully, Julia allows Tasks to be scheduled on many threads.

The Threads.@spawn macro can be used to rewrite the long_process example. For each process, create a Task and schedule it to run on any thread once it becomes available.

```
@elapsed begin
    s1 = Threads.@spawn long_process()
    s2 = Threads.@spawn long_process()
    s3 = Threads.@spawn long_process()
    (fetch(s1), fetch(s2), fetch(s3))
end
```

```
3.030350092
```

To prove that Threads.@spawn just creates and schedules tasks[2], see the return type below is indeed a Task.

```
Threads.@spawn sleep(1)
```

```
Task (runnable) @0x00007f05b13fb080
```

Spawning tasks across multiple available threads can be simplified with the Threads.@threads macro. In order to execute a multithreaded for loop, simply prefix the loop with @threads. For example, the resulting code block takes ~3 seconds to complete because each long_process is spawned as a new Task and distributed on an available thread (just as in the @spawn example).

---

[1] Great talk for deep dive Concurrency is not Parallelism by Rob Pike

[2] `Threads.@spawn` gives you a `Task`

```
Threads.@threads for _ in 1:3
    long_process()
end
```

> Setting Up Threads in Julia
>
> Before Julia can schedule tasks on other threads, it must know about the other threads. Add available threads when invoking Julia by passing the --threads flag (or equivalently -t).
>
> Setting Up Threads in VS Code
>
> VS Code users can control the number of threads with one of the following settings in their settings.json:
>
> - `"julia.NumThreads": 4`
> - `"julia.additionalArgs": ["--threads=4"]`

Scheduling

The @threads macro offers further control by assigning the scheduler.

```
Threads.@threads :static for _ in 1:3
    long_process()
end
```

The :static scheduler creates one task per thread. This functionality exists to support older Julia code (pre v1.3) and should generally be discouraged in favor of :dynamic.

```
Threads.@threads :dynamic for _ in 1:3
    long_process()
end
```

The :dynamic scheduler is the default and executes iterations dynamically to available worker threads.

Channels

The @threads macro makes scheduling tasks on available threads easy, but what happened to the task results? How can task results be retrieved when the for-loop does not explicitly assign variables to each Task?

The Channel provides a robust solution to this problem. While the solution is robust, the concept is simple. Think of a channel as a pipe - something is put into one end and taken out of the other. When creating a Channel, its **type** (declared the allowed input) and **size** (declares how many inputs are allowed at once) can be specified. For instance, Channel{String}(12) creates a channel that can hold a dozen string items at one time. When the type is omitted, the channel will allow inputs of type Any. So creating a channel of size 2 and type Any, run the following.

```
ch = Channel(2)
```

```
Channel{Any}(2) (empty)
```

Now, use the put! command to place an item in the channel.

```
put!(ch, "foo")
```

```
"foo"
```

This channel can hold mulitple types of inputs at once.

```
put!(ch, 1_000)
```

```
1000
```

> **Blocking `put!` calls**
>
> Once a channel reaches maximum capacity, an additional put! call will result in a blocking execution. In other words, the program will stop execution until the channel has sufficient space for the new item. For example, before any more items could be put! into ch, at least one item would need to be removed first.

Look at the channel now.

```
ch
```

```
Channel{Any}(2) (2 items available)
```

The channel is at capacity because the number of available items matches its size. The fetch command can be used to retrieve an item. Items will be obtained in a first in, first out (FIFO) manner - the first item put! into the channel will be the first item retrieved.

```
fetch(ch)
```

```
"foo"
```

An item was retrieved from the channel! How many items are still available?

```
ch
```

```
Channel{Any}(2) (2 items available)
```

There are still 2 items available. Often when an item is retrieved from the channel, it is useful to remove it as well. This pattern may continue until all items have been retrieved. Use the take! command to retrieve the next item and remove it from the channel. As indicated by the !, this function mutates the channel by removing the returned value.

```
take!(ch)
```

```
"foo"
```

There should be 1 more item available. We can be sure by using the isempty command.

```
isempty(ch)
```

```
false
```

Okay, the channel is **not** empty. It is safe to take! the next item.

```
take!(ch)
```

```
1000
```

> Blocking `take!` calls
>
> Once a channel is empty, an additional take! call will result in a blocking execution. I.e., the program will stop execution until the channel has another items available for retrieval. So if !isempty(ch) then it is safe to take!.

In addition to Type and size, a Channel can take other inputs, such as:

- a function: creates a new task bound to the channel and scheduled automatically (first argument or specified in a do-block)
- a task reference: if a reference to the created task is needed, then a Ref{Task} object can be passed to the taskref keyword argument
- an option to parallelize: if keyword argument spawn is set to true, then the Task created for the Function may be scheduled on another thread in parallel

See the full type signature:

```
Channel{T=Any}(func::Function, size=0; taskref=nothing, spawn=false)
```

Now see an example of these additional inputs.

```
taskref = Ref{Task}()
ch = Channel{String}(taskref=taskref, spawn=true) do c
```

```
    println(uppercase(take!(c)), "!")
end
```

Line 1

Create a reference to a Task so that we can see its status later

Line 2

Create a Channel of Type String with a task reference which may be spawned in parallel

Line 3

Define a function that is bound to the Channel and scheduled for execution (the function will uppercase any input and append an exclamation point)

```
Channel{String}(0) (empty)
```

Since the Channel is tied to the a Task reference, its status can be obtained from the following functions:

- `istaskstarted`
- `istaskfailed`
- `istaskdone`

At this point, the task is expected to be started, but not failed or done.

```
istaskstarted(taskref[]) && !istaskfailed(taskref[]) &&
!istaskdone(taskref[])

true
```

Given the status above, the Channel should be able to accept an item, execute its bound function (maybe on another available thread in parallel) and close.

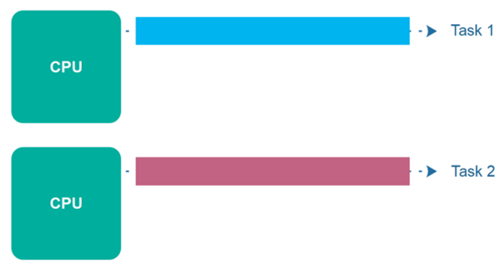
```
put!(ch, "hello");

HELLO!

istaskdone(taskref[])

true

ch

Channel{String}(0) (closed)
```

These mechanisms provide the foundation needed to break down a problem into distinct and manageable units of work (aka tasks) as well as handle results as they become available across all workers.

Distributed Computing

A standard library exists with tools for distributed parallel computing called Distributed.jl. These tools help explore possibilites of executing tasks on separate threads in a pool of workers (where threads may be on different machines).

> "Where concurrency is about dealing with many things at once, parallelism is about doing many things at once"[3]
>
> — Rob Pike



Parallel

Import the Distributed Standard Library.

```
using Distributed
```

To begin, use Distributed to provide basic information. For instance, how many processes are currently available?

```
nprocs()

1
```

How many workers are avilable? This should return one less than the number of processes (unless number of processes is one, then returns one).

```
nworkers()

1
```

Distributed can also add available processes. Launch workers via addprocs.

```
addprocs(4)
nworkers()
```

---

[3] Concurrency vs. Parallelism

```
4
```

Launch Julia with Added Processes

To achieve the same effect when Julia is invoked, pass the --procs=4 flag (equivalently -p 4).

Once a Julia instance is up and running with some workers, how can you put them to use? One example is to distribute a map call in parallel by using pmap. This is a great method to execute a function (first argument) on some collection (second argument) and distribute the work across all available workers in parallel.

```
pmap(x -> x*2, [1, 2, 3])

3-element Vector{Int64}:
 2
 4
 6
```

One thing to watch out for, any error will stop pmap. This can result in the specified function not being applied to all elements of the collection. However, these errors can be handled using the on_error keyword argument.

```
pmap(x -> iseven(x) ? error("foo") : x, 1:4; on_error=identity)

4-element Vector{Any}:
 1
  ErrorException("foo")
 3
  ErrorException("foo")
```

Above, errors are captured when they occur but the pmap call continues on through the entire collection. This functionality extends to handling errors in user a specified manner, such as returning 0 whenever an error occurs.

```
even_or_zero = pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=ex->0)

4-element Vector{Int64}:
 1
 0
 3
 0
```

Defining special behavior to more gracefully handle errors may be advantageous for downstream processing. For instance, say the collection above needs to be squared and then reduced over its sum. This is easily done using the @distributed macro which executes a distributed memory, parallel loop where the reducer (in this case +) is provided just before the for keyword.

```
@distributed (+) for x in even_or_zero
    x^2
end
```

```
10
```

Essentially, @distributed gives us an eaily parallelizable mapreduce where the last line in the body of the loop will be reduced using the specified reducer. If no reducer is given, then the loop will be executed asynchronously by spawing tasks on available workers and returning immediatedly. If the loop should wait for completion then the @sync macro should prefix @distributed.

```
@sync @distributed for _ in 1:4
    # do work
end
```

Composing yet another macro, it is easy to see that indeed these tasks are being executed in parallel in about the same amount of time for a single execution to complete.

```
@elapsed @sync @distributed for _ in 1:4
    sleep(2)
end
```

```
2.240532206
```

Can long_process be used in this fashion?

```
@distributed (+) for _ in 1:3
    long_process()
end
```

Error

```
TaskFailedException

  nested task error: On worker 2:
  UndefVarError: `#long_process` not defined
```

Why did that not work? After all it is almost identical to the prior example that also uses sleep. Well, when the work was assigned to the available workers, not every worker knew about the existence and behavior of the function long_process. Luckily, this can be easily fixed. Whenever workers need access to some code for their Task, it must be made available everywhere. To illustrate this, a new function will be defined with @everywhere.

```
@everywhere function other_long_process()
    sleep(3)
```

```
    return 10
end
```

Now rerun the example using the new function.

```
@distributed (+) for _ in 1:3
    other_long_process()
end

30
```

That worked! Once again (since there are available workers for every tasks) the elapsed time is reduced to roughly the time it takes to complete one execution. The @everywhere macro can also be applied to include and using statements which makes it easy to distribute code across available workers for any job.

Third Party Packages Enabling Distributed Compute

In the scenario where programmers already have dependencies for Third Party libraries, the JuliaParallel GitHub Organization has a collection of packages which provide support for many commonly used libraries such as MPI.jl and Elemental.jl.

GPU Computing

> High-performance GPU programming in a high-level language.[4]

Julia's high-level syntax and powerful compiler make it the perfect interface for productive and performant GPU programming. JuliaGPU is a Github organization created to unify the many packages for programming GPUs in Julia, and it is an excellent resource for learning more about the Julia + GPU landscape in.

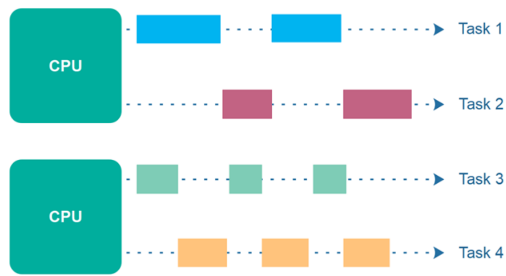
See this gentle introduction to get started.

Conclusion

Parallel computing support in the Julia programming language is special due to its unique combination of high-level expressiveness and efficient performance. Julia's built-in support for parallelism enables programmers to easily harness the power of multiple processors and distributed computing environments. The language features covered

[4] JuliaGPU

here are intuitive abstractions for parallel execution, such as multi-threading, asynchronous programming, and parallel loops. What sets Julia apart is its ability to seamlessly integrate parallelism with its just-in-time (JIT) compilation and dynamic type system. This allows for efficient execution of parallel code without sacrificing the flexibility and ease of use that Julia is known for. With its emphasis on both productivity and performance, Julia's parallel computing support makes it an ideal choice for tackling computationally intensive tasks in a parallel and distributed manner.



Parallel & Concurrent

Next Steps

Now, you have been introduced to all the tools to for concurrent and parallel programming in Julia. As always, the trick is to understand when each tool is the right one for the occasion. We can drastically improve the performance of a program by wisely choosing how to employ threads, tasks, processes and workers. Stay tuned for a future webinar to help explain how to make good programming decisions and effectively use these parallel programming paradigms!

In the meantime, deepen your understanding with the following resources:

- Official documentation on parallel computing
- Multi-Threading Using Julia for Enterprises webinar by Jeff Bezanson
- Announcing composable multi-threaded parallelism in Julia original blog post
- Source code for Parallelizing Data Science webinar by Dr. Elliot Saba