Parallelization

August 10, 2021

1 Parallel Computing

Julia supports: * coroutines (aka green threads) * multithreading (without a Global interpreter lock like CPython) * multiprocessing and distributed computing.

1.1 Coroutines

Here is coroutine version of a `fibonacci()` generator function:

[ ]:
```
function fibonacci(n)
    Channel() do ch
        a, b = 1, 1
        for i in 1:n
            put!(ch, a)
            a, b = b, a + b
        end
    end
end

for f in fibonacci(10)
    println(f)
end
```

`Channel() do ... end` creates a `Channel` object, and spawns an asynchronous `Task` to execute the code in the `do ... end` block.

The task is scheduled to execute immediately, but when it calls the `put!()` function on the channel to yield a value, it blocks until another task calls the `take!()` function to grab the `put!()` value.

`take!()` function is not impleteded explicitly in the code example, since it is executed automatically in the `for` loop, in the main task.

To demonstrate this, we can just call the `take!()` function 10 times to get all the items from the channel:

[ ]:
```
ch = fibonacci(10)
for i in 1:10
    println(take!(ch))
end
```

This channel is bound to the task, therefore it is automatically closed when the task ends.

So if we try to get one more element, we will get an exception:

[ ]:
```julia
try
    take!(ch)
catch ex
    ex
end
```

Here is a more explicit version of the `fibonacci()` function:

[ ]:
```julia
function fibonacci(n)
  function generator_func(ch, n)
    a, b = 1, 1
    for i in 1:n
        put!(ch, a)
        a, b = b, a + b
    end
  end
  ch = Channel()
  task = @task generator_func(ch, n) # creates a task without starting it
  bind(ch, task) # the channel will be closed when the task ends
  schedule(task) # start running the task asynchronously
  ch
end
```

And here is a more explicit version of the `for` loop:

[ ]:
```julia
ch = fibonacci(10)
while isopen(ch)
  value = take!(ch)
  println(value)
end
```

Note that asynchronous tasks (also called "coroutines" or "green threads") are not actually run in parallel: they cooperate to alternate execution.

Some functions, such as `put!()`, `take!()`, and many I/O functions, interrupt the current task's execution, at which point it lets Julia's scheduler decide which task should resume its execution.

1.2 Multithreading

Julia also supports multithreading.

You need to specify the number of available threads upon startup, by setting the `JULIA_NUM_THREADS` environment variable (or setting the `-t` argument).

[ ]:
```julia
if haskey(ENV, "JULIA_NUM_THREADS")
        @info "Number of threads:", ENV["JULIA_NUM_THREADS"]
else
        @warn "No treads; restart julia as `julia -t 8`"
```

```
end
```

The actual number of threads started by Julia may be lower than that, as it is limited to the number of available cores on the machine (thanks to hyperthreading, each physical core may run two threads).

Here is the number of threads that were actually started:

[ ]:
```
Base.Threads.nthreads()
```

Now let us run 10 tasks across these threads:

[ ]:
```
@Base.Threads.threads for i in 1:10
    println("thread #", Base.Threads.threadid(), " is starting task #$i")
    sleep(rand()) # pretend we're actually working
    println("thread #", Base.Threads.threadid(), " is finished")
end
```

Here is a multithreaded version of the `estimate_pi()` function.

Each thread computes part of the sum, and the parts are added at the end:

[ ]:
```
import BenchmarkTools

function parallel_estimate_pi(n)
  s = zeros(Threads.nthreads())
  nt = n ÷ Threads.nthreads()
  @Threads.threads for t in 1:Threads.nthreads()
      for i in (1:nt) .+ nt*(t - 1)
        @inbounds s[t] += (isodd(i) ? -1 : 1) / (2i + 1)
      end
  end
  return 4.0 * (1.0 + sum(s))
end

@BenchmarkTools.btime parallel_estimate_pi(100_000_000)
```

The `@inbounds` macro is an optimization: it tells the Julia compiler not to add any bounds check when accessing the array.

It is safe in this case since the `s` array has one element per thread, and `t` varies from `1` to `Threads.nthreads()`, so there is no risk for `s[t]` to be out of bounds.

Let's compare this with the single-threaded implementation:

[ ]:
```
function estimate_pi(n)
    s = 1.0
    for i in 1:n
        s += (isodd(i) ? -1 : 1) / (2i + 1)
    end
    return 4s
```

```
end

@BenchmarkTools.btime estimate_pi(100_000_000)
```

Julia has a `mapreduce()` function which makes it easy to implement functions like `parallel_estimate_pi()`:

[ ]:
```
function parallel_estimate_pi2(n)
    4.0 * mapreduce(i -> (isodd(i) ? -1 : 1) / (2i + 1), +, 0:n)
end

@BenchmarkTools.btime parallel_estimate_pi2(100_000_000)
```

The `mapreduce()` function is well optimized, so it's about twice faster than `parallel_estimate_pi()`.

You can also spawn a task using `Threads.@spawn`. It will get executed on any one of the running threads (it will not start a new thread):

[ ]:
```
task = Threads.@spawn begin
    println("Thread starting")
    sleep(1)
    println("Thread stopping")
    return 42
end

println("Hello!")

println("The result is: ", fetch(task))
```

The `fetch()` function waits for the thread to finish, and fetches the result. You can also just call `wait()` if you don't need the result.

You can also use channels to synchronize and communicate across tasks, even if they are running across separate threads:

[ ]:
```
ch = Channel()
task1 = Threads.@spawn begin
    for i in 1:5
        sleep(rand())
        put!(ch, i^2)
    end
    println("Finished sending!")
    close(ch)
end

task2 = Threads.@spawn begin
    foreach(v->println("Received $v"), ch)
    println("Finished receiving!")
end
```

```
wait(task2)
```

1.3 Multiprocessing & Distributed Programming

Julia can spawn multiple Julia processes upon startup if you specify the number of processes via the `-p` argument.

You can also spawn extra processes from Julia itself:

[ ]:
```
import Distributed
Distributed.addprocs(4)
Distributed.workers() # array of worker process ids
```

The main process has id 1:

[ ]:
```
Distributed.myid()
```

The `@Distributed.everywhere` macro lets you run any code on all workers:

[ ]:
```
@Distributed.everywhere println("Hi! I'm worker $(Distributed.myid())")
```

You can also execute code on a particular worker by using `@Distributed.spawnat <worker id> <statement>`:

[ ]:
```
@Distributed.spawnat 3 println("Hi! I'm worker $(Distributed.myid())")
```

If you specify `:any` instead of a worker id, Julia chooses the worker for you:

[ ]:
```
@Distributed.spawnat :any println("Hi! I'm worker $(Distributed.myid())")
```

Both `@Distributed.everywhere` and `@Distributed.spawnat` return immediately.

The output of `@Distributed.spawnat` is a `Future` object.

You can call `fetch()` on this object to wait for the result:

[ ]:
```
result = @Distributed.spawnat 3 1+2+3+4
fetch(result)
```

If you import some package in the main process, it is not automatically imported in the workers.

For example, the following code fails because the worker does not know what `pyimport` is:

[ ]:
```
using PyCall

result = @Distributed.spawnat 4 (np = pyimport("numpy"); np.log(10))

try
    fetch(result)
catch ex
    ex
end
```

You must use `@Distributed.everywhere` or `@Distributed.spawnat` to be able to use `using` of packages you need in each worker:

```
[ ]: @Distributed.everywhere using PyCall

     result = @Distributed.spawnat 4 (np = pyimport("numpy"); np.log(10))

     fetch(result)
```

Or simple you can use `import` which imports packages automatically to all workers.

```
[ ]: import PyCall

     result = @Distributed.spawnat 4 (np = PyCall.pyimport("numpy"); np.log(10))

     fetch(result)
```

Similarly, if you define a function in the main process, it is not automatically available in the workers. You must define the function in every worker:

```
[ ]: @Distributed.everywhere addtwo(n) = n + 2

     result = @Distributed.spawnat 4 addtwo(40)

     fetch(result)
```

You can pass a `Future` to `@Distributed.everywhere` or `@Distributed.spawnat`, as long as you wrap it in a `fetch()` function:

```
[ ]: M = @Distributed.spawnat 2 rand(5)

     result = @Distributed.spawnat 3 fetch(M) .* 10.0

     fetch(result)
```

In this example, worker 2 creates a random array, then worker 3 fetches this array and multiplies each element by 10, then the main process fetches the result and displays it.