Parallel computing with Julia

Raoul Lefmann
raoul.lefmann@tu-dortmund.com
Overview

1. Introduction
2. Managing workers
3. Communication between workers
4. Parallel loops and maps
5. Synchronization
6. Distributed Arrays
Julia is designed from the start having parallelization in mind. It implements the actor pattern, but on a higher level than sending and receiving messages. Processes are called *workers*.

**Figure:** a distributed memory system
Managing workers

- create $n$ workers at start of Julia session using the `-p` option
  \[
  \text{julia} \ -p \ n
  \]
- view number of workers + master process
  \[
  \text{nprocs()}
  \]
- create $n$ workers during a session
  \[
  \text{addprocs(n)}
  \]
- number of local cores on your machine (including virtual or hyperthreading cores)
  \[
  \text{CPU\_CORES}
  \]
workers are addressed by numbers (PIDs)
  - master process has PID 1
  - PIDs of workers

workers()

a process can get its PID by calling

myid()

remove workers

rmprocs(pid1,pid2,...)
Communication between workers

- **remote call**: requests a worker to execute a function with given arguments

  \[
  \text{rref} = \text{remotecall}(\text{pid}, f, \text{arg}1, \text{arg}2, \ldots)
  \]

- creates a *remote reference*
  - object of type `RemoteRef`
  - reference to the result once it is computed by the worker

- check if the calculation associated with a remote reference is completed

  \[
  \text{isready}(\text{rref})
  \]

- block the calling process until the calculation associated with a remote reference is completed and return the result

  \[
  \text{result} = \text{fetch}(\text{rref})
  \]
Example (square root)

```python
rref = remotecall(2, sqrt, 9)
result = fetch(rref)
```

Example (process id)

```python
for w in workers():
    rref = remotecall(w, myid)
    println(fetch(rref))
end
```

Communication between workers
just wait for completion, but don’t return the result

```julia
wait(rref)
```

immediately get the result of a remote call

```julia
remotecall_fetch(pid,f, arg1, arg2, ...)
```

more efficient then remotecall and then fetch

the `@spawnat` macro is easier to read than the remotecall syntax

```julia
remotecall(2,sqrt,100)
@spawnat 2 sqrt(100)
```

the `@spawn` macro lets Julia decide on which worker the calculation is performed

```julia
@spawn sqrt(100)
```
The @everywhere macro

- we can use a list comprehension to execute a function on each worker

```julia
rrefs = [@spawnat pid myid() for pid in workers()]
results = [fetch(rref) for rref in rrefs]
```

- instead we can use the @everywhere macro

```julia
@everywhere println(myid())
```

- can be used to generate a global variable

```julia
@everywhere x=5
```

- make program code available to all processes

```julia
@everywhere include("code.jl")
```
Structure of a parallel Julia program

**driver.jl**  Main process. Manages parallel execution and collects the result.

```julia
addprocs(CPU_CORES)
@everywhere include("functions.jl")
```

**functions.jl**  The function definitions that should be available to all workers.

Communication between workers
Parallel loops

- Use the `@parallel` macro

```
@parallel (reduction_operator) for i=1:n
  ...
end
```

- The operations inside the for loop must be independent of each other, because they will run in arbitrary order.

- Variables inside the loop get send to each worker.

**Example (coin tosses)**

```
@parallel (+) for i=1:1000
  rand(Bool)
end
```

Parallel loops and maps
Each Julia worker has its own memory, so the following code will not work as expected

```julia
a = zeros(10^8)
@parallel for i=1:10^8
    a[i] = 1
end
println(a)
```

Instead we can use a *shared array*

```julia
a = SharedArray(Float64, 10^8)
```

which is available to each worker
Parallel maps

- method for applying an operation to all elements in a collection (distributed among workers)

\[ \text{pmap}(f, \text{coll}) \]

- requires less time and memory than sequential manipulation

Example (squaring array entries)

\[
\begin{align*}
\text{arr} &= \text{collect}(1:100) \\
\text{pmap}(x \to x^2, \text{arr})
\end{align*}
\]
Synchronization

@async allows to compute code asynchronously. Wraps the code into a task. Similar to @spawn, but runs on the local processor.

@sync blocks enclose a number of @async, @spawn or @parallel calls. Execution waits at the end of the block for the completion of every statement inside.

```julia
@sync begin
    ...
    @async ...
    @async ...
end
```
Tasks

- functions whose calculation could be halted in between
- similar to yield keyword in Python
- task produces values and the code from which is called consumes them

```python
function numbers()
    i = 0
    while true
        produce(i)
        i += 1
    end
end
tsk = Task(numbers)
while true
    consume(tsk)
end
```
Distributed Arrays

Pkg.add("DistributedArrays")
@everywhere using DistributedArrays

- large-scale calculations often involve large arrays
- distribute arrays among several processes
- idea: communication via array indexing
- syntax is the same as for regular arrays, e.g.

  \[
  \text{dzeros}(100,100) \\
  \text{drandn}(10,10) \\
  \text{dfill}(x,10,10)
  \]

- often makes sense to limit the workers the array is distributed to, e.g.

  \[
  \text{dzeros}((100,100), [2,4])
  \]
distribute a local array

```python
a = fill(42, 10, 10)
distribute(a)
```

distributed arrays are different from shared arrays. Each worker can only access its local part.

per default the array is split equally along each dimension. To find the splits:

```python
a.indexes
a.pids
```

find the part of the shared array a worker can access:

```python
localpart(a)
localindexes(a)
```
- Some functions like `sum` or `minimum` work directly on distributed arrays.
- Others like `rank` need to be implemented by operations on the local parts.

**Example (maximum of a distributed array)**

```julia
@parallel (max) for w in workers()
    rref = @spawnat w maximum(localpart(a))
    fetch(rref)
end
```
References

- Ivo Balbaert - Getting Started with Julia Programming
- Julia Language documentation - parallel Computing
- Github - JuliaParallel/DistributedArrays
- Youtube - Parallel and Distributed Computing with Julia (Jeff Bezanson)
- Marc Moreno Maza - Parallel and Distributed Computing with Julia