Parallel & Distributed Computing: Lecture 16

Alberto Paoluzzi

November 28, 2017

Julia Parallel programming in Julia 2 — Summary

From Julia Manual

1. Built-in support for clusters
2. Data Movement
3. Shared arrays
4. Parallel Map and Loops

Built-in support for clusters

Process identifiers

- Each process has an associated identifier

Process identifiers

- Each process has an associated identifier
- The process providing the interactive Julia prompt always has an id=1

Process identifiers

- Each process has an associated identifier
- The process providing the interactive Julia prompt always has an id=1
- The processes used by default for parallel operations are referred to as "workers"

Process identifiers

- Each process has an associated identifier
- The process providing the interactive Julia prompt always has an id=1
- The processes used by default for parallel operations are referred to as "workers"
- When there is only one process, process 1 is considered a worker.

Process identifiers

- Each process has an associated identifier
- The process providing the interactive Julia prompt always has an id=1
- The processes used by default for parallel operations are referred to as "workers"
- When there is only one process, process 1 is considered a worker.
- Otherwise, workers are considered to be all processes other than process 1

Code Availability

```
$ ./julia -p 4
```

Most commonly you will be loading code from files or packages, and you have a considerable amount of flexibility in controlling which processes load code.

Code Availability

Consider a file, `largrid.jl` we want to execute

Look at the Repo https://github.com/cvdlab/larlib-literate

and suppose to work with the local copy

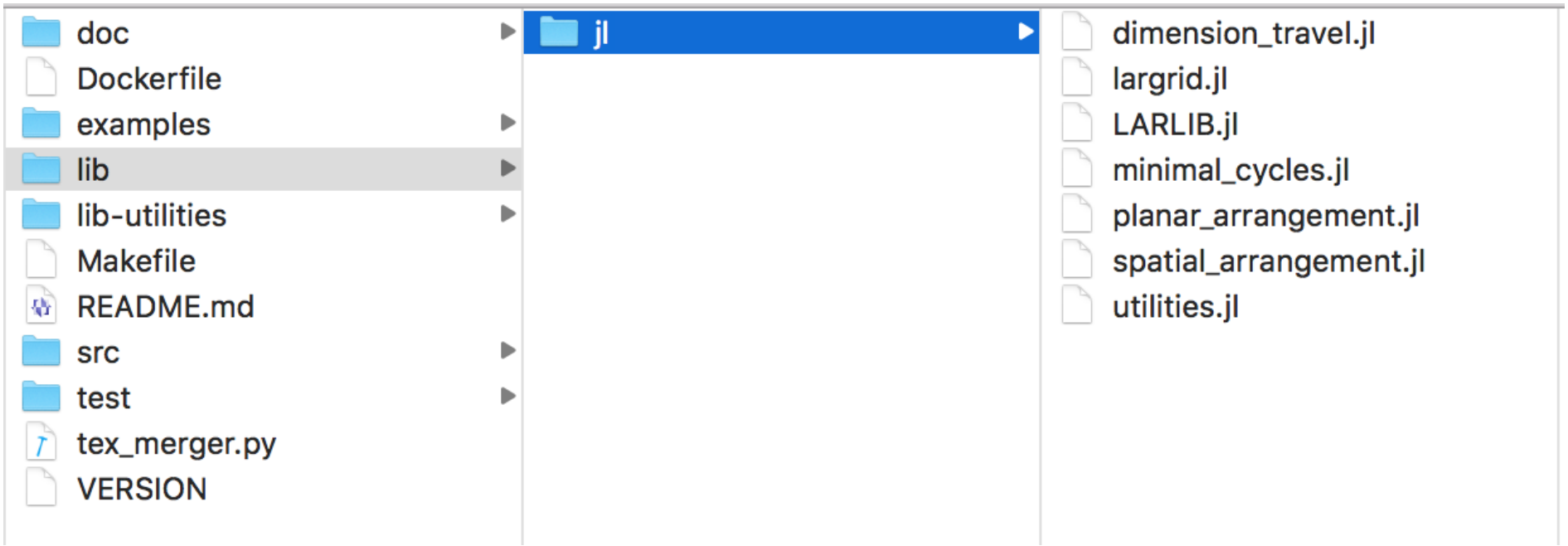

Figure 1: Local repository of `larlib-literate`

aaaaa

- `include("lib/jl/largrid.jl")` loads the file on just a single process (whichever one executes the statement)

```
@everywhere include("./lib/jl/largrid.jl")
```

aaaaa

- `include("lib/jl/largrid.jl")` loads the file on just a single process (whichever one executes the statement)
- `using LARGRID` causes the module to be loaded on all processes; however, the module is brought into scope only on the one executing the statement.

```
@everywhere include("./lib/jl/largrid.jl")
```

aaaaa

- `include("lib/jl/largrid.jl")` loads the file on just a single process (whichever one executes the statement)
- `using LARGRID` causes the module to be loaded on all processes; however, the module is brought into scope only on the one executing the statement.
- You can force a command to run on all processes using the `@everywhere` macro

```
@everywhere include("./lib/jl/largrid.jl")
```

Preloading at start-up

A file can also be preloaded on multiple processes at startup, and a driver script can be used to drive the computation:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

The Julia process running the driver script in the example above has an id=1, just like a process providing an interactive prompt.

Built-in support for two types of clusters

The base Julia installation has in-built support for two types of clusters:

- A local cluster specified with the -p option as shown above.

This uses a passwordless ssh login to start Julia worker processes (from the same path as the current host) on the specified machines.

Built-in support for two types of clusters

The base Julia installation has in-built support for two types of clusters:

- A local cluster specified with the -p option as shown above.
- A cluster spanning machines using the ‘–machinefile“ option.

This uses a passwordless ssh login to start Julia worker processes (from the same path as the current host) on the specified machines.

Cluster computing: `_addprocs()`, `rmprocs()`, `workers()_`

Functions

- addprocs(),

are available as a programmatic means of adding, removing and querying the processes in a cluster.

Cluster computing: `_addprocs()`, `rmprocs()`, `workers()_`

Functions

- addprocs(),
- rmprocs(),

are available as a programmatic means of adding, removing and querying the processes in a cluster.

Cluster computing: `_addprocs()`, `rmprocs()`, `workers()_`

Functions

- addprocs(),
- rmprocs(),
- workers(),

are available as a programmatic means of adding, removing and querying the processes in a cluster.

Cluster computing: `_addprocs()`, `rmprocs()`, `workers()_`

Functions

- addprocs(),
- rmprocs(),
- workers(),
- and others

are available as a programmatic means of adding, removing and querying the processes in a cluster.

Cluster computing

Note that workers do not run a .juliarc.jl startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

Other types of clusters can be supported by writing your own custom ClusterManager, as described in the ClusterManagers section of the manual.

Data Movement

bbbbbbbb

- Sending messages and moving data constitute most of the overhead in a parallel program.

bbbbbbbb

- Sending messages and moving data constitute most of the overhead in a parallel program.
- Reducing the number of messages and the amount of data sent **is critical* to achieving performance and scalability.

bbbbbbbb

- Sending messages and moving data constitute most of the overhead in a parallel program.
- Reducing the number of messages and the amount of data sent **is critical* to achieving performance and scalability.
- It is important to understand the data movement performed by Julia's various parallel programming constructs.

Explicit and implicit data movement

- `fetch()` can be considered an explicit data movement operation,

since it directly asks that an object be moved to the local machine.

but this is not as obvious, hence it can be called an implicit data movement operation.

Explicit and implicit data movement

- `fetch()` can be considered an explicit data movement operation,

since it directly asks that an object be moved to the local machine.

- `@spawn` (and a few related constructs) also moves data,

but this is not as obvious, hence it can be called an implicit data movement operation.

Explicit and implicit data movement: Example

Method 1:

```
julia> A = rand(1000,1000);
julia> Bref = @spawn A^2;
julia> fetch(Bref);
```

Explicit and implicit data movement: Example

Method 1:

```
julia> A = rand(1000,1000);
julia> Bref = @spawn A^2;
julia> fetch(Bref);
```

Method 2:

```
julia> Bref = @spawn rand(1000,1000)^2;
julia> fetch(Bref);
```

Conclusion

In this toy example, the two methods are easy to distinguish and choose from.

However, in a real program designing data movement might require more thought and likely some measurement

Global variables

```
A = rand(10,10)
remotecall_fetch(()->foo(A), 2)
```

Note that `A` is a global variable defined in the local workspace.

Worker 2 does not have a variable called A under Main.

The act of shipping the closure ()->foo(A) to worker 2 results in `Main.A` being defined on 2.

`Main.A` continues to exist on worker 2 even after the call `remotecall_fetch` returns.

Global variables

Remote calls with embedded global references (under Main module only) manage globals as follows:

- New global bindings are created on destination workers if they are referenced as part of a remote call.

Global variables

Remote calls with embedded global references (under Main module only) manage globals as follows:

- New global bindings are created on destination workers if they are referenced as part of a remote call.
- Global constants are declared as constants on remote nodes too.

Global variables

Remote calls with embedded global references (under Main module only) manage globals as follows:

- New global bindings are created on destination workers if they are referenced as part of a remote call.
- Global constants are declared as constants on remote nodes too.
- Globals are re-sent to a destination worker only in the context of a remote call, and then only if its value has changed. Also, the cluster does not synchronize global bindings across nodes.

Global variables

For example:

```
A = rand(10,10)
remotecall_fetch(()->foo(A), 2) # worker 2
A = rand(10,10)
remotecall_fetch(()->foo(A), 3) # worker 3
A = nothing
```

Executing the above snippet results in `Main.A` on worker 2 having a different value from `Main.A` on worker 3, while the value of `Main.A` on node 1 is set to nothing.

Global variables

As you may have realized:

- while memory associated with globals may be collected when they are reassigned on the master,

This will release any memory associated with them as part of a regular garbage collection cycle.

Global variables

As you may have realized:

- while memory associated with globals may be collected when they are reassigned on the master,
- no such action is taken on the workers as the bindings continue to be valid.

This will release any memory associated with them as part of a regular garbage collection cycle.

Global variables

As you may have realized:

- while memory associated with globals may be collected when they are reassigned on the master,
- no such action is taken on the workers as the bindings continue to be valid.
- `clear!` can be used to manually reassign specific globals on remote nodes to nothing once they are no longer required.

This will release any memory associated with them as part of a regular garbage collection cycle.

Global variables: example

comportamento differente per Julia 5.x and Julia 6.x

```
A = rand(10,10);

remotecall_fetch(()->A, 2);

B = rand(10,10);

let B = B
   remotecall_fetch(()->B, 2)
end;

@spawnat 2 whos();
```

The constructs introducing scope blocks are:

Scope name	block/construct introducing this kind of scope
Global Scope	module, baremodule, at interactive prompt (REPL)
Local Scope	Soft Local Scope: for, while, comprehensions, try-catch-finally, let
Local Scope	Hard Local Scope: functions (either syntax, anonymous & do-blocks), struct, macro

Figure 2: Kind of scope

Scope of Variables

The scope of a variable is the region of code within which a variable is visible. Variable scoping helps avoid variable naming conflicts.

Global Scope

Each module introduces a new global scope, separate from the global scope of all other modules; there is no all-encompassing global scope.

Local Scope

A new local scope is introduced by most code-blocks,

A local scope usually inherits all the variables from its parent scope, both for reading and writing.

There are two subtypes of local scopes, hard and soft, with slightly different rules concerning what variables are inherited.

Shared arrays

Shared arrays

Shared Arrays use system shared memory to map the same array across many processes.

While there are some similarities to a `DArray`, the behavior of a `SharedArray` is quite different.

In a `DArray`, each process has local access to just a chunk of the data, and no two processes share the same chunk;

in contrast, in a `SharedArray` each "participating" process has access to the entire array.

A `SharedArray` is a good choice when you want to have a large amount of data jointly accessible to two or more processes on the same machine.

Shared arrays

The constructor for a shared array is of the form:

```
SharedArray{T,N}(dims::NTuple; init=false, pids=Int[])
```

which creates an N-dimensional shared array of a bits type `T` and size `dims` across the processes specified by `pids`.

Unlike distributed arrays, a shared array is accessible only from those participating workers specified by the `pids` named argument (and the creating process too, if it is on the same host).

Shared arrays

EXAMPLE

Shared arrays

Parallel Map and Loops

bbbbbbbb

aaaa