Parallel & Distributed Computing: Lecture 15

Alberto Paoluzzi

November 23, 2017

Julia Parallel Computing 1 — Summary

1. Computational infrastructure
2. Parallel programming in Julia
3. Built-in support for clusters
4. Data Movement
5. Parallel Map and Loops

Computational infrastructure

Distribution of student accounts

SSH to enter:

$ `ssh username@tesla2.inf.uniroma3.it`

client ssh for windows

https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

Remarks

Available softwares on path (including Julia)
To move files, use `scp` (secure copy — by now, next OpenAFS)
Password changed at first access

Parallel programming in Julia

multiprocessing environment

Two main factors influence performance:

- speed of CPUs

multiprocessing environment

Two main factors influence performance:

- speed of CPUs
- speed of their access to memory

multiprocessing environment

Two main factors influence performance:

- speed of CPUs
- speed of their access to memory

multiprocessing environment

Two main factors influence performance:

- speed of CPUs
- speed of their access to memory

In a cluster, it's obvious that a given CPU will have fastest access to the RAM within the same computer (node).

Similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache.

multiprocessing environment

Two main factors influence performance:

- speed of CPUs
- speed of their access to memory

In a cluster, it's obvious that a given CPU will have fastest access to the RAM within the same computer (node).

Similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache.

Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

two primitives: remote references and remote calls

Remote reference

is an object that can be used from any process to refer to an object stored on a particular process.

Remote call

is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

- A remote call returns a remote reference to its result.

two primitives: remote references and remote calls

Remote reference

is an object that can be used from any process to refer to an object stored on a particular process.

Remote call

is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

- A remote call returns a remote reference to its result.
- Remote calls return immediately;

two primitives: remote references and remote calls

Remote reference

is an object that can be used from any process to refer to an object stored on a particular process.

Remote call

is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

- A remote call returns a remote reference to its result.
- Remote calls return immediately;
- the process proceeds to its next operation while the remote call happens somewhere else.

functions `wait()`, `fetch()`, `put!()`

- You can wait for a remote call to finish by calling `wait()` on its remote reference,

functions `wait()`, `fetch()`, `put!()`

- You can wait for a remote call to finish by calling `wait()` on its remote reference,
- you can obtain the full value of the result using `fetch()`.

functions `wait()`, `fetch()`, `put!()`

- You can wait for a remote call to finish by calling `wait()` on its remote reference,
- you can obtain the full value of the result using `fetch()`.
- You can store a value to a remote reference using `put!()`.

Example

```
julia> r = remotecall(2, rand, 2, 2)
RemoteRef(2,1,5)
```

Example

```
julia> r = remotecall(2, rand, 2, 2)
RemoteRef(2,1,5)
```

- The first argument to `remotecall()` is the index of the process that will do the work

Example

```
julia> r = remotecall(2, rand, 2, 2)
RemoteRef(2,1,5)
```

- The first argument to `remotecall()` is the index of the process that will do the work
- the second argument to `remotecall()` is the function to call

Example

```
julia> r = remotecall(2, rand, 2, 2)
RemoteRef(2,1,5)
```

- The first argument to `remotecall()` is the index of the process that will do the work
- the second argument to `remotecall()` is the function to call
- the remaining arguments will be passed to this function

Example

```
julia> fetch(r)
2x2 Float64 Array:
 0.60401   0.501111
 0.174572  0.157411
```

```
julia> s = @spawnat 2 1 .+ fetch(r)
RemoteRef(2,1,7)
```

The `@spawnat` macro evaluates the expression in the second argument on the process specified by the first argument.

The result of both calculations is available in the two remote references, `r` and `s`

```
julia> fetch(s)
2x2 Float64 Array:
 1.60401  1.50111
 1.17457  1.15741
```

function `remotecall_fetch()`

If you want a remotely-computed value immediately.

This typically happens when you read from a remote object to obtain data needed by the next local operation.

The function `remotecall_fetch()` exists for this purpose.

It is equivalent to `fetch(remotecall(...))` but is more efficient.

```
julia> remotecall_fetch(2, getindex, r, 1, 1)
0.10824216411304866
```

The macro @spawn

The syntax of `remotecall()` is not especially convenient.

The macro `@spawn` makes things easier.

It operates on an expression rather than a function, and picks where to do the operation for you:

```
julia> r = @spawn rand(2,2)
RemoteRef(1,1,0)

julia> s = @spawn 1 .+ fetch(r)
RemoteRef(1,1,1)

julia> fetch(s)
1.10824216411304866 1.13798233877923116
1.12376292706355074 1.18750497916607167
```

The macro `@spawn`

```
julia> r = @spawn rand(2,2)
RemoteRef(1,1,0)

julia> s = @spawn 1 .+ fetch(r)
RemoteRef(1,1,1)

julia> fetch(s)
1.10824216411304866 1.13798233877923116
1.12376292706355074 1.18750497916607167
```

Note that we used `1 .+ fetch(r)` instead of `1 .+ r`.

This is because we do not know where the code will run, so in general a `fetch()` might be required to move `r` to the process doing the addition.

In this case, `@spawn` is smart enough to perform the computation on the process that owns `r`, so the `fetch()` will be a no-op.

Built-in support for clusters

bbbbbbbb

```
aaaa
```

Data Movement

bbbbbbbb

```
aaaa
```

Parallel Map and Loops

bbbbbbbb

`aaaa`