1 Multiprocessing in Julia

- Add workers using `addprocs`
- Start a function as a Task on an available thread using `remotecall`.
- `fetch(task)` wait for the completion of the task and retrieve result
- `remotecall_fetch` does `remotecall` and `fetch`

```julia
using Distributed
using LinearAlgebra
using BenchmarkTools

addprocs(4)
```

[1]: 4-element Array{Int64,1}:
    2
    3
    4
    5

We can also do `addprocs([((hostname,n)])` to work on different hosts

List of workers

```julia
workers()
```

[2]: 4-element Array{Int64,1}:
    2
    3
    4
    5

Run a function on all threads

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

[3]:
Run a function on another worker and return its id multiplied by 10

```julia
[4]: @everywhere function run_on()
    return myid()*10
end
remotecall_fetch(run_on, 3)

[4]: 30
```

Distributed Arrays allow to distribute data to all workers

```julia
[5]: @everywhere using DistributedArrays

Now let us try to calculate a scalar product

Scalar product for two arrays

```julia
[6]: @everywhere function mydot(A::Array, B::Array)
    result=0.0
    @inbounds @fastmath for i=1:length(A)
        result+=A[i]*B[i]
    end
    return result
end
```

Scalar product for two distributed arrays

This uses an asynchronous map, where results are collected as they come in

```julia
[7]: function mydot(DA::DArray, DB::DArray)
    results=asyncmap(p->remotecall_fetch((DA, DB) -> mydot(localpart(DA),
                                           localpart(DB)), p, DA, DB), workers())
    reduce(+, results)
end
```

```julia
[7]: mydot (generic function with 2 methods)
```

```julia
[8]: A=rand(1_000_000)
    B=rand(1_000_000)
    DA=distribute(A)
    DB=distribute(B);
    res_s=@btime mydot($A,$B)
```
res_p = @btime mydot($DA,$DB)
res_s  res_p

392.439 s (0 allocations: 0 bytes)
428.211 s (418 allocations: 17.58 KiB)

[8]: true

- Due to communication and data distribution overhead, this is more efficient for coarser grained parallelism

---

*This notebook was generated using *Literate.jl.*