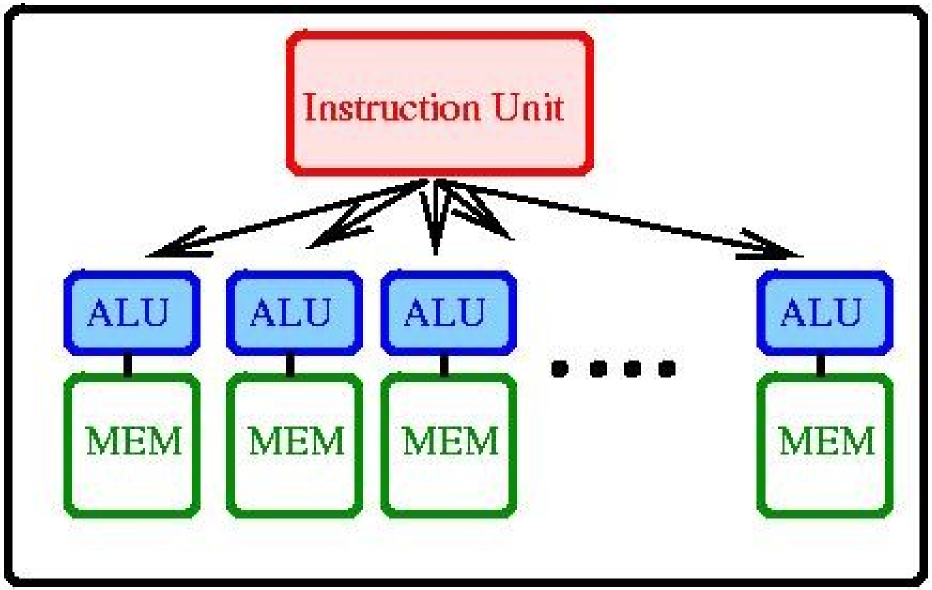
Parallel Computing in Julia

Yang Xu

Department of Automatic Control
Lund University

Background

- Most computers possess more than one CPU
- Two major factors that influence performance: CPU speed, speed of access to memory
- Parallel computing



Parallel computing

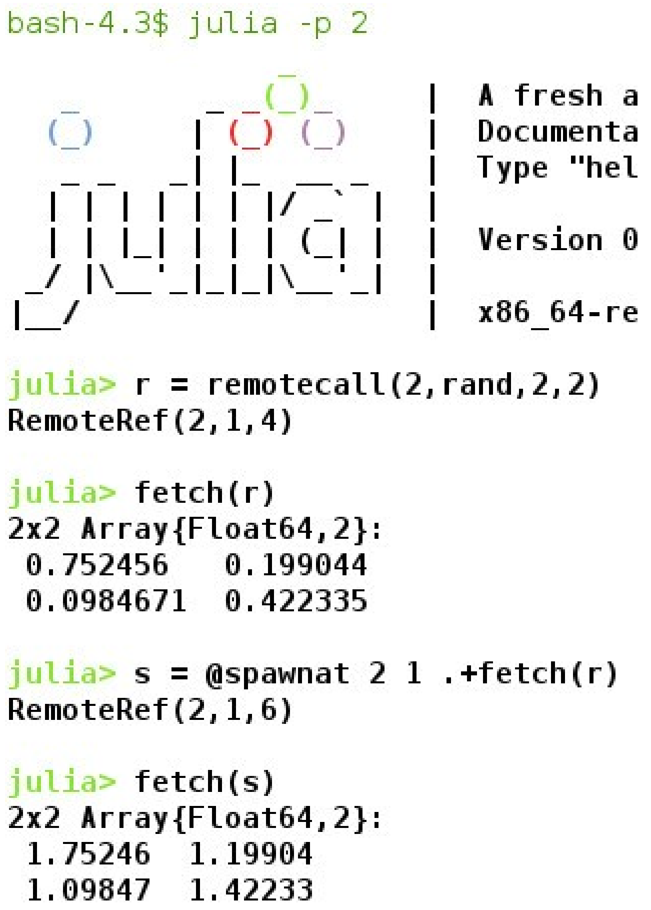
- Two primitives:

  remote reference: an object referring to an object stored on another process

  remote call: a request calling a function on another process
- *`wait()`*: wait for a remote call to finish
- *`fetch()`*: obtain full value of the result
- *`put!()`*: store a value to a remote reference

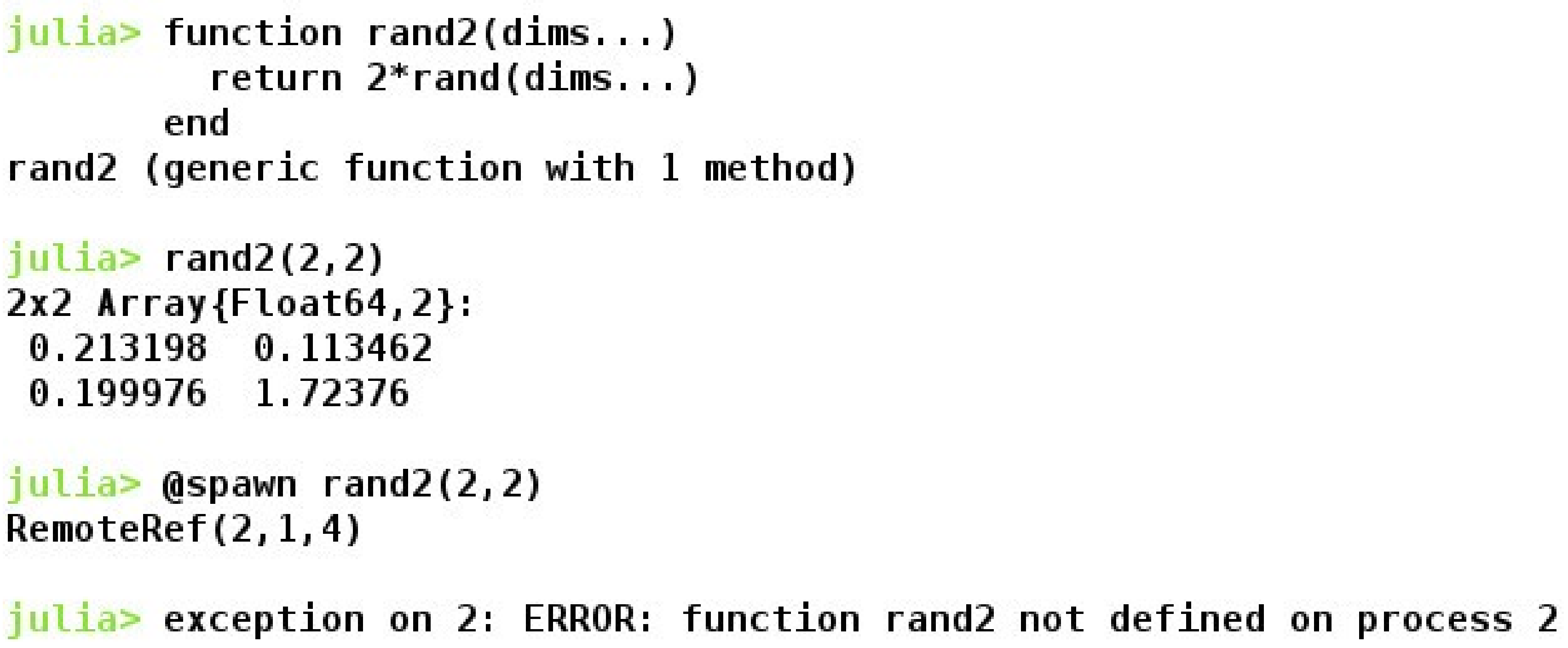
Parallel computing

- *Remotecall()*: low-level interface providing finer control
- *@spawnat*: evaluates the expression on a specified process
- *remotecall_fetch()*: a more efficient version of *fetch(remotecall())*
- *@spawn*: Execute an expression on an randomly-chosen process



```
bash-4.3$ julia -p 2

               _
   _       _ _(_)_     |  A fresh a
  (_)     | (_) (_)    |  Documenta
   _ _   _| |_  __ _   |  Type "hel
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0
 _/ |\__'_|_|_|\__'_|  |
|__/                   |  x86_64-re

julia> r = remotecall(2,rand,2,2)
RemoteRef(2,1,4)

julia> fetch(r)
2x2 Array{Float64,2}:
 0.752456   0.199044
 0.0984671  0.422335

julia> s = @spawnat 2 1 .+fetch(r)
RemoteRef(2,1,6)

julia> fetch(s)
2x2 Array{Float64,2}:
 1.75246  1.19904
 1.09847  1.42233
```

Code availability

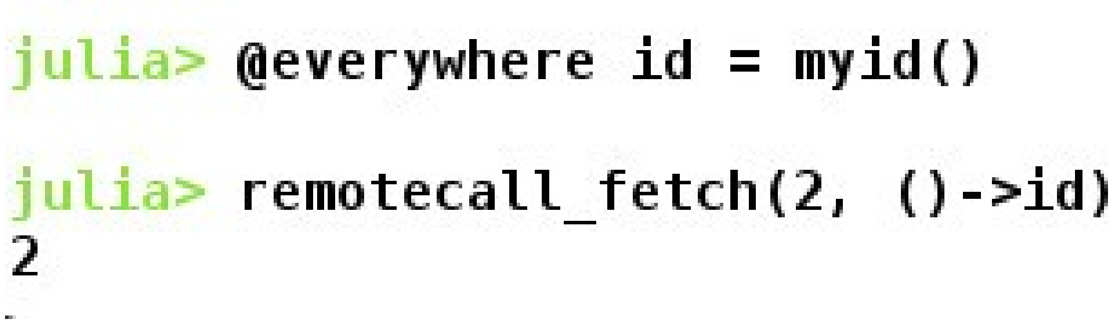


```
julia> function rand2(dims...)
         return 2*rand(dims...)
       end
rand2 (generic function with 1 method)

julia> rand2(2,2)
2x2 Array{Float64,2}:
 0.213198  0.113462
 0.199976  1.72376

julia> @spawn rand2(2,2)
RemoteRef(2,1,4)

julia> exception on 2: ERROR: function rand2 not defined on process 2
```

- Process 1 knew about the function `rand2`, but process 2 did not.
- How do we solve it?

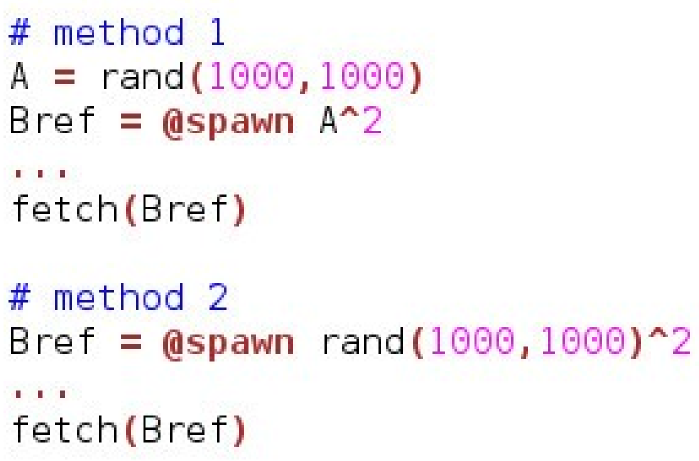
Solution

- *@everywhere*



```
julia> @everywhere id = myid()

julia> remotecall_fetch(2, ()->id)
2
```

Data movement



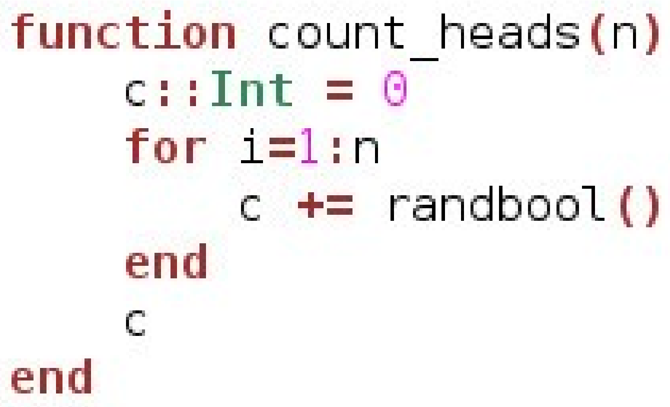
```
# method 1
A = rand(1000,1000)
Bref = @spawn A^2
...
fetch(Bref)

# method 2
Bref = @spawn rand(1000,1000)^2
...
fetch(Bref)
```

- Sending messages and moving data constitute most of the overhead in a parallel program.
- Method 2 sends much less data than method 1, and hence saves time.

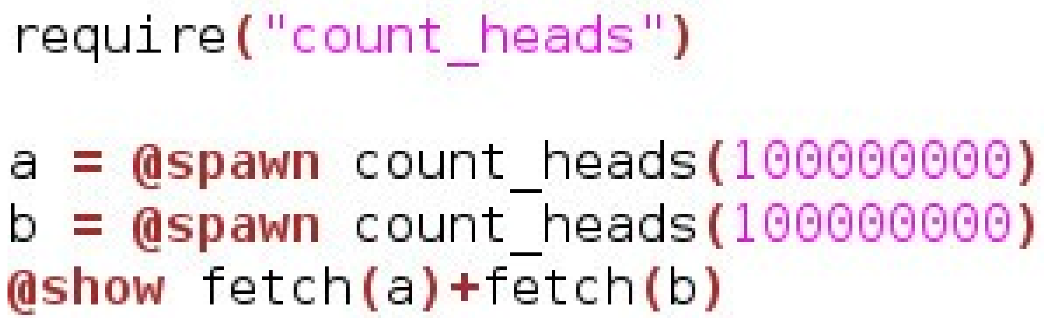
A Monte Carlo simulation

- Flip coins on two processes
- This computation does not require data movement
- Multiple processes can handle independent simulation trials simultaneously
- Method 1: *@spawn*
- Method 2: Parallel loop

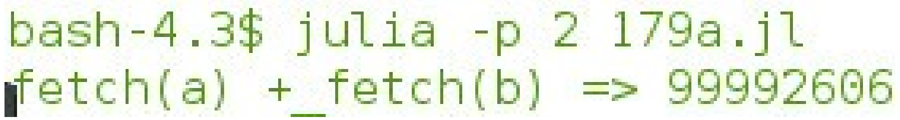
@spawn



```
function count_heads(n)
    c::Int = 0
    for i=1:n
        c += randbool()
    end
    c
end
```

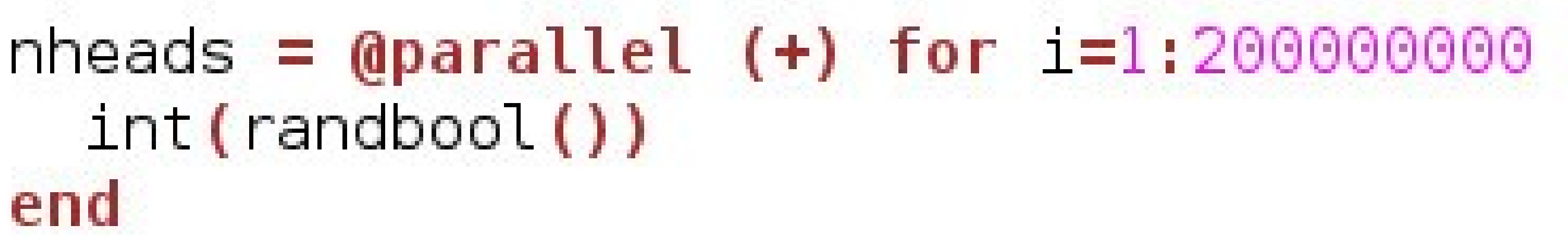


```
require("count_heads")

a = @spawn count_heads(100000000)
b = @spawn count_heads(100000000)
@show fetch(a)+fetch(b)
```

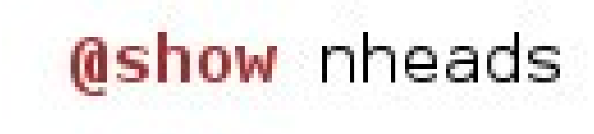


```
bash-4.3$ julia -p 2 179a.jl
fetch(a) + fetch(b) => 99992606
```

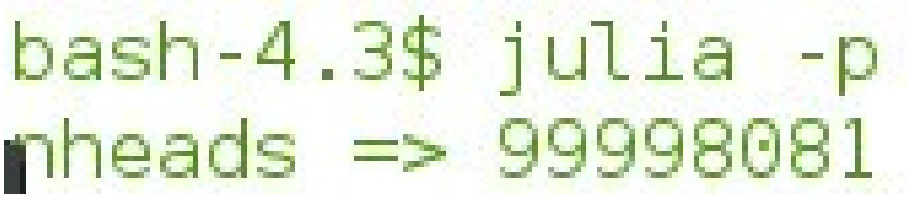
Parallel for-loop



```
nheads = @parallel (+) for i=1:200000000
  int(randbool())
end
```

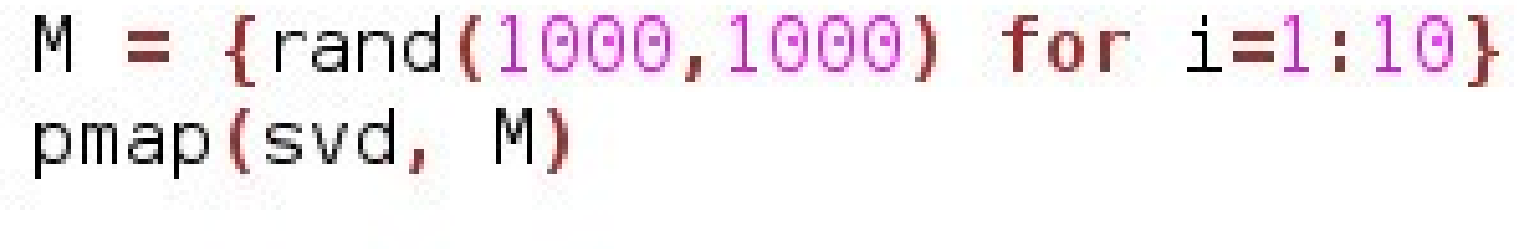


```
@show nheads
```



```
bash-4.3$ julia -p 2 179b.jl
nheads => 99998081
```

Parallel map



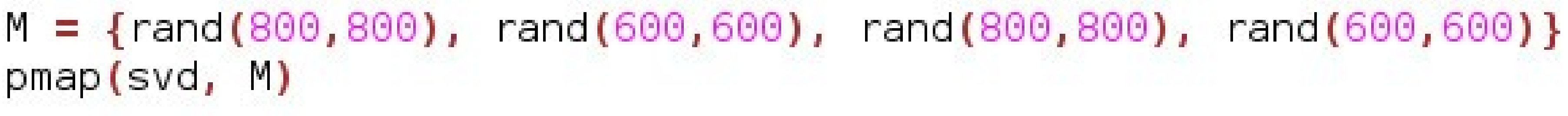
```
M = {rand(1000,1000) for i=1:10}
pmap(svd, M)
```

- Compute the singular values of several large random matrices in parallel

- Parallel map: each function call does a large amount of work
- Parallel loop: each iteration is tiny, perhaps merely summing two numbers

Scheduling

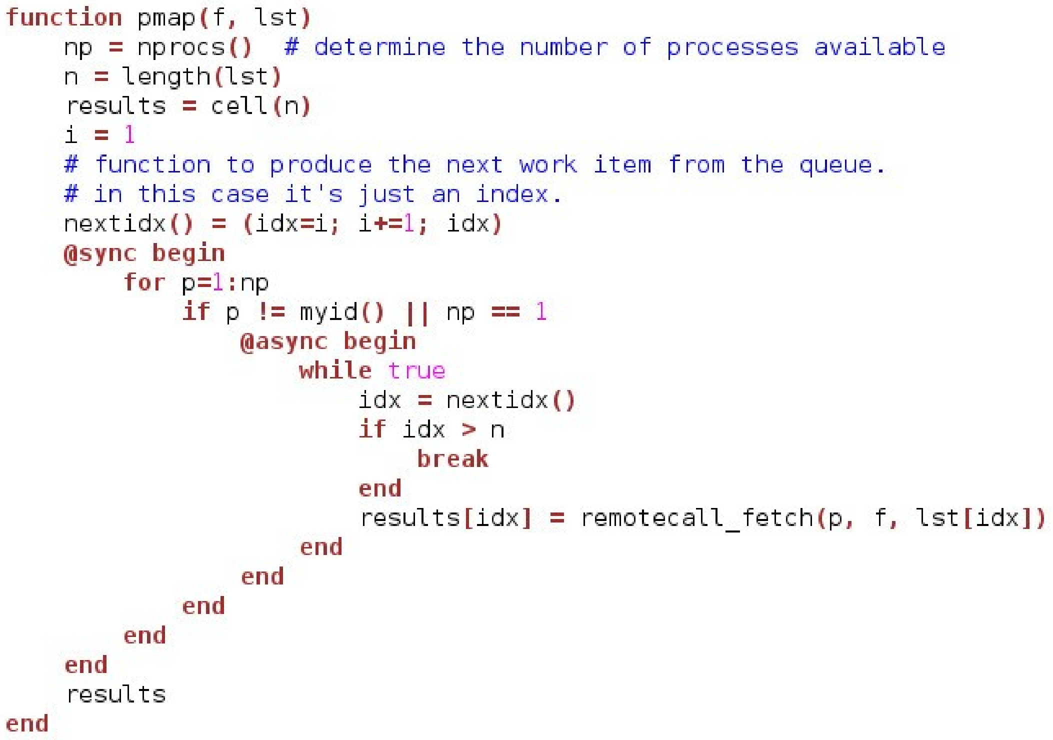
- Dynamic scheduling: a program decides what to compute or where to compute it based on when other jobs finish
- An example: computing the singular values of matrices of different sizes



```
M = {rand(800,800), rand(600,600), rand(800,800), rand(600,600)}
pmap(svd, M)
```

Dynamic scheduling

- `@async` runs task on the local processor
- “Feeder” task for other tasks
- Each task picks the next index that need to be computed, then waits for its job finish



```
function pmap(f, lst)
    np = nprocs()  # determine the number of processes available
    n = length(lst)
    results = cell(n)
    i = 1
    # function to produce the next work item from the queue.
    # in this case it's just an index.
    nextidx() = (idx=i; i+=1; idx)
    @sync begin
        for p=1:np
            if p != myid() || np == 1
                @async begin
                    while true
                        idx = nextidx()
                        if idx > n
                            break
                        end
                        results[idx] = remotecall_fetch(p, f, lst[idx])
                    end
                end
            end
        end
    end
    results
end
```

Homework

- Generate n processes (n is the number of cores on your computer) to simulate the answer of the following question:

  There are 9,784,445 people in Sweden. Everyone is going to roll 2 fair dice. How many people will get 12 totally?