**Produce-consume scheme\methods and the Task() way of execution**

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

# One procedure is generating values and another procedure is consuming them

# Julia provides the functions produce and consume, e.g., if we create the function:

julia> function producer()

produce("start")

for n=1:2

produce(2n)

end

produce("stop")

end

# To consume the values of this function, first the function producer() has to be wrapped bu the comand Task,

# creating in this way a new object: p,

julia> p = Task(producer)

# Than, we can call the consume() repeatedly on that object, p:

julia> consume(p)

"start"

julia> consume(p)

2

julia> consume(p)

4

julia> consume(p)

"stop"

# A Task can be used as an iterable object in a for loop, in which case the

# loop variable takes on all the produced values:

julia> for x in Task(producer)

println(x)

end

start

2

4

stop

# If now we write the usual loop:

julia> for x in [1,2,4] println(x) end

# the result will be:

1

2

4

# However, we can do the same by an "anonymous" and "runnable" Task, t:

julia> t = @task { for x in [1,2,4] println(x) end }

# This is onlay a definition of this task and the result from its execution will be:

Task (runnable) @0x0000000005478fe0

# if we ask now:

julia> istaskdone(t)

# the answer will be:

false

# if we noe ask: what is the state of the current Task:

julia> current\_task()

# the answer will be:

Task (waiting) @0x0000000004464f80

# as previosly, we can execute the task t by the method consume(t),

julia> consume(t)

1

2

4

1-element Array{Any,1}:

nothing

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

**Julia’s Prnciples for Parallel Computing**

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Julia's implementation of **message passing** is one-sided:

• the programmer needs to explicitly manage only one processor in a

two-processor operation

• **these operations** typically do not look like message send and message

receive but rather **resemble higher-level operations like calls to user**

**functions.**

Two key notions: **remote references** and **remote calls:**

• A **remote reference** is an object that can be used from any processor

to refer to an object stored on a particular processor.

• A **remote call** is a request by one processor to call a certain function

on certain arguments on another (possibly the same) processor. A

**remote call returns a remote reference**.

How **remote calls** are handled in the program flow:

• Remote calls **return immediately**: the processor that made the call

can then proceeds to its next operation, while the remote call happens

somewhere else.

• You can **wait** for a remote call to finish **by calling wait on its remote**

**reference**, and you can **obtain the full value** of the result using **fetch**.

xxxxxxxxxxxxxxxxxxxxxxxxxxxxx **Examples** xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

# First we run Julia on 4 processors on the local machine:

boian@Big~$ julia -p 4

# Next we make our first remote call, r:

julia> r = remotecall(2, rand, 2, 2)

RemoteRef(2,1,6)

• Here, we execute: Base.remotecall(id, func, args...), where

1) the first argument (2) in the remotecall(2, rand, 2, 2), is the index of the processor that will do the work.

2) the second argument (rand) is the function we are calling on the processor with id=2;

3) Next arguments denote the argument that the function rand() will use, in this case – rand has to produce a 2x2 matrix.

# Further, we can see the results be the command fetch():

julia> fetch(r)

2x2 Array{Float64,2}:

0.34652 0.69345

0.172677 0.62607

# Than we can it to add 1 to it:

julia> s = @spawnat 2 1+fetch(r)

RemoteRef(2,1,10)

# and of course we can to see the result of this operation by using again fetch():

julia> fetch(s)

2x2 Array{Float64,2}:

1.34652 1.69345

1.17268 1.62607

the macros @spawnat, execute the command, 1+fetch(r), on processor with id==2;