Julia: A third perspective
Parallel computing explained

Ge Baolai
SHARCNET | Compute Ontario | Compute Canada
Western University

A language for both prototyping and performance

Outline

We try to cover the following

- A quick review of what's covered in the previous talks
- Automatic parallelization in linear algebra operations
- Parallel and distributed computing
- Running julia on CCF systems

What's **NOT** covered

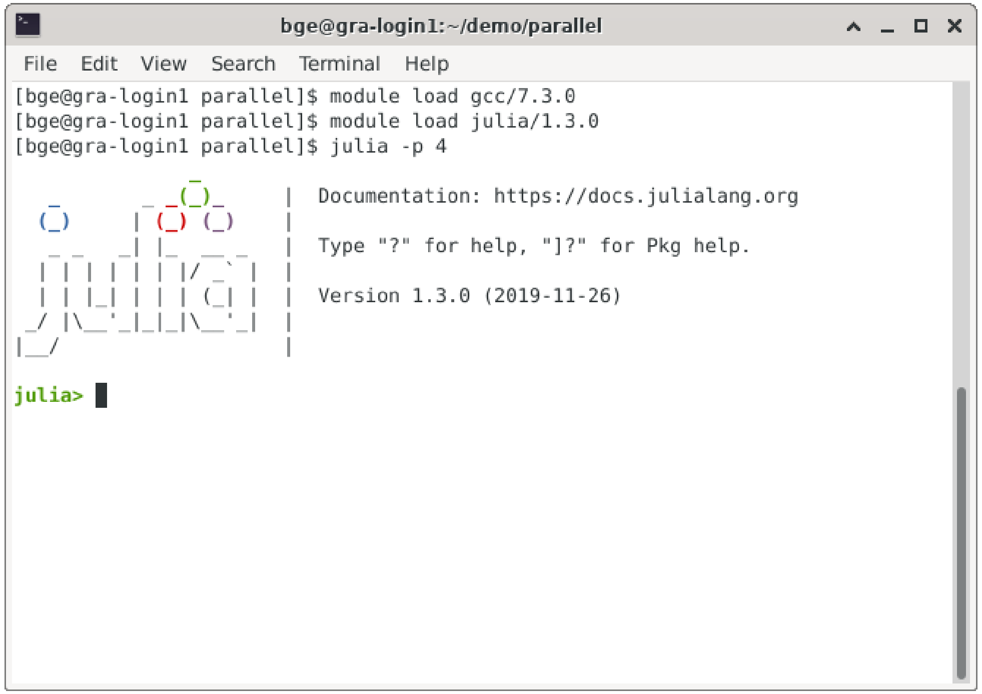
- Threaded computing (next separate talk)
- MPI and others

This is not a complete, systematic introduction but a collection of pointers for ones to explore.

A quick review

A quick review

- Available for Windows, Linux and Mac OS X (intel processors fully supported, limited support for ARM based processors).
- Rich programming language support.
- Support for parallel programming paradigms via the underlying MPI library.
- Support for linear algebra operations.
- Support data frames.
- Very fast, compared to R, Python, Matlab and even C/C++ and Fortran.
- Available on CCF systems.


bge@gra-login1:~/demo/parallel
File Edit View Search Terminal Help

```
[bge@gra-login1 parallel]$ module load gcc/7.3.0
[bge@gra-login1 parallel]$ module load julia/1.3.0
[bge@gra-login1 parallel]$ julia -p 4
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.3.0 (2019-11-26)
 _/ |\__'_|_|_|\__'_|  |
|__/                   |

julia> 
```

A quick review

Structure

```
struct Person
    name::AbstractString
    id::Int
end

people = Person[] # Create an empty array
push!(people,Person("Ge B",88544))
push!(people,Person("Tyson W",78910))
```

```
julia> people
2-element Array{Person,1}:
 Person("Ge B", 88544)
 Person("Tyson W", 78910)
```

A quick review

Dictionaries (Pair of key, value)

```
# Create a dictionary containing two entries
d = Dict("a"=>1,"b"=>2)

# Get the value corresponding to key “a”, otherwise return -1
get(d,"a",-1)

# Add an entry or a dictionary, but the original d does not change
merge(d,Dict(“c”=>99))

# Add an entry or a dictionary, now d has changed
merge!(d,Dict(“c”=>99))
```

A quick review

Operations	MATLAB	R	Julia	Note
Slicing	A(i,j), B(i:j,m:n)	A[i,j], B[i:j,m:n]	A[i,j], B[i:j,m:n]	
	A(end)	A[length(A)]	A[end]	The last element.
	A(1:end ~=k)	A[~k]	A[1:end .!=k]	All but the kth element.
	A(i,:), A(:,j)	A[i,], A[,j]	A[i,:], A[:,j]	
Assignment	B = A	B = A	copyto!(B,A)	With B=A, B is an alias to A. Use copyto!() to create a copy.
Sequence	*from*,*by*,*to*	seq(*from*,*to*,*by*)	collect(*from*,*by*,*to*)	
Filtering	A > b	A > b	A .> b	Return indices of elements > b.
	A(A > b)	A[A > b]	A[A .> b]	Return a subarray of elements > b.
Replacement	A(A > b)=val	A[A > b]=val	A[A .> b] .= val	Replace elements > b with val.
Delete an object	clearvar(A)	rm(A)	A=nothing	Just replace with one taking less mem and run garbage collection with gc()

Automatic parallelization in linear algebra operations

Linear algebra operations

Matrix-vector operations via OpenBLAS

We run this simple code first

```
n = 5000
A = randn(n,n)
B = randn(n,n)
C = zeros(n,n)

using LinearAlgebra
for i=1:4
    @time C = A*B
end
```

And then set environment variable

```
export OMP_NUM_THREADS=4
```

and run it again to see if there's any performance changes.

Do not spawn julia threads!

Linear algebra operations

Solving linear dense system

```
using LinearAlgebra
A = [2.0 -1 0 0 0
-1 2 -1 0 0
0 -1 2 -1 0
0 0 -1 2 -1
0 0 0 -1 2]
x = ones(5)
b = A*x
sol = A\b
```

Solving linear sparse system

```
using LinearAlgebra
A = [2.0 -1 0 0 0
-1 2 -1 0 0
0 -1 2 -1 0
0 0 -1 2 -1
0 0 0 -1 2]
using SparseArrays
A1 = sparse(A)
x = ones(5)
b = A1*x
sol = A1\b
```

Parallel computing

Implicit

- Less effort, no need to write explicit parallel code.
- Using built-in libraries, e.g. OpenBLAS for linear algebra operations using multi-cores.
- Using shared and distributed data objects.
- Debugging?

Explicit

- Need extra effort to write parallel code, having to know what you are doing.
- Explicit control of data transfers via send/recv operations among processes.
- One-sided communication via put/get operations.
- Debugging can be challenging.

Parallel and distributed computing

Parallel computing: Implicit parallelism

Matrix-vector operations via OpenBLAS

We run this simple code first

```
n = 5000
A = randn(n,n)
B = randn(n,n)
C = zeros(n,n)

using LinearAlgebra
for i=1:4
    @time C = A*B
end
```

And then set environment variable

```
export OMP_NUM_THREADS=4
```

and run it again to see if there's any performance changes.

Do not spawn julia threads!

If all your work is like this, then you are done. The rest is more advanced.

Parallel computing: Starting multiple processes

Launching from command line when starting julia

```
julia -p 8
```

or

```
julia --machine-file hostfile
```

Launching from within a julia process

```
using Distributed

# Start extra 8 processes to have 9 in total
addprocs(8)
```

Parallel computing: Starting multiple processes

Launching from command line when starting julia

```
julia -p 8
```

or

```
julia --machine-file hostfile
```

Launching from within a julia process

```
using Distributed

# Start extra 8 processes to have 9 in total
addprocs(8)
```

Dynamically creating or increasing the number of processes is not recommended.

This is for all jobs, e.g. R, Matlab, Python, etc on systems where the job schedule controls.

Parallel computing: Broadcasting a value to all processes

```
# Broadcast a value to all processes
using Distributed


@everywhere x = 12345 # This works


x0 = 12345
@everywhere x = x0 # This will fail, as x0 is local


@everywhere x = $x0 # This works! By "copying" x0 value
```

Parallel computing: Executing a function on all processes

Execute a locally defined function

```
using Distributed


# The scope of this function is within this process
function showid()
    println("My ID: ", myid())
end


# This is likely to fail on other processes
@everywhere showid()
```

Execute a globally defined function

```
using Distributed


# This function is defined on every process
@everywhere function showid()
    println("My ID: ", myid())
end


# Execute this procedure on every process
@everywhere showid()
```

Parallel computing: Executing a function on all processes

Execute a locally defined function

```
using Distributed

# The scope of this function is within this process
function showid()
    println("My ID: ", myid())
end

# This is likely to fail on other processes
@everywhere showid()
```

Execute a globally defined function

```
using Distributed

# This function is defined on every process
@everywhere function showid()
    println("My ID: ", myid())
end

# Execute this procedure on every process
@everywhere showid()
```

@everywhere *stmt*

Parallel computing: Executing a procedure remotely

```
using Distributed

println("Number of cores: ", nprocs())
println("Number of workers: ", nworkers())

# Fetch the ID of each worker and host the worker running on
for i in workers()
    id, pid, host = fetch(@spawnat i (myid(), getpid(), gethostname()))
    println(id, " " , pid, " Hello from ", host)
end
```

Parallel computing: Executing a procedure remotely

Julia uses the concept "future" referring to the remote execution.

To run a procedure on an automatically chosen process

```
f = @spawn (x.^2, myid())
```

To run a procedure on a specific process n

```
f = @spawnat n (x.^2, myid())
```

To get the result, one needs to "fetch" it by the reference.

```
fetch(f)
```

Parallel computing: Executing a procedure remotely

Julia uses the concept "future" referring to the remote execution.

To run a procedure on an automatically chosen process

f = @spawn (x.^2, myid())

To run a procedure on a specific process n

f = @spawnat n (x.^2, myid())

To get the result one needs to "fetch" it by the reference.

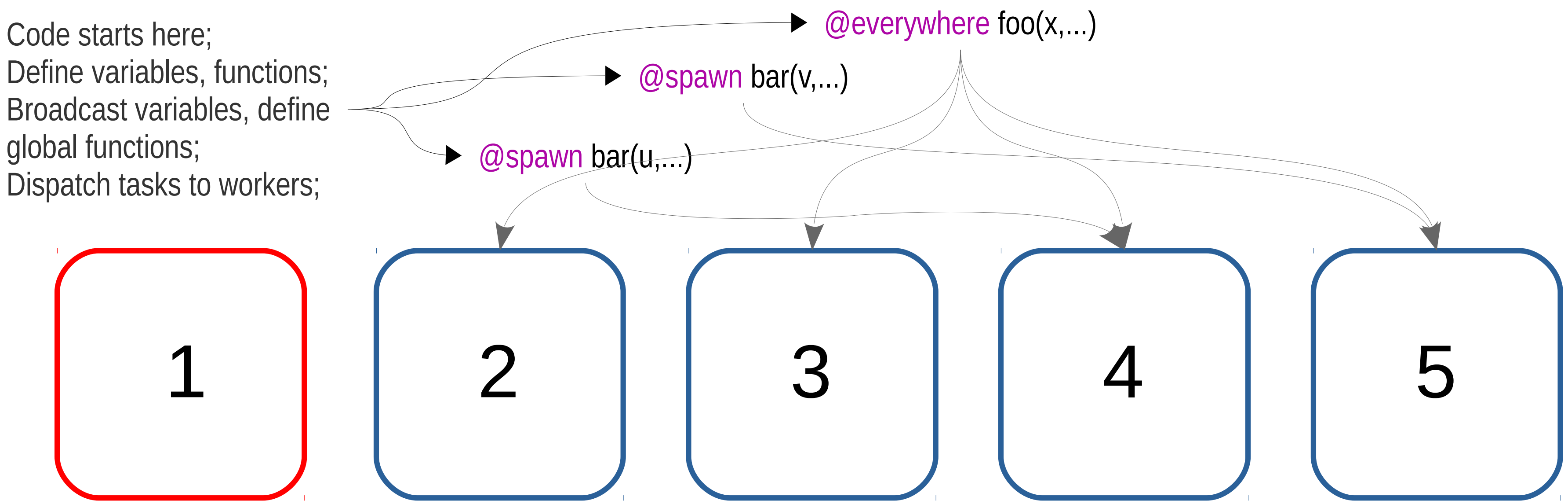
fetch(f)

@spawn *stmt*

@spawnat *proc stmt*

Parallel computing: Programming model

Julia always uses 1+***p*** processes: A control or Main process, plus ***p*** Worker processes

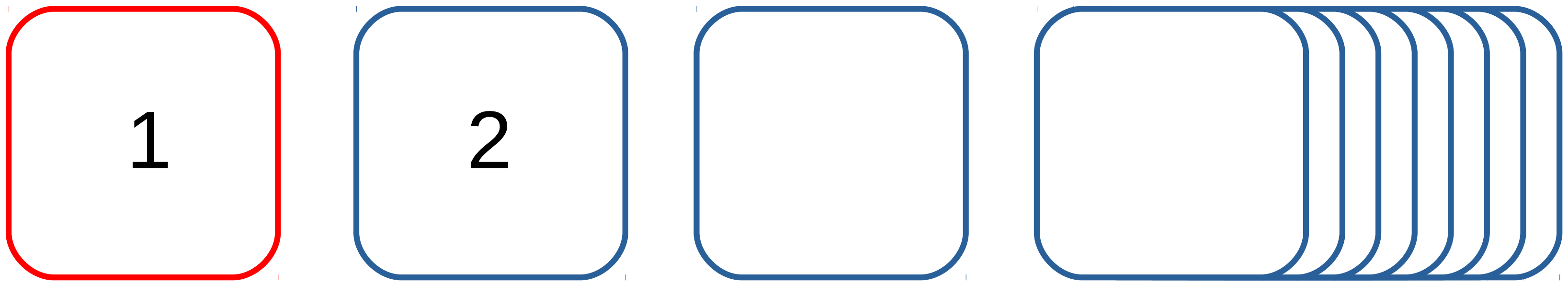


Code starts here;
Define variables, functions;
Broadcast variables, define global functions;
Dispatch tasks to workers;
@everywhere foo(x,...)
@spawn bar(v,...)
@spawn bar(u,...)
1
2
3
4
5

Tasks are dispatched and computed on workers, like jobs are done on compute nodes.

Parallel computing: Programming model

Who am I?



Parallel computing: Placing a remote call

Asynchronous call, non-blocking, returns immediately

```
f = remotecall( maximum, WorkerPool(workers()),   x   )
                  call            where          var
```

To get the result

```
r = fetch(f)
```

Synchronous call, combines remotecall() and fetch()

```
r = remotecall_fetch(maximum,WorkerPool(workers()),x)
```

Parallel computing: Producer-consumer model

A communication channel between “tasks” **Channel** can be used for communication between tasks.

```
c1=Channel(1024)

c2=Channel(1024)
```

Define a function that wraps producer-consumer pattern

```
function foo()
    while condition==true
        data = take!(c1)  # Take a task from c1
        Process data. If this is the last data set condition=false
        put!(c2, result)  # Put result to c2
    end
end
```

Parallel computing: Producer-consumer model

A communication channel between “tasks” **Channel** can be used for communication between tasks.

```
c1=Channel(1024)

c2=Channel(1024)
```

Define a function that wraps producer-consumer pattern

```
function foo()

    while condition==true

        data = take!(c1)  # Take a task from c1

        Process data. If this is the last data set condition=false

        put!(c2, result)  # Put result to c2

    end

end
```

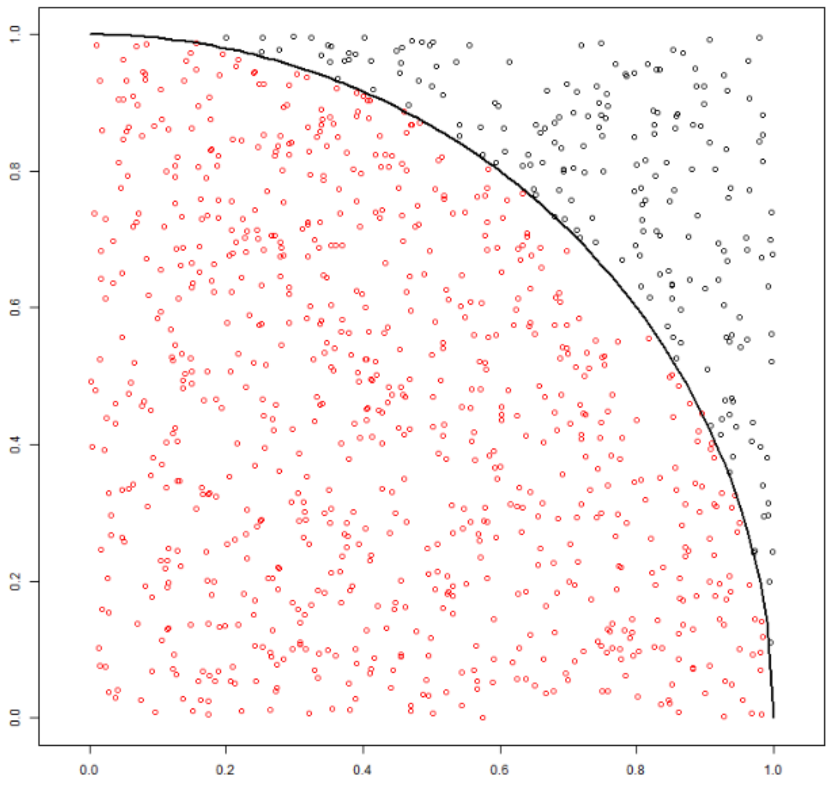
Then schedule n instances of foo to be active concurrently on **local** machine

```
for _ in 1:n

    @async foo()

end
```

Parallel computing: Calculating the approximation of pi

We compute the approximation of pi by counting the points uniformly tossed inside an 1/4 circle vs total number of points over the unit square (See Marc Marano Maza 2017).

$$\frac{\frac{1}{4}\pi a^2}{a^2} = \frac{n_{\text{in}}}{n} \implies \pi \approx 4\frac{n_{\text{in}}}{n}$$



Parallel computing: Calculating the approximation of pi

Create a file "**pi_dist.jl**", define a function that counts the number of points falling inside the circle

```
function points_inside_circle(n)
    n_in = 0
    for i=1:n
        x, y=rand(), rand()
        n_in += (x*x + y*y) <= 1
    end
    return n_in
end
```

In the same file, define a function wrapper that computes the approximation of pi in parallel

```
function pi_p(n)
    p = nworkers()
    n_in = @distributed (+) for i=1:p # A reduction call
        points_inside_circle(n/p)
    end
    return 4*n_in/n # The approximation of pi
end
```

This function executes on multiple cores in parallel and collects the result by reduction

@distributed *op procedure*

Parallel computing: Calculating the approximation of pi

Create a file "**pi_dist.jl**", define a function that counts the number of points falling inside the circle

```
function points_inside_circle(n)
    n_in = 0
    for i=1:n
        x, y = rand(), rand()
        n_in += (x*x + y*y) <= 1
    end
    return n_in
end
```

In the same file, define a function wrapper that computes the approximation of pi in parallel

```
function pi_p(n)
    p = nworkers()
    n_in = @distributed (+) for i=1:p # A reduction call
        points_inside_circle(n/p)
    end
    return 4*n_in/n # The approximation of pi
end
```

@distributed *op procedure*

N.B. This function executes on multiple cores in parallel and collects the result by reduction

@distributed *op procedure*

Parallel computing: Calculating the approximation of pi

Now we start julia with 4 workers using command

```
julia -p 4
```

Within julia, use the commands below

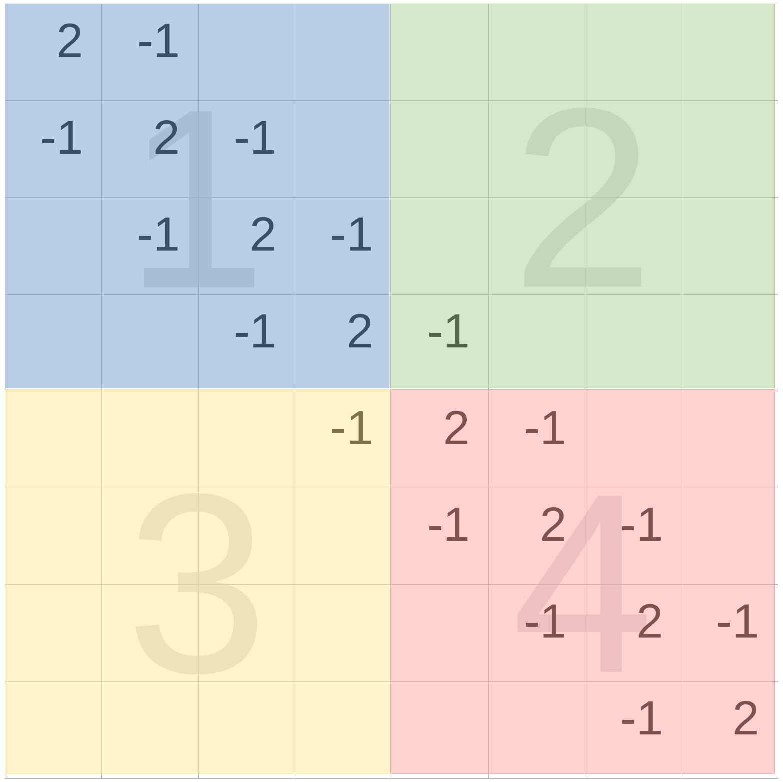
```
julia> using Distributed
julia> @everywhere include("pi_dist.jl") # Load functions on all processes

julia> pi_p(1_000_000) # pi_p() is defined in file “pi_dist.jl”
3.1419629999999996
```

Parallel computing: Distributed arrays

Example: A matrix stored across 4 processes on a 2x2 Cartesian processor grid

Proces 1 has the blue portion.

But it also has access to other portions stored remotely, *simply via indices.*



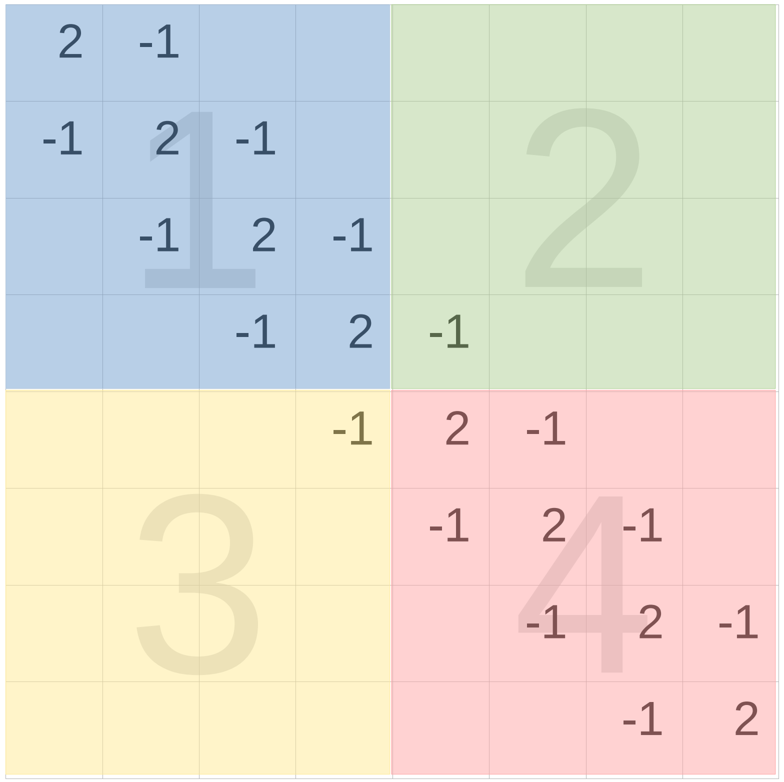
$$\begin{array}{cccc|cccc}
2 & -1 & & & & & & \\
-1 & 2 & -1 & & & & & \\
 & -1 & 2 & -1 & & & & \\
 & & -1 & 2 & -1 & & & \\
\hline
 & & & -1 & 2 & -1 & & \\
 & & & & -1 & 2 & -1 & \\
 & & & & & -1 & 2 & -1 \\
 & & & & & & -1 & 2
\end{array}$$

Suitable for handling large data sets that can NOT fit on a single machine.

Parallel computing: Distributed arrays

```
using Distributed, DistributedArrays
@everywhere using LinearAlgebra
@everywhere function aa(n)
    la = zeros(n,n)
    la[diagind(la,0)] .= 2.0
    la[diagind(la,-1)] .= -1.0
    la[diagind(la,1)] .= -1.0
    return la
end
@everywhere function b1(n)
    la = zeros(n,n); la[1,n] = -1.0;
    return la
end
@everywhere function b2(n)
    la = zeros(n,n); la[n,1] = -1.0;
    return la
end
```

Matrix A distributed on 4 processors on a 2x2 grid



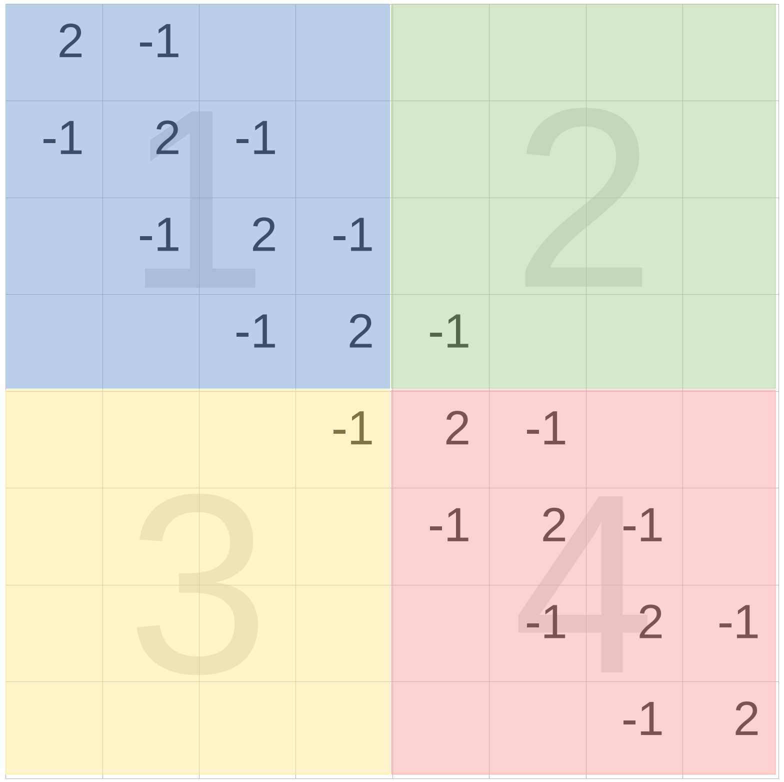
Parallel computing: Distributed arrays

```
# Call functions on workers to created local portions
d11 = @spawnat 2 aa(4)
d12 = @spawnat 3 b1(4)
d21 = @spawnat 4 b2(4)
d22 = @spawnat 5 aa(4)

# Create a distributed matrix on a 2x2 processor grid
DA = DArray(reshape([d11 d21 d12 d22],(2,2)));
```

NB:

- No (large) data communications between Main and workers;
- **d11**,**d12**,**d21**,**d22** are not matrices, but handles – futures. They are NOT taking up spaces;
- **DA** is NOT the whole matrix either, it's a reference;
- But one can access the entire matrix by simply using the index, e.g. DA[5000,5050] even though it's not local.

Matrix A distributed on 4 processors on a 2x2 grid



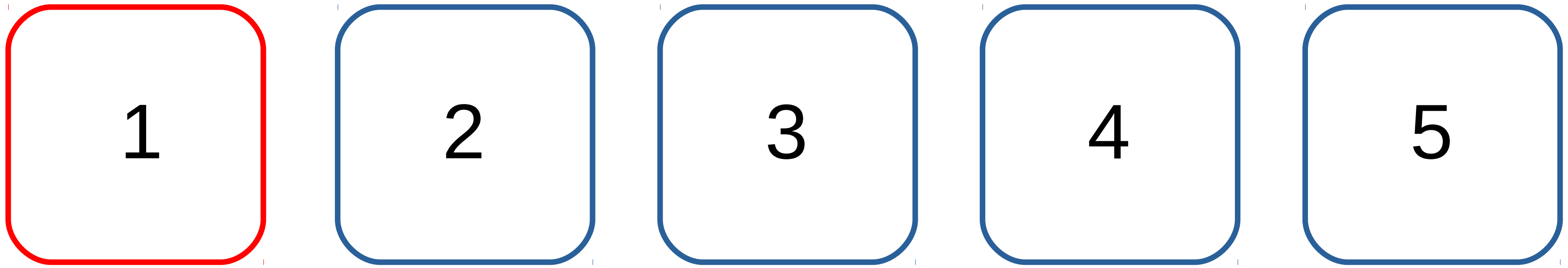
2	-1						
-1	2	-1					
	-1	2	-1				
		-1	2	-1			
			-1	2	-1		
				-1	2	-1	
					-1	2	-1
						-1	2

Parallel computing: Distributed arrays

Julia always uses 1+***p*** processes: A control or Main process, plus ***p*** Worker processes

```
varinfo()
```

To see vars on “me”



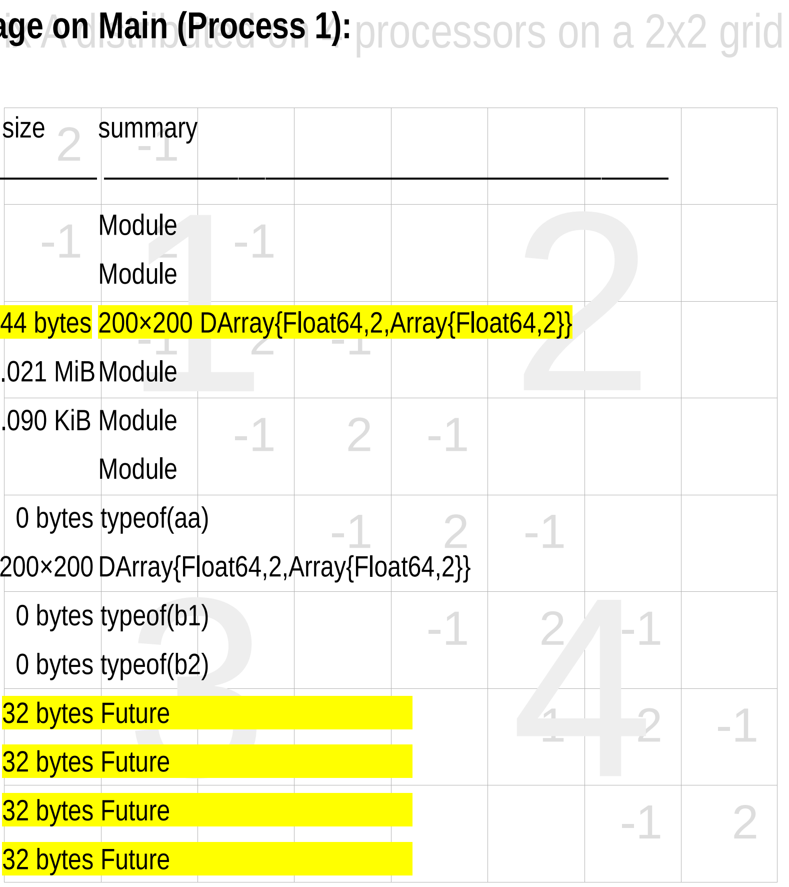
To see vars on others

```
@everywhere using InteractiveUtils
fetch(@spawnat p varinfo())
```

Parallel computing: Distributed arrays

```
# Call functions on workers to created local portions
n=100
d11 = @spawnat 2 aa(n)
d12 = @spawnat 3 b1(n)
d21 = @spawnat 4 b2(n)
d22 = @spawnat 5 aa(n)

# Create a distributed matrix on a 2x2 processor grid
DA = DArray(reshape([d11 d21 d12 d22],(2,2)));

# Examine storage on Main
varinfo()
```

Examining the storage on Main (Process 1):

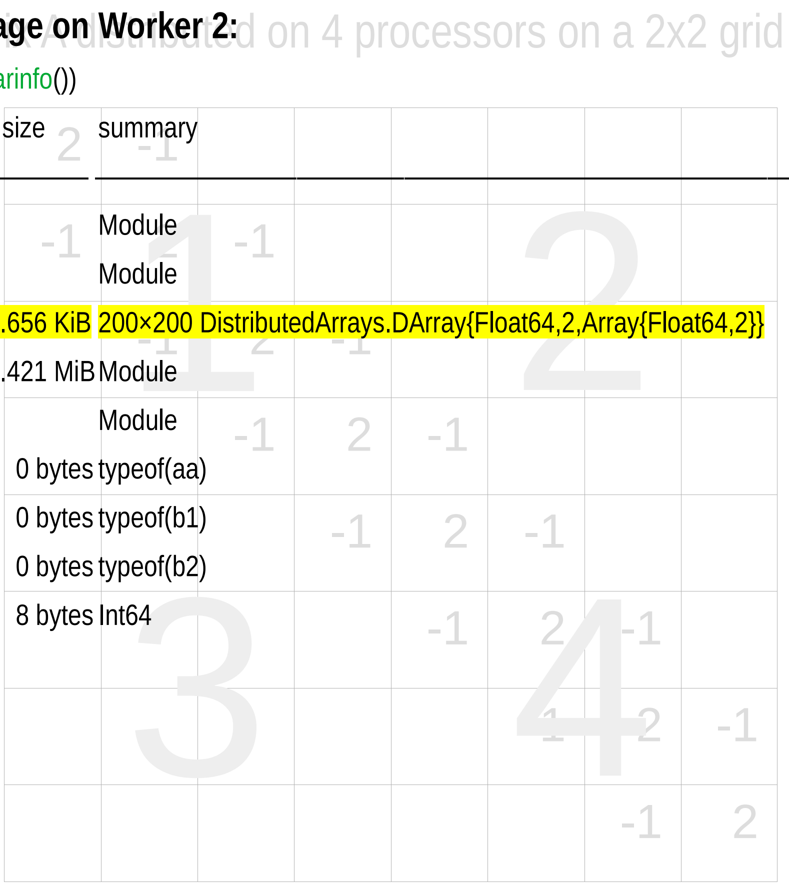


```
julia> varinfo()
  Name                        size summary
  ––––––––––––––– –––––––––– ––––––––––––––––––––––––––––––––––––
  Base                             Module
  Core                             Module
  DA                     544 bytes 200×200 DArray{Float64,2,Array{Float64,2}}
  Distributed            2.021 MiB Module
  InteractiveUtils     162.090 KiB Module
  Main                             Module
  aa                       0 bytes typeof(aa)
  ans               544 bytes 200×200 DArray{Float64,2,Array{Float64,2}}
  b1                       0 bytes typeof(b1)
  b2                       0 bytes typeof(b2)
  d11                     32 bytes Future
  d12                     32 bytes Future
  d21                     32 bytes Future
  d22                     32 bytes Future
  n                        8 bytes Int64
```

Parallel computing: Distributed arrays

```
# Call functions on workers to created local portions
n=100
d11 = @spawnat 2 aa(n)
d12 = @spawnat 3 b1(n)
d21 = @spawnat 4 b2(n)
d22 = @spawnat 5 aa(n)

# Create a distributed matrix on a 2x2 processor grid
DA = DArray(reshape([d11 d21 d12 d22],(2,2)));

# Examine remote storage on Worker 2
fetch(@spawnat 2 varinfo())
```

Examining the storage on Worker 2:

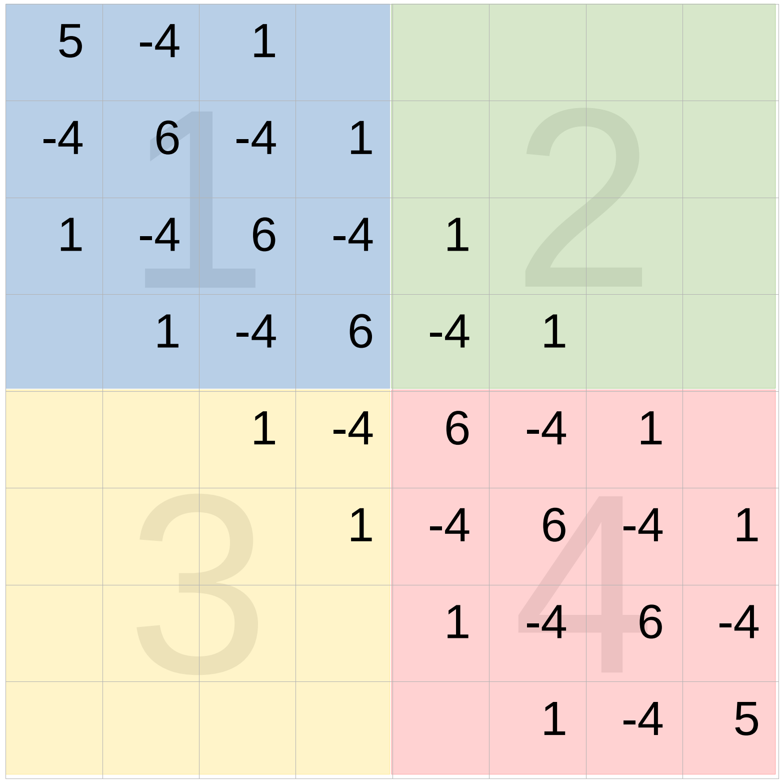


```
julia> fetch(@spawnat 2 varinfo())
Name                      size  summary
―――――――――――――――――――――――― ―――――――――― ――――――――――――――――――――――――――――――――――――――――――――――――――――
Base                            Module
Core                            Module
DA                  78.656 KiB  200×200 DistributedArrays.DArray{Float64,2,Array{Float64,2}}
Distributed          1.421 MiB  Module
Main                            Module
aa                     0 bytes  typeof(aa)
b1                     0 bytes  typeof(b1)
b2                     0 bytes  typeof(b2)
n                      8 bytes  Int64
```

Parallel computing: Distributed arrays

```
julia> # Perform A*A directly on distributed arrays
julia> DB = dzeros(8,8)
julia> DB = DA*DA

julia> # Check remote values on process 3
julia> f = @spawnat 3 DB.localpart # Remote call returns a future
julia> fetch(f)
4×4 Array{Float64,2}:
 0.0  0.0  1.0  -4.0
 0.0  0.0  0.0   1.0
 0.0  0.0  0.0   0.0
 0.0  0.0  0.0   0.0

julia> remotecall_fetch(localpart,3,DB) # Alternative
```

Result of A*A distributed on 4 processors

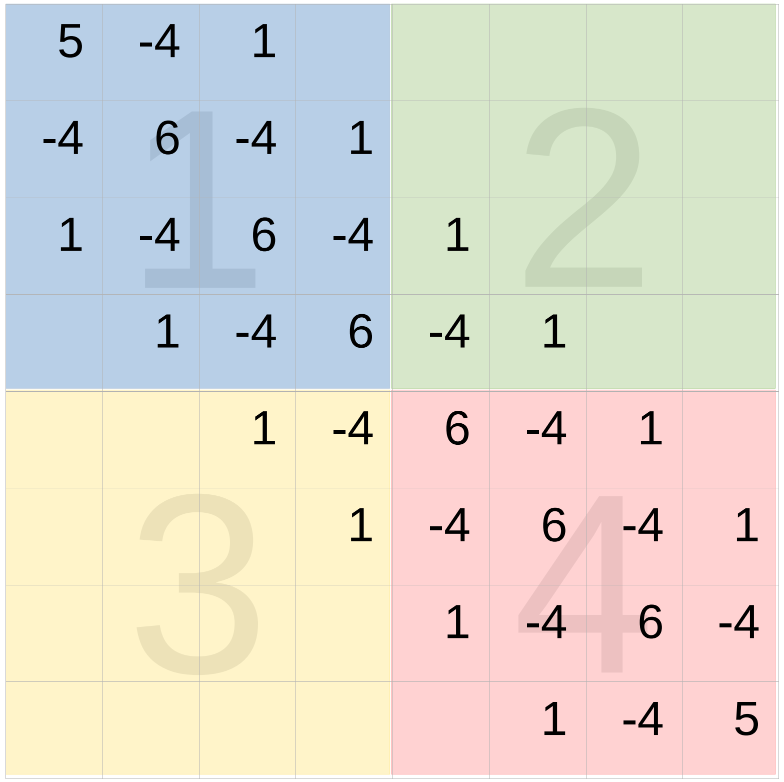


5	-4	1					
-4	6	-4	1				
1	-4	6	-4	1			
	1	-4	6	-4	1		
		1	-4	6	-4	1	
			1	-4	6	-4	1
				1	-4	6	-4
					1	-4	5

Parallel computing: Distributed arrays

```
julia> # Access components owned remotedly
julia> DB[5:8,1:4]
4×4 view(::DArray{Float64,2,Array{Float64,2}}, 5:8, 1:4) with eltype
Float64:
 0.0  0.0  1.0  -4.0
 0.0  0.0  0.0   1.0
 0.0  0.0  0.0   0.0
 0.0  0.0  0.0   0.0
```

Result of A*A distributed on 4 processors



5	-4	1					
-4	6	-4	1				
1	-4	6	-4	1			
	1	-4	6	-4	1		
		1	-4	6	-4	1	
			1	-4	6	-4	1
				1	-4	6	-4
					1	-4	5

Parallel computing: Distributed arrays

Summary:

- Define functions to be executed on workers, e.g. via @everywhere;
- Define global variables and broadcast to workers, e.g. via @everywhere;
- Create distributed arrays, by calling functions on workers, via @spawnat or remotecall();
- Perform the operations on the distributed arrays, as if they were local;
- This is a very different concept from the SPMD model (often seen in scientific applications, e.g. written in MPI)

Parallel computing: Distributed arrays

Summary (cont'd):

- So far not much self-contained functionalities are available, but only allows one to reference to global spaces by indexing to the elements.
- Each process has a global view of any distributed objects.
- It uses one-sided communication via underlying libraries (e.g. MPI). The other prominent programming language that supports global address access is Fortran.
- Support from third party libraries are expected.
- A few packages to look at
  - **Elemental** – hides the communication APIs and one can do linear algebra operations as is, such as svdvals(A) to get SVD values.
  - **PETSc** – contains explicit MPI like APIs.
  - **Trilinos** – contains explicit MPI like APIs.

Parallel computing: Shared arrays

Shared arrays via module SharedArrays provide a convenient way of accessing data among processes. The following creates a 5x4 integer array on each process

```
using SharedArrays
A = SharedArray{Int,2}((5,4))
```

Changes to A in one process also happen to A on other processes.

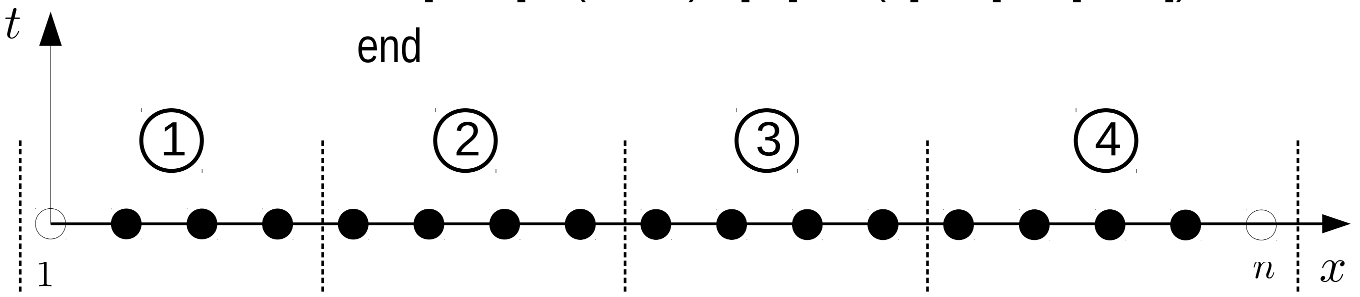
Parallel computing: Shared arrays

Example: 1D heat equation. A rod heated in the middle, the temperature distribution over time can be simulated by the following

$$T(x, t + \Delta t) = (1 - 2k)T(x, t) + k(T(x - \Delta x, t) + T(x + \Delta x, t))$$

Using a 2D array u[i,j] to store the temperature at spatial points (1st dim) and over time steps (2nd dim),

```
for i=i1:in
    u[i,k+1] = (1.0-2r)*u[i,k] + r*(u[i-1,k] + u[i+1,k])
end
```



The spatial points are partitioned into *p*, e.g. 4, worker groups, the temperature in each is updated concurrently independent of other groups.

Parallel computing: Shared arrays

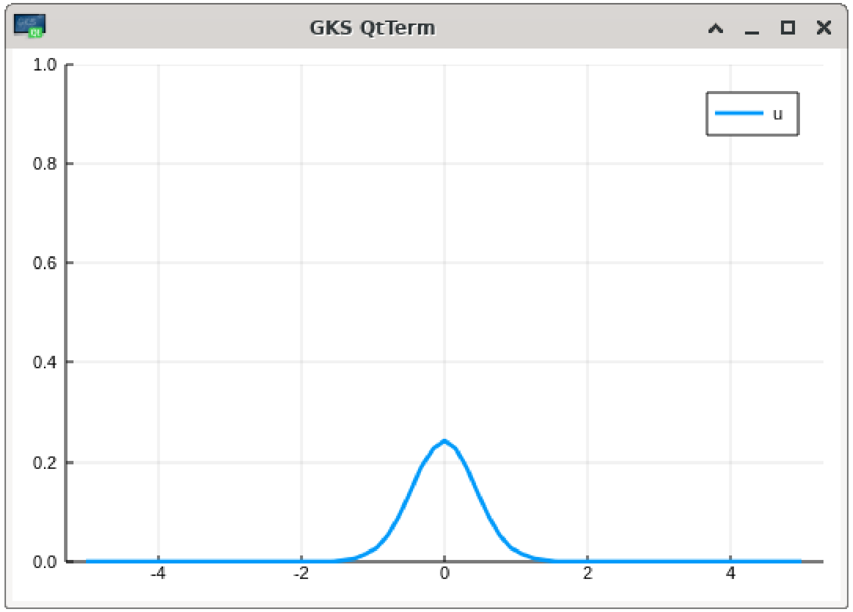
NB: The loop

```
for i=2:n-1
    u[i,k+1] = (1.0-2r)*u[i,k] + r*(u[i-1,k] + u[i+1,k])
end
```

can be replaced by the vectorized form

```
u[2:n-1,k+1] = (1.0-2r)*u[2:n-1,k] + r*(u[1:n-2,k] + u[3:n,k])
```

See our Python, Matlab/Octave and Fortran courses.

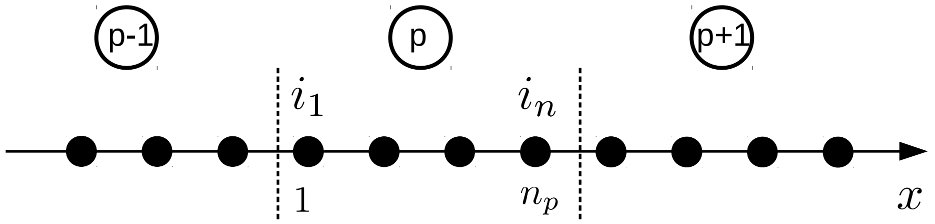


NB: 2r is not a typo, it is a legitimate literal expression in julia

Parallel computing: Shared arrays

Serial code (sketch)

```
u = zeros(n,nt);
… …
for k=1:nt-1
    u[2:n-1,k+1] = (1.0-2r)*u[2:n-1,k] + r*(u[1:n-2,k]+u[3:n,k])
    if (k % nt_disp == 0)
        display(plot(x,u[:,k],lw=3,ylim=(0,1)))
    end
end
```



Parallel code (sketch)

NB: Although the data of u is shared, but u itself is not. It must be passed to workers.

```
u = SharedArray{Float64,2}(n,nt);
u .= 0;
@everywhere function update(u,k,p)
    i1 = np*(p - 1) + 1; # Start index
    if (p == 1) # Skip the boundary point
        i1 = 2;
    end
    in = i1 + np + n % num_workers - 1; # End index
    if (p == num_workers) # Skip the boundary point
        in = n - 1;
    end
    u[i1:in-1,k+1] = (1.0-2r)*u[i1:in-1,k] + r*(u[i1-1:in-2,k]+u[i1+1:in,k])
end
```

Parallel computing: Shared arrays

Serial code (sketch)

```
for k=1:nt-1
    u[2:n-1,k+1] = (1.0-2*r)*u[2:n-1,k] + r*(u[1:n-2,k]+u[3:n,k])
    if (k % nt_disp == 0)
        display(plot(x,u[:,k],lw=3,ylim=(0,1)))
    end
end
```

Parallel code (sketch)

Three args passed to update(). NB: Although the data of u is shared, but the u itself as a reference must be passed to workers.

```
for k=1:nt-1
    @sync begin
        for p=1:num_workers
            @async remotecall(update,p+1,u,k,p);
        end
    end
    if (k % nt_disp == 0)
        display(plot(x,u[:,k+1],lw=3,ylim=(0,1)))
    end
end
```

remotecall(*func*, *pid*, *args_of_func*) – returns immediately

Parallel computing: Shared arrays

Summary

- Shared arrays are for the local computer only (Fortran's co-arrays can be across nodes);
- Shared arrays can be accessed via global indexing, hence convenient for parallel algorithms;
- For A = SharedArray{Float64,2}(n,n), the data is shared, but A is not. It's a reference and must be passed to participating workers via any of the following

  @everywhere function … end or @everywhere *var*=...

  @everywhere include(*code_script*)

  @remotecall(*func*, *worker_set*, *var_list*)
- Math and linear algebra operations apply to shared array objects as regular arrays;
- Lastly the diffusion example can also be implemented using distributed arrays, so it can run on clusters.

Threads in julia

Parallel computing: Threads

Example: Parallel loop. First start julia with say 4 threads by setting environment variable

```
export JULIA_NUM_THREADS=4
```

Then run jula. In julia, run the following commands

```
using Base.Threads

threadid() # Should be 1
nthreads() # Should be 4

n=10
a = zeros(n)
@threads for i=1:n
    a[i] = threadid()
end
```

Results

```
julia> a
10-element Array{Float64,1}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
 4.0
 4.0
```

Parallel computing: Threads

Example: Create threads with

```
export JULIA_NUM_THREADS=4
```

and have each one do some work in a function

```
using Base.Threads
nthreads()
function do_something()
   println("In Thread ", threadid())
   sleep(1)
end

@threads for _ in 1:nthreads()
   do_something()
end
```

Results

```
In Thread 4
In Thread 1
In Thread 2
In Thread 3
```

NB: Julia seems to only create threads up to the number of available physical cores.

Summary

- So far the threads module in julia is still experimental;
- The number of threads can be created seems to be limited by the physically available cores;
- There doesn’t to be a way of creating more threads on demand;
- We will have a separate talk dedicated to julia multi-threading programming;
- See Jeff Bezanson (Julia Computing), Jameson Nash (Julia Computing), Kiran Pamnany (Intel), “Announcing composable multi-threaded parallelism in Julia”, 2019.

Running julia on CCF systems

Running julia on CCF systems

Loading modules

Run the following commands

```
module spider julia
module spider julia/1.3.1
```

Then load dependencies and jula

```
module load julia/1.3.1
```

Example: “hello.jl” - Displaying IDs of all worker processes

```
using Distributed

println("Number of cores: ", nprocs())
println("Number of workers: ", nworkers())

# Each worker gets its id, process id and hostname
for i in workers()
    id, pid, host = fetch(@spawnat i (myid(), getpid(), gethostname()))
    println(id, " " , pid, " ", host)
end

# Remove the workers
for i in workers()
    rmprocs(i)
end
```

Running julia on CCF systems

Slurm job script: run_julia.sh – run across nodes, containing the following lines

```
#!/bin/bash
#SBATCH --ntasks=64                # Number processes
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1024M  # Memory, default 4GB
#SBATCH --time=0-00:05             # Run time (DD-HH:MM)
#SBATCH --account=def-bge          # Billing account
#SBATCH --output=hello.log

srun hostname -s > hostfile
julia --machine-file ./hostfile ./hello.jl
```

Submitting jobs

```
sbatch run_julia.sh
```

References

[1] Marc Marano Maza, Lecture Notes: Distributed and parallel systems, Department of Compute Science, Western University, 2017.

[2] Julia documentations: https://docs.julialang.org/en/v1/.

[3] Julia cheat sheet: https://juliadocs.github.io/Julia-Cheat-Sheet/.