Introduction to julia
Parallel Computing Revisited

Ge Baolai, Western University
Edward Armstrong, University of Guelph
SHARCNET | Compute Ontario | Compute Canada

A language for both prototyping and performance

Outline

We try to cover the following

- Examples of using parallelization enabled linear algebra libraries
- Examples of parallel processing support via **Distributed**
- Examples of using distributed arrays (**DistributedArrays**) and shared arrays (**SharedArrays**)
- A example of using threads

What will **NOT** be covered

- Using Julia in Jupyter Notebook
- Threaded computing details (a separate talk)
- MPI and others

This is not a tutorial, but rather a collection of pointers for ones to explore.

Using libraries

Parallel computing: Implicit parallelism

Example: 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!

Running on multiple processors

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)
```

On clusters using a scheduler, dynamically creating or increasing the number of processes is ***DISCOURAGED.***

Parallel computing: Starting multiple processes

Launching from command line when starting julia

```
julia -p 8
```

or

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

Launching on a cluster

```
#!/bin/bash
#SBATCH --ntasks=64              # number of MPI processes
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1024M
#SBATCH --time=0-00:05
#SBATCH --account=def-bge
#SBATCH --job-name=hello
#SBATCH --output=hello.log

srun hostname -s > hostfile
sleep 5
julia --machine-file ./hostfile ./hello.jl
```

Parallel computing: Programming model

Two simple mechanisms

- @everywhere
- @spawn, @spawnat

Parallel computing: Broadcasting a value to all processes

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

@everywhere x = 12345 # This works

X0 = 12345 # X0 is a local variable to the main process
@everywhere x = x0 # This MAY fail, as x0 is local, check the following
@everywhere println(x)

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

Parallel computing: Executing a function on all processes

Execute a locally defined function

```julia
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

```julia
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

Exercise: Type and run the following code

```
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

Note the performance difference in the following two calls

```
n=10_000
A=randn(n,n);

@time fetch(@spawnat :any sum(A.^2)) # Involves copying A to remote process
```

vs.

```
n=10_000
@time fetch(@spawnat :any sum(randn(n,n).^2)) # No data copy
```

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)

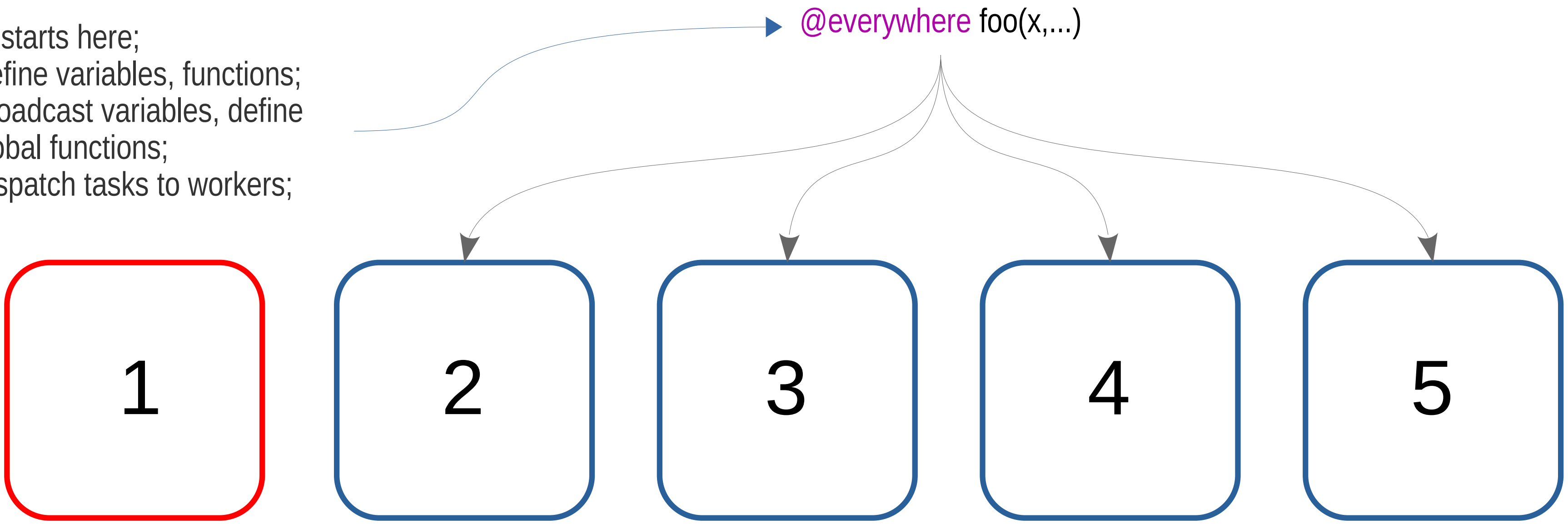
@spawn *stmt*

@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;

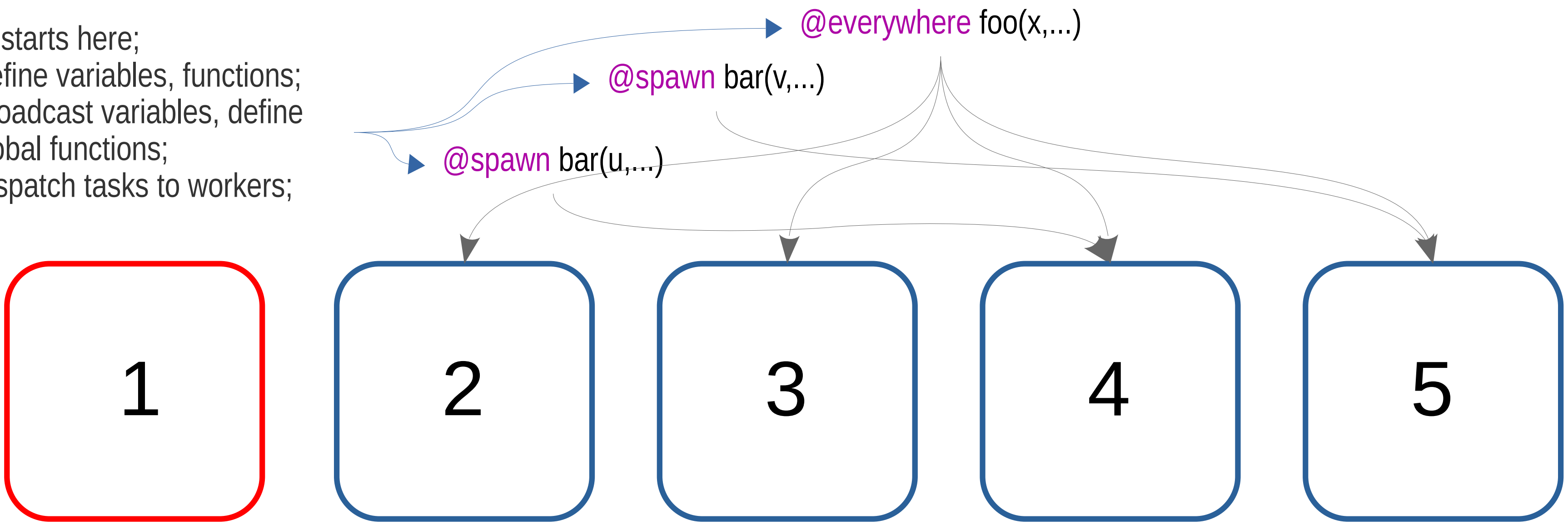


Tasks are dispatched and computed on workers, like jobs are done on compute nodes.

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;



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

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)
```

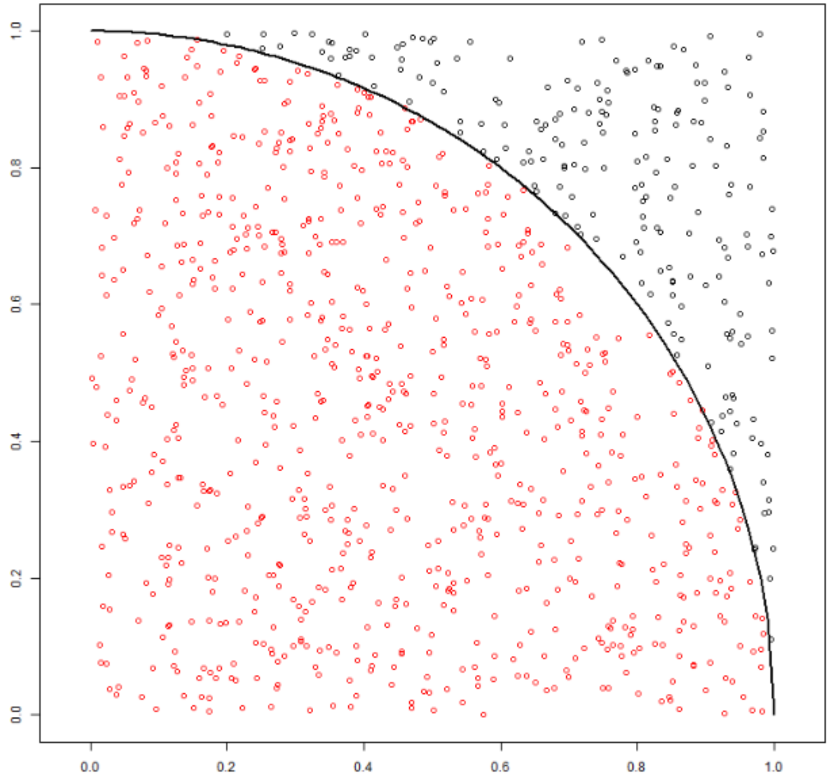
Synchronous call, combines remotecall() and fetch()

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

Parallel computing: Calculating the approximation of pi

Example: 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).

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



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
```

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

@distributed *op procedure*

@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
```

Using 4 cores, for n=1,000,000,000, it will take about 4 to 5 seconds.

Parallel computing: Calculating the approximation of pi

Create a file "**pi_pmap.jl**", define a function that estimates pi one local processor

```
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 = sum(pmap(x->points_inside_circle(x),
        [n/p for i=1:p]))
    return 4*n_in/n
end
```

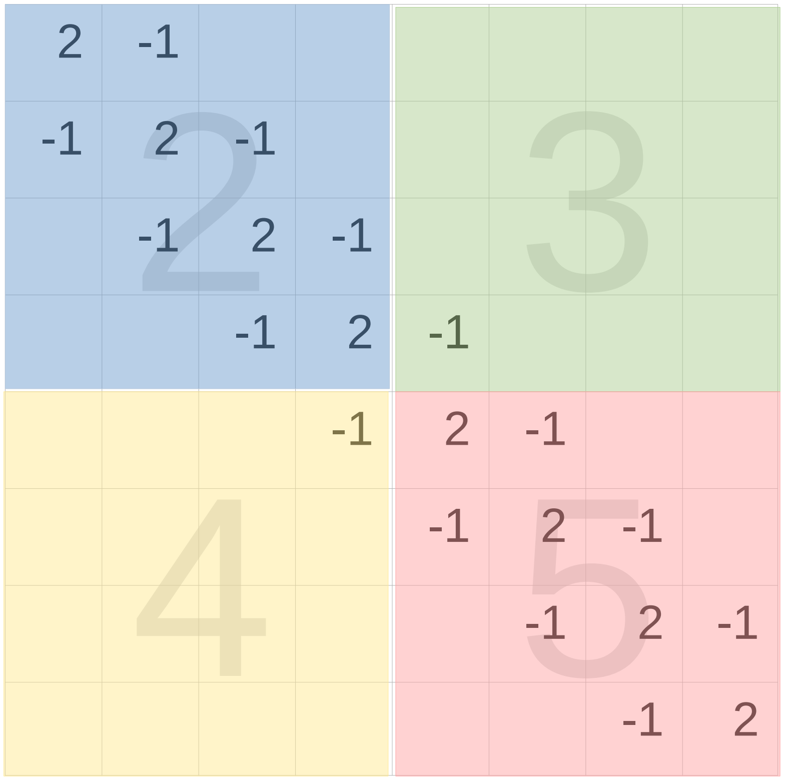
Using distributed arrays

Parallel computing: Distributed arrays

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

Process 1 has the blue portion.

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

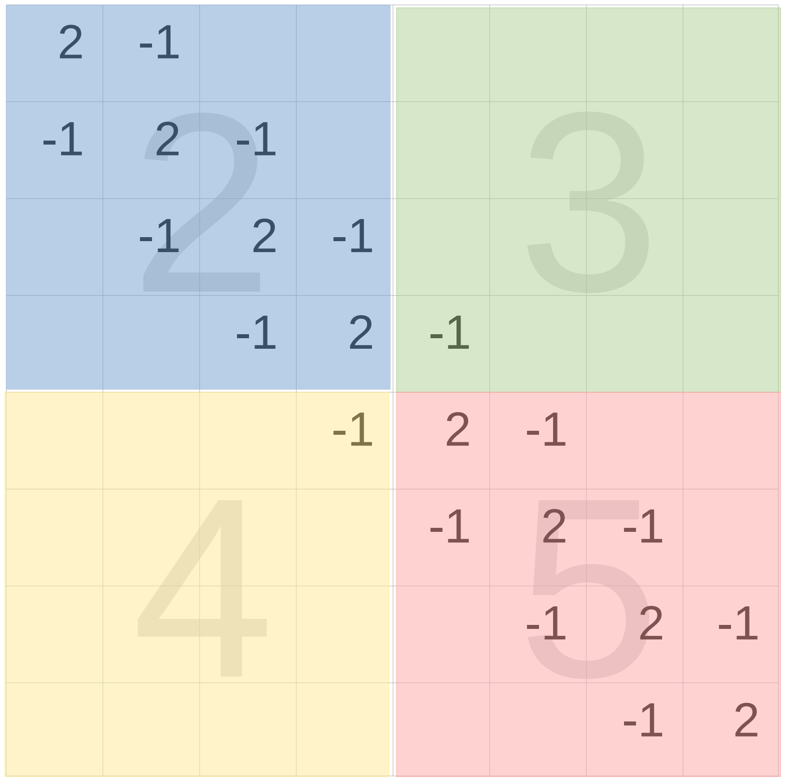


Suitable for handling large data sets that can NOT fit on a single machine.

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



$$\begin{array}{cccccccc} 2 & -1 & & & & & & \\ -1 & 2 & -1 & & & & & \\ & -1 & 2 & -1 & & & & \\ & & -1 & 2 & -1 & & & \\ & & & -1 & 2 & -1 & & \\ & & & & -1 & 2 & -1 & \\ & & & & & -1 & 2 & -1 \\ & & & & & & -1 & 2 \end{array}$$

Parallel computing: Distributed arrays

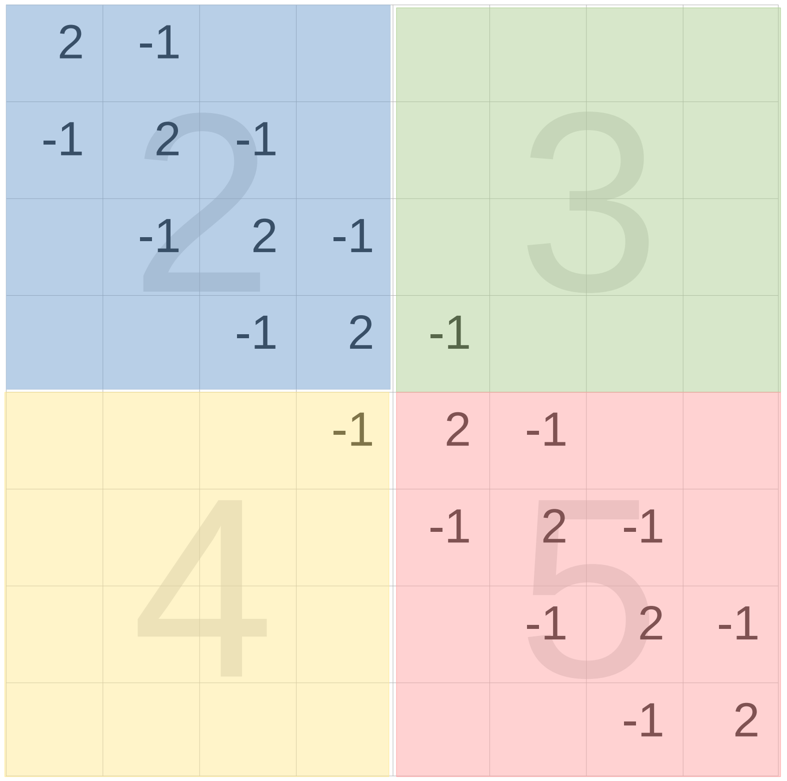
```
# Call functions on workers to created local portions
d11 = @spawnat 2 aa(4)
d12 = @spawnat 3 b1(4)
d21 = @spawnat 4 b2(4)
d22 = @spawnat 5 aa(4)

# 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



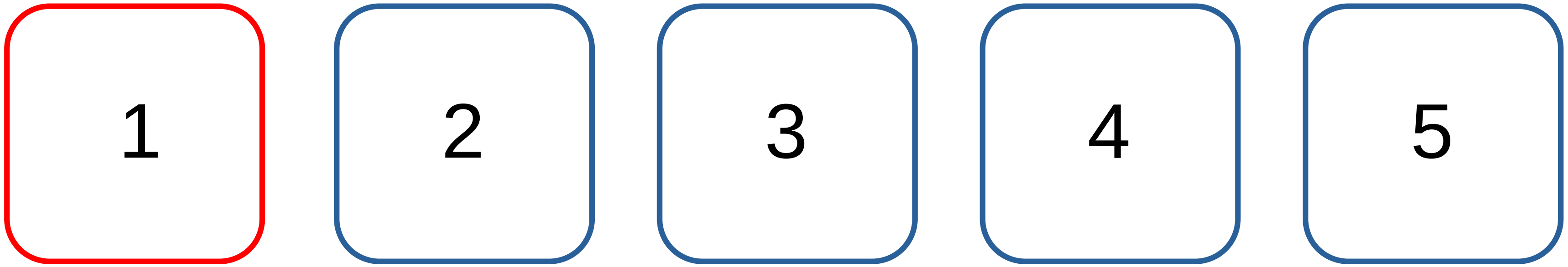
2	-1						
-1	2	-1					
	-1	2	-1				
		-1	2	-1			
			-1	2	-1		
				-1	2	-1	
					-1	2	-1
						-1	2

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())
```

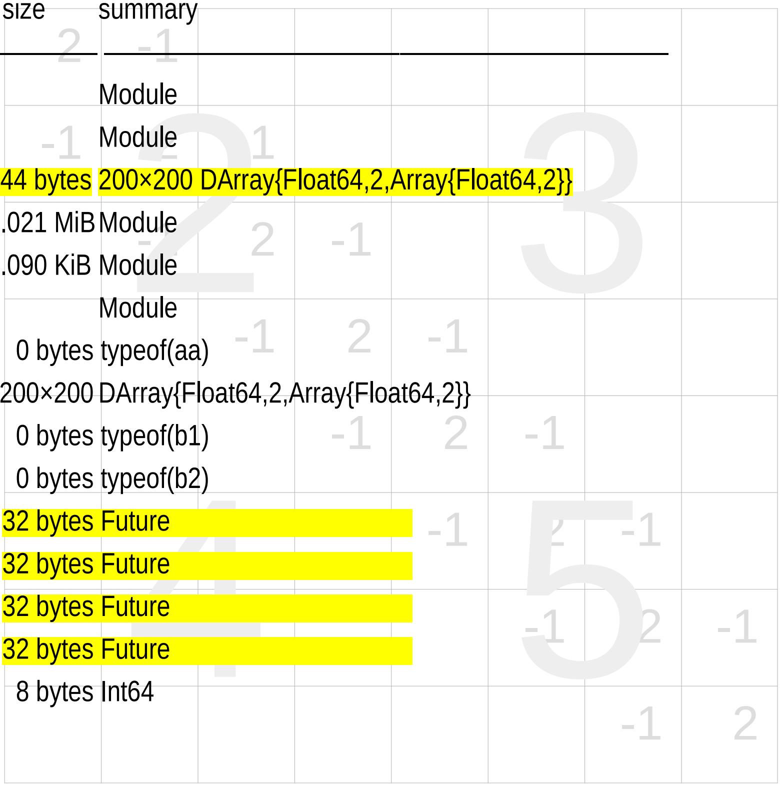
Parallel computing: Distributed arrays

```
# 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 d12 d21 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

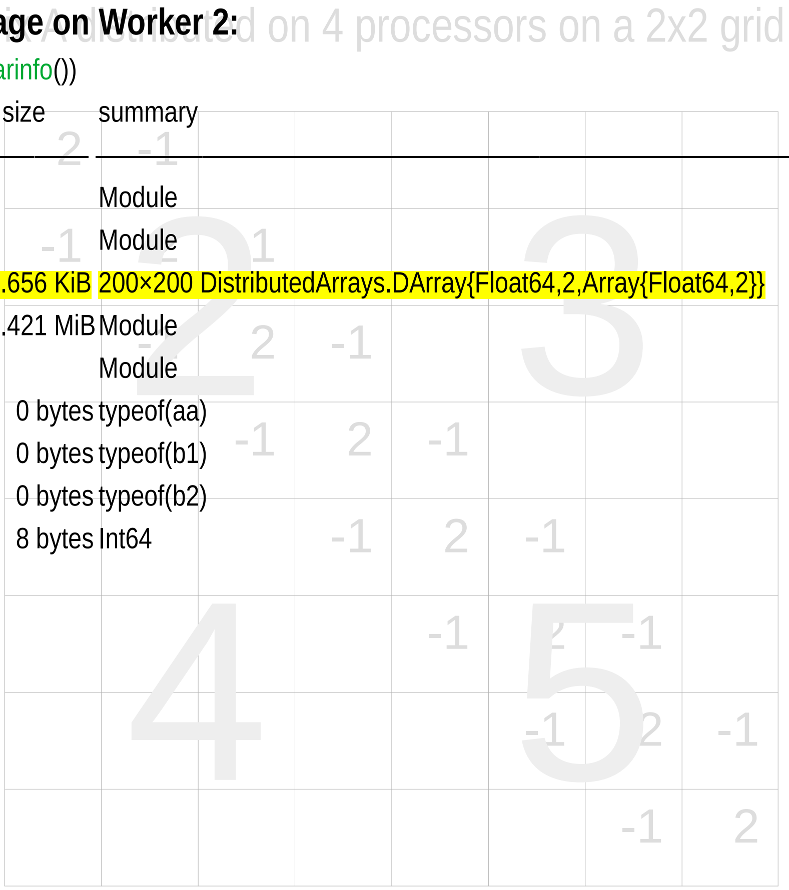
```
# Call functions on workers to created local portions
n=100
d11 = @spawnat 2 aa(n)
d12 = @spawnat 3 b1(n)
d21 = @spawnat 4 b2(n)
d22 = @spawnat 5 aa(n)

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



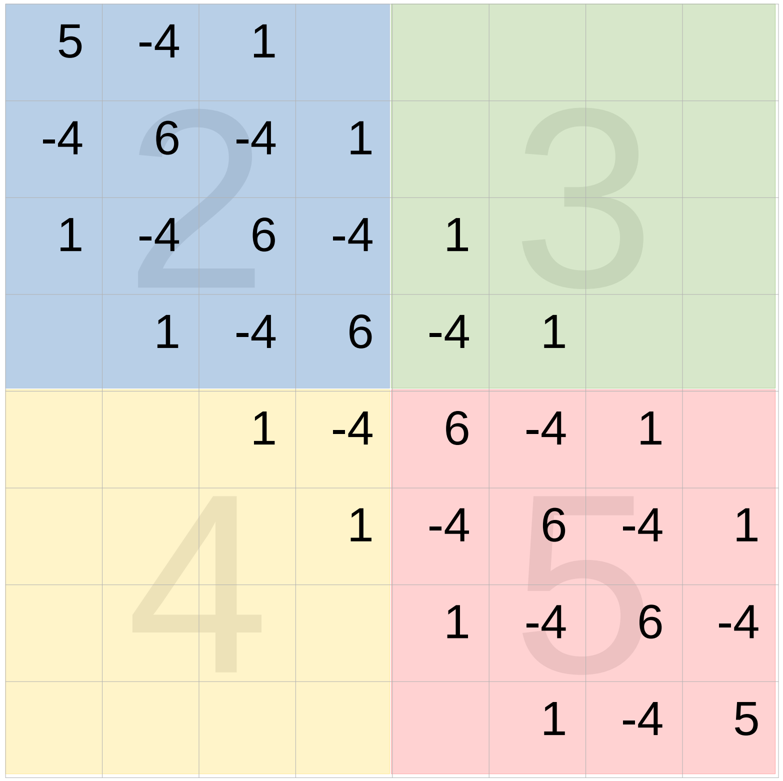
Parallel computing: Distributed arrays

```
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

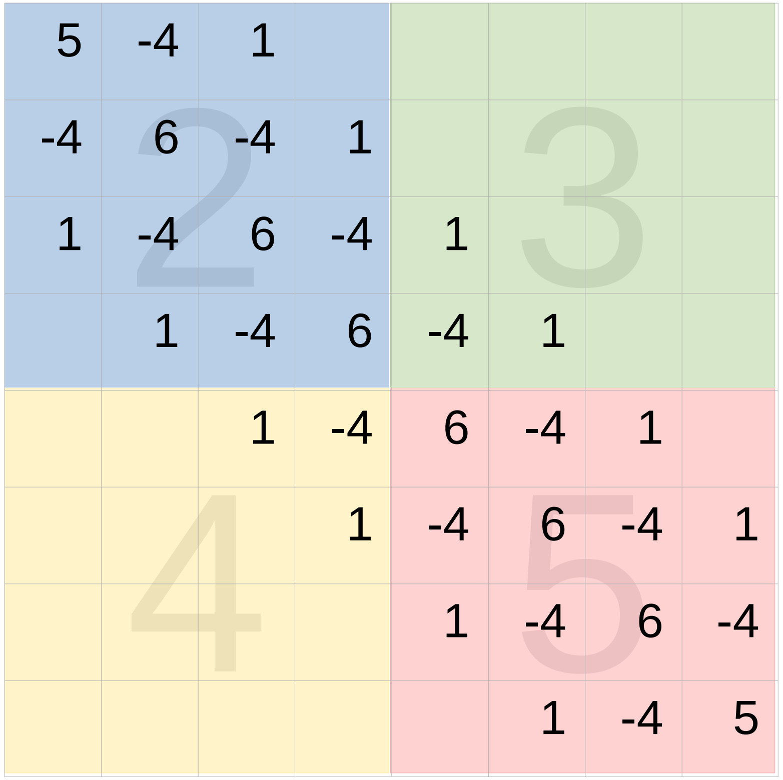


5	-4	1					
-4	6	-4	1				
1	-4	6	-4	1			
	1	-4	6	-4	1		
		1	-4	6	-4	1	
			1	-4	6	-4	1
				1	-4	6	-4
					1	-4	5

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.

Using shared arrays

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

```
julia> using SharedArrays
julia> A = SharedArray{Int,2}((5,4))
5×4 SharedArray{Int64,2}:
 0  0  0  0
 0  0  0  0
 0  0  0  0
 0  0  0  0
 0  0  0  0
julia> A[:,3] = collect(11:15)
5-element Array{Int64,1}:
 11
 12
 13
 14
 15
```

```
julia> # Get the the 3rd column of A on worker 4
julia> @fetchfrom 4 A[:,3]
5-element Array{Int64,1}:
 11
 12
 13
 14
 15
```

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

Parallel computing: Shared arrays

Remote calls

The following isn't what you intended

```
n=16
a=zeros(n) # a is local
@distributed for i=1:n
    a[i] = i
end
# a now is available on other workers!

@fetchfrom 2 a # Only see the first 4 elements are assigned values
```

NB: Surprise!

- The code results in a copy of a on each process.
- Only a portion of a gets assigned values on each process.

Using SharedArray

```
using SharedArrays
n=16
a = SharedArray{Float64}(n)
@distributed for i=1:n # Each process does a portion of the loop
    a[i] = i
end
```

NB: Each process has access to the entire chunk of the array a. In other words, a is shared among participating processes.

SharedArray objects are used on the same machine.

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.

Using threads, an example

Parallel computing: Multi-threading

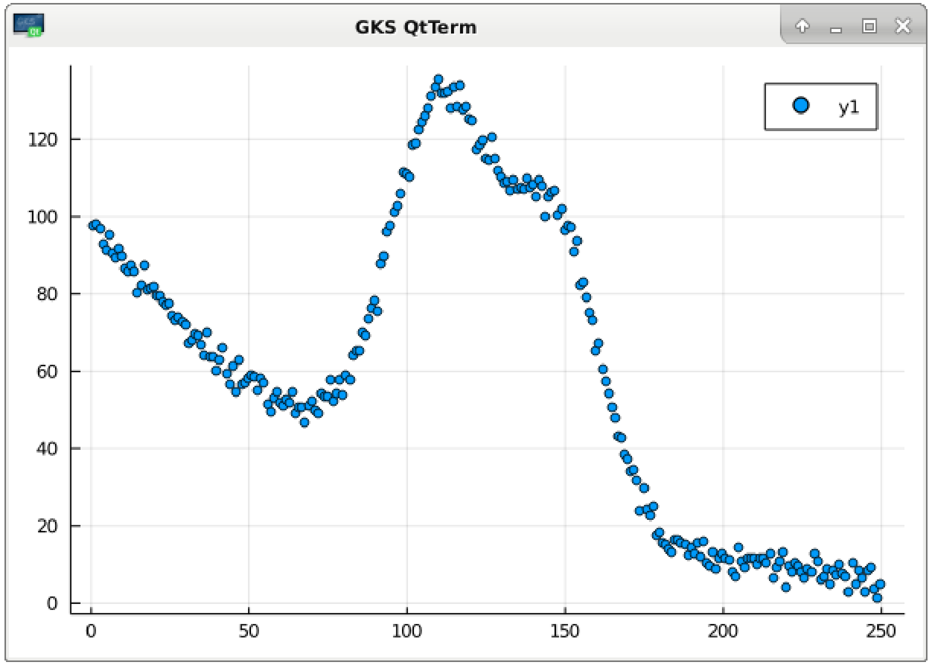
Example: Given the observation data (right, blue dots) and a model containing 8 parameters (dim D=8),

$$\begin{aligned} y &= f(x;p) \\ &= p_1 \exp(-p_2 x) \\ &+ p_3 \exp[-(x-p_4)^2/p_5^2] \\ &+ p_6 \exp[-(x-p_7)^2/p_8^2]. \end{aligned}$$

find the parameters that best fit the observation data in the least squares sense[1]

$$\min_p \|y - f(x;p)\|_2.$$

This example is extracted from a work by Armin Sobhani, Ge Baolai and Pawel Pomorski.

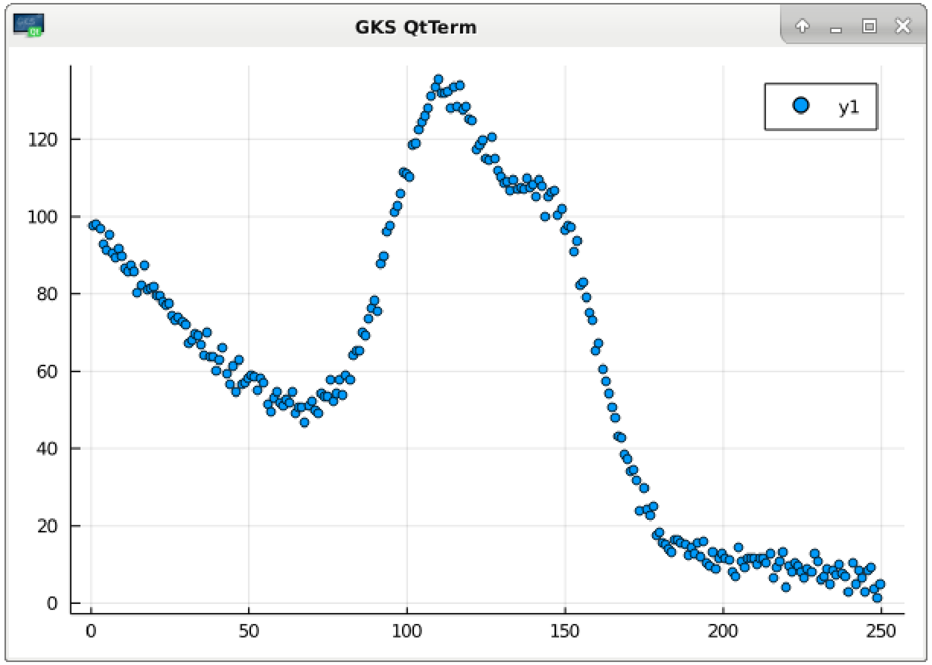


[1] https://www.itl.nist.gov/div898/strd/nls/data/gauss3.shtml

Parallel computing: Multi-threading

Example: Given the observation data (right, blue dots) and a model containing 8 parameters (dim D=8),

$$\begin{aligned} y &= f(x;p) \\ &= p_1 \exp(-p_2 x) \\ &+ p_3 \exp[-(x-p_4)^2/p_5^2] \\ &+ p_6 \exp[-(x-p_7)^2/p_8^2]. \end{aligned}$$



find the parameters that best fit the observation data in the least squares sense[1]

$$\min_p \|y - f(x;p)\|_2.$$

Solution: Find the 8 parameters using Monte-Carlo approach.

[1] https://www.itl.nist.gov/div898/strd/nls/data/gauss3.shtml

Parallel computing: Multi-threading

Example: Non-linear fitting (cont'd) using a Monte-Carlo method:

1) Generate N points of p (of dim D=8) , with each element of p uniformly distributed in the corresponding dimension within its range;

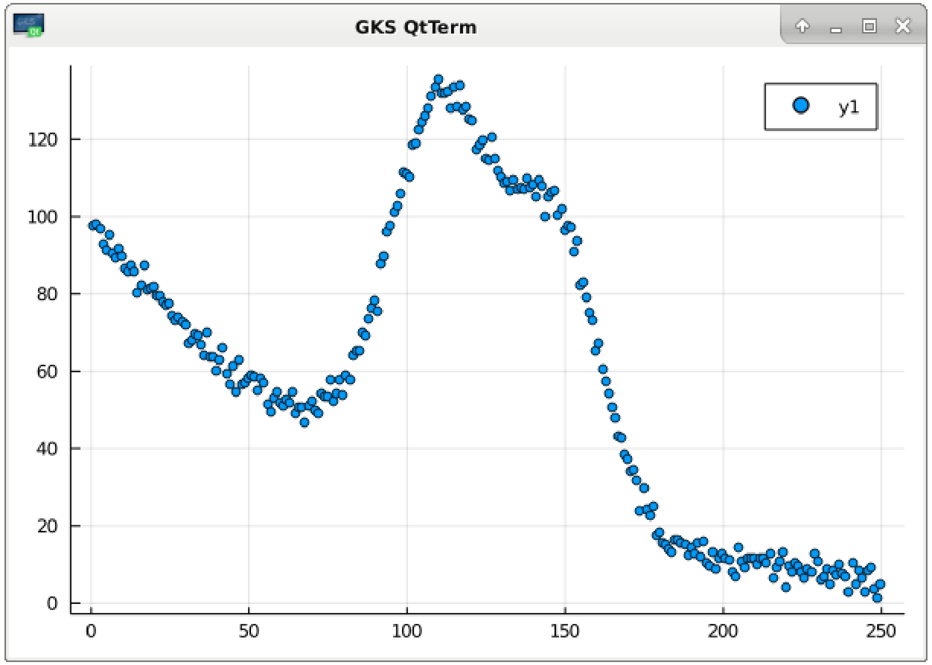
2) For each point p, compute the error (a scalar)

   z[i] = ||y – f(x;p)||, i=1...N

3) Sort z in ascending order, pick the first M corresponding points p as new candidates; find the minimum z_min (and the best p);

4) Adjust the range for each dimension of p that encloses the M selected candidates;

5) If z_min <= tol && iter <= num_iters STOP; else GOTO 1).



$$
\begin{aligned}
y &= f(x;p) \\
&= p_1 \exp(-p_2 x) \\
&+ p_3 \exp[-(x-p_4)^2/p_5^2] \\
&+ p_6 \exp[-(x-p_7)^2/p_8^2].
\end{aligned}
$$

Parallel computing: Multi-threading

Example: Non-linear fitting (cont'd) using a Monte-Carlo method:

1) Generate N points of p (of dim D=8) , with each component of p uniformly distributed in corresponding dimension within its range;
2) For each point p, compute the error (a scalar)

   z[i] = ||y – f(x;p)||, i=1...N

3) Sort z in ascending order, pick the first M corresponding points p as new candidates; find the minimum z_min (and the best p);
4) Adjust the range for each dimension of p that embraces the M selected candidates;
5) If z_min <= tol && iter <= num_iters STOP; else GOTO 1).

Sketch of serial code

```
while (z_min > tol && iters <= num_iters)
   # Generate N parameter points params[D,N]
   for i in 1:N
      params[:,i] .= llims .+ rand(D).*intervals;
      z[i] = costfun(y,x,params[:,i],ym[:,i]);
   end
   … ...
end
```

$$\text{param[1:D,1:N]} = \begin{bmatrix} p_1^{(1)} & p_1^{(2)} & \cdots & \cdots & p_1^{(N)} \\ p_2^{(1)} & p_2^{(2)} & \cdots & \cdots & p_2^{(N)} \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ p_D^{(1)} & p_D^{(2)} & \cdots & \cdots & p_D^{(N)} \end{bmatrix}$$

Parallel computing: Multi-threading

Example: Non-linear fitting (cont'd) using a Monte-Carlo method:

1) Generate N points of p (of dim D=8) , with each component of p uniformly distributed in corresponding dimension within its range;
2) For each point p, compute the error (a scalar)

   z[i] = ||y – f(x;p)||, i=1...N
3) Sort z in ascending order, pick the first M corresponding points p as new candidates; find the minimum z_min (and the best p);
4) Adjust the range for each dimension of p that embraces the M selected candidates;
5) If z_min <= tol && iter <= num_iters STOP; else GOTO 1).

Sketch of serial code

```
while (z_min > tol && iters <= num_iters)
   # Generate N parameter points params[D,N]
   for i in 1:N
      params[:,i] .= llims .+ rand(D).*intervals;
      z[i] = costfun(y,x,params[:,i],ym[:,i]);
   end

   # Sort the vector z and find the smallest one
   iz_sorted[:] = sortperm(z);
   iz_min = iz_sorted[1]; z_min = z[iz_sorted[1]];
   … ...
end
```

Parallel computing: Multi-threading

Example: Non-linear fitting (cont'd) using a Monte-Carlo method:

1) Generate N points of p (of dim D=8) , with each component of p uniformly distributed in corresponding dimension within its range;
2) For each point p, compute the error (a scalar)

   z[i] = ||y – f(x;p)||, i=1...N
3) Sort z in ascending order, pick the first M corresponding points p as new candidates; find the minimum z_min (and the best p);
4) Adjust the range for each dimension of p that embraces the M selected candidates;
5) If z_min <= tol && iter <= num_iters STOP; else GOTO 1).

Sketch of serial code

```
while (z_min > tol && iters <= num_iters)
   # Generate N parameter points params[D,N]
   for i in 1:N
      params[:,i] .= llims .+ rand(D).*intervals;
      z[i] = costfun(y,x,params[:,i],ym[:,i]);
   end

   # Sort the z; pick the first M corresponding points of p
   iz_sorted[:] = sortperm(z);
   iz_min = iz_sorted[1]; z_min = z[iz_sorted[1]];
   elite_view = view(params,:,iz_sorted[1:num_elites]);

   # Update the range of each of the parameters
   llims[:] = minimum(elite_view,dims=2);
   uims[:] = maximum(elite_view,dims=2);
   intervals .= ulims .- llims;
   iter += 1;
end
```

Parallel computing: Multi-threading

Example: Non-linear fitting (cont'd) using a Monte-Carlo method:

1) Generate N points of p (of dim D=8) , with each component of p uniformly distributed in corresponding dimension within its range;

2) For each point p, compute the error (a scalar)

   z[i] = ||y – f(x;p)||, i=1...N

3) Sort z in ascending order, pick the first M corresponding points p as new candidates; find the minimum z_min (and the best p);

4) Adjust the range for each dimension of p that embraces the M selected candidates;

5) If z_min <= tol && iter <= num_iters STOP; else GOTO 1).

Sketch of parallel code using threads

```
while (z_min > tol && iters <= num_iters)
   # Generate N parameter points params[D,N]
   @threads for i in 1:N
      params[:,i] .= llims .+ rand(D).*intervals;
      z[i] = costfun(y,x,params[:,i],ym[:,i]);
   end

   # Sort the z; pick the first M corresponding points of p
   iz_sorted[:] = sortperm(z);
   iz_min = iz_sorted[1]; z_min = z[iz_sorted[1]];
   elite_view = view(params,:,iz_sorted[1:num_elites]);

   # Update the range of each of the parameters
   llims[:] = minimum(elite_view,dims=2);
   uims[:] = maximum(elite_view,dims=2);
   intervals .= ulims .- llims;
   iter += 1;
end
```

Parallel computing: Starting multiple threads

From command line (ver 1.5 and newer)

```
julia -t 8
```

or

```
julia --threads 8
```

Or via environment variable

```
export JULIA_NUM_THREADS=8
```

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/.

[4] Jeff Bezanson, Stefan Karpinski, *State of Julia*, JuliaCon 2020 (YouTube).