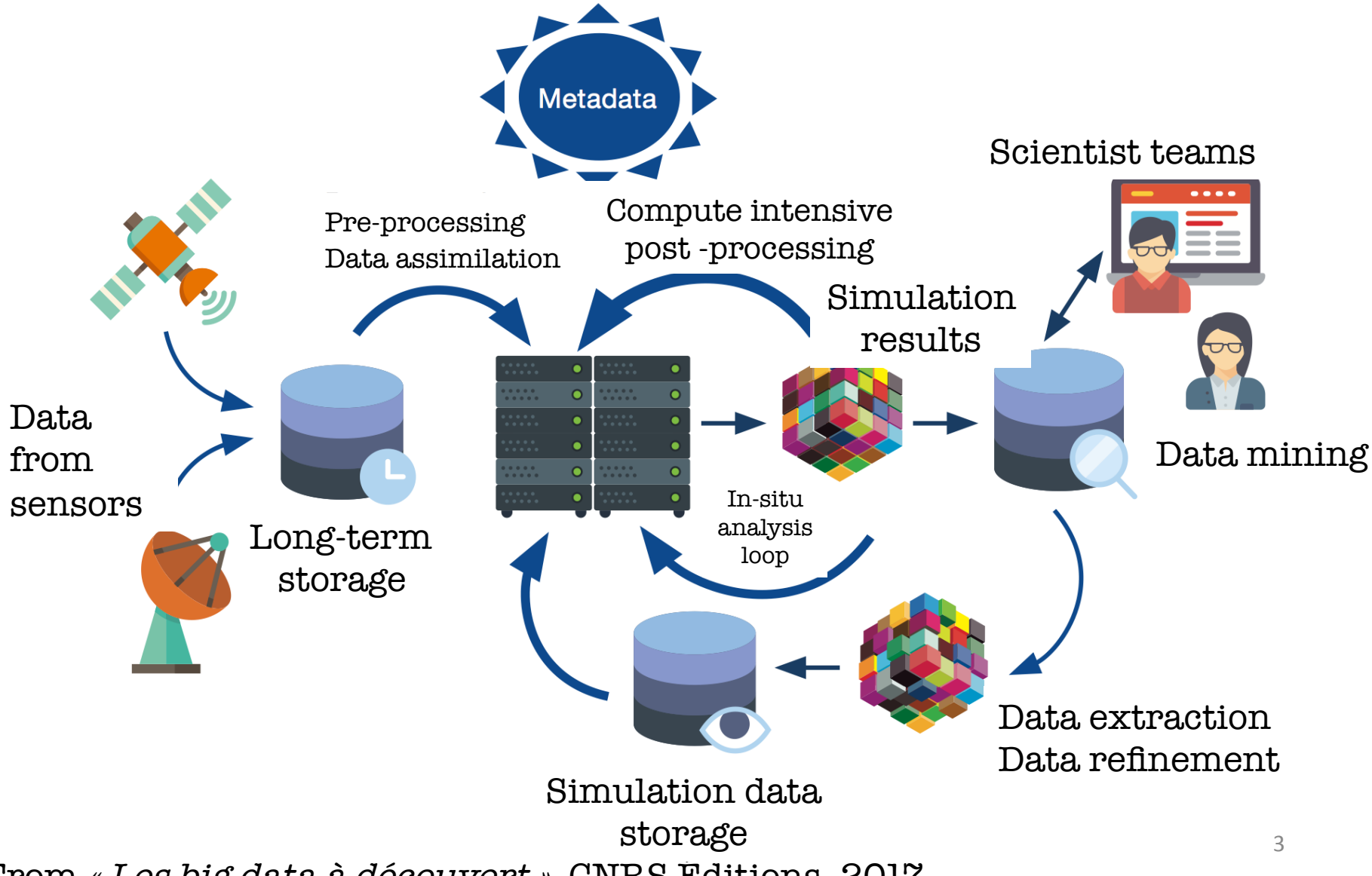
Le langage Julia pour le calcul scientifique

François Bodin
University of Rennes / IRISA

Introduction

- Convergence computing and data in the "Exascale" context implies deploying
  - In-situ data analytic
  - Complex workflows/dataflows composed of heterogeneous set of tasks
  - Domain Specific Languages (DSL)
  - etc.
- The use of a language such as Julia maybe part of the solution

Introduction cont.
Complex Workflows



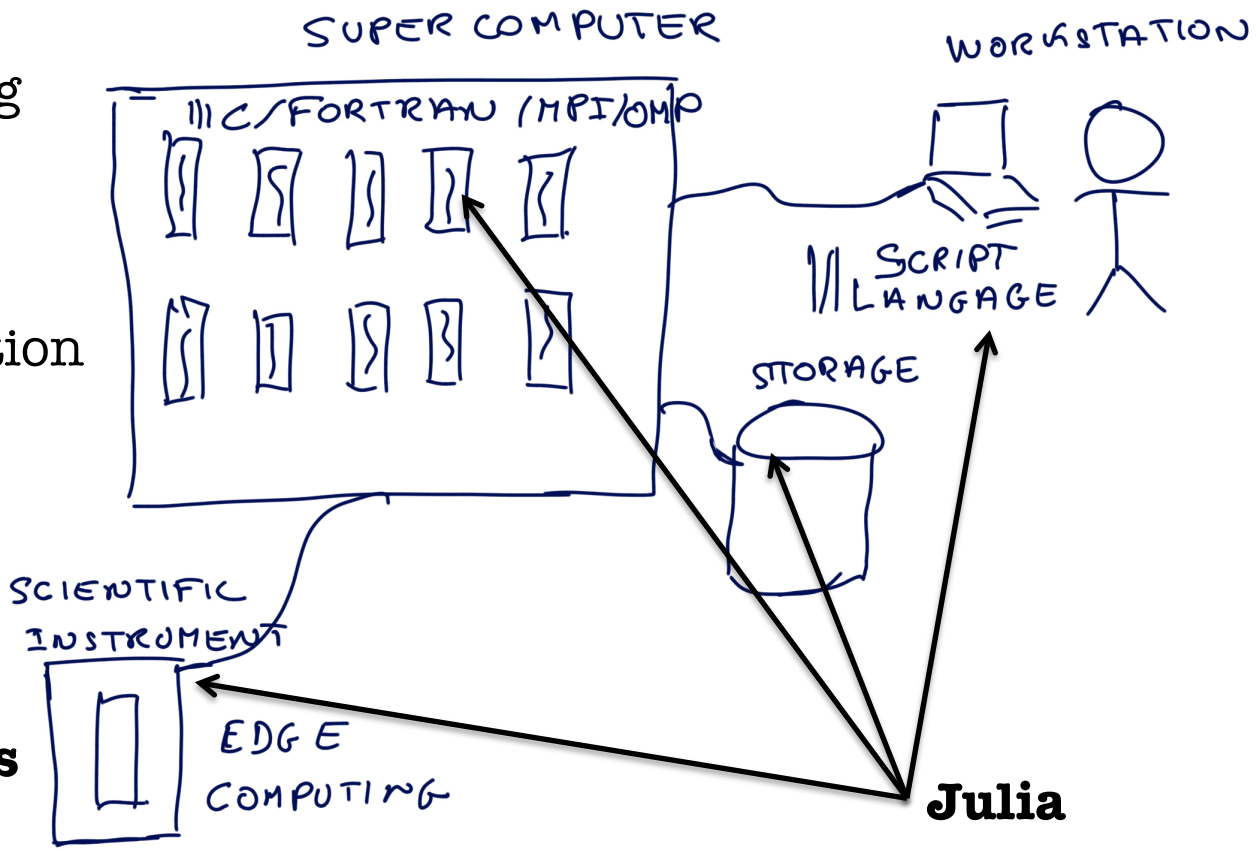
From « *Les big data à découvert* », CNRS Editions, 2017

Introduction cont.

- This talk aims at showing how a language such as Julia may help while “standard” languages are usually a must

In-situ analysis
Application monitoring
Edge computing
Data mining
Pre/post processing
Workflow implementation
In-transit computing

Many of this tasks are volatile in nature to adjust to scientist’s needs



Outline

- The “Exascale” context
- Julia
- The Workshop Agenda

Petascale to Exascale

- Petascale to Exascale transition is raising many issues
  - Not only related to technology
  - Not happening in isolation
  - In a context of scientific (observational) data deluge
- Well summarized in the USA National Strategic Computing Initiative (NSCI)
  - "NSCI seeks to drive the convergence of compute-intensive and data-intensive systems"
- We are potentially on a paradigm change denoted Exascale but meaning computing *generation transition*

Why Peta-Exa is Not Similar to Tera-Peta Transition?

- The main Tera-Peta transition was performed before during the Giga-Tera transition
  - Adaption of codes to distributed memory machines
  - Tera to Peta was smooth and with minimum (side-) effects for most HPC users
- Data issue is changing the game for Peta-Exa
  - New software stack and algorithms
  - Questions the discovery process (e.g The Fourth Paradigm)
  - Data analytics and machine learning
  - Data localization

What Exascale is Not

- Exascale == 10^{18} flops of interest for a small community
  - Such as LQCD and field based on embarrassingly parallel methods (e.g. Monte-Carlo)
- Exascale transition for most people is not about the next increment in machine features
  - The next generation of machines is likely to create a practice and organization disruption
  - It is easy to compute anywhere (c.f. PRACE) but moving data around is (very) slow
  - Adherence to a system (including storage and networking) is likely to increase

Exascale-Wise Applications Characterization*

1. Workload
2. Workflow
3. Code
4. Scalability
5. Operating System
6. I/O
7. HPC Community
8. Hardware
9. Visualization
10. Interactivity
11. Data management and analysis
12. Impact on Science/ Society

*Computer science point of view

Exascale Transition Impact on Workflows

	Tera-to-Peta	Peta-to-Exa
Complexity	Code coupling	More multiphysics, multiphase models, data assimilations, data analytics, edge computing, …
Heterogeneity	Mostly homogeneous	**Mix of data analytics and simulation, heterogeneous bricks**
Localization	All in one system	Data may come from large scientific instruments, or a large number of small instruments
I/O constrained	Solvable issue	Cannot move the data around, not sure it can be solved
Allocation	Batch mostly	Batch, **interactive (guided simulation and analysis)**, (soft) real-time* (visualization, …)

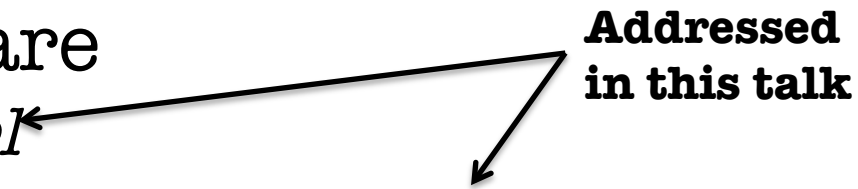
*Big data assimilation for Extreme-scale NWP, Takemasa Miyoshi

Julia

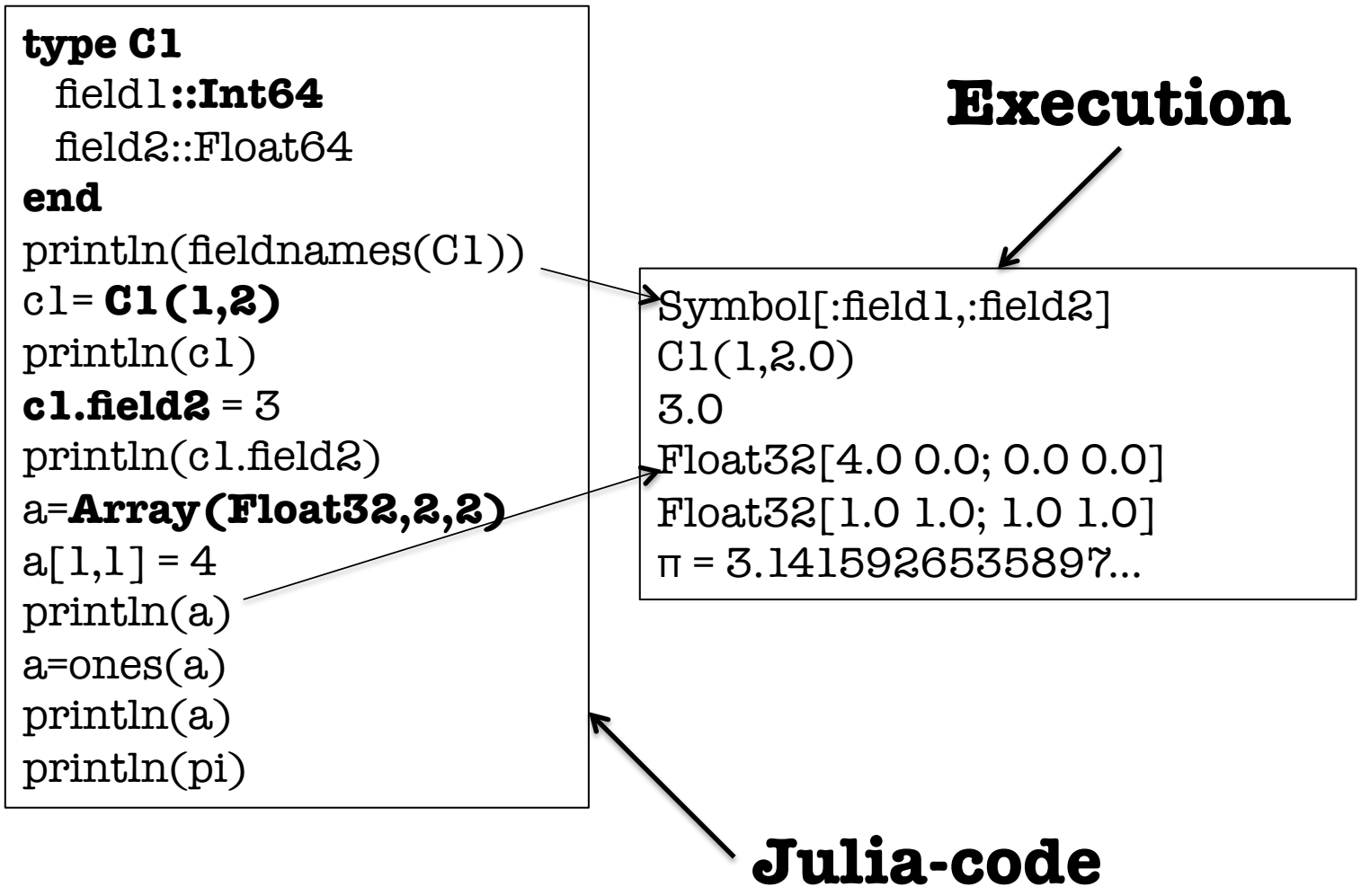
Julia's Interpreted Language

- Three main potential uses of Julia
  - As a glue for tasks in a workflow
  - As a (online) tool to run inside simulation
  - As a user tool on scientist's workstation
  - As a tool for Domain Specific Languages (DSL)

- Then requirements are
  - *Simple parallel model*
  - *Interoperability with C and Fortran*
  - "Good enough" performance (LLVM JIT)
  - Meta-programming capabilities



Some Julia's Types



Julia's Functions

```
function f1(y::Int64)
   local x::Int8 = 100+y
   return x
end
function f1(y::Float64)
   local x::Float64 = 100+y
end
function f1(y=1)
   local x = 100+y
end

println(methods(f1))

println(typeof(f1))
println(typeof(f1(5)))
println(f1(7.0))
println(f1())
```

Julia-code

Execution

```
#4 methods for generic function "f1":
f1() at /.../example2.jl:9
f1(y::Float64) at /.../example2.jl:6
f1(y::Int64) at /.../example2.jl:2
f1(y) at /.../example2.jl:9
#f1
Int8
107.0
101
```

Julia's Loops

```
i=1
while i <= 5
  println(i)
  i += 1
end
for i = 1:5
 if i == 1
  println("-> $(i)" * "st ite")
 else
  println("-> $(i)" * "th ite")
 end
end
```

Julia-code

Execution

```
1
2
3
4
5
-> 1st ite
-> 2th ite
-> 3th ite
-> 4th ite
-> 5th ite
```

Parallelism in Julia

- Parallel programming in Julia is built on two main primitives
  - remote references
  - remote calls
- Communication in Julia is generally "one-sided"
- Cluster-wide

Julia's Channel

Julia-code

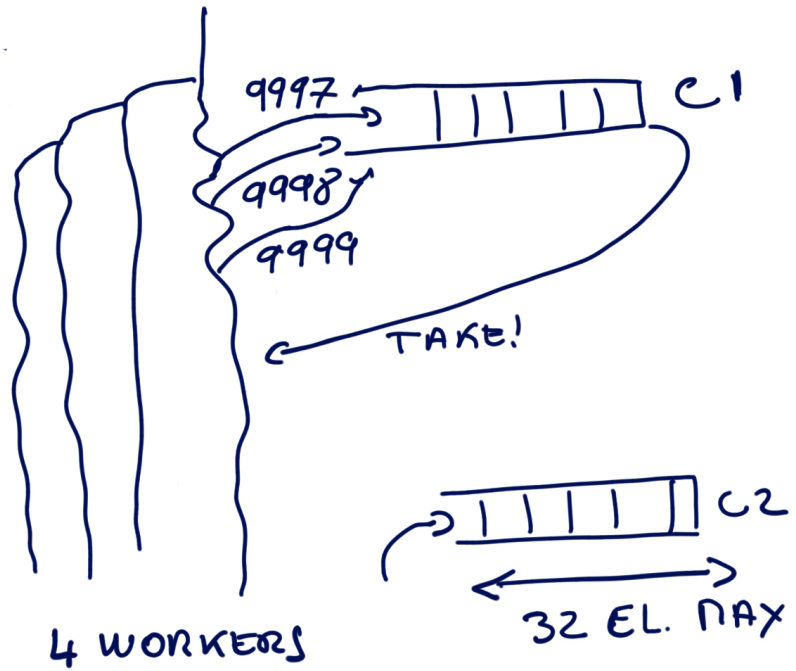
```
addprocs(3)
nw = nprocs()
println("Number of workers is $(nw)")

c1 = RemoteChannel(()->Channel(32))
c2 = RemoteChannel(()->Channel(32))

put!(c1,9997)
put!(c1,9998)
put!(c1,9999)

while isready(c1)
  println(take!(c1))
end

put!(c1,7777)
put!(c1,7778)
put!(c1,7779)
```



Execution

```
Number of workers is 4
9997
9998
9999
```

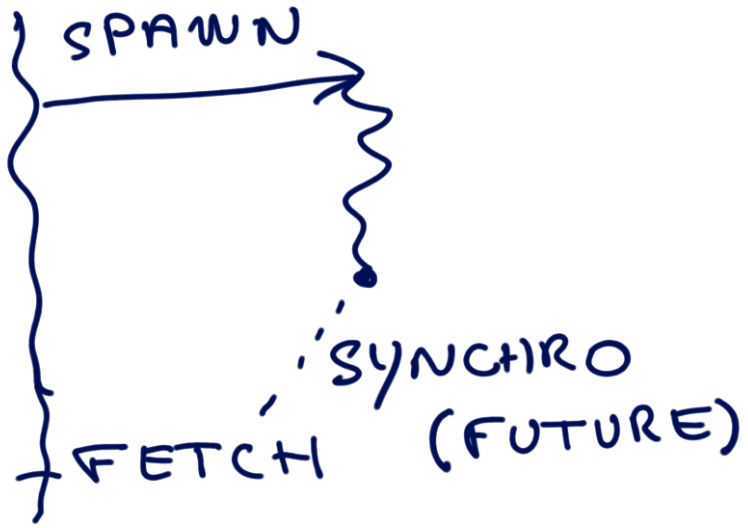
Spawning and Remote Accesses

Julia-code

```
#function to be executed remotely
@everywhere function rmfc(cin,cout)
   global id=myid()
   local vc = "empty"
   if (isready(cin))
     vc = take!(cin)
   end
   put!(cout,"channel v$(id)")
   println("Running on worker $(id),
      cin channel value is $(vc)")
   sleep(1)
   return id
end
rc11 = @spawnat 2 rmfc(c1,c2)
rc12 = @spawnat 3 rmfc(c1,c2)
rc13 = @spawnat 4 rmfc(c1,c2)
println("Worker 2 id is
    $(remotecall_fetch(()->id, 2))")
r1 = fetch(rc11)
r2 = fetch(rc12)
r3 = fetch(rc13)
println("1-Results $(r1) $(r2) $(r3)")
```

Execution

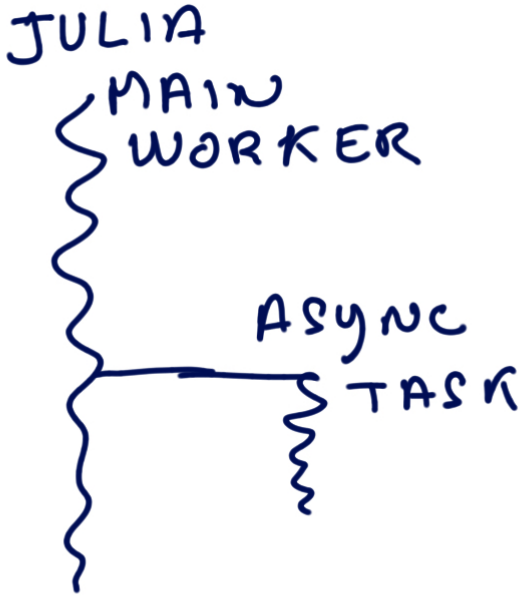
```
Worker 2 id is 2
      From worker 4: Running on worker 4,
      From worker 4: cin channel value is 7779
      From worker 3: Running on worker 3,
      From worker 2: Running on worker 2,
      From worker 3: cin channel value is 7778
      From worker 2: cin channel value is 7777
1-Results 2 3 4
```



Asynchronous Tasks

```
@everywhere function asyncfc()
   local id=myid()
   println("Async running on worker $(id)")
   sleep(2)
   return id
end

@async begin
      asyncfc()
end
println("After the async call")
```



Julia-code

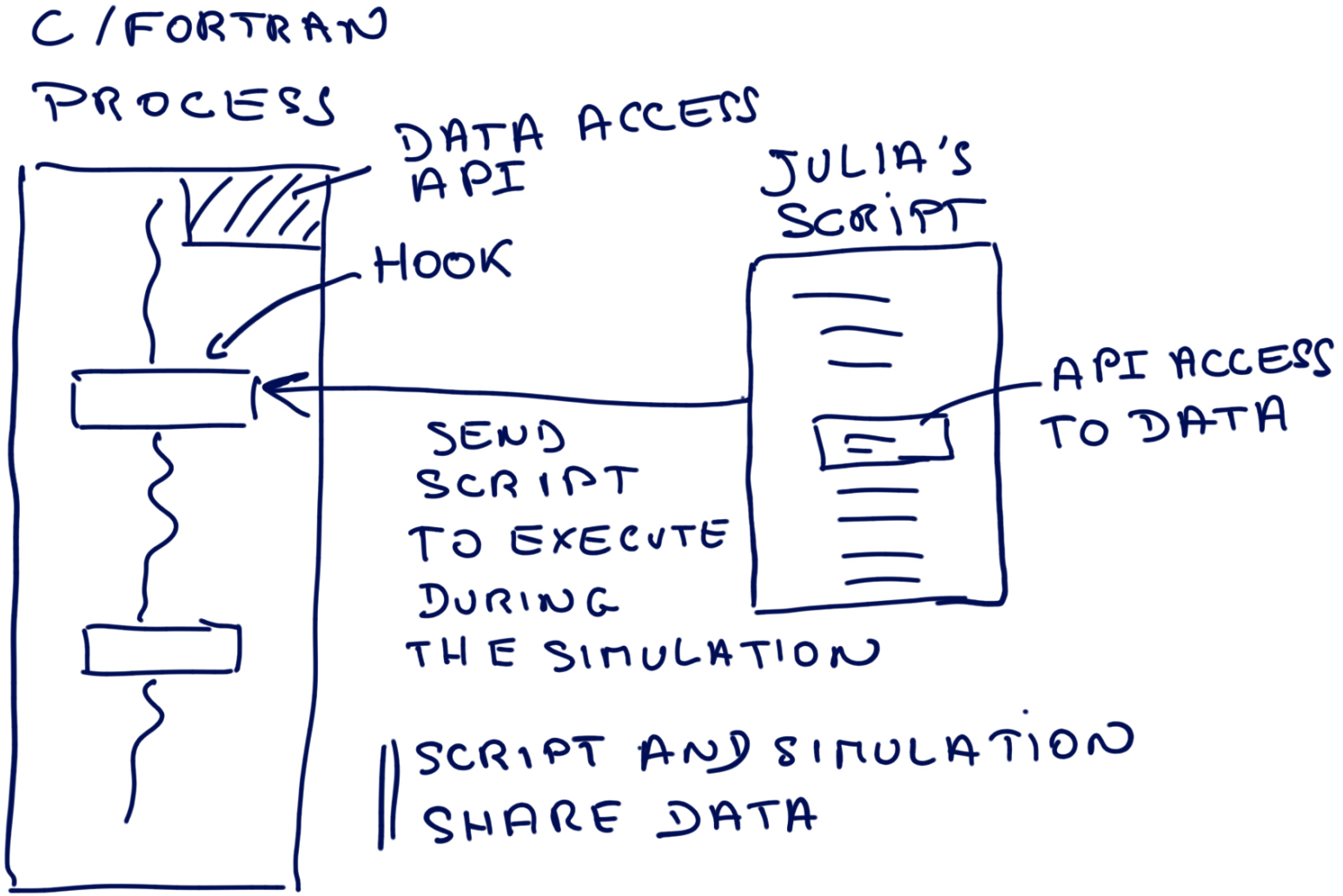
Execution

```
After the async call
Async running on worker 1
```

Julia's Interoperability-1

- Julia in C/Fortran codes
  - Add scripting capabilities in usual C/ Fortran simulation to interact/drive/ extract data at runtime
  - Implement flexible monitoring
- Julia's calling C/Fortran codes
  - Scripting as a glue to call optimize C/ Fortran codes

Example:
Runtime In-situ Data Inspection

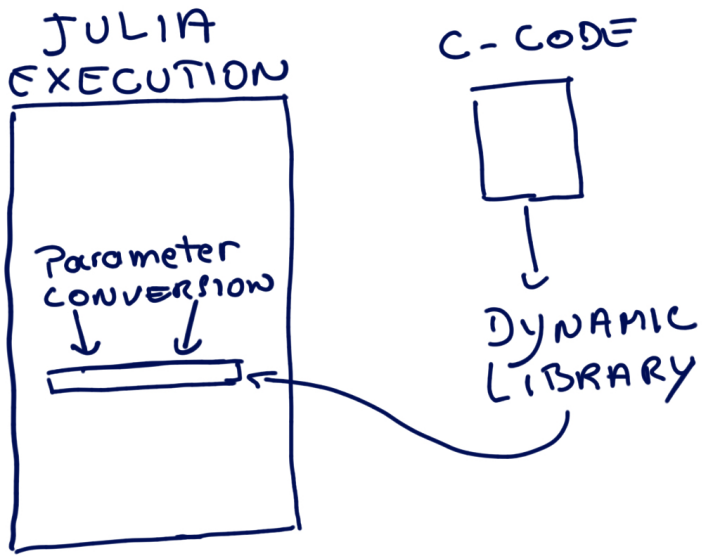


C-Code in Julia's Code

C-code

```
#include<stdio.h>

void cfunc(int n, char *s){
  if (!s) printf("Null pointer\n");
  printf("Hello world from C : %d %s\n",n,s);
}
```



Execution

```
Hello world from C : 3 data
```

Julia-code

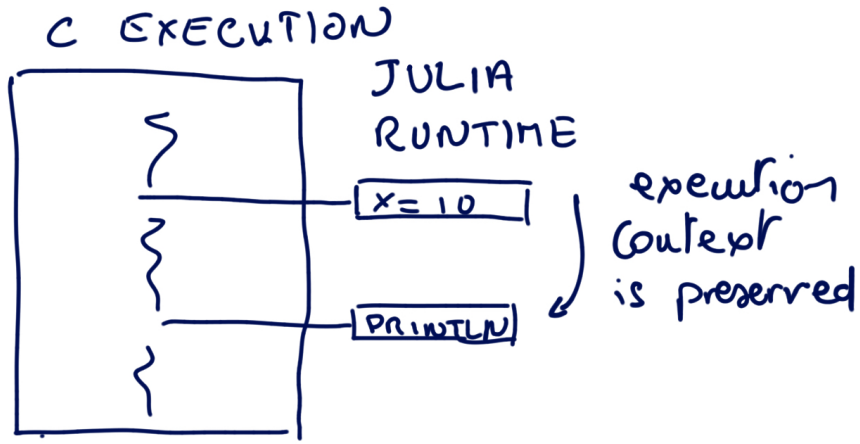
```
#Calling c function
ccall(("cfunc","libsomeccode"),Void, (Int32,Cstring),3,"data")
```

Julia-Code in C-Code

```
#include <julia.h>
#include<stdio.h>

int main(int argc, char *argv[]) {
   jl_init(JULIA_INIT_DIR);
   l_value_t *ret1 = jl_eval_string("x=10");
   jl_value_t *ret2 = jl_eval_string("println(\"value of x is $(x)\")");
   jl_atexit_hook(0);
   return 0;
}
```

C-code



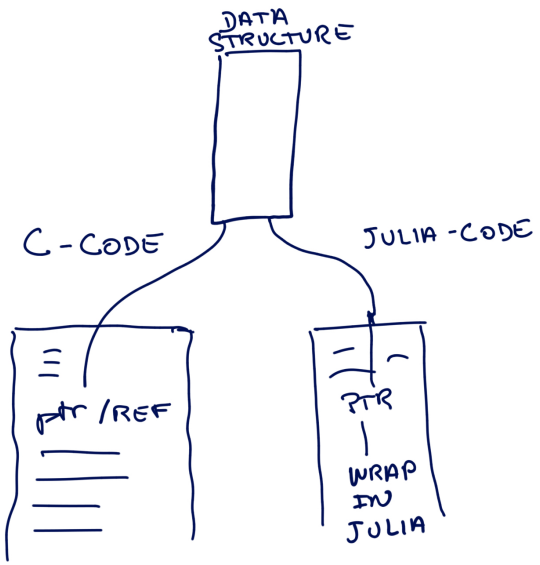
Execution

value of x is 10

Sharing Data Structures

```
#include <julia.h>
#define ASIZE 5
double A[ASIZE];
int arraysize(){
 return ASIZE;
}
double *cfunc2(int n, char *s){
 printf("cfunc2 - Hello world from C : %d %s\n",n,s);
 return A;
}
```

C-code




```
int main(int argc, char *argv[]) {
 for(int i=0;i<ASIZE; i++) A[i] = 1.0;
 printf("Initial values for A\n");
 for(int i=0;i<ASIZE; i++) printf("%f\n",A[i]);
 jl_init(JULIA_INIT_DIR);
 jl_eval_string("include(\"myjuliascript.jl\")\n");
 printf("New values for A\n");
 for(int i=0;i<ASIZE; i++) printf("%f\n",A[i]);
 jl_atexit_hook(0);
 return 0;
}
```

Sharing Data Structures cont.

Julia-code
(myjuliascript.jl)

```
asize = ccall(("arraysize",),Int32,(),)
println("size of the array is $(asize)")
ptr = ccall(("cfunc2",),Ptr{Cdouble},
    (Int32,Cstring),3,"data")
println(ptr)
println(unsafe_load(ptr,1))
unsafe_store!(ptr,777.0,4)
A = unsafe_wrap(Array{Cdouble,1},
     ptr,(asize))
A[1] = 999.0
println(A)
```

Execution

```
Initial values for A
1.000000
1.000000
1.000000
1.000000
1.000000
size of the array is 5
cfunc2 - Hello world from C : 3 data
Ptr{Float64} @0x0000000100754030
1.0
[999.0,1.0,1.0,777.0,1.0]
New values for A
999.000000
1.000000
1.000000
777.000000
1.000000
```

C

C

Other Julia's Topics I Did Not Address

- Parallel Loops & Parallel Arrays
- Meta-Programming
- Accelerators
- …

Conclusion

- Heterogeneity in software inevitable to cover all functionalities

- Julia may cover many of the needs
  - Scripting inside C/Fortran applications
  - Gluing tasks in a workflows
  - Parallel programming
  - DSL building

The Workshop Agenda

Outline

- **Utiliser Python dans le cadre du HPC et de l'analyse de données**
  - *Asma Farjallah, INTEL*
- **Environnement logiciel pour l'apprentissage profond dans un contexte HPC**
  - *Gunter Roth, NVIDIA*
- **Orchestration et distributions de flux de tâches à l'aide d'une extension de docker swarm et d'un langage dédié construit à l'aide de SIRIUS**
  - *Olivier BARAIS, UNIVERSITÉ DE RENNES 1*
- **Les nouveaux défis de l'analyse de données climatiques**
  - *Christian PAGÉ, CERFACS*
- **Réduire le temps et le coût des échanges de données en améliorant l'allocation de ressources réalisée par SLURM**
  - *Thomas CADEAU and Yiannis GEORGIOU, BULL-ATOS*