The Julia `MPI` wrapper

March 24th, 2021

Overview

Introduction to MPI

What is MPI ?

- "Message Passing Interface (MPI) is a standardized and portable message-passing standard", *Wikipedia*.
- MPI is a library not a language : it is written in C,C++ and Fortran.
- MPI processes are like different jobs on Unix

What do I use it for ? : Distributed computing and memory. On large HPC clusters : MPI for distributing between nodes, shared memory on each node.

Why MPI when you have Julia multiprocessing ?

- "MPI remains the dominant model used in high-performance computing today", *Wikipedia*.
- Easy translation from another language, wider community.
- Portability

MPI.jl : Tools

What is MPI.jl ?

- A Julia Wrapper for MPI in C
- Inspiration is taken from mpi4py

Ressources

- https://juliaparallel.github.io/MPI.jl/stable/
- https://github.com/JuliaParallel/MPI.jl
- http://www.idris.fr/formations/mpi/ (Pure MPI tutorial and courses)
- https://people.bordeaux.inria.fr/coulaud/Enseignement/PG305/pg305-MPI.pdf (pdf in french, a lot of resources in pure MPI)

MPI.jl : Installation

How to install on local ?

- You need MPI (OpenMPI, MPICH...),
- Add MPI.jl and build (commands on juliaparallel.github.io)
- Configure julia execution wrapper mpiexecjl

What about notebooks ? And why we wont use it! ;)

You cannot actually use directly MPI , you need MPIClusterManagers, Distributed and then use Distributed with MPI commands i.e. not actual MPI syntax.

Hello world !

```julia
using MPI
MPI.Init() # mandatory to initialize MPI

# the communicator containing all the processes
comm = MPI.COMM_WORLD

# print the rank (=id) of the process
println("Hello world, I am $(MPI.Comm_rank(comm))
         of $(MPI.Comm_size(comm))")

# wait for all processes to reach this point
MPI.Barrier(comm)

# optional, automatically called when Julia exits
MPI.Finalize()
```

```
mpiexecjl -n 4 julia hello_world.jl
```

```
Hello world, I am 2 of 4
Hello world, I am 3 of 4
Hello world, I am 0 of 4
Hello world, I am 1 of 4
```

First communication : send / receive

Sending a message from a process to another:

```
Send(obj, dst::Integer, tag::Integer, comm::Comm)
```

Receiving a message from a process:

```
Recv(::Type{T}, src::Integer, tag::Integer, comm::Comm)
```

→ returns a tuple of the object of type `T` and the status of the receive.

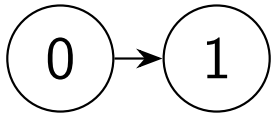
```
Recv!(data, src::Integer, tag::Integer, comm::Comm)
```

→ returns the status of the receive, but data has to be a `Buffer` or any object for which `Buffer(data)` is defined (typically arrays, but not floats or integers) ! (see `https://juliaparallel.github.io/MPI.jl/stable/buffers/#MPI.Buffer` for more details)

Example 1

```julia
using MPI
MPI.Init()

# the communicator containing all the processes
comm = MPI.COMM_WORLD
# number of processes
numprocs = MPI.Comm_size(comm)
# id of the process
procid = MPI.Comm_rank(comm)

# this test is only for 2 processes
if numprocs != 2
    MPI.Abort(comm, 0)
end
```

```
# process 0 sends data to process 1
if procid == 0
    pipi = 3.14
    MPI.Send(pipi, 1, 0, comm)
    println("ProcID $(procid) sent value
            $(pipi) to ProcID 1")
end

# process 1 receives data from process 0
if procid == 1
    pipi, status = MPI.Recv(Float64, 0, 0, comm)
    println("ProcID $(procid) received value
            $(pipi) from ProcID 0")
end

MPI.Finalize()
```



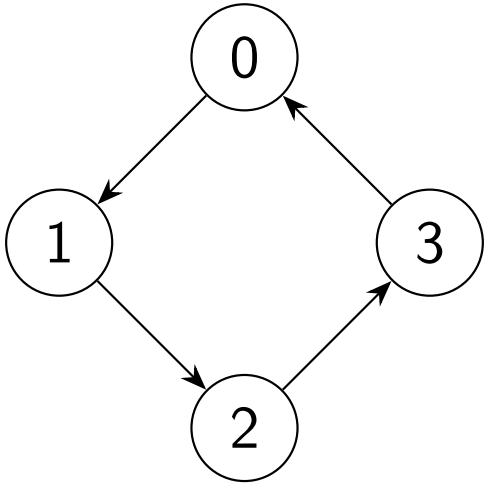

```
mpiexecjl -n 2 julia send_receive.jl
```

```
ProcID 0 sent value 3.14 to ProcID 1
ProcID 1 received value 3.14 from ProcID 0
```

Example 2

```
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
numprocs = MPI.Comm_size(comm)
procid = MPI.Comm_rank(comm)

dst = mod(procid+1, numprocs) # destination for sending
src = mod(procid-1, numprocs) # source for receiving

N = 3
send_mesg = Array{Float64}(undef, N)
recv_mesg = Array{Float64}(undef, N)
# fill the message to send with the procid
fill!(send_mesg, Float64(procid))
```

```
# sending message
println("ProcID $(procid) sending $(send_mesg)
        to ProcID $(dst)")
MPI.Send(send_mesg, dst, procid+32, comm)

# receiving message
MPI.Recv!(recv_mesg, src, src+32, comm)
println("ProcID $(procid) received $(recv_mesg)
        from ProcID $(src)")

MPI.Barrier(comm)
MPI.Finalize()
```




```
mpiexecjl -n 4 julia send_receive_n.jl
```

```
ProcID 1 sending [1.0, 1.0, 1.0] to ProcID 2
ProcID 3 sending [3.0, 3.0, 3.0] to ProcID 0
ProcID 2 sending [2.0, 2.0, 2.0] to ProcID 3
ProcID 0 sending [0.0, 0.0, 0.0] to ProcID 1
ProcID 3 received [2.0, 2.0, 2.0] from ProcID 2
ProcID 1 received [0.0, 0.0, 0.0] from ProcID 0
ProcID 2 received [1.0, 1.0, 1.0] from ProcID 1
ProcID 0 received [3.0, 3.0, 3.0] from ProcID 3
```

Broadcasting

Broadcasting is used to send a message from one process to all others, typically when you load data, collect user entries, ...

```
Bcast!(buf, root::Integer, comm::Comm)
```

→ broadcasts the buffer `buf` from `root` to all processes in `comm`. Works only with buffer compatible objects !

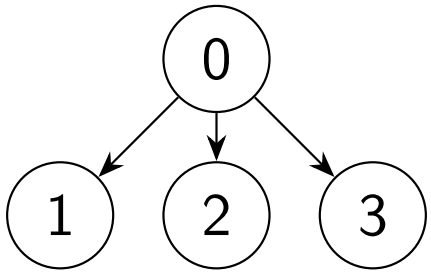
```
bcast(obj, root::Integer, comm::Comm)
```

→ broadcasts the object `obj` from `root` to all processes in `comm`, returns `obj` to the current process. This is able to handle arbitrary data.

Example

```
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
N = 5
root = 0

if MPI.Comm_rank(comm) == root
    println("Running on $(MPI.Comm_size(comm)) processes")
end
MPI.Barrier(comm)
```

```
if MPI.Comm_rank(comm) == root
    A = [Float64(i) for i = 1:N]
else
    A = Array{Float64}(undef, N)
end
# broadcast with buffer compatible object
MPI.Bcast!(A, root, comm)
println("rank = $(MPI.Comm_rank(comm)), A = $A")

if MPI.Comm_rank(comm) == root
    B = Dict("foo" => "bar")
else
    B = nothing
end
# brodcast without buffer compatible object
B = MPI.bcast(B, root, comm)
println("rank = $(MPI.Comm_rank(comm)), B = $B")
```



mpiexecjl -n 4 julia broadcast.jl

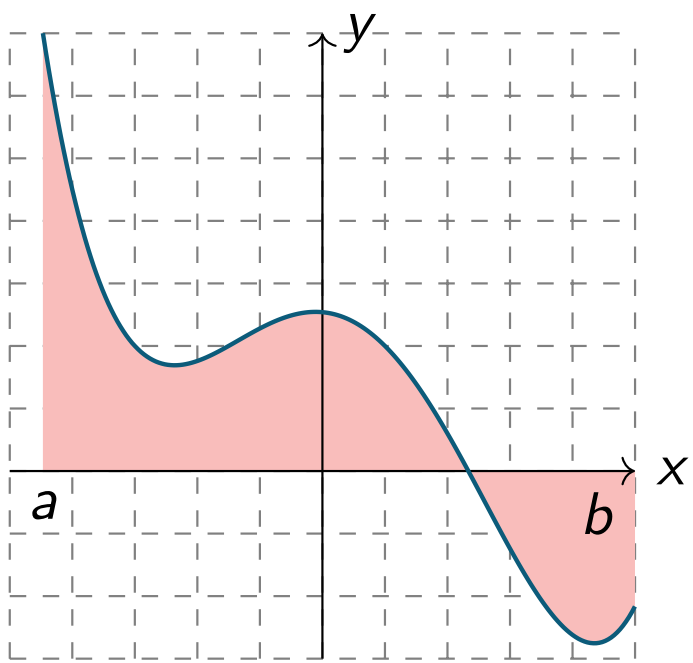
```
Running on 4 processes
rank = 2, A = [1.0, 2.0, 3.0, 4.0, 5.0]
rank = 1, A = [1.0, 2.0, 3.0, 4.0, 5.0]
rank = 3, A = [1.0, 2.0, 3.0, 4.0, 5.0]
rank = 0, A = [1.0, 2.0, 3.0, 4.0, 5.0]
rank = 0, B = Dict("foo" => "bar")
rank = 2, B = Dict("foo" => "bar")
rank = 1, B = Dict("foo" => "bar")
rank = 3, B = Dict("foo" => "bar")
```

Standard integration

Trapezoid rule for integration of a function f:

$$\int_a^b f \approx \delta x \times \left(\sum_{k=1}^{N-1} f(x_k) + \frac{f(x_0) + f(x_N)}{2} \right)$$

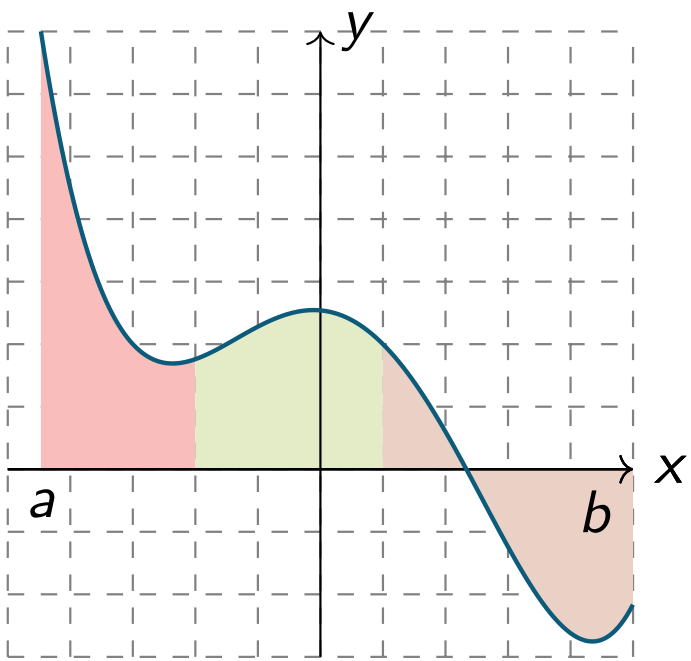
```
f(x) = ...

function trap(a, b, n, h)
    integral = (f(a) + f(b)) / 2.0
    x = a
    for i = 1:(n-1)
        x += h
        integral += f(x)
    end
    integral * h
end
```



Parallel integration

1. Each process calculates its interval of integration;
2. each process estimates the integral of f over its interval using the trapezoidal rule;
3. each process different than 0 sends its integral to 0;
4. process 0 sums the calculations received from the individual processes and prints the result.

Warning We assume that the number of trapezoids is divisible by the number of processes !



```julia
using MPI
include("integrate.jl")
# First, we define the variables common to all processes
a = 0.0                          # left endpoint
b = 1.0                          # right endpoint
n = 2048                         # number of trapezoids
h = (b-a)/n                      # integration step
dst = 0                          # destination process

MPI.Init()
comm = MPI.COMM_WORLD
procid = MPI.Comm_rank(comm)
numprocs = MPI.Comm_size(comm)

MPI.Barrier(comm)
start_time = MPI.Wtime()    # just to measure execution time
```

```
# another common variable
local_n = n / numprocs      # number of trapezoids per process
@assert mod(local_n,1) == 0 # assert that n is a multiple of
                            # the number of processes

# variables local to the process
# this process deals with interval
# [a + local_n*i*h, a + local_n*(i+1)*h]
local_a = a + local_n * procid * h
local_b = local_a + local_n*h
integral = trap(local_a, local_b, local_n, h)

# /!\ Now, each process has calculated its interval
# we can sum everything up
```

```
if procid == 0
    total = integral
    for src = 1:(numprocs-1)
        rcv_integral, status = MPI.recv(src, src+32, comm)
        println("$(procid) received integral from $(src)")
        global total
        total += rcv_integral
    end
else
    println("$(procid) sending integral to $(dst)")
    MPI.send(integral, dst, procid+32, comm)
end

MPI.Barrier(comm)
end_time = MPI.Wtime()
```

```
julia integrate_serial.jl
```

```
With 2048 trapezoids : 0.3333333730697632
Computation time = 0.009713 seconds
```

```
mpiexecjl -n 4 julia integrate_parallel.jl
```

```
2 sending integral to 0
3 sending integral to 0
1 sending integral to 0
0 received integral from 1
0 received integral from 2
0 received integral from 3
With 2048 trapezoids and 4 processes : 0.3333333730697632
Computation time = 0.410386 seconds
```

$\rightarrow$ **communication takes time !**

```
julia integrate_serial.jl
```

```
With 2048000000 trapezoids : 0.3333333553798291
Computation time = 2.470623 seconds
```

```
mpiexecjl -n 4 julia integrate_parallel.jl
```

```
2 sending integral to 0
1 sending integral to 0
3 sending integral to 0
0 received integral from 1
0 received integral from 2
0 received integral from 3
With 2048000000 trapezoids and 4 processes : 0.3333333424451355
Computation time = 1.086678 seconds
```

$\rightarrow$ not very useful case, but shows the trade-off between size of the problem and communication time.

Deadlock

What is a deadlock ?

Everyone is waiting for *something* to happen (ex. everyone receiving but no one sending, or the opposite)

```
send_message = Array{Float64}(undef, 1)
receive_container = Array{Float64}(undef, 1)

if rank == 0
    MPI.Recv!(receive_container, 1, 1, comm)
    MPI.Send(send_message, 1, 0, comm)
end
if rank == 1
    MPI.Recv!(receive_container, 0, 0, comm)
    MPI.Send(send_message, 0, 1, comm)
end
```

Solving the deadlock

Solutions

1. Check communication order

```
send_message = Array{Float64}(undef, 1)
receive_container = Array{Float64}(undef, 1)

if rank == 0
    MPI.Send(send_message, 1, 0, comm)
    MPI.Recv!(receive_container, 1, 1, comm)
end

if rank == 1
    MPI.Recv!(receive_container, 0, 0, comm)
    MPI.Send(send_message, 0, 1, comm)
end
```

Solving the deadlock

Solutions

1. Check communication order
2. SendRecv! blocking communication

```
send_message = Array{Float64}(undef, 1)
rcv_container = Array{Float64}(undef, 1)

if rank == 0
    MPI.Sendrecv!(send_message, 1, 0, rcv_container, 1, 1, comm)
end
if rank == 1
    MPI.Sendrecv!(send_message, 0, 1, rcv_container, 0, 0, comm)
end
```

Solving the deadlock

Solutions

1. Check communication order
2. `SendRecv!` blocking communication
3. Non-blocking communications. Warning : you don't know when the message is actually sent, modifying the variable is risky. You may then need (`MPI_Wait`, `MPI_Test`, `MPI_Probe`, `MPI_Cancel`...)

```
if rank == 0
    MPI.Irecv!(receive_container, 1, 1, comm)
    MPI.Isend(send_message, 1, 0, comm)
end
if rank == 1
    MPI.Irecv!(receive_container, 0, 0, comm)
    MPI.Isend(send_message, 0, 1, comm)
end
```

Load balancing

- All processes must have the same *amount of work*

```
N = 48
q = trunc(Int, N / size)
if rank == size - 1
    q += N % size
end
```

```
mpiexecjl -n 7 -oversubscribe julia load_balancing.jl
```

```
My rank is 0, I have 6 jobs to do.
My rank is 4, I have 6 jobs to do.
My rank is 5, I have 6 jobs to do.
My rank is 6, I have 12 jobs to do.
My rank is 1, I have 6 jobs to do.
My rank is 2, I have 6 jobs to do.
My rank is 3, I have 6 jobs to do.
```

Load balancing

- All processes must have the same *amount of work*

```
N = 48
q = trunc(Int, N / size)
if rank < N % size
    q += 1
end
```

```
mpiexecjl -n 7 -oversubscribe julia load_balancing.jl
```

```
My rank is 4, I have 7 jobs to do.
My rank is 0, I have 7 jobs to do.
My rank is 1, I have 7 jobs to do.
My rank is 2, I have 7 jobs to do.
My rank is 3, I have 7 jobs to do.
My rank is 6, I have 6 jobs to do.
My rank is 5, I have 7 jobs to do.
```

Smart matrix-vector product

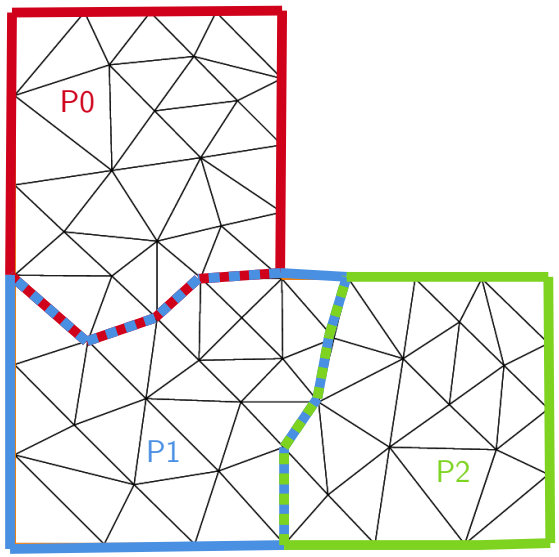
Data is supposed splitted on the different processes

- The matrix is distributed by row blocks
- The vector is computed on different processes then updated

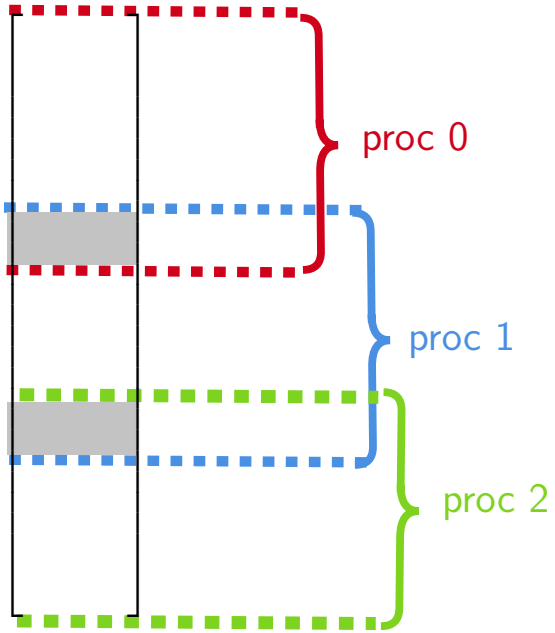
```
# rand simulates different computations on procs
# complete M (size*n x size*n)
M = rand(n, size * n) # block on n rows /proc
u = rand(n * size)  # a complete u on each proc
MPI.Allreduce!(u, +, comm) # communicate u

v = M * u # block M*u on each proc
completeV = MPI.Allgather(v, comm) # concatenate result
# --> completeV is on all the processes
```

Deal with duplicated data

- Domain decompisition (finite elements), graph partition ...
- A *node* at the interface is shared between processes $\rightarrow$ data is duplicated



A domain decomposition between 3 procs



Place of data in the vector of unknowns (value at the nodes)

Merci !