Calling a parallel simulation code from Julia

Marc Fuentes, INRIA

July 16, 2020

Code is available at Github

0.0.1 Rationale

We want to do a parameter optimization for a physical process simulated by a parallel software written in Fortran or C. Thus, implementing a robust optimization method in a low-level language could be a waste of time, particularly when you are doing a Ph.D for instance. The aim of this poster is to share some recipes (technical and numerical), to achieve that in Julia. The knowledge level remains deliberately basic, so it can be seen as a tutorial. Our hypotheses are the following :

- distributed memory paradigm for parallelism (i.e. MPI),
- direct problem is in Fortran or C,
- possibly, implementing a gradient computation will is done in the low level language,
- we focus on continuous optimization (using `Optim.jl`), but principle is the same using JuMP for combinatorial optimization.

First of all, we want to explain how to call a piece of an external code and how to run Julia scripts in a MPI environment.

0.0.2 calling Fortran or C code

As documented here , we need the *Julia* statement `ccall`:

```
ccall((:funcName, library), returnType, (argType1, argType2, ...), (argVal1, argVal2, ...))
```

where `function_name` is the mangled name of the C function in the shared library. Roughly speaking, languages like Fortran or C++ must encode their function names when they interoperate with C.

Remarks

- `library` is only *formally* a string: you could use `"./mylib.so"`, but not `string(pwd(),"/mylib.so")`
- if the library is not in `.`, add path to `LD_LIBRARY_PATH` before launching Julia

For example, the following C function `int addTwo(int x) { return x+2; }` could be called with

```
run(`gcc -o addTwo.so --shared addTwo.c`);
w = ccall((:addTwo, "./addTwo.so"), Int32, (Int32,), 12); println("$w")
```

This example deserves some explanations:

1. to build a *shared* library we add the flag `--shared`, which is compiler-dependent. To improve portability in the provided codes, we use **CMake** to generate Makefiles and compile: `add_library(name_lib SHARED name_src)`
2. When using C, which by default passes function arguments by value, we need pointers if we want modify arguments in the function's body. In that case, Julia's code must provide a **reference** to `ccall`.

On the contrary, in Fortran, since args are passed by reference, we always use reference for non-pointer types. For example, the fortran code

```
subroutine addTwoF(x) bind(C, name ="addTwoF")
    integer(c_int), intent (inout) :: x
    x = x + 2
  end subroutine
```

could be called using

```
z = Ref{Int32}(12) # !VERY IMPORTANT!
w = ccall((:addTwoF, "./addTwoF.so"), Cvoid, (Ref{Int32},), z);println(z[])
```

We use the statement `bind` to attach a C name to `addTwoF` that will override the mangled name. For better compatibility Fortran90 has C equivalent types such as `real(c_double)` or `integer(c_int)` defined in `iso_c_binding`.

To finish, Julia arrays can be converted to pointers without any problem when using `ccall`. For instance, we can call the function

```
void changeArray(int n, double * x) { if (n > 1) x[0] += 3 ; }
```

with

```
a = [1:3.;]; ccall((:changeArray, "./changeArray.so"), Cvoid,(Int32, Ptr{Float64},),
     size(a,1), a); println("$a")
```

0.0.3 Interacting with MPI

For interacting with MPI, we could use the packages `MPI.jl` and `MPIClusterManagers`. Using `MPI.jl` one can run

```
using MPI
MPI.Init()
println("Hi from $(MPI.Comm_rank(MPI.COMM_WORLD))!")
flush(stdout)
```

directly from shell with `mpirun -np 2 julia examples/hello_world.jl`.

Doing so, since `mpirun` calls Julia, this code will be JIT-compiled before being executed each time we run the script. Furthermore, we must run the code out of the Julia REPL, which is not very convenient for some experiments. To avoid that, we can use macro `@mpi_do` from `MPIClusterManagers` package, after the following preamble

```
using MPIClusterManagers, Distributed
manager = MPIManager(np=4)
```

```
addprocs(manager)
@everywhere import MPI.
```

In this way, we can modify and run the following block without restarting Julia

```
@mpi_do manager begin
    comm = MPI.COMM_WORLD;p = MPI.Comm_size(comm);r = MPI.Comm_rank(comm)
    s_loc=sum(1+r* 100/p:100/p * (r+1)); s=MPI.Allreduce(s_loc, +, comm)
    println("s=$s")
end
```

Unfortunately, the present version of MPIClusterManagers does not have an `@mpi_fetchcall` macro to retrieve the result of the computation on the master (which is not part of the MPI Cluster).

0.0.4 Distributed Minimization

Formally, we want to solve a problem like

$$\min_{x_1,\cdots,x_p} f(x_1,\cdots,x_p),$$

where $x_i \in \mathbb{R}^{n_i}$ and the main function f is assumed decomposable, i.e. we could write it as a maximum or a sum of functions defined on each $\mathbb{R}^{n_i}$. For instance,

$$f(x_1,\cdots,x_p) = \sum_{i=1}^{p} f_i(x_i).$$

In almost all optimization methods there is a **linesearch** step, which tries to find a step length α to advance in the descent direction. Since the computation is distributed, each algorithm owns a version of α. Fortunately, since the **linesearch** uses only the value of the function, which it is the same on each process, α remains the same across processes.

A dummy example: the squared L_2 norm Suppose we want to minimize $f(x) = \frac{1}{2}\|x-c\|^2$ where c is a constant vector. The solution is trivially equal to c. We also remark that if we cut $\mathbb{R}^n$ into some chunks, the f function is clearly decomposable and each f_i is just $f_i(x_i) = \frac{1}{2}\|x_i - c_i\|^2$ where c_i has the chunk components of c. Even if in this case we do not need a gradient to find the solution, we can compute it very easily using

$$\nabla f(x) = \bigoplus_{i=1}^{p} \nabla f_i(x_i) = \bigoplus_{i=1}^{p} (x_i - c_i).$$

Since this example is very simple, we directly build the partition of the state vector x in Julia and computation of $f_i, \nabla f_i$ is done in Fortran. To divide x we compute the chunk sizes according to

```
function partition(n::Integer, p::Integer)
 r = n % p
 m = ceil(Integer, n / p)
 part = fill( m, p )
 part[ (r+1) * (r> 0) + (p+1) * (r==0): p ] .= m - 1
 return part
end
```

Then, Julia gathers the chunks on the root process

```
x_glob = MPI.Gatherv(x_min,Cint[i for i in partition(n,p)] , 0, comm)
```

The low-level part is simply a computation of the local squared norm and with `MPI_ALLREDUCE` we share the global sum among all processes.

```
subroutine compute_error(n, x, c, f, df) bind(C, name="compute_error")
(...)
  f = 0.d0
  ! computing objective
  f_loc = 0.5d0 * sum( (x - c)**2 )
  call MPI_ALLREDUCE( f_loc, f, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD, ierr )
  ! computing gradient
  df = x - c
end subroutine compute_error
```

Now, it remains to write the interface using `ccall`

```
function simu!(n::Integer, x::Array{Float64,1}, df::Array{Float64,1})
  f = Ref{Float64}(0.)
  c = cos.(1:n)[slice[1]:slice[2]]
  ccall((:compute_error, "./libpar_error.so"),
  Cvoid, (Ref{Int32}, Ptr{Float64}, Ptr{Float64}, Ref{Float64}, Ptr{Float64}), size(x, 1), x,
  return f[]
end
```

For simplicity, we choose $c_i = \cos(i)$ and the slices are shared as a global variable.

We can now choose our favorite minimization algorithm. A lot of people like Nelder-Mead algorithm because it does not require a gradient, but please do not use it! Proofs of convergence towards non-stationary points exist in the literature Torczon.

Knowing the gradient, we can choose a "1.5" order method like `LBFGS`.

```
res = optimize(cost, grad!, x_loc, LBFGS(), Optim.Options(... ))
```

where `cost` and `grad!` are defined by `cost = x -> simu!(n, x, df_dummy)` and `grad! = (g,x)-> simu!(n,x,g)`. When we do not need a gradient, we can avoid to compute it using a branching condition.

A more advance example: controlling the 2D heat equation Before talking about adjoint methods, let us describe our "toy" problem (which requires 1600 sloc). We heat a square, by imposing a constant but not uniform temperature on the boundary Γ of the square. We want to obtain for $t = t_{final}$ values of the temperature inside the square as close as possible to a target function. The continuous problem is

$$\min_{p} \quad \int_{[0,1]^2} (u(x, t_{final}) - u_{target}(x))^2 \, dx \tag{1}$$

$$\text{such that} \quad \frac{\partial u}{\partial t}(x,t) = \Delta u(x,t), \, \forall (x,t) \in [0,1]^2 \times [0, t_{final}], \tag{2}$$

$$u(x,t) = p(x), \, \forall (x,t) \in \Gamma \times [0, t_{final}.] \tag{3}$$

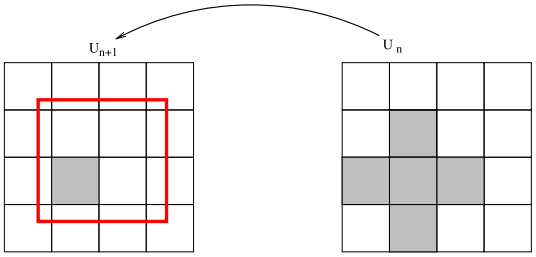
After discretizing, using finite difference scheme for space and Euler formula for time, we obtain

$$\min_{p} \quad \|U^n - U^{target}\|^2_{inside} \tag{4}$$

$$\text{such that} \quad U^{n+1} = F(U^n, p) = U^n - \frac{\delta t}{h^2} \cdot K2D(U^n, p) \tag{5}$$

$$U^0 = 0. \tag{6}$$

where $K2D$ is the heat kernel with p on the boundary, which can be sketched as



In this 4-stencil, each interior point of the matrix is computed as a linear combination of its value and its four neighbour values. The direct problem is computed through a recursive sequence of matrices. By controlling matrix boundary terms, we want to minimize the error relative to a target matrix at the last iteration. See `heat_seq.jl` as a Julia sequential implementation of the algorithm. In this case, all the parallel part is done in Fortran. The Julia program transfers the vector p representing the boundary Γ to the Fortran program which could return the cost or the gradient: Fortran's `optional` feature for functions arguments is used to factorize code, when we need only the cost f.

Adjoint method Adjoint methods are used when we want to minimize a function $g(x, p)$ relatively to p under the state's constraint $F(x, p) = 0$. A good reference on this topic is note of about reccurence S. G Johnson. To sum up, to compute a gradient of $g(p) = \|U^n(p) - U^{target}\|^2$, we need to apply a direct recursion to compute the final state and the final error, and then we backwardly compute contributions to the gradient using adjoint state.

Some software tools using *automatic differentiation* exist to generate a program which computes the gradient. In Julia, one could use JuliaDiff. For C or Fortran codes, see adifor or tapenade. In our case, the gradient can be computed analytically. Since F is linear, partial differential of F relative to U and p is F herself, zeroing the other component.

$$\partial_u F(u, p)[v] = v - \frac{\delta t}{h^2} \cdot K2D(v, 0) \text{ and } \partial_p F(u, p)[q] = -\frac{\delta t}{h^2} \cdot K2D(0, q).$$

One could remark that the two operators are very sparse, which is practical.

Future works

- benchmarking: check at least that Fortran parallel code beats Julia sequential.
- create a package in the same spirit as MPI.jl but for OpenCoarrays. MPI is well suited for parallelism, but it remains a library in Fortran. Coarrays are now part of the standard and they are more suited to write parallel algorithms.