Capítulo
2

High Performance Computing in Julia

Roberto Machado Velho
Sarpen Quant Investments, São Paulo, Brazil

Rafael Benchimol Klausner
PSR Energy Consulting and Analytics, Rio de Janeiro, Brazil

Matheus da Silva Serpa
Parallel and Distributed Processing Group (GPPD)
Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

Adriano Maurício de Almeida Côrtes
Applied Mathematics Department (DMA) and
High-Performance Computing Center (NACAD)
Federal University of Rio de Janeiro, Rio de Janeiro, Brazil

Abstract

These are a set of notes designed for a short course addressing the use of the Julia Programming Language for High Performance Computing (HPC). They target the reader that has some basic knowledge of HPC and distributed systems. First goal is to show how to use Julia from zero on a typical HPC cluster environment. Second goal is showing which changes are necessary for parallelizing a code using Julia. The main purpose of the use of Julia is that it is easy to install, easy to code, and it can still deliver great computational performance on a HPC system.

2.1. Introduction

Julia is a modern language that is easy to use and to learn. It was designed to be used in scientific computing while being fast and allow multiple types of parallelization. The main goals of the notes is to enable the reader to prototyping simple parallel applications in Julia, and to run already existing complex applications, under multicore, distributed CPUs, or GPU architectures, the typical environment of nowadays clusters found in a High Performance Computing (HPC) Center. Our examples use the Process Manager SLURM.

These notes assume some familiarity of the reader with HPC systems. We will describe how easy is to prototyping a parallel code in Julia. Later, via examples, we discuss which kind of modifications are necessary when converting some application (eventually composed of multiple functions) from its serial version to a parallel one.

We start the notes by presenting Julia and how to install its binary file and its packages on a HPC system in section 2.2. We then pass to analyze some typical aspects of HPC programming, on how to measure performance of an application and how to organize code for maximum efficiency, on section 2.3. Section 2.4 discusses how the use of Julia for multicore programming, while Section 2.5 goes through distributed computing. Finally, in Section 2.6 we bring some simple analysis of speed up for the examples presented in the multicore and distributed sections, while in Section 2.7 we present graphics processing unit (GPU) programming with Julia.

Since these notes target an intermediary course on HPC, we avoided complex discussions on Input/Output (I/O) aspects in Julia. One of the reasons is the already not mature capabilities of the language to perform I/O for many forms of parallelizing a code. For more information about Parallel Computing in Julia, you can access the parallel computing documentation [1].

2.2. Basics of Julia

Our goal along this section is presenting the language Julia and how to install it and its packages on a HPC system.

2.2.1. Why Julia?

Julia is a high-level, dynamic, general-purpose programming language that is specifically aimed for numerical computing and computational science. Julia benefits from a technique of compiling in advance the necessary libraries a piece of code will run. This can bring high speed on running codes. It is also easy to programming, resembling Python or Matlab syntaxes. Its ecosystem has already a large amount of libraries for solving from basic problems in computer science to specific industrial applications.

Julia allows the use of Unicode characters, giving the programmer access to a large extension of symbols to construct variables. Among them are Greek characters, very widely used in science, in particular in mathematics, what makes the process of writing a code with variable names that reflect the scientific problem easier. To input such characters, use the Latex code and press Tab. A list of permitted characters is in here:

[1] https://docs.julialang.org/en/v1/manual/parallel-computing/

https://docs.julialang.org/en/v1/manual/unicode-input/

Julia permits calls to pieces of code of other languages. This is a great advantage on prototyping code in HPC systems, as you can call a performative code from a library in C or Fortran, and at the same time assemble the results and visualize it with the modern libraries from Julia's ecosystem. This is very beneficial for non-experts on parallel programming, that can assemble pieces of code in Julia while using previously assembled performative libraries written in other languages by experts on HPC systems and in the domain of knowledge in question, e.g., biology, chemistry, geophysics, or mathematics. There are several Julia case studies available at https://juliacomputing.com/case-studies/.

2.2.2. Installing Julia and its packages

In order to download Julia, go to the website https://julialang.org/downloads/. All code examples run in the current stable release v1.7.2 (at the time of writing). The Julia Language has a thorough documentation in https://docs.julialang.org/en/v1/. Julia can be used in an interactive command-line interface, called REPL (read-eval-print loop), by calling `julia` in the command-line.

```
$ julia

julia>
```

Also, you can run a code from a julia script.

```
$ julia script.jl
```

To download a specific package in Julia, open Julia's terminal and press the button closed square bracket (]) to go to the Package Manager (Pkg).

```
$ julia

julia>

(@v1.7) pkg>
```

To install an specific package use the keyword add:

```
(@v1.7) pkg> add Example
```

It is also possible to specify multiple packages at once.

```
(@v1.7) pkg> add JSON StaticArrays
```

To remove a package, use the keyword rm.

```
(@v1.7) pkg> rm JSON StaticArrays
```

Use update to update an installed package.

```
(@v1.7) pkg> update Example
```

To update all installed packages, use update without any arguments.

```
(@v1.7) pkg> update
```

Julia has packages environments just like Python language, but it is a lot easier to use. To activate an environment use the keyword activate.

```
(@v1.7) pkg> activate my_env
  Activating environment at `/home/user/my_env/`

(my_env) pkg>
```

The Pkg let you know which environment you are using in the brackets. To see what packages you have already installed, use status.

```
(my_env) pkg> status
  Status `/home/user/my_env/` (empty project)
```

When a environment is just started, its condition is empty. The packages metadata are stored in the `Project.toml` file in the folder `/home/user/my_env/`, so it is possible to use the same environment in different computers. This is important for two or more persons working on the same project, or a single person working on different computers, e.g., a personal laptop and the HPC system where a parallel code will run. Let us add a package.

```
(my_env) pkg> add StatsBase
...

(my_env) pkg> status
    Status `/home/user/my_env/`
  [2913bbd2] StatsBase v0.33.16
```

The environment now has `StatsBase` in its dependencies. You can start Julia with a specific project with the `--project` flag.

```
  $ julia --project=my_env

julia>
```

```
(my_env) pkg>
```

For more information about Package Manager access https://pkgdocs.julialang.org/v1/.

This course uses a specific `Project.toml` to guarantee the code will be reproducible in all computers.

Listing 2.1. Project.toml

```
[deps]
BenchmarkTools = "6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf"
CUDA = "052768ef-5323-5732-b1bb-66c8b64840ba"
ColorSchemes = "35d6a980-a343-548e-a6ea-1d62b119f2f4"
Images = "916415d5-f1e6-5110-898d-aaa5f9f070e0"
Plots = "91a5bcdd-55d7-5caf-9e0b-520d859cae80"
StatsBase = "2913bbd2-ae8a-5f71-8c99-4fb6c76f3a91"
```

It is highly recommended to create different environments for each Julia repository, because the package has a set of dependencies and, if you have a high number of different packages (about 20 or more), there is a high probability of dependency conflict. More on this topic is discussed in https://pkgdocs.julialang.org/v1/managing-packages/#conflicts.

2.3. High Performance Computing aspects in Julia

This section is dedicated to describe some aspects of HPC in Julia. We start by presenting how to measure time of a Julia application. Later, we go through some specificities of Julia interpreter and how to organize a serial for good performance even before an attempt to parallelize it.

2.3.1. Analyzing Julia Performance

As a high performance language, Julia has several tools to analyze and benchmark code performance. The most simple and yet the most useful tool is the `@time` macro.

```
julia> @time sqrt(rand())
  0.000001 seconds
  0.5857267559291746
```

The `@time` macro evaluates how much time Julia took to give aresult, in this case 0.000001 seconds, and provides the output of the function `sqrt(rand())` right below. Let us see more examples:

```
julia> @time sqrt.(rand(10))
  0.000002 seconds (2 allocations: 320 bytes)
10-element Vector{Float64}:
 0.7798392052070687
 0.5408866082279589
 0.994391535760473
 0.5254650346521715
 0.805928150879407
 0.4298229174632851
```

```
 0.5083361019219698
 0.5073599184347386
 0.8721449074432472
 0.7945992718819369
```

In the previous case we used Julia broadcasting attribute in the function `sqrt` by adding the dot after the function call, so that the function is applied to all elements in the array. Without the dot, Julia generates an error.

There is a more verbose version of `@time`, the `@timev` macro.

```
julia> @timev sqrt.(rand(1000));
  0.000005 seconds (2 allocations: 15.875 KiB)
elapsed time (ns): 5500
bytes allocated:   16256
non-pool GC allocs:2
```

Along the examples in this notes we will often run a piece of code twice. On the first call, the code gets compiled. This call should not be taken into account for the total time, just the second one. It is possible to speedup this latency with PackageCompiler [2].

For a more complete benchmarking, Julia provides a package called BenchmarkTools. Details are available at https://juliaci.github.io/BenchmarkTools.jl/stable/. On it, there is a macro called `@btime`, that provides the same information as `@time`.

Along our examples, we have made use of a macro called `@my_time`. The reason we used this macro, instead of the original @time, is to retrieve both the information about the result and the time elapsed. For more information about macros, look at the Julia Manual [3]. Our macro is defined as following:

Listing 2.2. My time

```
macro my_time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        Δt = (t1-t0)/1e9
        val, Δt
    end
end
```

2.3.2. Organizing your serial code even before parallelization

As we already know, Julia is a dynamic language that balances the best of two worlds: the high-level flexibility of a pure interpreted language and the speed of compiled language. To do so, a JIT (Just-in-Time) Compiler amalgamates several design decisions and features of Julia.

[2] https://github.com/JuliaLang/PackageCompiler.jl
[3] https://docs.julialang.org/en/v1/manual/metaprogramming

In this section, we elaborate on some of these features that should be taken into account by the programmer before going to parallelization, that is, some guidelines to achieve performance in the serial version of the code. We start with a description of the four main steps of the JIT compilations process, from the user's source code (script) to the machine-level code. At each step, we touch upon some of the features the programmer should be aware of.

Julia's JIT compilation process can be disassembled in four main stages. The first ones are promoted by Julia, and generates intermediate representations, already implementing some code optimization, and the last one is target-specific, and is promoted by the LLVM[4] compiler framework, promoting final optimizations targeting the binary machine code. Julia provides code macros for the programmer be able to inspect the intermediate representations (IR) of the main steps, as we see bellow. Despite generally being hard to read them all in detail, important performance tips and checks can pop up inspecting them. We will describe them in sequence, as summarized in Figure 2.1.

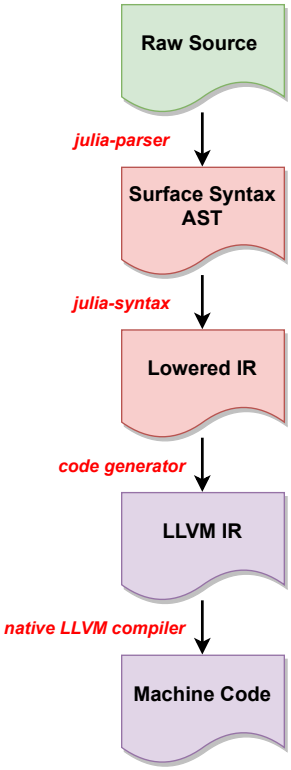

Figure 2.1. Julia's JIT compilation process diagram.

As a first step, the raw source code is parsed by Julia, producing an abstract syntax tree (AST) called *Surface Syntax AST*. In this tree representation, for example, operators are replaced by their respective function call. We can inspect this step by using the functions `dump` and `Meta.parse`. Try on the REPL `dump(:(1+2))`.

[4]originally an acronym for Low Level Virtual Machine

```
julia> dump(:(1+2))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 2
```

As we can see, we have an expression with three nodes being the *head* node the call to addition. Now try on the REPL `dump(:(1+2-3))`.

```
julia> dump(:(1+2-3))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol -
    2: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol +
        2: Int64 1
        3: Int64 2
    3: Int64 3
```

We can see now two expressions in a hierarchical (tree) organization. First, the lower head node is the addition operation call, which results is a operand node for the upper head node: the subtraction operation call. Let us try a bit more complex code, now making use of the function `Meta.parse`. Try on the REPL `Meta.parse("sum = 0; for i in 1:5 sum+=i end")`.

```
julia> Meta.parse("sum = 0; for i in 1:5 sum+=i end")
:($(Expr(:toplevel, :(sum = 0), :(for i = 1:5 #= none:1 =# sum += i end))))
```

The output now is an expression, where the *head* node is the top level (tree root), and the child nodes are expressions (the ones preceded by :). We can now use it as an input to the `dump` function.

```
julia> dump(Meta.parse("sum = 0; for i in 1:5 sum+=i end"))
Expr
  head: Symbol toplevel
  args: Array{Any}((2,))
    1: Expr
      head: Symbol =
      args: Array{Any}((2,))
        1: Symbol sum
        2: Int64 0
    2: Expr
      head: Symbol for
      args: Array{Any}((2,))
        1: Expr
          head: Symbol =
          args: Array{Any}((2,))
            1: Symbol i
            2: Expr
              head: Symbol call
```

```
            args: Array{Any}((3,))
              1: Symbol :
              2: Int64 1
              3: Int64 5
        2: Expr
          head: Symbol block
          args: Array{Any}((2,))
            1: LineNumberNode
              line: Int64 1
              file: Symbol none
            2: Expr
              head: Symbol +=
              args: Array{Any}((2,))
                1: Symbol sum
                2: Symbol i
```

The Julia AST is then parsed by the *julia-syntax* producing an intermediate representation, known as *Lowered IR*. In this representation, the high-level Julia syntax is replaced by a smaller set of instructions, from the standard modules for example. Additionally, all control flow is converted to explicit branches (go to) and sequences of statements, and all macros are expanded. We can inspect this lowered IR using some functions and macros. The first macro is `Meta.@lower`. When applied to the **`for`** code snippet above we obtain:

```
julia> Meta.@lower begin sum = 0; for i in 1:5 sum+=i end end
:($(Expr(:thunk, CodeInfo(
    @ REPL[228]:1 within `top-level scope'
1 —       sum = 0
│   %2  = 1:5
│         #s564 = Base.iterate(%2)
│   %4  = #s564 === nothing
│   %5  = Base.not_int(%4)
└──       goto #4 if not %5
2 — %7  = #s564
│         i = Core.getfield(%7, 1)
│   %9  = Core.getfield(%7, 2)
│         sum = sum + i
│         #s564 = Base.iterate(%2, %9)
│   %12 = #s564 === nothing
│   %13 = Base.not_int(%12)
└──       goto #4 if not %13
3 —       goto #2
4 —       return nothing
))))
```

Another option is to encapsulate the code snippet in a function and use the macro `@code_lowered`.

```
julia> function my_sum()
       sum = 0; for i in 1:5 sum+=i end
       end
my_sum (generic function with 2 methods)

julia> @code_lowered my_sum()
CodeInfo(
1 —       sum = 0
│   %2  = 1:5
│         @_2 = Base.iterate(%2)
│   %4  = @_2 === nothing
│   %5  = Base.not_int(%4)
```

```
└──       goto #4 if not %5
2 ─ %7  = @_2
│         i = Core.getfield(%7, 1)
│   %9  = Core.getfield(%7, 2)
│         sum = sum + i
│         @_2 = Base.iterate(%2, %9)
│   %12 = @_2 === nothing
│   %13 = Base.not_int(%12)
└──       goto #4 if not %13
3 ─       goto #2
4 ─       return nothing
)
```

We see that the **`for`** statement is transformed to a sequence of `go to` together with `Base.iterate` function calls. Lowered IR is an important link in the Julia's compilation chain. It has a significant impact on Julia's performance, since it is used for type inference, optimizations like inlining, and code generation. Types and type inference, together with Multiple Dispatch, are some language design features that guarantees Julia's performance. The type inference outcome can be inspected by the macro `@code_type`. For example, try the commands bellow:

```
julia> function my_sum(n)
           sum = 0; for i in 1:n sum+=i end
           return sum
       end
my_sum (generic function with 2 methods)

julia> @code_typed my_sum(5)
CodeInfo(
1 ── %1  = Base.sle_int(1, n)::Bool
│    %2  = Base.ifelse(%1, n, 0)::Int64
│    %3  = Base.slt_int(%2, 1)::Bool
└──       goto #3 if not %3
2 ──       goto #4
3 ──       goto #4
4 ── %7  = φ (#2 => true, #3 => false)::Bool
│    %8  = φ (#3 => 1)::Int64
│    %9  = φ (#3 => 1)::Int64
│    %10 = Base.not_int(%7)::Bool
└──       goto #10 if not %10
5 ── %12 = φ (#4 => %8, #9 => %21)::Int64
│    %13 = φ (#4 => %9, #9 => %22)::Int64
│    %14 = φ (#4 => 0, #9 => %15)::Int64
│    %15 = Base.add_int(%14, %12)::Int64
│    %16 = (%13 === %2)::Bool
└──       goto #7 if not %16
6 ──       goto #8
7 ── %19 = Base.add_int(%13, 1)::Int64
└──       goto #8
8 ── %21 = φ (#7 => %19)::Int64
│    %22 = φ (#7 => %19)::Int64
│    %23 = φ (#6 => true, #7 => false)::Bool
│    %24 = Base.not_int(%23)::Bool
└──       goto #10 if not %24
9 ──       goto #5
10 ─ %27 = φ (#8 => %15, #4 => 0)::Int64
└──       return %27
) => Int64
```

Note that all the annotated types and the final line indicating the returned type `Int64` of

the function. Now we suggest to run the line command `@code_typed my_sum(5.0)` in the REPL and compare the result with the output above. Notice that we obtain `Union{Float64, Int64}`, depicting an important feature, related to type stability, that we describe later. By now, let us return to the description of the compilation process chain.

The lowered IR passes through the code generator, see Figure 2.1, to produce the LLVM IR. Actually, beginning in Julia 0.7, parts of the compiler use a new SSA-form (*static single assignment form*) intermediate representation. The code above already brings elements of this representation, like the ϕ functions. They also appear in the LLVM IR, that can be inspected by the `@code_llvm` macro, for example,

```
julia> @code_llvm my_sum(5)
;  @ REPL[231]:1 within `my_sum'
define i64 @julia_my_sum_4154(i64 signext %0) {
top:
;  @ REPL[231]:2 within `my_sum'
; ┌ @ range.jl:5 within `Colon'
; │┌ @ range.jl:287 within `UnitRange'
; ││┌ @ range.jl:292 within `unitrange_last'
     %.inv = icmp sgt i64 %0, 0
     %1 = select i1 %.inv, i64 %0, i64 0
; └└└
  br i1 %.inv, label %L12.preheader, label %L27

L12.preheader:                                    ; preds = %top
  %2 = shl nuw i64 %1, 1
  %3 = add nsw i64 %1, -1
  %4 = zext i64 %3 to i65
  %5 = add nsw i64 %1, -2
  %6 = zext i64 %5 to i65
  %7 = mul i65 %4, %6
  %8 = lshr i65 %7, 1
  %9 = trunc i65 %8 to i64
  %10 = add i64 %2, %9
  %11 = add i64 %10, -1
;  @ REPL[231]:3 within `my_sum'
  br label %L27

L27:                                              ; preds = %L12.preheader, %top
  %value_phi9 = phi i64 [ 0, %top ], [ %11, %L12.preheader ]
  ret i64 %value_phi9
}
```

Finally, the last representation, target-specific, is the *LLVM assembly code*, hereafter called the *machine code*. Following our recurrent example, we have

```
 julia> @code_native my_sum(5)
        .section        __TEXT,__text,regular,pure_instructions
; ┌ @ REPL[231]:2 within `my_sum'
; │┌ @ range.jl:5 within `Colon'
; ││┌ @ range.jl:287 within `UnitRange'
; │││┌ @ range.jl:292 within `unitrange_last'
        testq   %rdi, %rdi
; │└└└
        jle     L43
; │ @ REPL[231] within `my_sum'
        movq    %rdi, %rax
        sarq    $63, %rax
        andnq   %rdi, %rax, %rax
; │ @ REPL[231]:2 within `my_sum'
```

```
        leaq    -1(%rax), %rdx
        leaq    -2(%rax), %rcx
        mulxq   %rcx, %rcx, %rdx
        shldq   $63, %rcx, %rdx
        leaq    (%rdx,%rax,2), %rax
        decq    %rax
; | @ REPL[231]:3 within `my_sum'
        retq
; | @ REPL[231] within `my_sum'
L43:
        xorl    %eax, %eax
; | @ REPL[231]:3 within `my_sum'
        retq
        nop
; L
```

Julia is a language targeting scientific computing applications, as such, vector, matrices, and, more generally, arbitrary dimension arrays, a data container provided by `Base` library. They are parameterized by the data type `T` (for example `Int64, Float64`, etc) and dimension `N` in `Array{T,N}`. For vectors, `N=1`. In fact, the definition `Vector{T}` is an alias for `Array{T,1}`. For matrices, `N=2`, the alias `Matrix{T}` is an alias for `Array{T,2}`. The indexing rule for `N`-dimensional arrays is from the last index to the first. In terms of matrices, this means that the memory layout of data storage is column-wise. To exemplify this and propose some gradual performance improvements we will resort to an important Linear Algebra kernel, the matrix-vector product, hereafter referred as `matvec`.

The code in 2.3 implements a row-wise `matvec`, that is, taking inner products of the matrix row with the column vector, whiel code in 2.4 implements a column-wise `matvec`, that is, forming a linear combination of the matrix columns, with the vector entries as coefficients.

Listing 2.3. Matvec row-wise (row-column inner products).

```
function matvec_row!(y, A, x)
    for i in 1:size(A,1)
        for j in 1:size(A,2)
            y[i] += A[i,j]*x[j]
        end
    end
end
```

Listing 2.4. Matvec column-wise (linear combination of the column)

```
function matvec_col!(y, A, x)
    for j in 1:size(A,2)
        for i in 1:size(A,1)
            y[i] += A[i,j]*x[j]
        end
    end
end
```

As one would expect, by the matrix data layout in memory, `matvec_col` is faster than `matvec_row`. Indeed, using the `@btime` macro, we obtain

```
julia> @btime matvec_row!(y,$A,$x) setup=(y = zero(x))
  2.682 ms (0 allocations: 0 bytes)

julia> @btime matvec_col!(y,$A,$x) setup=(y = zero(x))
  688.710 μs (0 allocations: 0 bytes)
```

Julia has bounds checking, and if the programmer can guarantee to have full control of it, as in our case, since we have a simple loop by using the function `size`), one can make use of the macro `@inbounds`. This macro implies checking is skipped, impacting positively to the performance. In listing 2.5, we make use of it to reduced the time to run `matvec_col`.

Listing 2.5. Matvec column-wise with `@inbounds` macro.

```
function matvec_col_inbounds!(y, A, x)
    @inbounds for j in 1:size(A,2)
        for i in 1:size(A,1)
            y[i] += A[i,j]*x[j]
        end
    end
end
```

And we observe an expressive gain in performance:

```
julia> @btime matvec_col_inbounds!(y,$A,$x) setup=(y = zero(x))
  261.954 μs (0 allocations: 0 bytes)
```

But as one can note, we are not using any high-level syntax, aside from that, our code looks very similar to a C/C++ vanilla implementation. The listing 2.6 implements a naive column-wise `matvec` using the array slicing `A[:,j]` to get the matrix columns.

Listing 2.6. Matvec column-wise using naive high-level syntax.

```
function matvec_col_hl_naive!(y, A, x)
    for j in 1:size(A,2)
         y += A[:,j]*x[j]
    end
end
```

In this case, we obtain a timing that is even worse than the `matvel_row`. Indeed, we get

```
julia> @btime matvec_col_hl_naive!(y,$A,$x) setup=(y = zero(x))
  3.057 ms (3000 allocations: 23.25 MiB)
```

Notice that the number of allocations increased from 0 to 3000, causing a considerable degradation in performance when compared to `matvec_col`. The reason for that is the array slicing `A[:,j]`, that promotes a copy of the matrix column. For such case, one can use the macro `@view`, as in the Listing 2.7

Listing 2.7. Matvec column-wise using high-level syntax fixing slicing allocations.

```
function matvec_col_hl_fix1!(y, A, x)
    for j in 1:size(A,2)
         y += @view(A[:,j])*x[j]
    end
end
```

The macro `@view` prevents these copies, then increasing performance:

```
julia> @btime matvec_col_hl_fix1!(y,$A,$x) setup=(y = zero(x))
  2.140 ms (2000 allocations: 15.50 MiB)
```

But we still note a non-negligible number of allocations, 2000. The reason now is that we are still allocating temporary vectors to perform the scalar-vector multiplication and the vector-vector addition. The fixing now is by use of one of the performance gain players of Julia, namely, the broadcasting. Broadcasting is achieved by using the dot operator, `.`, as in the Listing 2.8

Listing 2.8. Matvec column-wise using high-level syntax still allocations with broadcast.

```
function matvec_col_hl_fix2!(y, A, x)
    for j in 1:size(A,2)
         y .+= @view(A[:,j]).*x[j]
    end
end
```

We finally zeroed the number of allocations, and obtained a comparable performance, as we can see from

```
julia> @btime matvec_col_hl_fix2!(y,$A,$x) setup=(y = zero(x))
  262.494 μs (0 allocations: 0 bytes)
```

In this simple progressive exercise, we were able to show the most important aspects related to working with arrays, in such a way to obtain a performative serial code. We will pass then to aspects of parallelization in Julia.

2.4. Multicore Computing

Modern computer CPUs have architectures that allow multiple cores to operate simultaneously. While one of the possibilities is that different applications use different cores, in scientific computing we would like that a single application of interest use part or all the available cores of a single CPU. This is achieved via the so-called threading. Along this section we describe the basic aspects of threading in Julia and we include examples.

Two remarks must be addressed. First is about taking care while using Input/Output operations when using code containing multithread calls. The second is the use of nested threading, that must be avoided. This last issue may appear even when using basic scientific computing libraries and one thread implementation in Julia. Several basic libraries,

as for linear algebra, include already parallelized versions within it. An attempt of using the basic threading in Julia while making calls for functions of such libraries may lead to an unexpected nested threading and the computing time may end up getting worse when compared with the serial version call of the same function. This effect happens because the code may try to use twice the maximum number of threads possible.

2.4.1. Threading in Julia

Threads are independent parts computation that run on a CPU core. As known in HPC techniques, we should use at maximum the number of threads as the number of cores available. One point of attention is when the CPU you work with has the hyper-threading option. Intel's CPUs have such option and operational systems may see the number of cores available as the double of the real (physical) CPU cores. For intense CPU computation, you must use at maximum the number of physical cores in the CPU. One form of embedding in the Julia code the information about the number of physical cores of the machine being used is through the package **Hwloc**, and then with the use of the command `Hwloc.num_physical_cores()`.

In Julia the number of threads is fixed at startup. Thus, to set it, you can export the environment variable `JULIA_NUM_THREADS` setting the number the threads you would like or you can call Julia with the flags `-t` or `--threads`. The command `nthreads()` shows the current number of threads in use.

```
$ export JULIA_NUM_THREADS=4
$ julia

julia> using Base.Threads

julia> nthreads()
4
```

In Julia the use of threads is done via the `@threads` macro, as we can see in the Hello World file example 2.9.

Listing 2.9. Hello World

```
using Base.Threads

Threads.@threads for thread = 1:nthreads()
    println("$(Threads.threadid()) of $(nthreads()) - Hello World!")
end
```

When in the command-line, we call Julia using 4 threads to run such code, obtaining as result:

```
$ julia -t 4 .\hello-world.jl
  1 of 4 - Hello World!
  2 of 4 - Hello World!
  3 of 4 - Hello World!
  4 of 4 - Hello World!
```

The iteration space is split among the threads, each one having its thread ID, and each thread prints "Hello World!". The macro `@threads` is similar to the use of OpenMP pragmas in the C++ language [5].

2.4.2. Synchronization with threads

Since multiple cores can access simultaneously the same memory, we may have different cores changing the same piece of information in memory at the same time. This is often called *race condition*. In order to avoid it, Julia offers the possibility of changing values atomically, similarly to the SYNCHRONIZATION Pragmas in OpenMP. Example 2.10 demonstrates the use.

Listing 2.10. Vector sum

```
using Base.Threads

N = 1000
acc1 = Ref(0) # Pointer with value 0
@threads for i in 1:N
    acc1[] += 1 # Add 1 with race condition
end
println("Sum wrong is $(acc1[])")

acc2 = Atomic{Int64}(0) # Atomic pointer with value 0
@threads for i in 1:1000
    atomic_add!(acc2, 1) # Add 1 thread safely
end
println("Sum atomic is $(acc2[])")
```

When the previous code is run, we obtain:

```
 $ julia -t 4 .\vsum.jl
Sum wrong is 989
Sum atomic is 1000
```

Such strange behavior occurs because our use of a pure function[6]. If a function modify its arguments, we might end up with a wrong answer. To avoid such issue, it suffices to encapsulate the sum in a function, then obtaining the right answer.

2.4.3. Selection Sort Algorithm via threads

We now pass to an example of the use of threads in Julia. We implement a serial and a parallel version of the algorithm *selection sort*. We can the observe several aspects on the following code: show how we can adapt a serial Julia code to its parallel counterpart, regard how to prepare the time measurements of both codes running, and finally regard the speed up achieved. Both versions of the algorithm are in the Listing 2.11, and the experiment of speed up was performed using 88 threads (more about speed up of this

[5] https://www.intel.com/content/www/us/en/develop/documentation/oneapi-dpcpp-cpp-compiler-dev-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-pragmas-summary.html

[6] https://en.wikipedia.org/wiki/Pure_function

code and other examples is presented in Section 2.6.

Listing 2.11. Selection sort

```
using Base.Threads

macro my_time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        Δt = (t1-t0)/1e9
        val, Δt
    end
end

function selectionsort!(arr::Vector{<:Real})
    len = length(arr)
    if len < 2 return arr end
    for i in 1:len-1
        lmin, j = findmin(arr[i+1:end])
        if lmin < arr[i]
            arr[i+j] = arr[i]
            arr[i] = lmin
        end
    end
    return arr
end

function selectionsortparallel!(arr::Vector{<:Real})
    len = length(arr)
    if len < 2 return arr end
    @threads for i in 1:len-1
        lmin, j = findmin(arr[i+1:end])
        if lmin < arr[i]
            arr[i+j] = arr[i]
            arr[i] = lmin
        end
    end
    return arr
end

v = rand(50000)
val_ser, Δt_ser = @my_time selectionsort!(v)
val_ser, Δt_ser = @my_time selectionsort!(v)
println("Serial:   time = ",Δt_ser, " seconds")
val_par, Δt_par = @my_time selectionsortparallel!(v)
val_par, Δt_par = @my_time selectionsortparallel!(v)
println("Parallel: time = ",Δt_par, " seconds")
@assert val_ser == val_par
```

The computing times were:

```
  $ julia -t 88 .\selection_sort.jl
Serial:   time = 3.065008172 seconds
Parallel: time = 0.374435835 seconds
```

showing the parallel version was more than 8 times faster compared to the serial version.

2.4.4. Final Remarks on Threads

The first remark concerns the use of I/O operations inside threads, what is advisable to avoid. In particular, the use of the command `println` for debugging multithreaded code. A safer way is the command `Core.println`.

The second remark is about Julia's garbage collector. It does not run in threaded mode and an attempt to use it will block all other threads, slowing down the performance.

The final remark is about Threaded libraries. Many libraries already use threads inside its code to speed up calculations. An example is the linear algebra library *OpenBLAS*. Thus, using it in your code already gives you a multicore performance automatically. However, since nested threads are not safe in Julia, it is not advisable to use calls to such threaded libraries inside your own threaded code. An attempt to do it, could in fact reduce the performance compared to a simple serial call to a function of the threaded library.

We will pass now to aspects of parallelization of Julia code on distributed computing.

2.5. Distributed Computing or Multi-Machine Computing in Julia

Most of HPC systems have many computer nodes. In this context, scientific applications can be distributed across all system processors/cores aiming to achieve high performance and meet the computational demands of accurate simulations. Using the Julia language, we can take advantage of this approach, dividing the work of an application between several computational nodes, each one possessing multiple CPUs.

2.5.1. Introduction to Julia Clusters

We discuss now how to operate with such architectures using Julia. The abstraction used in Julia for such tasks is called *Clusters*. This abstraction can also be used on a single computer for calculations with multiple cores, without concerning about synchronization issues. This is similar to the use of the MPI abstraction in C++ Language on a single machine, what is largely described in the literature of HPC systems. Also, there is a interface of MPI in Julia called `MPI.jl`[7].

One of the forms of working with distributed computing in Julia is via the **Distributed** package. If we wish to use it, we first need to start Julia on a local cluster using the flag `-p`. To start it with 2 workers, load the package, and check how many workers we have set, proceed as:

```
 $ julia -p 2

julia> using Distributed

julia> workers()
2-element Vector{Int64}:
 2
 3
```

Alternatively, we can use the function `addprocs` from the `Distributed` package:

[7] https://github.com/JuliaParallel/MPI.jl

```
 $ julia

julia> using Distributed

julia> addprocs(2)
2-element Vector{Int64}:
 2
 3
```

The advantage of the method `addprocs()` is that, being it a Julia's pure function, it can be called from the REPL or from another Julia code.

On both previous cases, 2 additional processes were created besides the master process. Communication in Julia is one-sided. So, a master process is used to receive inputs from the user and this master process controls all the other processes. In the example above, there is one master process and two workers' processes.

Since we now have the option to perform calculations on multiple machines, we need to have access to them. A simple way of doing that is with the help of cluster managers. They are tools allowing us from a single machine to perform calculations on other machines located in the physical cluster (containing multiple CPUs). There are different ways of organizing machines inside a cluster, the usual one is via *partitions*. They are simply abstractions on how to group different machines. You can also think of that as an alias to a set of machines where calculations will be performed. Cluster managers control also the time a single user can have on one or several partitions, the priority of an user compared to others, the amount of time that can be allocated on a partition, etc.

Along the examples in the notes, we have made use of SLURM as a cluster manager. We quickly introduce SLURM and how to work with it in what follows.

2.5.2. Slurm Workload Manager

Slurm [Yoo, Jette e Grondona 2003] (Simple Linux Utility for Resource Management) is an Open-source job scheduler for Linux-based systems created by LLNL (Lawrence Livermore National Laboratory). Currently, Slurm is deployed at many supercomputers worldwide, including the Brazilian super computer SDumont. Slurm allows users to allocate and deploy batch jobs to a series of resources, ensuring that detailed resources, like CPUs, memory, or full nodes, are accurately shared among users and jobs without wrong interaction. Primarily, each set of equal resources is aggregated on a division of resources with an individual queue for submissions of jobs. Users can send jobs and allocate entire nodes of different partitions.

2.5.2.1. Interactive Jobs on Slurm

Interactive jobs is a manner of working directly at the operational system of each machine and having a direct screen feedback of the operations being performed. We will decide for such jobs in Julia while using the REPL.

Aiming to submit interactive jobs, it is necessary to use the salloc command, requesting the resources to be used. When salloc allocates the requested resources to the job, it

informs the user, who can then access the node (via ssh), perform its tasks locally, and run the application. See examples on how to use salloc below.

Example 1 Request to allocate any machine present in the *draco* partition for 5 hours:

```
salloc -p draco -J NAME-JOB -t 05:00:00
```

Example 2 Request to allocate two computational nodes present in the *hype* partition for 24 hours:

```
salloc -p hype -N 2 -J NAME-JOB -t 24:00:00
```

where `NAME-JOB` is the name given to such job and that it will be listed on the queue of jobs, visible to all other users on the cluster.

Besides salloc, there are two other relevant SLURM commands: **sinfo** and **squeue**. Command sinfo presents information about SLURM nodes and partitions, while squeue is used to view jobs in queue or running.

2.5.2.2. Non-Interactive Jobs on SLURM

Non-interactive jobs are those in which the user submits a script that runs on the machines. SLURM permits this kind of scripting using directives #SBATCH. Every line starting with this command will have its content passed straight to SLURM at allocation time. Those non-interactive jobs are the ones we will use to run an entire piece of Julia code saved in a file.

A simple non-interactive script to a Julia code for the *selection_sort.jl* file is in the Listing 2.12. Line 1 is used to instruct the operating system to use bash as a command interpreter. Lines 2 and 3 set the partition as Blaise and ask for one node, respectively. Line 5 sets the number of tasks to 88, which is the number of threads of the Blaise node. Line 6 request the amount of time to run the job, here as 10 minutes. Finally, in the last line, there is the call to run the code contained in the Julia file.

Listing 2.12. Non-interactive Julia Example

```
#!/bin/bash

#SBATCH --partition=blaise
#SBATCH --nodes=1
#SBATCH --ntasks=88
#SBATCH --time=0:10:00

julia-1.7.2/bin/julia -t 88 ./selection_sort.jl
```

Once the SBATCH script is ready, for example with the name run_julia_selection_sort.slurm, you can request a non-interactive job to run it as:

```
$ sbatch run_julia_selection_sort.slurm
```

You must be aware that you can not request a longer computing time than the maximum allowed for a given partition, a value predefined by the cluster administrator. Also, if your computation is not done by the time it was requested for computing, you may loose the results.

2.5.3. Back to Julia Clusters

In order to start all workers with a certain project it is necessary to set the system environment variable `JULIA_PROJECT` as the directory path of the `Project.toml` file. Once this is done, we can check which packages and versions were loaded.

```
$ export JULIA_PROJECT="/home/users/rbklausner/Julia_HPC"

$ julia -p 2

julia> using Pkg

julia> Pkg.status()
      Status `~/Julia_HPC/Project.toml`
  [6e4b80f9] BenchmarkTools v1.2.2
  [052768ef] CUDA v3.8.0
  [35d6a980] ColorSchemes v3.16.0
  [916415d5] Images v0.25.1
  [91a5bcdd] Plots v1.25.7
  [2913bbd2] StatsBase v0.33.14
```

Julia can perform distributed computing on a high-level and on a low-level of control and communication to workers. Using at high-level we simply distribute computational effort along workers without worrying about the specific calculation being performed at a specific worker. At low-level programming, we are able to designate which computation will be performed at which worker in our cluster. The high-level option is easier while writing parallel code.

2.5.3.1. High-level distributed computing

The main tools for high-level distributed programming in Julia are the `@distributed` macro and the parallel map called `pmap`. To illustrate the use of both we first show how a simple function can be defined in Julia. This will be useful for understanding the behavior of `pmap`. We define a simple function that calculates the square of a number.

```
julia> f = i -> i^2

julia> f(12)
       144
```

In the example of high-level computing below we exemplify on how to call a distributed piece of code using `@distributed` and `pmap`. We start by calling Julia with 4 workers besides the master process. Then, using `nprocs()` we check the total number of processes, 5. The method `workers()` retrieves the ID of the workers, while `nworkers()` retrieves the number of available workers.

```
 $ julia -p 4

julia> nprocs()
5

julia> procs()
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> workers()
4-element Vector{Int64}:
 2
 3
 4
 5

julia> nworkers()
4

julia> myid()
1

julia> pmap(i -> println("Worker $(myid()), working on i=$i"), 1:10)
      From worker 4:    Worker 4, working on i=1
      From worker 4:    Worker 4, working on i=5
      From worker 4:    Worker 4, working on i=6
      From worker 4:    Worker 4, working on i=7
      From worker 4:    Worker 4, working on i=8
      From worker 4:    Worker 4, working on i=9
      From worker 4:    Worker 4, working on i=10
      From worker 3:    Worker 3, working on i=2
      From worker 2:    Worker 2, working on i=3
      From worker 5:    Worker 5, working on i=4

julia> @sync @distributed for i in 1:10
         println("Worker $(myid()), working on i=$i")
       end
      From worker 3:    Worker 3, working on i=4
      From worker 3:    Worker 3, working on i=5
      From worker 3:    Worker 3, working on i=6
      From worker 4:    Worker 4, working on i=7
      From worker 4:    Worker 4, working on i=8
      From worker 2:    Worker 2, working on i=1
      From worker 5:    Worker 5, working on i=9
      From worker 5:    Worker 5, working on i=10
      From worker 2:    Worker 2, working on i=2
      From worker 2:    Worker 2, working on i=3
```

The behavior of `@distributed` and `pmap` seems to be very similar. `pmap` works applying a function on a one-dimensional list or array. It passes each element of the list/array to each worker node, one at a time. Once a worker finishes the computation and retrieves the result, a new element of the list/array is sent to this worker. The process goes until the last element of the list is sent and computation performed.

Now, the `@distributed` macro works on a different manner. It splits the list/array in a set of blocks, with the number of blocks equal to the number of workers available. Then, each entire block is sent to each worker, at one shot, at the beginning of computation. Since `@distributed` executes the code asynchronously, it is necessary to wait for completion before going to the next instruction in Julia. For that purpose, we can use `@sync`.

Not waiting for a process to complete may generate sync errors.

Since the `pmap` requires a larger amount of communication steps, it will perform better if the calculation on a single element consumes a good amount of time.

We must introduce another macro, the `@everywhere`, that will help us on the distributed computing. Is is used to run the same piece of code over all processes in the cluster. An example of this use is calculating the realization of a stochastic function for different seeds.

```
julia> @everywhere println(rand())
0.856476480924345
      From worker 2:    0.3882556897855819
      From worker 5:    0.4721670866061788
      From worker 3:    0.22741379804937245
      From worker 4:    0.21971772218390695
```

On the output above, the master process produces the first result, followed by the workers printing their results.

The example below presents the use of `map`, `pmap`, and `@distributed` for two functions, one of fast calculation with simple evaluation, the other one, a slow one, composed of a nested loop.

Listing 2.13. Pmap example

```
@everywhere function fast(x::Float64)
    return x^2+1.0
end

@everywhere function fast_distributed(arr::StepRangeLen)
    @sync @distributed for i in arr
        fast(i)
    end
end

@everywhere function slow(x::Float64)
    a = 1.0
    for i in 1:1000
        for j in 1:5000
            a+=asinh(i+j)
        end
    end
    return a
end

@info "Precompilation"
map(fast,range(1,1000,1000))
pmap(fast,range(1,1000,1000))
map(slow,range(1,1000,10))
pmap(slow,range(1,1000,10))
fast_distributed(range(1,1000,1000))

@info "Testing slow function"
@time map(slow,range(1,1000,10))
@time pmap(slow,range(1,1000,10))
@info "Testing fast function"
@time map(fast,range(1,1000,1000))
@time pmap(fast,range(1,1000,1000))
@info "Testing @distributed fast function"
```

```
@time fast_distributed(range(1,1000,1000))
```

We can regard the different running times below.

```
  $ julia -p 32 .\pmap.jl
[ Info: Precompilation
[ Info: Testing slow function
  2.411691 seconds (1 allocation: 144 bytes)
  0.350282 seconds (1.28 k allocations: 59.188 KiB)
[ Info: Testing fast function
  0.000011 seconds (1 allocation: 7.938 KiB)
  0.077671 seconds (46.37 k allocations: 1.568 MiB)
[ Info: Testing @distributed fast function
  0.018515 seconds (5.49 k allocations: 201.750 KiB)
```

And we can conclude, as expected, that, for the slow function, `pmap` has gains over the use of a simple `map`, more than 7 times faster. However, for the fast function, `pmap` has a worse behavior, being more than 7000 times slower, while `@distributed` is about 1600 times slower than `map`.

All such degrading behavior is due to the unnecessary distributed computing that includes communication performed. This also shows that `@distributed` is a better fit for simple computations per node, as `pmap` will be the successful one when intense (long) computations are done per node. Long here meaning that the amount of time spent on communication from the master process to the worker and back is negligible compared to the time spent on the computation.

2.5.3.2. Estimation of Pi via Monte Carlo

The next example calculates an approximation for π using the Monte Carlo technique. The idea is to sample N_{total} 2D points in a square of side 1 unit. From the sampled points, $N_{inner} < N_{total}$ will be inside the unit circle. Since the circle has area $\pi/4$, the ratio N_{inner}/N_{total} will be approximately $\pi/4$.

We make use of `pmap` for the computation on a distributed system with 88 processes. At this example we can see how easy is to transform a serial version to a parallel one using a high-level distributed computing technique of Julia. This requires one extra line of code and one use of `@everywhere`. At the end of computation we compare the speed of both codes.

Listing 2.14. Monte Carlo pi estimative

```
using Distributed

macro my_time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        Δt = (t1-t0)/1e9
        val, Δt
```

```
    end
end

@everywhere function darts_in_circle(N)
    n = 0
    for i in 1:N
        if rand()^2 + rand()^2 < 1
            n += 1
        end
    end
    return n
end

function pi_distributed(N, loops)
    n = sum(pmap((x)->darts_in_circle(N), 1:loops))
    4 * n / (loops * N)
end

function pi_serial(n)
    return 4 * darts_in_circle(n) / n
end

N = 20_000_000
loops = 100
pi_approx_ser, t_ser = @my_time pi_serial(N*loops)
pi_approx_ser, t_ser = @my_time pi_serial(N*loops)
pi_approx_par, t_dist = @my_time pi_distributed(N, loops)
pi_approx_par, t_dist = @my_time pi_distributed(N, loops)
println("Serial = $pi_approx_ser in $t_ser seconds")
println("Distributed = $pi_approx_par in $t_dist seconds")
```

Has the result :

```
  $ julia -p 88 .\monte_carlo.jl
Serial = 3.141549988 in 8.427463928 seconds
Distributed = 3.141601936 in 0.307187824 seconds
```

We can see that the parallel version was about 28 times faster. Similar calculations, for different number of processes are analyzed in Section 2.6. We pass now to a brief discussion of the low-level aspects of distributed computing with Julia.

2.5.3.3. Low-level distributed computing

The manner Julia implements low-level control on distributed computing is via **remote calls** and **remote references**. A remote call is a request to execute a function on a particular process. You need to wait (with `wait()`) the process to complete in order to fetch (with `fetch()`) the result into the master process. A remote reference is a reference to data living on a specific process.

The remote call is an asynchronous process. The remote call finishes almost immediately and it returns a **Future** object, which is the remote reference for the result of such call. The way we carry out a remote call is specifying, in sequence, which function we will apply, the number of the process where this function will be calculated, and the argument(s) to such function. The caller then waits (`wait()`) on the reference and once the call is completed, it can `fetch()` the result of the computation and use it. A example of such

use is below, where we call the function `sqrt` with the argument `4.0` in the process 2.

```
julia> a = remotecall(sqrt, 2, 4.0)
Future(2, 1, 6, ReentrantLock(nothing, Base.GenericCondition{Base.Threads.
    SpinLock}(Base.InvasiveLinkedList{Task}(nothing, nothing), Base.Threads.
    SpinLock(0)), 0), nothing)

julia> wait(a)
Future(2, 1, 6, ReentrantLock(nothing, Base.GenericCondition{Base.Threads.
    SpinLock}(Base.InvasiveLinkedList{Task}(nothing, nothing), Base.Threads.
    SpinLock(0)), 0), nothing)

julia> fetch(a)
2.0
```

Another way of low-level control on distributed computing is via the use of `spawnat`. As an example, we create first a remote call **r**. In the example we do not specify in which node it will be computed, since we used as second argument of `@spawnat` the option `:any`. The computation will be then performed on one of the nodes. We then create the variable **s** that adds one to the result of the first remote call. As described before, we need to use `fetch(r)` to pick the result of this first call, and then we can use this result for further calculations. While creating the variable `s`, we called `@spawnat`, again with the option `:any`, meaning that the operation of adding 1 to the result of fetch will be performed on any node. It is possible to specify different nodes for the two different calculations. To select a specific node, we would need just to define the desired node instead of using `:any`. On an example with 4 nodes, we could run the first calculation on node 3, while the second one on node 2.

```
julia> r = @spawnat :any rand(2,2)
Future(2, 1, 25, ReentrantLock(nothing, Base.GenericCondition{Base.Threads.
    SpinLock}(Base.InvasiveLinkedList{Task}(nothing, nothing), Base.Threads.
    SpinLock(0)), 0), nothing)

julia> s = @spawnat :any 1 .+ fetch(r)
Future(3, 1, 26, ReentrantLock(nothing, Base.GenericCondition{Base.Threads.
    SpinLock}(Base.InvasiveLinkedList{Task}(nothing, nothing), Base.Threads.
    SpinLock(0)), 0), nothing)

julia> fetch(s)
2×2 Matrix{Float64}:
 1.12129  1.28538
 1.75886  1.34516
```

Such level of control on distributed computing may be necessary on some applications. In the following, we present a large algorithm that makes use of `@spawnat`.

2.5.3.4. Application - Ant Colony Optimization Algorithm

Ant colony optimization (ACO) is a probabilistic technique to find minimum paths. Its use range from solving mathematical problems to concrete industrial applications. The algorithm can be seen as part of metaheuristic optimization algorithms.

The purpose of this example is to illustrate how a more sophisticated problem, whose solution makes use of several predefined functions, can be simply parallelized using Julia. Example 2.15 presents, first, an adaptation of the serial code available in https://github.com/schiegl/AntColony.jl. Then, it depicts a parallel version written by one of the authors of these notes, R.B. Klausner. The main changes to the serial code are one call of `@spawnat`, creating a dictionary, and coordinate calls using `@sync` and `@async`, besides calling the necessary packages for the computations and the **Distributed** one for the parallel computing.

Listing 2.15. Ant Colony Optimization

```
using Distributed

macro my_time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        Δt = (t1-t0)/1e9
        val, Δt
    end
end

@everywhere using LinearAlgebra, Random, StatsBase

# Make a path based on the Probability Matrix
@everywhere function travel(
        P::Matrix{Float64},
        start_node::Int
    )
    n_nodes = size(P,1)
    not_visited = collect(1:n_nodes)
    path = [start_node]
    P_copy = copy(P)

    node = start_node

    for i in 2:n_nodes
        # Sample a new node
        next_node = sample(1:n_nodes,Weights(P_copy[node,:]))
        # You can't revisit the node
        P_copy[:,1:end .== node] .= 0.0
        # Normalize the other nodes probabilities, to sum 1
        P_copy ./= sum(P_copy,dims=2)
        # Go to the next node
        node = next_node
        push!(path, node)
    end
    return path
end

# Calculate the distances from a path
@everywhere function edge_distances(
        dist_mat::AbstractMatrix{<:Number},
        path::AbstractArray{Int}
    )
    [dist_mat[to, from] for (from, to) in edges(path)]
end

# Return a array of all edges in a path
@everywhere function edges(path::AbstractArray{Int})
    path_len = length(path)
    [(path[i], path[mod1(i + 1, path_len)]) for i in 1:path_len]
end
```

```
# Calculate the Serial Ant Colony Optimization Algorithm
function ACO(dist_mat,max_iter=50,nants=5;
    β=2,α=1,Q=1,ρ = 0.5,verbose = true)

    # Attractiveness of the move, calculated a priori
    η = (1 ./dist_mat) .^ β
    # Trail level
    τ = ones(size(dist_mat)) .^ α

    best_path = nothing
    best_cost = Inf
    no_improv = 0
    n_nodes = size(dist_mat,1)

    for i in 1:max_iter

        solutions = []
        # Create the probability matrix
        P = η.*τ
        P ./= sum(P,dims=2)

        # Run all ants in serial
        for k in 1:nants
            # Start randomly in a node
            start_node = rand(1:n_nodes)
            # Need to return to the same node
            end_node = start_node
            # Calculate path and costs
            path = travel(P, start_node)
            cost = sum(edge_distances(dist_mat, path))
            push!(solutions, (cost, path))
        end

        # Sort solutions
        sort!(solutions)
        best_local_cost, best_local_path = solutions[1]
        # Check if found a better cost path
        if best_local_cost < best_cost
            if verbose
                println("Better solution found with cost
                $(best_local_cost) at iteration $(i)")
            end
            best_path = best_local_path
            best_cost = best_local_cost
            no_improv = 0
        else
            no_improv += 1
        end

        # deposit pheromones
        τ .*= (1-ρ)
        for (cost, path) in solutions
            Δτ = Q / cost
            for (from, to) in edges(path)
                τ[to, from] += Δτ
            end
        end
    end
    best_path,best_cost
end

# Calculate the Parallel Ant Colony Optimization Algorithm
function ACO_parallel(dist_mat,max_iter=50,nants=5;
    β=2,α=1,Q=1,ρ = 0.5,verbose = true)

    # Attractiveness of the move, calculated a priori
    η = (1 ./dist_mat) .^ β
    # Trail level
    τ = ones(size(dist_mat)) .^ α
```

```
    best_path = nothing
    best_cost = Inf
    no_improv = 0
    n_nodes = size(dist_mat,1)

    for i in 1:max_iter

        solutions = []
        # Create the probability matrix
        P = η.*τ
        P ./= sum(P,dims=2)

        # Create a dictionary with Process and Future
        # to fetch later
        par_results = Dict{Int,Future}()
        @sync for k in 1:nants
            # Choose which process will calculate this path
            w = mod1(k, nprocs())
            # Spawn the call at `w` process
            @async par_results[k] = @spawnat w begin
                start_node = rand(1:n_nodes)
                end_node = start_node
                path = travel(P, start_node)
                cost = sum(edge_distances(dist_mat, path))
                path,cost
            end
        end

        for k in 1:nants
            w = mod1(k, nprocs())
            # Retrieve the call from `w` process calculated earlier
            path,cost = fetch(par_results[k])
            push!(solutions, (cost,path))
        end

        # Sort solutions
        sort!(solutions)
        best_local_cost, best_local_path = solutions[1]
        # Check if found a better cost path
        if best_local_cost < best_cost
            if verbose
                println("Better solution found with cost
                $(best_local_cost) at iteration $(i)")
            end
            best_path = best_local_path
            best_cost = best_local_cost
            no_improv = 0
        else
            no_improv += 1
        end

        # deposit pheromones
        τ .*= (1-ρ)
        for (cost, path) in solutions
            Δτ = Q / cost
            for (from, to) in edges(path)
                τ[to, from] += Δτ
            end
        end
    end
    best_path,best_cost
end

# Fix Random seed
Random.seed!(123)

# Create weights of the graph
n = 300
dist_mat = rand(n, n)
```

```
for i in 1:n
    # No self loop (1/Inf = 0.0)
    dist_mat[i,i] = Inf
end
max_iter = 50;
nants = 100;
verbose = false

(best_path_ser,best_cost_ser),Δt_ser = @my_time ACO(dist_mat,max_iter,nants,
    verbose=false)
(best_path_ser,best_cost_ser),Δt_ser = @my_time ACO(dist_mat,max_iter,nants,
    verbose=false)
println("Serial:   best_cost = ",best_cost_ser)
println("Serial:   time = ",Δt_ser)
(best_path_par,best_cost_par),Δt_par = @my_time ACO_parallel(dist_mat,max_iter,
    nants,verbose=false)
(best_path_par,best_cost_par),Δt_par = @my_time ACO_parallel(dist_mat,max_iter,
    nants,verbose=false)
println("Parallel: best_cost = ",best_cost_par)
println("Parallel: time = ",Δt_par)
```

Running the serial versus the parallel version we can observe an speed up of more than 12 times, as shown in Listing 2.16.

Listing 2.16. Ant Colony Speed up

```
  $ julia -p 88 .\aco_parallel.jl
Serial:   best_cost = 59.01948498666938
Serial:   time = 208.416440638
Parallel: best_cost = 57.180082172457624
Parallel: time = 16.431587189
```

2.5.4. Final remarks on distributed computing with Julia

We have seen that Julia permits high and low-level control on distributed computed and how to perform calculations on both cases.

We would like to bring attention to the fact that, a typical **Cloud service** (as AWS, Azure, etc) for computation is just a cluster that can allow distributed and multicore computations. It is then possible to run a Julia application on such services. You can use the `Distributed` package or specific interfaces like `MPI.jl` to parallelize the code. In order to make your code reproducible in a cluster that does not have Julia installed, you can create an executable app built by `PackageCompiler`[8].

Julia has integration with cloud machine learning platforms like Amazon Sagemaker [9] and Azure (with a preconfigured Julia installation). Also, there is a specific Julia platform for cloud computing called JuliaHub [10].

2.6. Multicore and distributed speed up analysis

Once we have seen how to adapt serial code to its parallel version under the multicore and distributed settings, and we have slightly touched on their speed up performance for

[8] https://github.com/JuliaLang/PackageCompiler.jl
[9] https://d1.awsstatic.com/whitepapers/julia-on-sagemaker.pdf
[10] https://juliahub.com/lp/

one single setup of workers/threads/processes, we decided to illustrate how three of the previously described algorithms behave under a variable number of processes.

The table in Figure 2.2 presents the running time in seconds for 8 different number of processes. Then, figure 2.3 depicts the speed up obtained under such settings. The experiments were conducted using the partition *Blaise* on the **PCAD** cluster (see Section 2.8.1), a computational node with the following specifications: 2*x* Intel Xeon E5-2699 v4 Broadwell (Q1'16) 2.2 GHz, each one with 44 physical cores, adding up to 88 threads and 256 GB DDR4 RAM. The speed up analysis described here must not be taken as the best possible behavior Julia could provide us. The goal here is simply illustrate the behavior of the described techniques along these notes while using Julia for parallelizing code.

	Number of Processes							
Instance	1	2	4	8	16	32	64	88
Selection Sort (Threads)	2.987	2.233	1.385	0.966	0.596	0.456	0.328	0.298
Pi Monte Carlo (Distributed)	8.446	4.225	2.185	1.197	0.721	0.411	0.312	0.309
Ant Colony Optimization (Distributed)	208.4	141.0	85.9	52.8	30.7	20.2	17.1	16.4

Figure 2.2. Table with execution time of the instances in seconds.

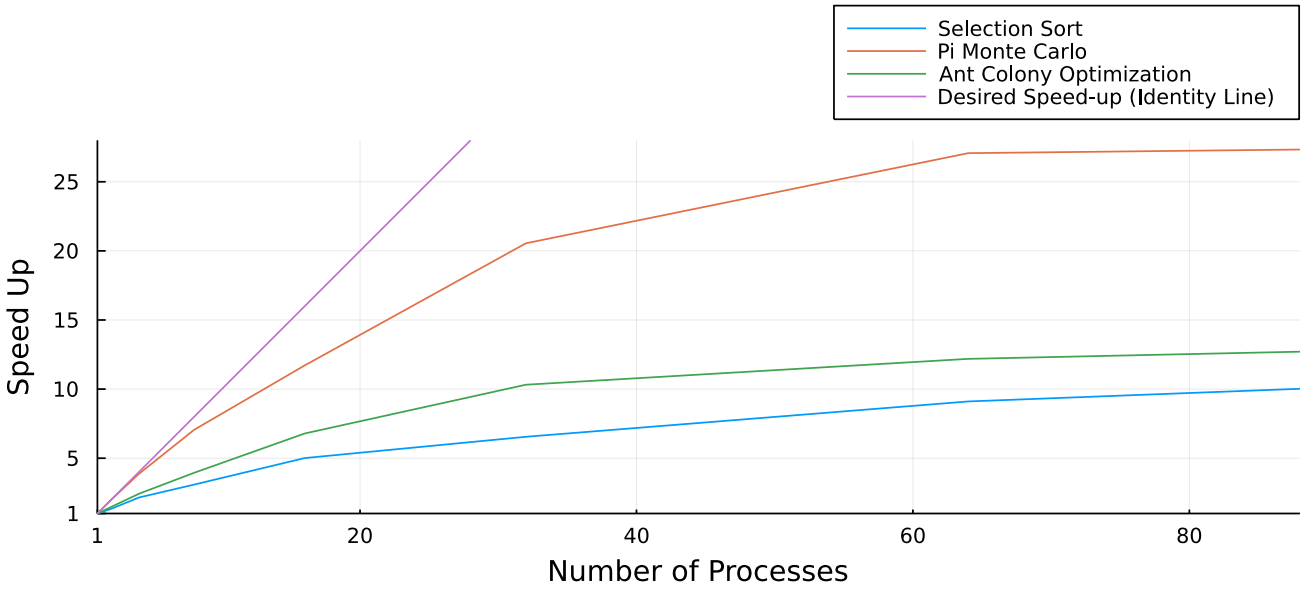

Figure 2.3. Speed Up

2.7. High-level GPU programming in Julia

High Performance and significant energy savings can be achieved by combining multicore processors and Graphics Processing Units (GPUs) [Nasciutti, Panetta e Lopes 2019, Serpa et al. 2019, Serpa et al. 2021]. It has become a popular choice for the top-ranked

HPC systems and for the yet to come exascale platforms [Dongarra e Strohmaier 2021]. GPUs have thousands of processing elements and different memory levels. Julia has support for programming in GPUs and can deliver high performance on these architectures.

Along this section, you need to have a computer with a compatible GPU and have installed CUDA [11]. We present here examples on how to use one single GPU only for computation with Julia, but the same package allows computations on multiple GPUs.[12]

Since we are restricting ourselves to work with one single GPU, GPU programming with Julia resembles multicore programming with CPUs. All examples of this section were computed using a NVidia Tesla P100, Pascal GPU with 3584 CUDA threads.

On the first example, Listing 2.17, we demonstrate GPU computations through high-level programming using the `CUDA.@sync` macro. The function of interest is just a simple sequential addition. We depict a serial version, a parallel one using threads, and a CUDA one. The only difference on the CUDA version of the function is the use of `@Cuda.@sync` . Also, at the moment of calling the function with CUDA version, we make use of `CUDA.fill` instead of the regular `fill` in Julia.

Desiring to measure the computational performance of these three versions, we use the macro `@benchmark`, from the **BenchmarkTools** package. This outputs on screen the behavior of the application. We can see in Figure 2.4 how such three versions performed.

Listing 2.17. Benchmarks

```
using CUDA
using Base.Threads
using BenchmarkTools


function sequential_add!(y, x)
    for i in eachindex(y, x)
        @inbounds y[i] += x[i]
    end
    return nothing
end

function parallel_add!(y, x)
    Threads.@threads for i in eachindex(y, x)
        @inbounds y[i] += x[i]
    end
    return nothing
end

function gpu_add!(y, x)
    CUDA.@sync y .+= x
    return
end


N = 2^23
x = fill(1.0f0, N)
y = fill(2.0f0, N)

println("Sequential Add:")
bench1 = @benchmark sequential_add!(x,y)
```

[11] https://developer.nvidia.com/cuda-downloads

[12] You can check also https://arxiv.org/pdf/1604.03410.pdf for further analysis on GPU programming with Julia, comparisons with C++ language and speed up analyses.

```
display(bench1)
println()

x = fill(1.0f0, N)
y = fill(2.0f0, N)

println("Parallel Add:")
bench2 = @benchmark parallel_add!($x,$y)
display(bench2)
println()

x_d = CUDA.fill(1.0f0, N)  # a vector stored on the GPU filled with 1.0 (Float32
    )
y_d = CUDA.fill(2.0f0, N)  # a vector stored on the GPU filled with 2.0

println("GPU Add:")
bench3 = @benchmark gpu_add!($x_d,$y_d)
display(bench3)
println()
```

The parallel version using threads is about 21 times faster, while the GPU version is about 28 times faster, all compared to the serial version. Also, the GPU version is about 30% faster than the parallel CPU version.

2.7.1. Example: Mandelbrot set

The next example constructs a plot of the Mandelbrot set[13]. This is the set of all complex numbers c for which the sequence defined by the quadratic recurrence equation

$$z_{n+1} = z_n + c,$$

with $z_0 = c$, is bounded. The plot is constructed by a simple escape time algorithm: the recurrence is evaluated for each (x,y) point in the plot area and depending on how fast it diverges, a color is given to the point. We present both the serial and the GPU versions of functions for such calculations. We also use the GPU code to produce the final picture of the set, seen in Figure 2.5. The analysis is inspired on https://github.com/vinifda/Mandelbrot-julia/blob/main/src/Mandelbrot_gpu.ipynb.

The main differences between the serial and the GPU version is the use of the `CUDA.CuArray` type, instead of the simple `Array` type Julia has by default, and specific CUDA programming fuctions, that are intuitive, and calculate the same as their homologue ones, as `CUDA.abs2`, `CUDA.Complex`, `CUDA.@time`. As in the first example, we also need to make use of `CUDA.@sync`.

Listing 2.18. MandelBrot CUDA

```
using CUDA, Images

function coloring(n)
    a = 0.1;
    return RGB(0.5 * sin(a * n) + 0.5,
            0.5 * sin(a * n + 2.094) + 0.5,
            0.5 * sin(a * n + 4.188) + 0.5)
end
```

[13] https://mathworld.wolfram.com/MandelbrotSet.html

```
/home/users/rbklausner/Julia_HPC
Sequential Add:
BenchmarkTools.Trial: 878 samples with 1 evaluation.
 Range (min … max):  5.638 ms …   7.765 ms  ┊ GC (min … max): 0.00% …
0.00%
 Time  (median):     5.654 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   5.680 ms ± 158.604 µs  ┊ GC (mean ± σ):  0.00% ±
0.00%
```

```
  5.64 ms      Histogram: log(frequency) by time      6.73 ms <

 Memory estimate: 0 bytes, allocs estimate: 0.
Parallel Add: 88 threads
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  226.814 µs …  16.809 ms  ┊ GC (min … max): 0.00% …
93.12%
 Time  (median):     251.220 µs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   270.542 µs ± 336.821 µs  ┊ GC (mean ± σ):  2.90% ±
2.32%
```

```
  227 µs           Histogram: frequency by time          336 µs <

 Memory estimate: 42.05 KiB, allocs estimate: 441.
GPU Add:
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  201.902 µs … 251.993 µs  ┊ GC (min … max): 0.00% …
0.00%
 Time  (median):     205.429 µs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   205.669 µs ±   1.730 µs  ┊ GC (mean ± σ):  0.00% ±
0.00%
```

```
  202 µs           Histogram: frequency by time          210 µs <

 Memory estimate: 1.61 KiB, allocs estimate: 23.
```

Figure 2.4. Benchmarks

```
# Image definitions
width, height = 301*2,241*2
width -= 1
height -= 1
xmin = -2.2
xmax = 0.8
ymin = -1.2
ymax = 1.2
h_x = (xmax-xmin)/width
h_y = (ymax-ymin)/height
xs = xmin:h_x:xmax
ys = ymin:h_y:ymax
xsize = length(xs)
ysize = length(ys)
maxiter = 1024;

# Serial function to compute the number of iterations
# necessary to the recurrence diverge
function get_steps(c::Complex, max_steps::Int64)
    z = Complex(0.0, 0.0) # 0 + 0im
    for i=1:max_steps
        # Calculate the recurrence
        z = z^2+c
        # Diverged, return the number of iterations
        if abs2(z) >= 400
            return i
        end
    end
    # The recurrence is bounded
    return max_steps+1
end

mandel(x,y) = get_steps(Complex(x, y), maxiter)
@time Z = [mandel(x,y) for y in ys, x in xs];
@time Z = [mandel(x,y) for y in ys, x in xs];

# GPU function to compute the number of iterations
# necessary to the recurrence diverge
function get_steps_gpu(c::CUDA.Complex, max_steps::Int64)
    z = CUDA.Complex(0.0, 0.0) # 0 + 0im
    for i=1:max_steps
        z = z^2+c
        if CUDA.abs2(z) >= 400
            return i
        end
    end
    return max_steps+1
end

Z = CUDA.CuArray([Complex(x,y) for y in ys, x in xs]);
cu_steps = CuArray(zeros(Int, (length(ys), length(xs))));
CUDA.@time CUDA.@sync cu_steps .= get_steps_gpu.(Z, maxiter);
CUDA.@time CUDA.@sync cu_steps .= get_steps_gpu.(Z, maxiter);
values = Array(cu_steps);
img = coloring.(collect(values))
save("mandelbrot_CUDA.png", img);
```

```
  $ julia --project=.. .\mandelbrot.jl
  0.312863 seconds (389.37 k allocations: 22.755 MiB, 31.05% compilation time)
  0.258803 seconds (87.25 k allocations: 6.455 MiB, 16.52% compilation time)
 19.873111 seconds (35.58 M CPU allocations: 1.870 GiB, 4.33% gc time)
  0.000968 seconds (23 CPU allocations: 1.281 KiB)
```

The GPU version is about 270 times faster than the serial one. The CUDA first call takes a long time to compile and should not be taken into account to make a time comparison.

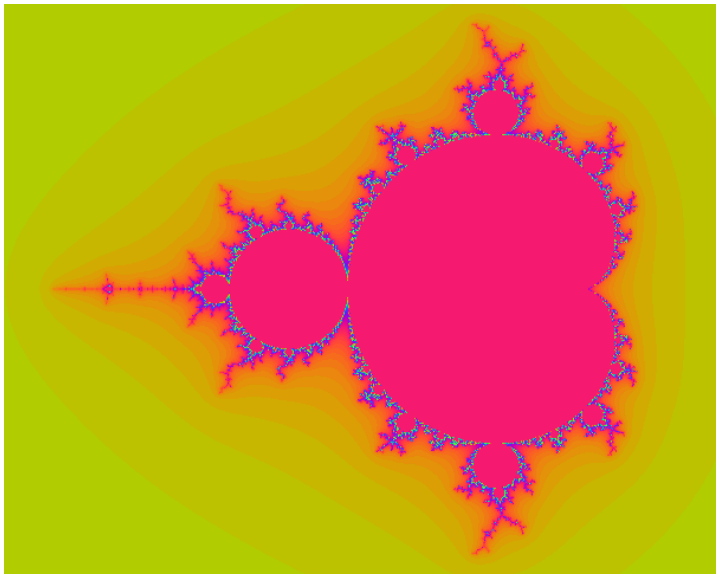

Figure 2.5. Mandelbrot plot

2.8. Conclusion

Along these notes we presented the use of the Julia language for HPC systems. We covered from its installation on a typical HPC cluster to parallel programming via multicore, distributed, and GPU architectures. Our motivation for the notes was the lack of a good reference with easy to access and updated to the current version of Julia for students and enthusiasts of HPC systems. Also, the complete absence of Julia on courses for HPC systems. We expect that these notes can serve as a first step for the HPC user that wishes to use Julia or for the Julia user that would like to speed up his/her applications running in parallel.

2.8.1. Acknowledgements

R.M. Velho would like to acknowledge Prof. Eduardo S. L. Gastal from the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS) - Brazil for the long transfer of knowledge on Julia language and its use for R&D

A.M.A. Côrtes would like to acknowledge Prof. Ricardo Rosa from the Applied Mathematics Department of the Federal University of Rio de Janeiro (UFRJ) - Brazil for helpful discussions regarding Julia's performance.

All the authors would like to thank the High Performance Computing Group (PCAD) http://gppd-hpc.inf.ufrgs.br/ at the Federal University of Rio Grande do Sul - Brazil for making available the use of its multiple clusters while the codes used in these notes were written and tested.

Several examples were inspired on the book [Sengupta 2019]. One of the few published book on the topic of HPC with Julia. We thank the author for the effort on gathering all that information.

References

[Carrijo Nasciutti et al. 2019] Carrijo Nasciutti, T., Panetta, J., and Pais Lopes, P. (2019). Evaluating optimizations that reduce global memory accesses of stencil computations in gpgpus. *Concurrency and Computation: Practice and Experience*, 31(18):e4929. páginas

[J. Dongarra and Strohmaier 2021] J. Dongarra, H. M. and Strohmaier, E. (2021). Top500 supercomputer: November 2021. https://www.top500.org/lists/2021/11/. [Accesed in: 3 Mar. 2022]. páginas

[Sengupta 2019] Sengupta, A. (2019). *Julia High Performance Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond*. Packt Publishing. páginas

[Serpa et al. 2019] Serpa, M. S., Moreira, F. B., Navaux, P. O., Cruz, E. H., Diener, M., Griebler, D., and Fernandes, L. G. (2019). Memory performance and bottlenecks in multicore and gpu architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 233–236. IEEE. páginas

[Serpa et al. 2021] Serpa, M. S., Pavan, P. J., Cruz, E. H., Machado, R. L., Panetta, J., Azambuja, A., Carissimi, A. S., and Navaux, P. O. (2021). Energy efficiency and portability of oil and gas simulations on multicore and graphics processing unit architectures. *Concurrency and Computation: Practice and Experience*, 33(18):e6212. páginas

[Yoo et al. 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer. páginas