Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Design and Optimisation of Large Scale Quantum Physics Simulations

Relatore: *Claudia Raibulet*

Co-relatori: *Dolf Huybrechts & Wouter Verstraelen*

Relazione della prova finale di:

Michele Pugno

Matricola 830513

Anno Accademico 2019-2020

Table of Contents

A. Preface

Modern computing embraces a wide range of science and engineering fields, continuously searching for new standards of reliability and efficiency while keeping up with the fast paced world we live in: this often leads to a dichotomy between performance and ease of use and development.

In high performance numerical computing, top notch libraries and frameworks are usually implemented in C and its dialects and, subsequently, wrapped in higher level code (e.g. Python) in order to be quickly deployed. Even though this solution may lead to impressively good results, a simpler and more efficient stack may be game-changing.
Conceived by Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman, Julia[1] is a young, high-performance, general purpose programming language released for the first time in 2012 under MIT license. The ambition of the creators was to provide a JIT free programming language which was both fast and high-level.

Due to an impressive community effort, the Julia environment is growing extremely fast from scientific to general-purpose computing: the possibility to directly call C and Fortran code with almost no overhead, while being able to interoperate with languages, such as Python and R, gives the community extra preexisting exploitable ecosystems.
Interest is also awoken by its easy to read syntax and by its full fledged array of parallelisation capabilities.

Hereupon, the CMA laboratory and the Physics department of Antwerp University decided to approach this new technology aiming to lower their computing facility workload, while increasing the precision and scale of their quantum physics simulations executed on the cluster.

This document embeds the knowledge gained while optimising and re-designing pre-existing HPC(High performance computing) software, offering design solutions to Differential Equations based problems. The aim of the work is to provide a viable introduction to whom may move its first steps in High-Performance computing using Julia.

The main content revolves around 2 projects:

- The former (Section C) one focuses on the difference between Julia and Matlab, which currently is the main technology used by the Physics department. Design techniques used to yield a working supercomputer Montecarlo ODE simulation with a bespoke scheduler are here discussed.
  The versione of Julia used is 1.4.1.
- The latter (Section D) exploits the benefits of a renewed Julia distribution which provides enhanced and stable multi-threading machinery. The different nature of the Physics system, described via SDE, and the new language version led to new implementation choices.
  The versione of Julia used is 1.5.1.

Besides, the first section aims to provide a brief walkthrough about the Julia language focusing on its most interesting peculiarities used for these projects. This should not be considered entirely an original work, but a set of useful knowledge collected by the author. Referencing has been tackled as consistently as possible to give credit to the sources with special regard to academic publications, cited in their reserved section.

[1] Julia: A Fresh Approach to Numerical Computing. Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah. (2017) SIAM Review, 59: 65–98. doi: 10.1137/141000671. pdf

B. The Julia Language

1. Introduction and Preliminary Tips

References and Primers

The main tool for every developer is the official documentation. It is strongly suggested to read its introductory section, especially during the very first steps.
Tons of learning resources can be found on the official website.

Aurelio Amerio's guide, master student in Physics at the University of Turin, is probably the best one for scientists not directly involved in the computer science field.

Widely used IDEs have their own extension supporting Julia including VSCode and the Atom based Juno, which is by far the most feature-rich and maintained one.

For anyone interested in the language development or source code, the main references are the official repository with the latest updates and the official releases.

The Performance Tips section is intended for advanced users already confident with Julia.

Eventually, a cheat sheet may result handy.

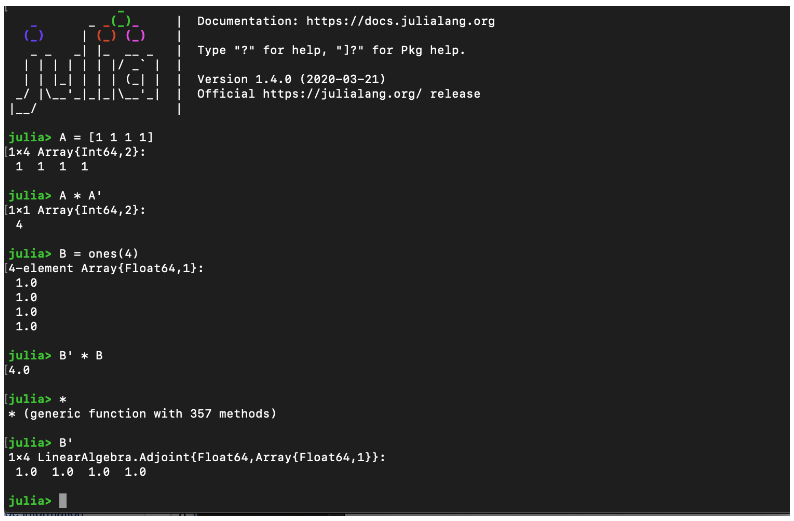
User Interface

Julia interface is conceived as an interactive environment the user mainly interacts with using a command line called REPL (read-eval-print loop).

Variables, constants, functions and more can be defined or imported in global scope, then called similarly to Python IDLE.

Basic, not interactive, usage of Julia is:

Bash:~/$ julia myScript.jl



```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.4.0 (2020-03-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> A = [1 1 1 1]
1×4 Array{Int64,2}:
 1  1  1  1

julia> A * A'
1×1 Array{Int64,2}:
 4

julia> B = ones(4)
4-element Array{Float64,1}:
 1.0
 1.0
 1.0
 1.0

julia> B' * B
4.0

julia> *
* (generic function with 357 methods)

julia> B'
1×4 LinearAlgebra.Adjoint{Float64,Array{Float64,1}}:
 1.0  1.0  1.0  1.0

julia> 
```

2. Compiler and Type Stability

Compilation and Primitive Types

Julia is a LLVM compiled language: compilation is done just-in-time during the first run of a function, yielding the corresponding machine code.

That means the first execution includes an overhead, sometimes not negligible.
In order to avoid it, packages, also called modules, may be pre-compiled ahead and the machine code, thus generated, stored for future use.
User code is instead evaluated and compiled on-the-fly.

The output quality of the toolchain may vary accordingly to the source code: even though Julia is defined as a dynamic language, massive type specialisation and type inference are its key strengths.

When a function is called, a specialised version of it is compiled given the arguments in input.
Multiple dispatch allows the programmer to create different variants of the same function in order to match all the specific use cases.
Each implementation is referred to as 'method'.

```julia
function mySum(a::Int64, b::Int64)
    a + b + Int32(1)
end

function mySum(a::Int32, b::Int32)
    a + b + Int32(2)
end

mySum(Int64(4), Int64(3)) # calling first function -> 8 of type Int64
mySum(Int32(4), Int32(3)) # calling second function -> 9 of type Int32
mySum(Int32(4), Int64(3)) # error, no method matching types
```

Different behaviour is shown by 'generic functions': their type is not explicitly assigned and it is the compilers duty to find the best solution according to the input.

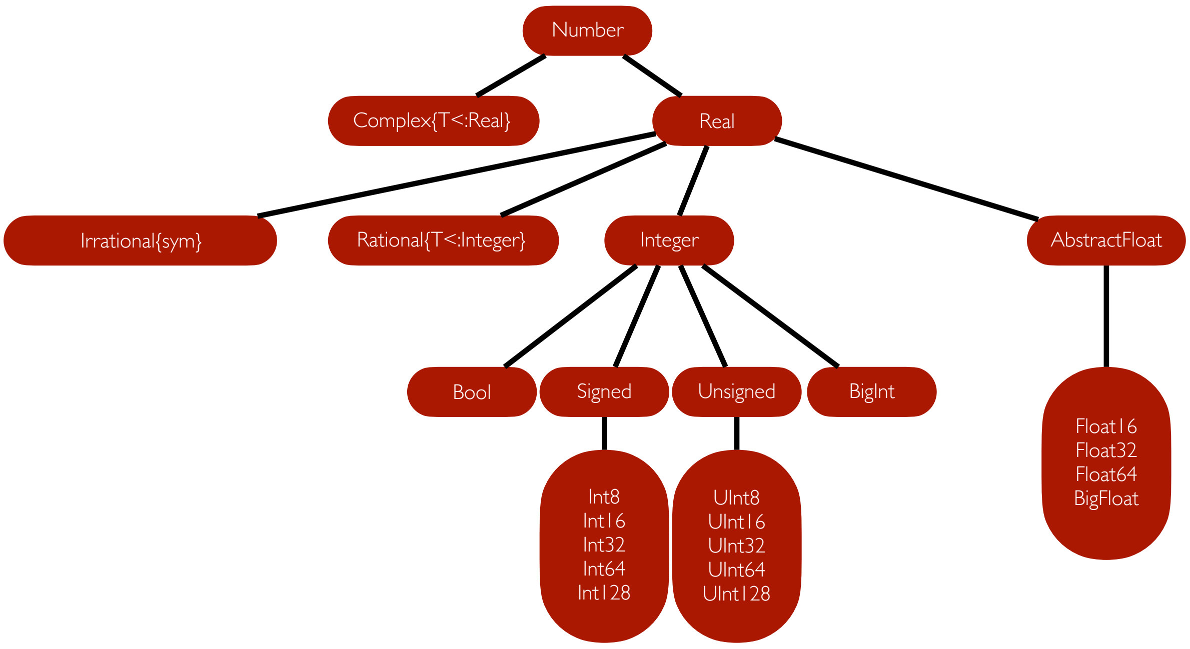
Similarly, the same approach is used for the output, hence writing it explicitly could be useful in a broad range of situations.

```julia
function mySum(a::Int64, b::Int64)::Int64 # output is defined as Int64
    a + b + 1
end

function mySum(a::Int32, b::Int32)::Int32 # output is defined as Int32
    a + b + 2
end

function mySum(a, b)
    a + b
end

mySum(Int64(4), Int64(3)) # calling first function -> 8 of type Int64
mySum(Int32(4), Int32(3)) # calling second function -> 9 of type Int32
mySum(Int32(4), Int64(3)) # calling generic function -> 7 of type Int64
mySum(4, 4)               # calling generic function -> 8 of type Int64
mySum(4.0, 4)             # calling generic function -> 8.0 of type Float64
mySum(4.0, 4.0)           # calling generic function -> 8.0 of type Float64
```

In the last four scenarios a different optimised version of the generic function is compiled for each call and stored for further use during execution.

The examples shown referred to Primitive Types, however, above them, distributed following a hierarchical scheme, there is another layer of abstraction formed by Abstract Types.

Abstract Types[2]



Hierarchy of numeric types

Abstract Types cannot be instantiated, but allow programmers to write generic functions that can later be used as the default method by many combinations of concrete types. Thanks to multiple dispatch, the programmer has full control over whether the default or more specific method is used.

When the compiler cannot figure out the input type consistently, it produces a generic compiled version of the function that dynamically dispatches after each call resulting in greater overhead.
For instance, the abstract type Any triggers this:

```
function myFunction(values)
    values[end]
end

input = ["I", "Am", "number", 1] # of type Array{Any,1}
myFunction(input) # dynamically dispatched
```

An Array of type Any could contain whatever piece of information, leaving the compiler unable to infer anything about it.

Returned elements are as important as function arguments because they are treated in the same way: usually, to achieve good outcome in performance critical code, the function output should be type consistent with the input whenever possible.

[2] Official Documentation: Types Some material has been taken from the official documentation.

Structs and Immutable Types[3]

Similarly to C, *struct* is the keyword to create user defined composite types.
They are divided into struct and mutable struct :

- It is not permitted to modify a value of an immutable type. Its immutability allows the compiler to create code that copies it freely and, if small enough, stack allocates its value or passes bit types, like integers or float, directly to functions in registers.
- A mutable type is heap allocated and passed to functions by pointers.

```
struct solid
    abstractNumber::Number
    generic
    integer::Int64
end

mutable struct fluid
    abstractNumber::Number
    generic
    integer::Int64
end

Immutable = solid(4.0, "string", 1)
Mutable = fluid(4.0, "string", 1)

Mutable.generic = ("TwoElement", "Tuple")
```

It must be underlined that in this snippet both ***solid*** and ***fluid*** are ambiguous: in fact the *generic* entry can by assigned to any type forcing the compiler to generate extra instructions in order to resolve it at run-time resulting in slower code.
Being an abstract type, a similar slow down is delivered by the *abstractNumber* entry.

It is clear that very little can be inferred about those objects, therefore they should be treated carefully.

Arrays and Tuples

Fundamental and complementary built-in types are Arrays and Tuples: the former is a mutable type, thus strictly heap allocated, while the latter is immutable.

```
myTuple = ("Example", 1)
typeof(myTuple) # -> Tuple{String,Int64}

myArray = [1, 3, "Example"]
typeof(myArray) # -> Array{Any,1}
```

Due to tuple immutability, each *myTuple* entry is inspected by Julia even though not type consistent, while a completely different behaviour is applied to *myArray*.
Array type has two fields: the entry type and the number of dimensions.

It must be underlined that the compiler adapts its choices according to the array type without digging into its entries, therefore passing *myArray* as argument to a function is suboptimal: in fact compiler outputted machine code must be able to handle a broader range of scenarios at runtime, yielding a non optimised execution.

[3] Official Documentation: Types Some material has been taken from the official documentation.

Code Availability

As previously said, Julia code can be packed in modules: to make a module available, two main keywords can be used: **using** and **import**.
These behave as shown below:

```
using LinearAlgebra # importing the package
import Statistics   # importing the package

mul!(arg1, arg2, arg3)          # LinearAlgebra.mul!() function
value = Statistics.mean(arg3) # Statistics.mean() function
```

It is remarkable about the **using** keyword that a new user defined function cannot be named as one contained in the target package because all of them are imported in the current namespace: therefore, to avoid conflict, no new method for the *mul!()* function can be created. This obligation is not applied to the *mean()* function due to the use of the **import** keyword which preserves distinction between namespaces.

New modules can be easily added with the help of the built-in package manager.

```
import Pkg

Pkg.add("newModule")
```

The *include()* function, similarly to Python 'import' keyword, is used to execute a user defined Julia file making as a consequence all of its functions and structs available during execution.

```
include("file.jl")  # running file.jl
```

More about code loading can be found in its dedicated page[4].

Macro

Before going on, it is recommended to consult the macros documentation, the main keyword is **@**.

Composability and CUDA

How typing interacts with the compiler is a gigantic topic which cannot be discussed here exhaustively. One of the most remarkable things about the Julia programming language, is how well the packages **compose**. You can almost always reuse pre-existing code and extend it with ease. For instance, the user may create a custom made matrix type and extend the built-in library to support it while being able to run it on GPUs or cluster just by changing the array type which the matrix is made of.
This capability is also referred as **composability.**

For deeper understanding, the devdocs and manual dedicated sections are gold mines of knowledge.

The reference package for CUDA acceleration is CUDA.jl.

[4] https://docs.julialang.org/en/v1/manual/code-loading/

3. Allocations management and Benchmarking

Heap Allocations

One important parameter to consider while writing Julia code is the number of heap allocations the program does.
Moreover the reader should keep in mind Julia is garbage collected, hence it is not possible to directly deallocate memory; although the collector can be manually forced into execution via *GC.gc()* function.

As previously mentioned, mutable data structures are heap allocated, these include built-in arrays. Furthermore temporary data structures are produced and cache stored while attempting not in-place operations.

A conscious use of cache is absolutely reasonable, but the developer should mind memory use, especially while handling large amounts of data or repeatedly executed actions in order to not exceed cache size, lower the time spent by Julia garbage collector to free space and reduce expensive allocation requests Julia sends to the system whether more cache is required.

Let's consider the following snippet:

```julia
using LinearAlgebra     # importing package that includes mul! function

matrix = rand(200, 200) # Array{Float64,2} size 200x200 with rand entries
out = similar(matrix)   # Allocating an array similar to matrix


@time result = matrix * matrix
# out -> 0.703573 seconds (2.87 M allocations: 136.538 MiB, 5.14% gc time)
@time result = matrix * matrix
# out -> 0.000659 seconds (2 allocations: 312.578 KiB)
@time mul!(out, matrix, matrix)
# out -> 0.002306 seconds (32 allocations: 2.156 KiB)
@time mul!(out, matrix, matrix)
# out -> 0.000396 seconds
```

The **@time** macro yields the execution time, number and dimension of the allocations and the amount of time in percentage the garbage collector has spent freeing memory.
The **mul**! function is the in-place matrix multiplication operation imported from the linear algebra built-in library (the in-place behaviour is described by the best-practise of appending an exclamation mark "**!**").

The first call of each function shows a higher cost due to the compilation overhead discussed in the previous section.

The in-place variant, not forced to use a new piece of memory to store the output, which will be saved in the pre-existing ***out*** bi-dimensional array, does not need extra space once compiled, thus it is faster and more memory efficient at runtime.

BenchmarkTools

@time macro is a basic tool and lacks reliability while doing benchmarking.

The more sophisticated **BenchmarkTools** package includes macros like **@btime** and **@benchmark** and many more which test an expression or function call multiple times in order to reduce noise.

Extensive documentation is available in the GitHub /doc folder, also dealing with the creation of a suitable benchmarking environment in Linux.

Profiling

Profiling in Julia is anything but intuitive.
It combines startup flags, a dedicated built-in package and 3rd part modules, becoming more an art than a straightforward operation at this point of its development.

The **Profile** module provides a wide set of tools to help developers improve the performance of their code and represent the centre around which the entire profiling ecosystem revolves.

It mainly works by periodically taking a backtrace sample during the execution of any task. Beside being possible to analyse single functions at a time, enabling automatic analysis at program startup might be desirable; this is possible thanks to the following flags:

- `--code-coverage={none|user|all}, --code-coverage`
- `--track-allocation={none|user|all}, --track-allocation`

Profiling output can be quickly studied using the text based **Coverage.jl** module.
Graphical browsers are also available both exploiting Atom-Juno IDE or the **ProfileView.jl** package.

Essential readings with further details are the guides written by Constatin Berzan and Ole Kröger.

4. BLAS and mathematical routines

BLAS frameworks

In order to guarantee maximum performance, Julia implements by default OpenBLAS: functions part of the **LinearAlgebra** package act for certain inputs as a wrapper for BLAS or LAPACK API, moreover they can be called explicitly using **LinearAlgebra.BLAS** and **LinearAlgebra.LAPACK**.

There are highly optimised implementations of BLAS available for every computer architecture, and sometimes in high-performance linear algebra routines it is useful to call the BLAS functions directly.

Usually a function has 4 methods defined, one each for Float64, Float32, ComplexF64 and ComplexF32 arrays, but not for SparseArrays.

Optionally, Julia can be compiled with Intel MKL as BLAS vendor: this can be done manually from scratch or automatically by the **MKL.jl** module.

Similarly to OpenBLAS, the MKL build will not include sparse matrices routines. To enable them seamlessly, install the **MKLSparse.jl** package.

BLAS Threads

BLAS threads allow Julia to automatically exploit multi-core capabilities of the host machine while attempting a heavy load computation.

Their number is set at 8, or the number of physical cores, by default.
The user can set the number of threads the BLAS library should use via the *LinearAlgebra.BLAS.set_num_threads(n)* function.

There is no specific Julia function to get the current number of BLAS threads in use, thus a C call is necessary:

```
ccall((:openblas_get_num_threads64_, Base.libblas_name), Cint, ()) # using
openblas
ccall((:mkl_get_max_threads, Base.libblas_name), Cint, ())     # using MKL
```

SparseArrays

Support for sparse vectors and matrices, only in compressed sparse column (CSC) format, is provided by the **SparseArrays** standard library module.

The main object is structured as following

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column j is in colptr[j]:(colptr[j+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

The largest part of the numerical methods specifically designed for SparseArrays are written in pure Julia and do not take advantage of the BLAS framework.

StaticArrays

The **StaticArrays** module implements statically sized arrays in Julia using the abstract type *StaticArray{Size,T,N} <: AbstractArray{T,N}*. Its subtypes will provide fast implementation of common array and linear algebra operations.

Statically sized does not mean necessarily immutable: in fact the package also provides concrete types for specific use cases.Moreover, the implementation allows Julia to stack allocate these data structures whenever possible. While yielding a substantial speedup in a wide range of mathematical routines, due to the stress put on the compiler, this benefit is limited to arrays smaller than, roughly, 100 elements.

5. Parallel Computing[5]

For newcomers to multi-threading and parallel computing it can be useful to first appreciate the different levels of parallelism offered by Julia. They can be divided in three main categories :

1. Julia Coroutines (Green Threading)
2. Multi-Threading
3. Multi-Core or Distributed Processing

As usual, the official documentation is still a cardinal point of reference.

Green Threading

Julia **Tasks** (aka coroutines) allow the user to suspend and resume the computation with full control of inter-Tasks communication without having to manually interface with the operating system scheduler.

The behaviour resembles the feeder-consumer one adopted by Python threading library where a program decides what to compute and in which order based on when other jobs finish.
More concisely, when a piece of computing work is designated as a task, it can be interrupted switching to another task. Later it can be resumed, at which point it will pick up right where it left off. Even though this could sound similar to a function call, switching can occur in any order and does not consume the call stack.

In order to allow Inter-Task communication, Julia provides a Channel mechanism. Channels are FIFO queue like data structures useful to pass data and messages between running tasks.

```
myChannel = Channel{Any}(10) # creating a channel of size 10 able to contain Any
type

put!(myChannel, "FirstEntry") # putting a new value
getter = take!(myChannel)     # getting the first available value

close(myChannel)              # closing the channel
```

- Channels are created via the *Channel{T}(sz)* constructor. The channel will only hold objects of type T.
- If a Channel is empty, readers (on a *take!* call) will block until data is available.
- If a Channel is full, writers (on a *put!* call) will block until space becomes available.
- *isready(channel)* tests for the presence of any object in the channel, while *wait* waits for an object to become available.
- A Channel is in an open state initially. This means that it can be read from and written to freely via *take!* and *put!* calls. *close* closes a Channel. On a closed Channel, *put!* will fail.

[5] Official Documentation: Parallel Computing Some material has been taken from the official documentation.

Multi-Threading

Not to be confused with BLAS threads, **Julia threads** are natively supported and a substantial part of the machinery may be considered stable starting from Julia version 1.5.

By default every Julia process is single threaded, this can be checked with the *Threads.nthreads()* call.

The number of threads Julia starts up with is configurable with the startup flag *-t* similarly to the *-p* flag discussed in the next section.

Threads in the same pool have access to the same memory location, therefore the user must consider if operations are thread-safe.

Creating a shared memory parallel computing program in Julia is usually straightforward, the dedicated section of the official manual shows in details the procedure.

Multi-Process

Distributed memory parallel computing is provided by the **Distributed** module as part of the standard library.

Most modern computers have more than a single CPU, additionally different machines can be combined together forming a cluster.
Harnessing this greater computational workforce is essential to complete a large variety of massive tasks in a reasonable amount of time: the creation of different Julia processes is its foundation.
Julia's IPC (inter process communication) implementation is not MPI based: generally one-sided from a user perspective, it resembles high-level operations like function calls more than message-pass/message-receive ones.

Distributed programming is based upon two primitives: *remote references* and *remote calls:*

- Remote references come in two forms: Future and RemoteChannel:
  - A Future is an object which will store the returned value of a remote async call.
    This reference flavour is suggested when the returned object is large or monolithic.
  - A RemoteChannel works exactly as a common Channel, but each different process, owning its reference, can write and read from it.
    RemoteChannels are an incredibly powerful tool to synchronise operations between a large number of processes thread-safely.
- A remote function call returns immediately a Future.
  The program can be instructed to wait for the remote call to finish in order to fetch it or to move on asynchronously and fetch in a second time.
  An interesting async call with no returned Future is *remote_do()*.

Starting Julia with the *-p n* flag provides n worker processes on the local machine and implicitly loads the Distributed module.
Workers can also be spawned at runtime as follows:

```
using Distributed
addprocs(2) # spawning 2 workers
```

The new workers will have id 2 and 3, id 1 is reserved to the master process that on UNIX systems acts as fork parent to workers.
The master process should be left idle when workers are busy: in fact it coordinates its IPC and other hidden subroutines.
For the same reason it is suggested to assign RemoteChannels to the master process.

Julia also includes a machine-file mechanism, common also to other technologies, to dispatch processes across multiple computers with ease. There is a dedicated small section about the *--machine-file* start-up flag in the Getting Started page.

It is not uncommon to need a working solution in a short amount of time and lines of code, thus Julia offers a set of macro to quickly write parallel code: this could be handy and efficient, but the user might lose fine control over the execution as a whole. For interested readers, the official **documentation** shows common macro use and more about Parallel Computing in Julia.

SharedArrays and DistributedArrays

SharedArrays are peculiar objects which use shared memory to map an array over multiple processes. Each one has full access to the entire data structure.

SharedArrays are a good choice when a large amount of data requires to be shared across the entire worker pool in a memory efficient manner.
Indexing works just like regular arrays and the object can be efficiently used as it is also in common serial application if required.

The SharedArray is assigned to the process that created it and stored on the physical machine which hosts that process, thus the user has to consider when a **DistributedArray** is the correct alternative: in fact, whether working on a cluster or a computer network and not on a single node, a shared memory object would not be appropriate.

Performance Tips on Clusters

There are 2 major factors to take into account while developing in a distributed memory environment:

- memory access time, which could be not uniform through the entire system.
- communication overhead between processes, which may also be under execution on different physical hardware.

Memory access should not be considered a trivial task.
Each computing node is composed by several processors, each of them with a different set of memory controllers associated to specific groups of cores.The outcome of diverse set-ups are different NUMA (Not Uniform Memory Access) domains.
The final user has to be aware of such a hardware configuration in order to squeeze all the possible power from it and to avoid memory bottlenecks.

By default, Julia imports the operative system NUMA configuration and sets process affinity accordingly, but this should be checked manually whatsoever. Anyhow, it happened in the past that this automatic feature was removed from the official source code.

Manual affinity can be forced using the ClusterManager.jl module.
This piece of software packs also more interfaces for a wide range of batch schedulers.
It is recommended to download it directly from the GitHub page.

The second crucial tip is to lower the amount of IPC executed by the program. While working with threads, this is not such a problem because they share the same memory, however this cannot be done in a distributed system, like a multi-node task, or in a worker pool.

6. LowLevel Optimisation and SIMD

The Compiler Output

Julia LLVM based JIT compiler usually operates under the hood, anyway, during development, studying its output may be a game-changer in performance critical code.

The **@code_** family is a set of 5 macros designed for that purpose:

1. **@code_warntype** verifies type stability of the given expression and may be crucial during the first optimisation phases.
2. **@code_typed** is useful to check which specific method has been chosen by the compiler for a given input after type inference and inlining.
   The output is not supposed for human consumption, however could displays quite interesting pieces of information.
3. A similar output is given by **@code_lowered**, but with less typing information because is a step behind in the inference process if compared to @code_typed.
4. **@code_llvm** shows the LLVM byte-code.
5. **@code_native**, eventually, yields the machine code.

More details are available from the community and the official documentation.

SIMD

Julia has full support for instruction level parallelisation.
Modern CPUs have the ability to run the equivalent of more than one instruction in a single clock cycle exploiting a specific SIMD instruction set such as Intel AVX.

The Julia compiler is largely autonomous while dealing with SIMD instructions, but some clever syntax should be followed.

Multiple vectorised operations (familiar to Matlab users) can be fused in a single loop without allocating any extraneous array.
The **.** turns every operator in its vectorised element-wise counterpart:

```
a = rand(40,40) # random 40x40 float matrix
b = rand(40,40)
c = similar(b);

c .= a .* b
```

The whole computation will be fused in a single loop with similar performance to its hand-written devectorised version:

```
for index = 1:length(a) # linear index size of a
    c[index] = a[index] * b[index]
end
```

Syntactic sugar is not the only reason why loop-fusion should be considered.
Looking at the LLVM byte-code, it is clear the vectorised alternative is natively compiled exploiting SIMD processor capabilities.

Consider the following part of LLVM code generated by the *c .= a .* b* expression:

……

```
; │││ @ simdloop.jl:77 within `macro expansion' @ broadcast.jl:910
; │││┌ @ multidimensional.jl:545 within `setindex!' @ array.jl:828
      %321 = add i64 %159, %index
      %322 = getelementptr inbounds double, double addrspace(13)* %138, i64 %321
      %323 = bitcast double addrspace(13)* %322 to <4 x double> addrspace(13)*
      store <4 x double> %314, <4 x double> addrspace(13)* %323, align 8
```

……

<4 x double> in LLVM language means multiple data operation, precisely AVX2 in this case, so the compiler is clever enough to choose the best solution automatically.

However, the explicitly devectorised version, as it is, does not make the compiler aware of the needed piece of information to thrive on SIMD, yielding worse performance and a completely SIMD free LLVM byte-code.

Array Bound Checking

In order to reach the same result of the vectorised code, the compiler must be told to skip array bound checking while iterating the loop.
This can be achieved using the **@inbounds** macro:

```
@inbounds for index = 1:length(a) # linear index size of a
    c[index] = a[index] * b[index]
end
```

Starting with Julia *--check-bounds=yes* enables bound checking whenever possible overriding all the @inbounds statements embedded in the code.

More about SIMD

The community is pushing the Julia compiler even further with the ambition to reach high-end native performance.

A remarkable piece of work is LoopVectorization by ChrisElrod. Displayed in a community post, the package author discussed benchmarks comparing native Julia code, compiled with the help of his module, and the most widely used BLAS libraries, including MKL, in a series of common routines.
Results are remarkable: Julia with LoopVectorization arises as one of the fastest implementation, outperforming MKL in some scenarios.

More about Julia instruction level parallelism is available on the official Intel site and Kristoffer Clarsson's blog.

Inlining

Inlining is an important low-level optimisation technique which replaces a function call site with the body of the whole function. This procedure trades memory for speed, but has intricate effects on performance and should be carefully implemented.

This compiler level operation can be triggered manually with the **@inline** macro or globally controlled with the starting flag *--inline={yes,no}.*

More Compiler Flags

Two others noteworthy startup flags are:

- *-O* flag triggers the compiler optimisation level to 3. This may improve performance.
- *--math-mode={ieee,fast}* flag Disallows or enables unsafe floating point optimizations, overriding **@fastmath** declarations.

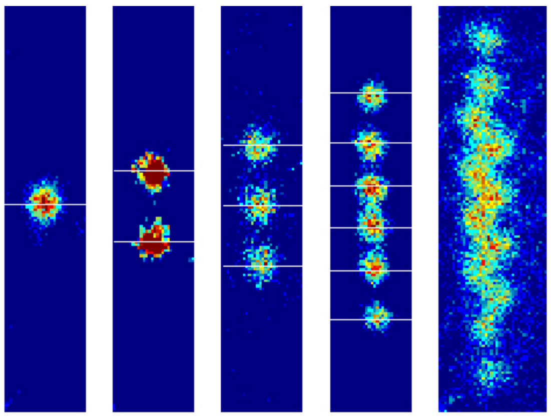
C. From Matlab to Julia

1. Problem Statement

Introduction

The aim of the project was to implement in Julia, improving performance and reliability, a pre-existing Matlab code, part of D. Huybrechts and M. Wouters's "**Cluster methods for the description of a driven-dissipative spin model**"[6], and to study the similarities of the technologies.

The program describes the dynamics of a lattice of interacting qubits. The strength of these interactions can be tuned and the code attempts to describe the systems properties for a range of these interactions.


False color of 1, 2, 3, 6, and 12 trapped magnesium ions lattice, a way to physically implement qubits [credits] [Wikipedia]

A qubit is the quantum version of a traditional computing bit. A classical bit can take on a value of zero or one at a given time. In the world of quantum mechanics, a qubit can also take on a value, also referred as state, equal to zero or one at a given time, but the laws of quantum mechanics also allow it to be in a superposition of these two states. This means it can take on both values at the same time. This property gives rise to a lot of interesting physics, but can also make the description of multiple qubits very hard. After all, if two qubits are considered then they can be described by four possible states (00, 01, 10 and 11) and a superposition of them. The same goes for three qubits, where there are eight possible states (000, 001, 010, 100, 011, 101, 110 and 111), and again a superposition of them.

Note that a collection of traditional bits could also attain any of these states, but never all at the same time. This growth of the number of states, as more qubits are introduced, is exponential and already for a small number of qubits one needs to keep track of a lot of states simultaneously, which becomes computationally infeasible very quickly. As an example, the number of states describing a collection of 275 qubits is already well above the estimated number of atoms in the universe! It should be clear that since these problems can be computationally very demanding, efficient technologies and simulation algorithms are an absolute necessity.

The above figure gives an experimental example of a system of magnesium ions trapped in a lattice, in which qubits can be implemented.

Considering a lattice of finite size, including a certain number of ions, each of them representing a piece of quantum information, and a given set of parameters describing the initial state of the system, the algorithm attempts a Montecarlo Simulation of the wave function in order to study their magnetic interaction and evolution over time.

[6] https://journals.aps.org/pra/abstract/10.1103/PhysRevA.99.043841

Dynamic System Representation

The time evolution is modelled as the evolution of a differential equation combined with a jump operation, which occurs when certain conditions are met and is stochastically sampled.

The time evolution of the equation is described by

$$i\hbar\frac{d\psi}{dt} = H\psi$$

with ψ the state of the system and H the Hamiltonian matrix, which is defined at the beginning of the simulation and contains information on the interactions present in the system. $\hbar$ is the Planck constant, but we will use units in which it is equal to one. The problem under consideration includes nearest-neighbour interactions between the qubits on a lattice, of which the strength can be tuned. They will be referred to as the coupling parameters in the x, y and z-direction.

ψ is the superposition of each possible realisation of the system and can be written in vector notation

$$\psi = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix}$$

In other words, each entry c_i belongs to one of the states the system can attain (e.g. the 00 or 10 state for a two qubit system), and is a measure for the probability of finding the system in this state. This means the differential equation can be written as

$$\frac{d}{dt}\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} H_{11} & H_{12} & \dots & H_{1n} \\ H_{21} & H_{22} & \dots & H_{2n} \\ \vdots & \vdots & \dots & \vdots \\ H_{n1} & H_{n2} & \dots & H_{nn} \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix}$$

In this form, it is clearly a system of ordinary differential equations.
The coefficient c_i evolves as

$$\frac{dc_i}{dt} = \sum_j^n H_{ij}c_j$$

It should be clear that under the influence of the entries of the matrix H, the different states can be transformed into each other. This should not come as a surprise since it has previously been noted that the interactions in the system are contained in the matrix H.

Additionally, a jump occurs when a certain condition, that depends on a randomly chosen ϵ value, is reached: more specifically, when $|\psi|^2 \leq \epsilon$. The jump then takes place and also has an influence on the state of the system. Physically, it represents the flip of a qubit due to an interaction with the environment. After the jump has occurred, another random value is assigned to ϵ and the system again evolves according to the differential equation until the condition is again satisfied, and so on.

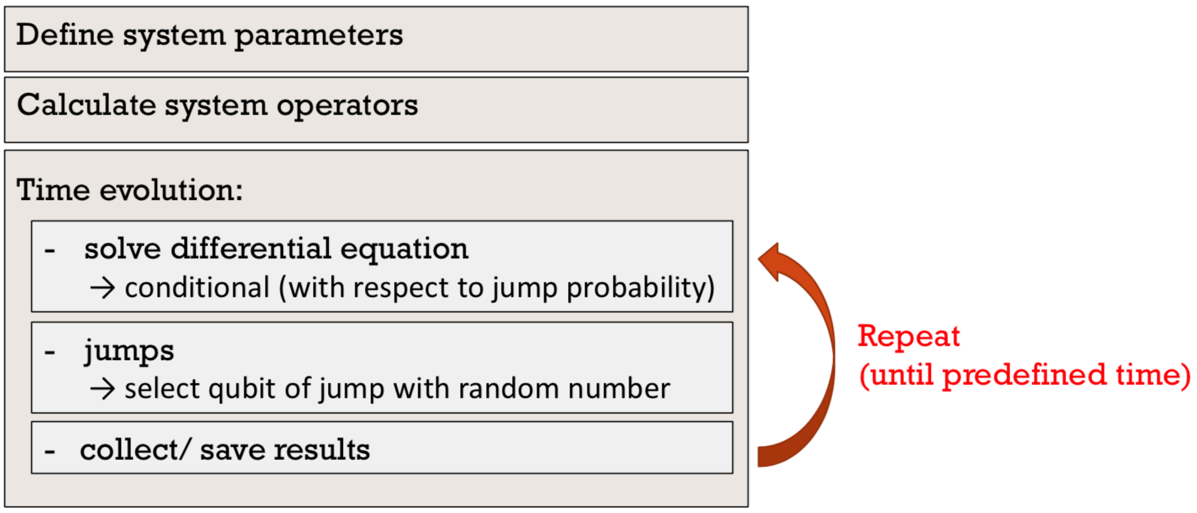
Problem Complexity

The ψ vector has 2^n entries and the Hamiltonian matrix H has 2^{2n} entries: therefore, the space cost grows exponentially $O(2^n)$, where n is the number of qubits that the user wants to simulate.

2. Matlab Implementation

Overview

A brief overview of original code-base will be essential in the next sections. Its structure is synthetically described in the picture below:


Define system parameters
Calculate system operators
Time evolution:
- solve differential equation
→ conditional (with respect to jump probability)
- jumps
→ select qubit of jump with random number
- collect/ save results
Repeat
(until predefined time)

The system parameters are embedded inside the script and the source code has to be modified to change them :

- *Nx{Int}:* represent the number of qubits inside the lattice over the x dimension.
- *Ny{IntArray, 1}:* represent the number of qubits inside the lattice over the y dimension.
- *gamma{Float}:* the coefficient of the coupling parameters.
- *Jy{FloatArray, 1}:* represent the set of desired coupling parameters for the y dimension.
- *Jx {Float}*: coupling parameter x dimensions.
- *Jz {Float}*: coupling parameter z dimensions.
- *T{Int}:* size of the differential equation time window(referred as "predefined time" in the picture).
- *dt{Int}:* after each 'dt', the program makes a snapshot of the state of the system.
- traj{int}: optional; it describes the number of times the simulation with a specific configuration and *T* has to be repeated in order to increase stochastic reliability.

The program folder is depicted in the tree graph aside:

- The .m files in green are the main script for parallel and serial execution.
- The red file contains the function dedicated to the study of the time evolution of a system of N qubits with a certain starting condition.

```
CodeBase
├── CalcAlternativeOperator.m
├── CalcExpSig.m
├── CalcNeighbours.m
├── CalcNeighbours1D.m
├── CalcRho.m
├── CalcSss.m
├── CalculateSssxx_Exact.m
├── ChanceInterval.m
├── EffH.m
├── GetAllOperatorsSparse.m
├── GetOperatorsSparse.m
├── GetSite.m
├── H1_exact.m
├── Jump.m
├── Mod.m
├── computation_time_parallel.m
├── computation_time_system_size.m
├── myEvent.m
├── odeC.m
└── superkronSparse.m
```

Parallelisation

The given code is parallelised only over the *Jy* array using the ***parfor*** Matlab keyword. Specialised main scripts had to be written in order to iterate over other coupling parameters or variables.

3. Julia Implementation

Mindset and First Attempt

While the main purpose was to achieve higher efficiency, one important aspect of the work was to stay as close as possible to the original code-base, including extra simulation capabilities whenever possible in order to find similarities between the different technologies. This proved to be not trivial because of the complexity of the topic, the NP growth of the problem and the intrinsic differences between the two languages.

The entire process followed a learning-by-doing approach starting from a straightforward translation. The inception of this very first phase took place in the “leaves function” of the original code-base, the ones that do not call other user defined functions, subsequently climbing the tree up to the main script. The parallelisation was achieved thanks to a simple macro which mimics ***parfor*** behaviour parallelising over a set of workers.

This method provided a solution that was in contrast with Julia best practices. In order to achieve the best possible result, a code redesign and a different approach were necessary.

Before proceeding further discussing the actual re-implementation of the code, some remarks, about a few critical topics, are due.

Cell Arrays

Cell arrays are common objects in Matlab, usually including other nested structures within. Unfortunately a corresponding object does not exist in Julia which relays on built-in Arrays and Dictionaries.

Operating a successful mapping between them turned out to be not straightforward, especially because of the strong typing required while instantiating and manipulating Julia objects: this important topic is discussed in the Compiler and Type Stability section.

Shared Memory Limitations

Julia 1.4.1 does not provide a stable multithreading interface. In order to yield a software usable with future releases, the parallelisation was performed through the already stable distributed memory interface. Shared memory objects such as SharedArrays are used to overcome limitations inside each node.

Futures and DistributedChannels are therefore heavily exploited to provide IPC.

Differential Equations in Julia[7]

The **DifferentialEquation.jl** module, part of the great SciML, is a suite for numerically solving differential equations in Julia and available for use also in Python and R. A vast realm of equations is supported by the package. Easy GPU acceleration and arbitrary precision are just two of the features inherited from the Julia ecosystem.

Even though the official documentation is more than exhaustive, a few more primers and guidelines are helpful to understand how the library internally works and how it is possible to elegantly optimise the code:

- Provided by Christopher Rackauckas, one of the authors of the package, the following is a marvellous guide. Moreover, the his personal blog is definitely noteworthy.
- More tutorials can be found on the dedicated git repository or in its indexed version.

A wide range of high-performance solving algorithms written in pure Julia, and also wrapped from different external suites and libraries, are freely available. For instance, MATLABDiffEq.jl provides interface bindings to use Matlab DE (differential equation) built-ins within julia DE framework. Additionally custom made routines can be edited.

The general workflow for using the package is:

1. Define a problem: this can be of various forms(e.g. ODE, Discrete, SDE…)
2. Solve the problem: the interface is ***sol = solve(problem, algo, kwargs)***. Consider reading The common solver options section to learn how to control the execution. Unfortunately, not all of them are always compatible with the algorithm in use.
3. Plot the solution.

Once the basics are understood, the reader should investigate further how the most important internals work:

- The integrator interface, from a developer perspective, gives the ability to dive into the solving process step by step. Called by the *solve* function, this is the centre of the entire framework whatsoever. A good understanding of its functionalities is crucial.
- Event handling and Callbacks are essential to describe intricate systems. Their realisation is tightly dependent on the integrator interface.

As said at the beginning of the chapter, the core of the simulation is embodied by a differential equation with the addition of a jump function.
In the matlab code-base, the original author resorted to the built-in Dormand-Prince Runge-Kutta 4/5th order algorithm(*ode45*).

[7] Check section H. References

After a series Benchmarking, omitted in this dissertation, the Julia implementation of the same algorithm(*DP5()*) arose as the best performing one for this specific problem.

Here a snippet similar to the actual Julia code for ODE solving:

```julia
#Parameters:
#   t           time variable
#   u           also called Cin, coefficient value the ODE will calculate
                (the ψ vector).
#   p[1]        also called H1, matrix containing the Hamiltonian
#               independent of time(a SparseMatrix)
#   p[2]        epsilon

# Inplace function
function odeC!(du, u, p, t)
    # mul!(out, A, C) -> mul!(out, A, C,1,0)
    mul!(du, p[1], u)
end

# The condition
function condition(u, t, integrator)
    # if result < 0 do affect!
    real(dot(u, u)) - integrator.p[2]
end

# Terminate computation, event effect
function affect!(integrator)
    terminate!(integrator)
end

using DifferentialEquations

# event/callback handler
cb = ContinuousCallback(condition, affect!)

# problem statement, (0, dt) = timeInterval, (H1, 1.0) = parameters
# the {true} statement describes
prob = ODEProblem{true}(odeC!, Cin, (0, dt), (H1, 1.0), callback=cb)

# solve(prob, algo)
sol = solve(prob, DP5())
```

Considerations over BLAS

Matlab speeds up linear algebra routines with the embedded MKL BLAS interface. The JuliaMatlab repository includes an extensive set of benchmarks comparing the two languages, but none of them deals with pure Julia algebraic routines.

As shown in the SparseArray section, pure Julia code is executed while dealing with sparse objects, or on SuiteSparse in some rare circumstances. As the most expensive line in the whole code, susceptible to exponential growth following the desired number of qubits, is a sparse matrix-vector multiplication located in the *odeC!* function, extra care is required while handling the sparse matrix. The function *dropzeros()* removes unnecessary zeros entries, a common leftover of matrix creation, which can lead to great inefficiency.

An interesting case of study, not implemented in this work, consists in creating a custom matrix type and extend the set of *mul!* methods with a specifically designed one.

Redesign

In order to provide an interface for the PCB scheduler, the main call has been revisited to provide a thinner interface for the user:

```
include("PARALLEL.jl")

# parallel run, include first precompiling run
time_parallel, Ss_N, Mz_N = parallel([2,2,3,4,5,6,7,8],         # system size
array aka Ny, change accordingly
                                     1,             # Nx
                                     0.9:0.1:0.9,  # Jx StepRangeLen
                                     1.2:0.001:2,  # Jy StepRangeLen
                                     1.0,           # Jz Float64 value
                                     1,             # trajectories (traj)
                                     20.0,          # Time window (T)
                                     1.0,           # gamma
                                     1.0);          # dt

@info "time parallel: $time_parallel"                # time output
println()
```

While enriching the new version with new simulation capabilities, allowing the program to crawl in parallel over multiple coupling parameters and thus to make a better use of the infrastructure, it is important to exploit already allocated memory.
This procedure bears also substantial enhancement for what concern the parallelisation. Each worker allocates and calculates all the necessary data structures to initialise, then iterates over the *Jy, Jx* and *traj* parameters in a series of nested loops computing just what is needed at the right time. These improvements provide extra scalability to the program.

```
# defining H1_Jz component, it is fixed
H1_Component!(H1_Jz, Jz, 3, Nx, systemSize, gamma, sig)
# inspecting Jx dimension
for x = args[1]:args[2]
    # computing H1_jx component only when changing Jx
    H1_Component!(H1_Jx, Jx[x], 1, Nx, systemSize, gamma, sig)
    # inspecting Jy dimension
    for i = args[3]:args[4]
        # computing H1_jy component only when changing Jy
        H1_Component!(H1_Jy, Jy[i], 2, Nx, systemSize, gamma, sig)
        # building H1, summing up  the three components
        #and (-gamma / 2) * sig[i, 4]
        H1_exact!(H1, H1_Jx, H1_Jy, H1_Jz, Nx, systemSize, gamma, sig)
        # inspecting trajectories
        for traj = args[5]:args[6]
            # here the simulation takes place…(ODE and jumps for T times)
            # larger the number of trajectories, greater precision the
            # simulation yields stochastically speaking
```

The computational load has to be distributed intelligently, thus the coupling parameters are fragmented according to the dimension of the worker-pool into smaller jobs (6-tuples of indices) and stored into a RemoteChannel, whose reference is known to the entire pool.
The maximum amount of jobs is equal to n^3 where n is the number of workers.

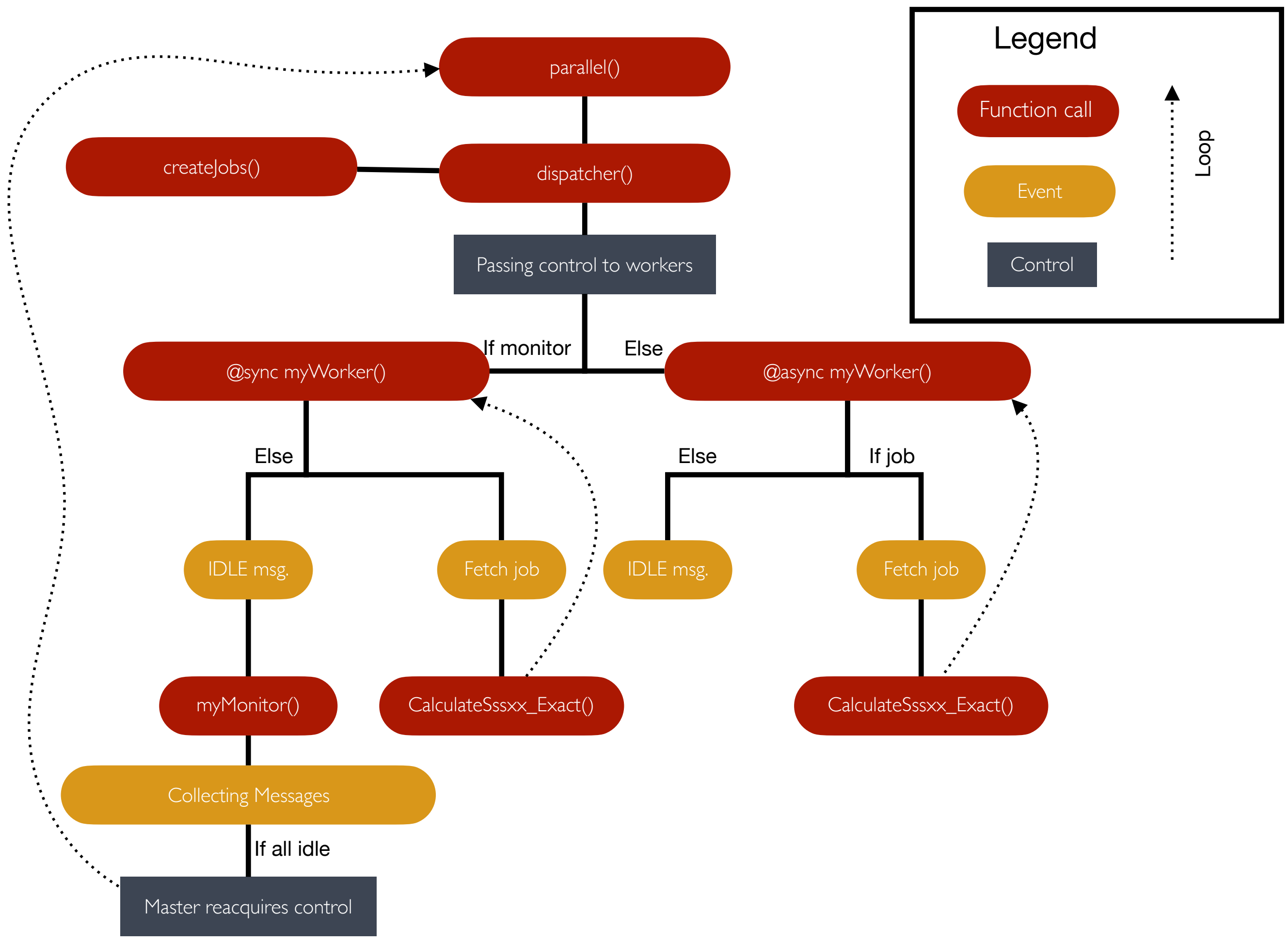
Other two parameters, which influence the complexity of the simulation without being considered during the phase of job creation, are *Ny* and *T*:

- *T* represents the time window of the actual simulation and it is fixed for all the different combinations of coupling parameters, therefore it is not relevant from a distributed design perspective.
- *Ny* array entries influence the number of qubits simulated, which is equal to *Ny[x].* When the value changes, all the previous data structures lose their value and must be recomputed: this happens when all the jobs for a specific *Ny* value are over.
  Therefore we are considering a 1 dimension qubits chain where *Nx = 0* always.

Apart from the master process, that handles job creation and task dispatching, also one randomly chosen worker, synched with the master, has a unique set of tasks: it manages synchronisation across the pool when the job channel becomes empty, retrieving all the messages sent by other workers and checking their idle status. When every one is idle, the master reacquires control from the synchronisation worker and merges the retrieved futures into a single main solution.
At that point, it is possible to proceed with the next qubit lattice and create a new job list.
IPC is hence managed with little overhead if compared to the overall cost.

Another important difference from the first rough implementation is that the largest part of the numeric computation is done with almost no use of the cache.

The main structure of the program is here described by a flowchart:



Results

Different benchmarks, whose plots are displayed in the appendix, have been done on various hardware, including CalcUA HPC facility. Julia always outperformed Matlab both in computational time and memory efficiency.

The **first machine** is the oldest one and does not ship AVX2 vectorial extension. The use of different BLAS interfaces within Julia does not show any performance differences (same with machine 2). It is immediately visible that with a higher number of qubits Julia delivers a lower performance boost. This is due to the fact that the cost of the ODE, which also the profiler identifies as the most expensive part of the code both in Julia and Matlab, becomes so large that the benefit of a good resources management and low level optimisation is crushed by the exponential growth of the linear algebra operation embedded in the ODE function.

A similar behaviour is registered while running on the **second machine**. Both programs take advantage of the greater number of cores and memory.

The **third machine** is a tier 2 node from the CalcUA facility: because of its nature, we were able to attempt larger simulation up to 16 qubits. Within this range, it is evident that the gain is limited and very little can be done in terms of software engineering: actually, only by developing a custom parallel ODE solver and a new algebraic parallel routine for that specific Hamiltonian matrix, a better outcome might be achievable.
While going further into the > 16 *Ny* area, Julia and Matlab performance would probably match or they might even turn favourable to Matlab due to the MKL interface.

Eventually, the last benchmark has something else to unravel: studying the timings for systems with *Ny* within the [5, 12] scope reveals that the performance multiplier is not consistent with the data retrieved from the other machines: in fact it shrank unexpectedly if compared with the other results.
To understand what is going on we must consider that:

1. Each node has 4 different memory controllers which provides a higher memory throughput.
2. While Matlab continues to parallelise over its memory inefficiencies, Julia reaches the point beyond which having more cores do not improve performance: this is a consequences of Amdahl's Law.

This is shown in the last benchmark: cutting down the number of cores in half does not impact significantly the overall simulation timing in Julia, but yields a better average performance, if compared to Matlab, while using half the resources. Similarly, increasing the size of the simulation by a factor 2,5 shows a greater average performance multiplier.
It is also interesting to notice that with 14 cores we have almost the same timing delivered by machine 2, which ships a 12 cores CPU with higher clock rate.
As a general rule, it is reasonable to assign a simulation span of 1000 different elements every 14 cores in order to avoid unnecessary resource consumption on the cluster.

Multi-node scaling results perfectly map the ones just discussed.

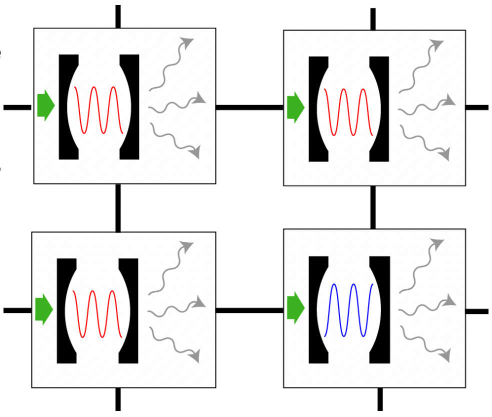
D. Layered Parallel Computing

1. Problem Statement

Introduction

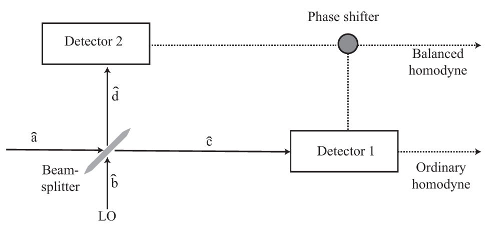
The project dealt with the design of the Montecarlo simulation over the Ito SDE (Stochastic Differential Equation) described in the supplementary material of W. Verstraelen's and M. Wouters's and collaborators "**Gaussian trajectory approach to dissipative phase transition: the case of quadratically driven photonic lattices**"[8]. Previous pre-existing implementations were constricted in their scalability by a proprietary software license, limiting the maximum number of exploitable nodes to one.
The objective was to exploit properly the CalcUA computing facility in order to shrink the absolute time cost of the program and unchain the users from strict licensing.

The system under consideration describes the time evolution of the quantum states of light in a lattice of cavities. Each cavity can contain a certain number of photons and as expected in quantum mechanics, a superposition of certain number of photons. Photons are injected pairs of two in the cavities with a two-photon laser. Naturally, the cavities will also leak out radiation.



In this concept picture a four elements lattice is depicted, highlighting the two possible phases of light with different colours.

As mentioned in section C, the number of possible quantum states increase exponentially with the number of cavities. In the case of photons this gets out of hand even more quickly since it scales as $Nphotons^{Ncavities}$ rather than2^{Nspins}, with $Nphotons$ the maximum allowed number of photons in a cavity (which is allowed to be infinite). If one wishes to study systems with a large amount of cavities and photons one quickly has to resorts to approximative methods that work with a smaller portion of the Hilbert space. The Gaussian trajectory approach is such a method where a Gaussian ansatz for the states of the system is assumed, decreasing the complexity from a exponential scaling with system size to a polynomial one since one only needs to keep track of theGaussian moments. For a more technical explanation I refer to "Gaussian trajectory approach to dissipative phase transition: the case of quadratically driven photonic lattices".



Setup for homodyne (and heterodyne) detection [Figure A1][source]

Contrary to the project in the previous section we will no longer study a continuous time evolution with events (i.e. quantum jumps) but rather a time evolution that is influenced by a stochastic variable. The reason for this difference stems from the assumed detection method of the system in this simulation. Contrary to the algorithm with the jumps where one detects this jump we now use a

[8] https://journals.aps.org/prresearch/abstract/10.1103/PhysRevResearch.2.022037

continuous measurement scheme as depicted in the figure aside(A1). This detection scheme results in a stochastic term in the time evolution that consists of complex Wiener noise. This detection method is called heterodyne detection, a common technique in experimental quantum optics.

Dynamic System Representation

The evolution of the Gaussian moments is here described.

The SDE is simplified disregarding the two-photon loss ($\eta = 0$) as implemented in the actual code

$$
\begin{aligned}
d\alpha_n &= [(-\frac{\gamma}{2} + i\Delta)\alpha_n + i\frac{J}{z}\sum_{n'}\alpha_{n'} - (\eta + U_i)(|\alpha_n|^2\alpha_n + 2\alpha_n v_{nn} + \alpha_n^* u_{nn}) - iG\alpha_n^*]dt \\
&\quad + \sqrt{\gamma}\sum_i (v_{in}dZ_i + u_{in}dZ_i^*) \\
du_{nm} &= [(-\gamma + 2i\Delta)u_{nm} + i\frac{J}{z}(\sum_{n'}u_{n'm} + \sum_{m'}u_{nm'}) - iG(v_{nm} + v_{mn} + \delta_{n,m}) \\
&\quad -(\eta + Ui)(v_{nm}(\alpha_n^2 + u_{nn}) + v_{mn}(\alpha_m^2 + u_{mm}) + 2u_{nm}(|\alpha_n|^2 + |\alpha_m|^2 + v_{nn} + v_{mm}) + \delta_{n,m}(\alpha_n\alpha_m + u_{nm})) \\
&\quad -\gamma\sum_i (u_{mi}v_{in} + u_{ni}v_{im})]dt \\
dv_{nm} &= [iU(2v_{nm}(|\alpha_n|^2 - |\alpha_m|^2 + v_{nn} - v_{mm}) + u_{nm}(\alpha_n^{*2} + u_{nn}^*) - u_{nm}^*(\alpha_m^2 + u_{mm})) \\
&\quad -i\frac{J}{z}(\sum_{n'}v_{n'm} - \sum_{m'}v_{nm'}) + iG(u_{nm} - u_{nm}^*) - \gamma v_{nm} \\
&\quad -\gamma\sum_i (v_{ni}v_{im} + u_{ni}^* u_{im})]dt
\end{aligned}
$$

Where:

- $dZ_i = \dfrac{1}{\sqrt{2}}(dW_{x,i} + idW_{p,i})$ is a complex Wiener process satisfying $|dZ_i|^2 = dt$.
- $d\alpha_n$ is a vector with n equal to the number of cavities. These values describe the expectation value of the photon field in each individual cavity
- du_{nm} and dv_{nm} are square matrices with n, m equal to the number of cavities. These matrices describe the (Gaussian) quantum correlations between each pair of cavities.

The first term $d\alpha_n$ does not match the standard multiplicative form:

$dx(t) = f(x(t))dt + g(x(t))dW$

In order to yield an orthodox structure we may rewrite the noise part of $d\alpha_n$ in its real components

$$\sqrt{\gamma}\sum_i (v_{in}dZ_i + u_{in}dZ_i^*) = \frac{\sqrt{\gamma}}{\sqrt{2}}\sum_i (v_{i,n} - u_{i,n})idW_p + \frac{\sqrt{\gamma}}{\sqrt{2}}\sum_i (v_{i,n} + u_{i,n})dW_x$$

The resulting term satisfies

$$dx(t) = f(x(t))dt + g^1(x(t))idW_1 + g^2(x(t))dW_2$$

Problem Complexity

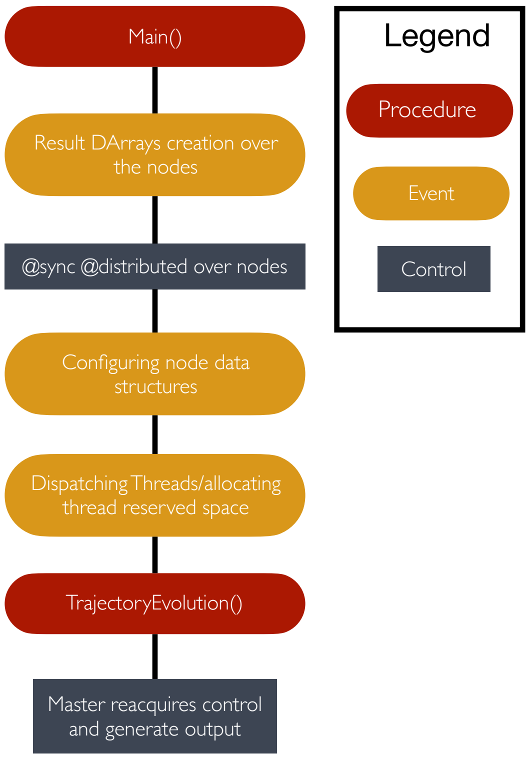
The space cost of the algorithm increases asymptotically to $O(n^2)$, where n is equal to the number of nonlinear photonic cavities. The overall cost of the algorithm is $O(n^3)$.

2. Implementation

Design

Due to the nature of the evolution and the needs of the Physics department, just the time evolution of multiple trajectories underwent parallelisation: every node hosts one Julia process, associated to its unique worker id, which ships n threads where n is equal to the number of active cores of the cluster node.

The distribution of the result arrays was tackled with the use of DitrubutedArrays(DArray) objects. Each worker, which runs independently on its node, has locally stored a piece of the DArray that is eventually retrieved by the master process to compose the final output.


Main()
Result DArrays creation over the nodes
@sync @distributed over nodes
Configuring node data structures
Dispatching Threads/allocating thread reserved space
TrajectoryEvolution()
Master reacquires control and generate output
Legend
Procedure
Event
Control

The availability of a stable multithreading machinery lifted the team from the need of a custom-made scheduler based over the distributed memory framework. This premise greatly simplified the codebase and improved software scalability and performance avoiding unnecessary overhead.

In order to maintain type stability and lower the amount of pointers passed as function arguments, ArrayPartition objects, children of the AbstractArray type, were widely used.

The algorithm chosen to numerically solve the SDE was the Euler-Mayurama method, which is a generalisation of the common Euler method for ordinary differential equations to stochastic differential equations. The implementation does not resort entirely to the DifferentialEquation.jl package: the domain and codomain of the system terms lean on the matrix space, which is not completely supported by the library, especially in the SDE machinery with the use of not scalar noise. The need of an implementation from scratch arose as consequence.

The author contributed[9] to the GitHub community actively opening and fixing issues discovered during the development.

The whole array of optimisation techniques, spanning from inlining, instruction level parallelism, to allocation and garbage collector management, has been used to achieve high computational performance.

[9] https://github.com/SciML/StochasticDiffEq.jl/issues/353

Results

Due to different hardware and parameters, a comparison with the implementation provided by the author of the paper results difficult. The most remarkable outcome is that the computing facility can now be fully exploited with no limitations due to licensing.

As shown in the benchmark, a simulation of 100 trajectories and 100 cavities over 4 cluster nodes, for a total of 25 active cores each, spent ~2000 seconds to complete.
The polynomial growth ratio is also clearly visible.

The parallelisation over multiple nodes showed a maximum overhead of the $2{,}5\,\%$ if compared to the time cost of a single trajectory evolution with the same number of cavities.

The arisen difficulties of a straightforward implementation based on the DifferentialEquation.jl package did not clear the way for the use of more precise higher order solving algorithms, whose cost would be greater given the same time-step, but might allow to make the time-step larger, thus decreasing the total number of steps.

Additionally, the traditional common Euler-Maruyama method, the one adopted in this work, is a fixed time-step procedure: a time window of to $[0{,}200]$ seconds, with a $dt = 2e^{-4}$ seconds, produces one million steps per trajectory, which represents a suboptimal part of the execution.

Unlike for ordinary differential equations, the comprehension of adaptive time-stepping for SDEs is still an open area with many issues regarding stability and convergence.
An interesting paper about this topic is "**An Adaptive Euler-Maruyama Scheme For SDEs: Convergence and Stability**"[10] by H. Lamba, J.C. Mattingly and A.M. Stuart: the work provides a simple procedure, root of a whole class of methods for adaptive integration of SDEs, which may significantly improve overall performance.
The actual adoption is planned in future works.

[10] https://arxiv.org/abs/math/0601029

E. Conclusions

The results show benefit for both the facility and the scale of the simulations with different given priority according to the final objective.

The correct exploitation of the available infrastructure requires a good knowledge of its internals and clever design choices.
A wide spectrum of variables must be taken into account to yield a good result, from physical architecture to software related optimisation.
Additionally, a wise selection of a valid software stack, while providing a better outcome, lifts the system from unnecessary load.

Julia, which takes advantage of the most recent discoveries in the field of computational sciences, is an open-source emerging language with a strong focus on numerical computing. Shipping a modern high level syntax, it is a promising technology with the potential to become a new standard in many fields due to its unique blend of readability and efficiency, given by its high quality compiler, portability and code composability.

The extensive use of cluster computing in the field of numerical analysis paves the way for the adoption of Julia as one of HPC core technologies in the near future: its high level parallelisation interface, both in distributed and shared memory scenarios, seamless low-level optimisation capabilities and open-source policy are key strength for the fast deployment of high performance use cases.

Anyway, the current state of development limits its catchment area to people interested in taking part in its evolution, moving it away from mission critical and industrial applications that need a more stable, standardised technology.
The academic environment offers the required blend of skill and interest to carry on the community driven project and exploit its still rough potential.

F. Appendix 1

This appendix is linked to section C

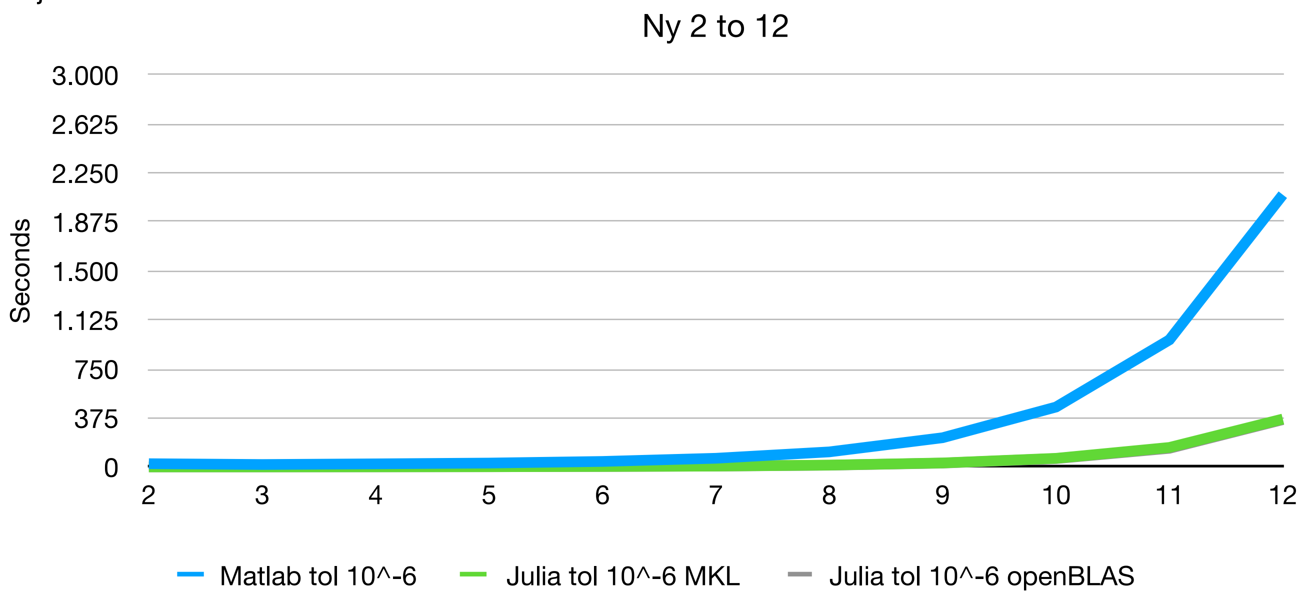
1. Machine 1

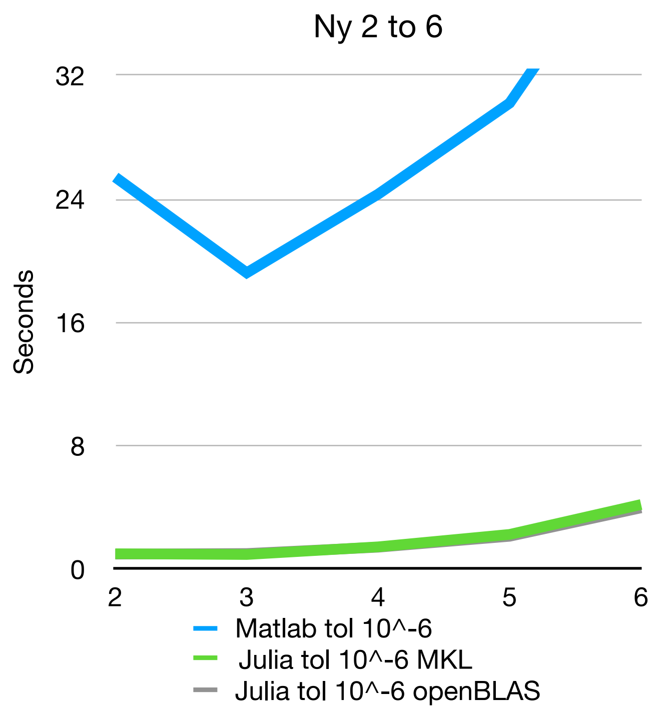
Intel(R) Xeon(R) W3570 @ 3.20GHz

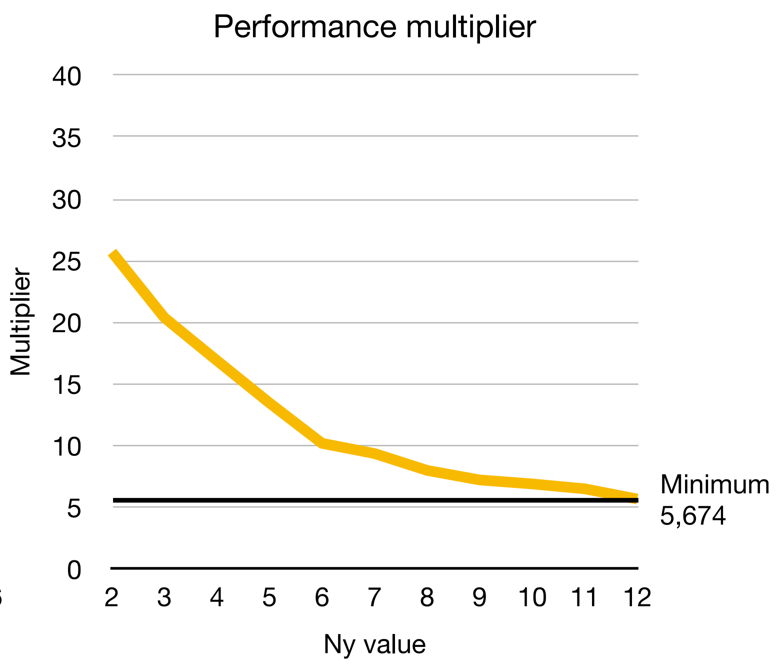
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	8192K

RAM: 20 GB

Parameters:
Time Window = 20
Jy = 1,2:0,001:2,0
Jx = 0,9:0,1:0,9
Trajectories = 1



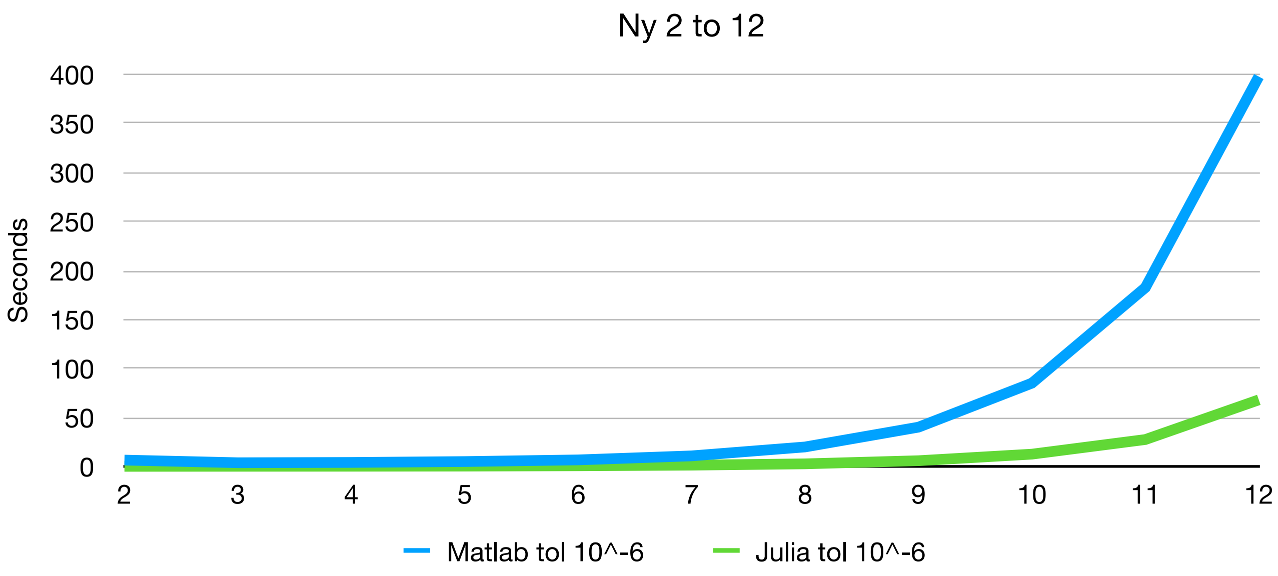


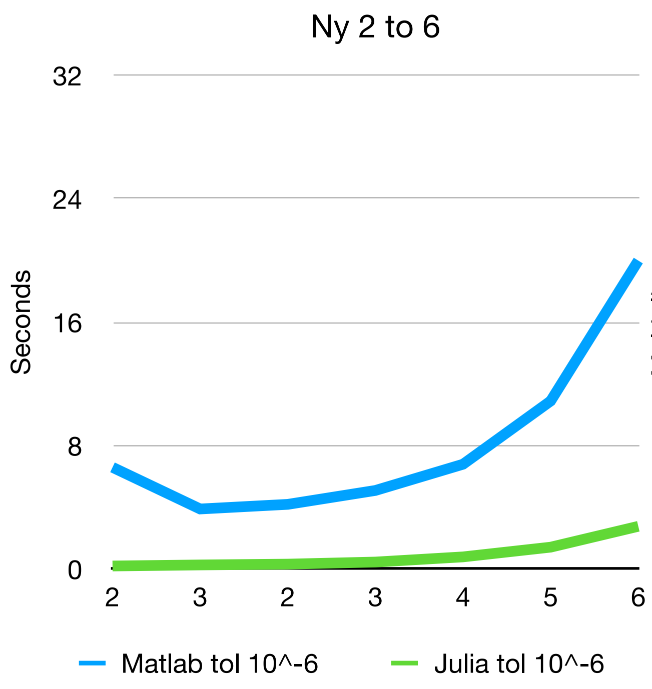


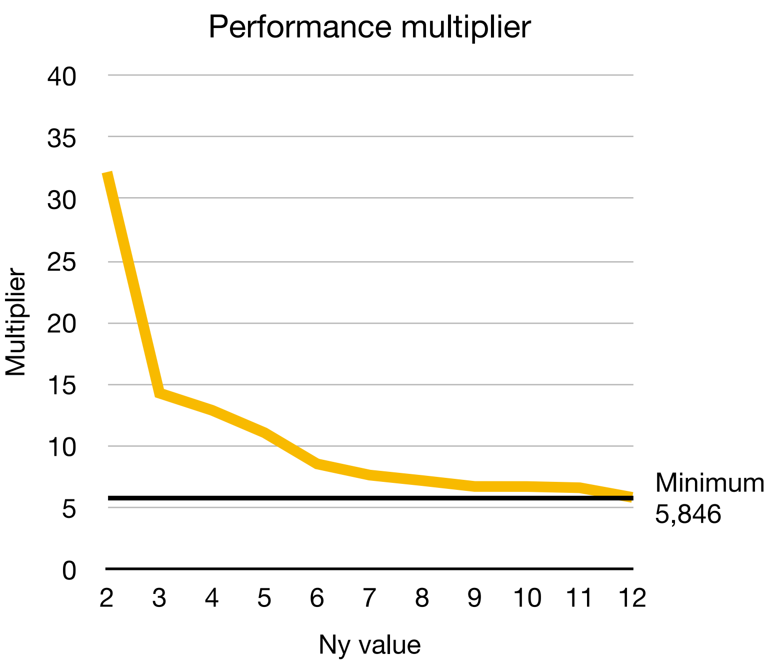
2. Machine 2

MotherBoard: MS-7A90
Processor: Intel(R) Core(TM) i9-7920X CPU @ 2.90, 12 cores
768KiB L1 cache
12MiB L2 cache
16MiB L3 cache
RAM: 32GB DDR4 2400MHz

Parameters:
Time Window = 20
Jy = 1,2:0,001:2,0
Jx = 0,9:0,1:0,9
Trajectories = 1





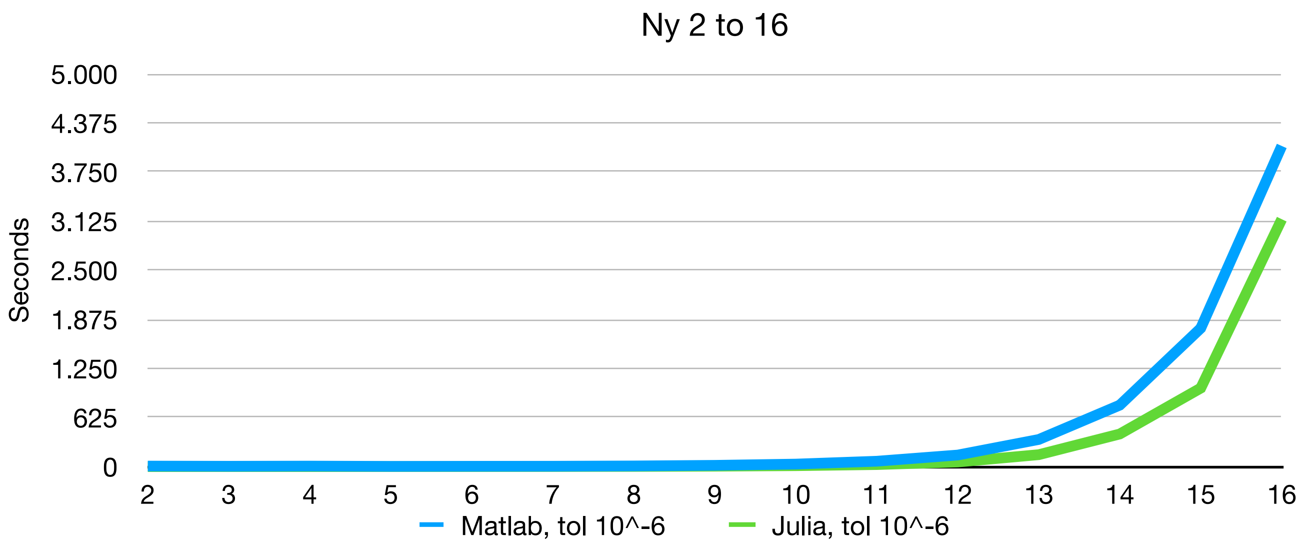


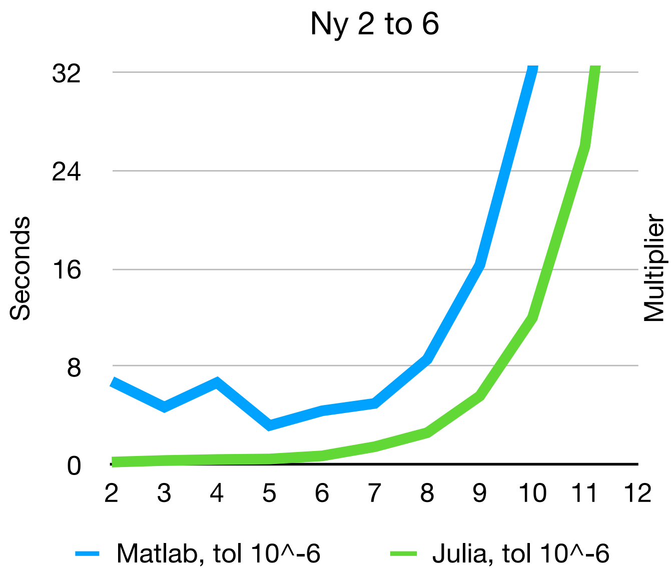
3. Machine 3

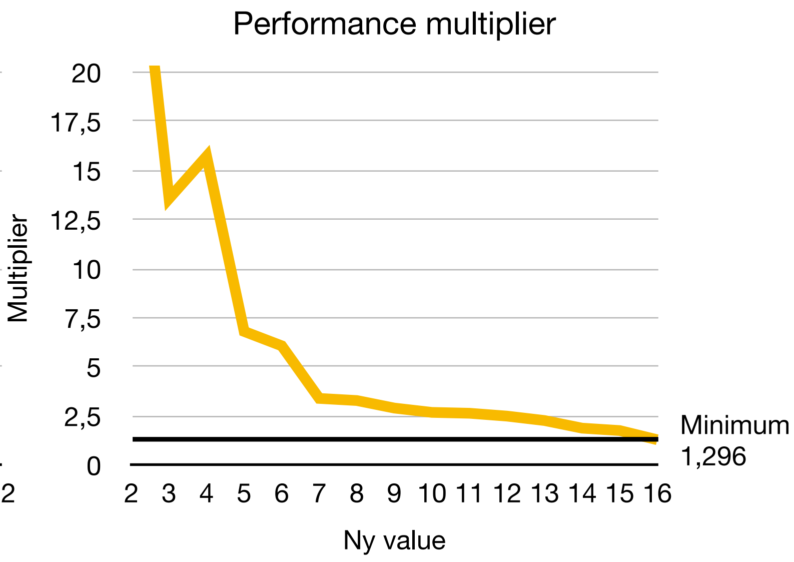
CalcUA Tier 2 Node

RAM: 128 GB
2 Xeon E5-2680v4 CPUs@2.4 GHz (Broadwell), 14 cores each

Parameters:
Time Window = 20
Jy = 1,2:0,001:2,0
Jx = 0,9:0,1:0,9
Trajectories = 1

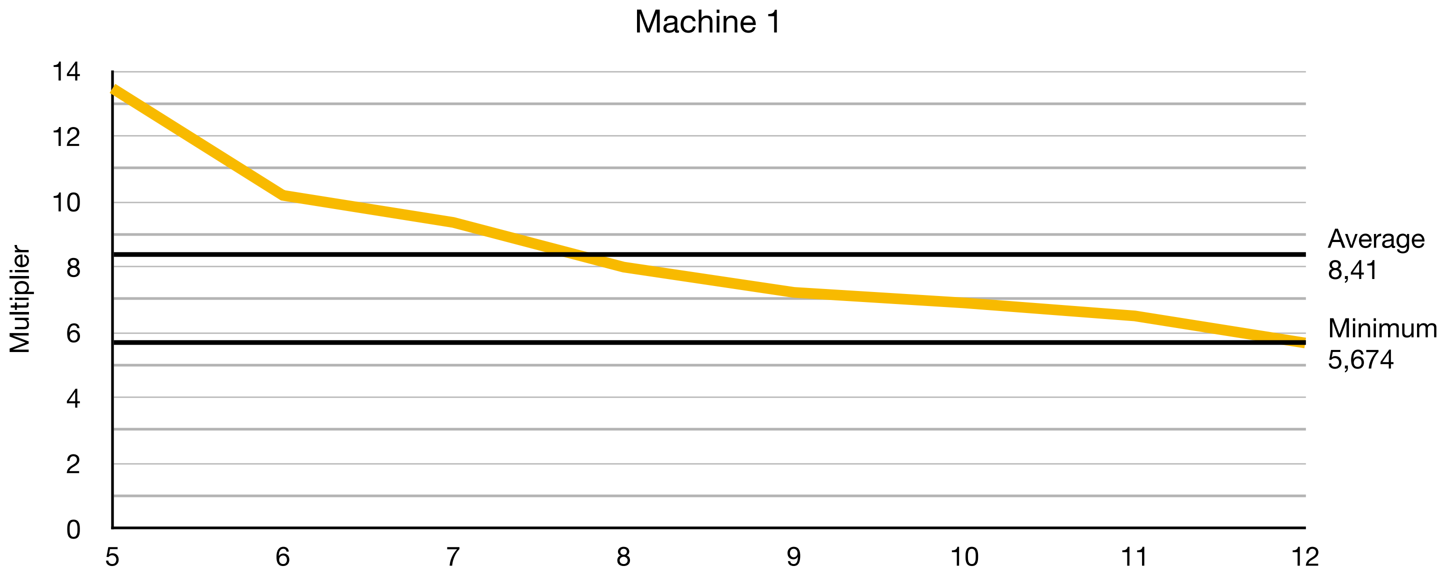


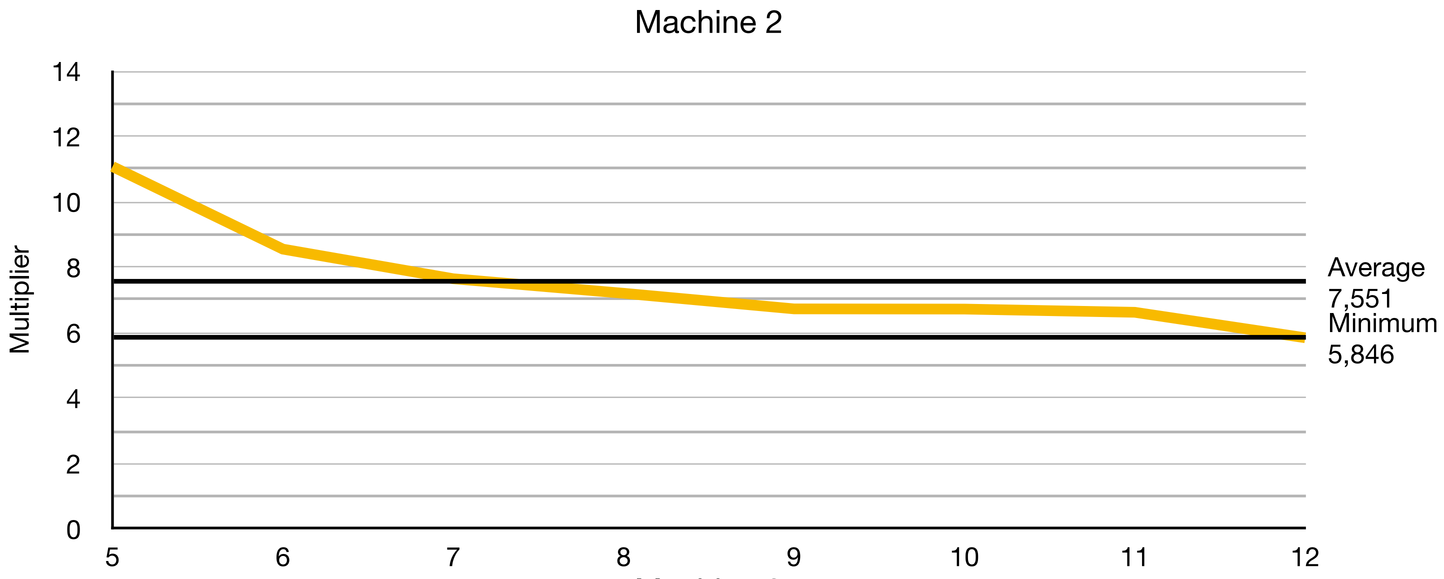


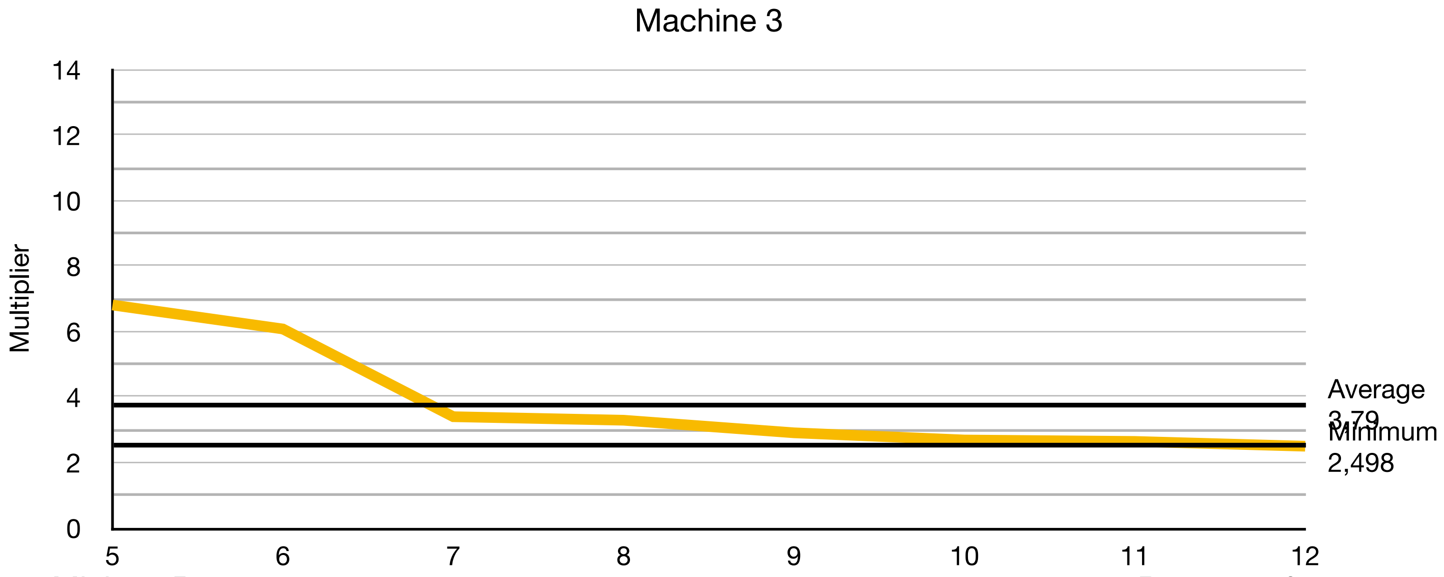


Multiplier Inconsistency

Same parameter as in the previous sections. $Ny \in [5,12]$
The average of the multiplication factor over the various number of qubits and the minimum are highlighted. This benchmark is useful to appreciate the degrading performance on the cluster node(aka. Machine 3) due to a little domain size whether compared to the hardware nature.

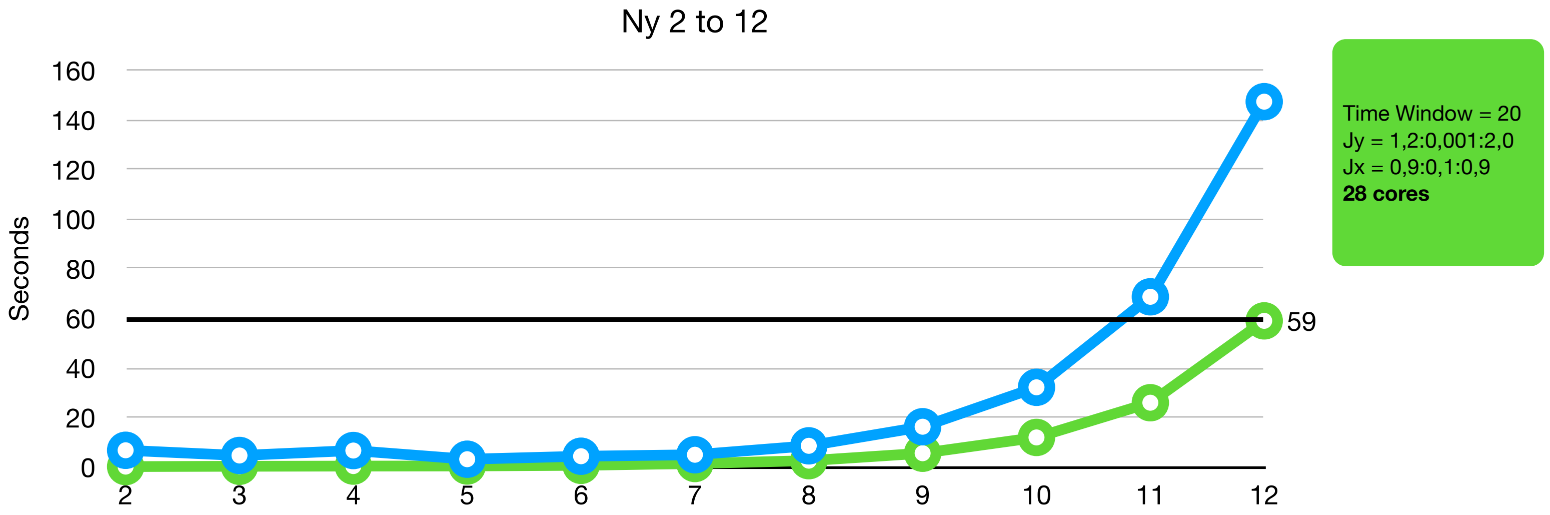


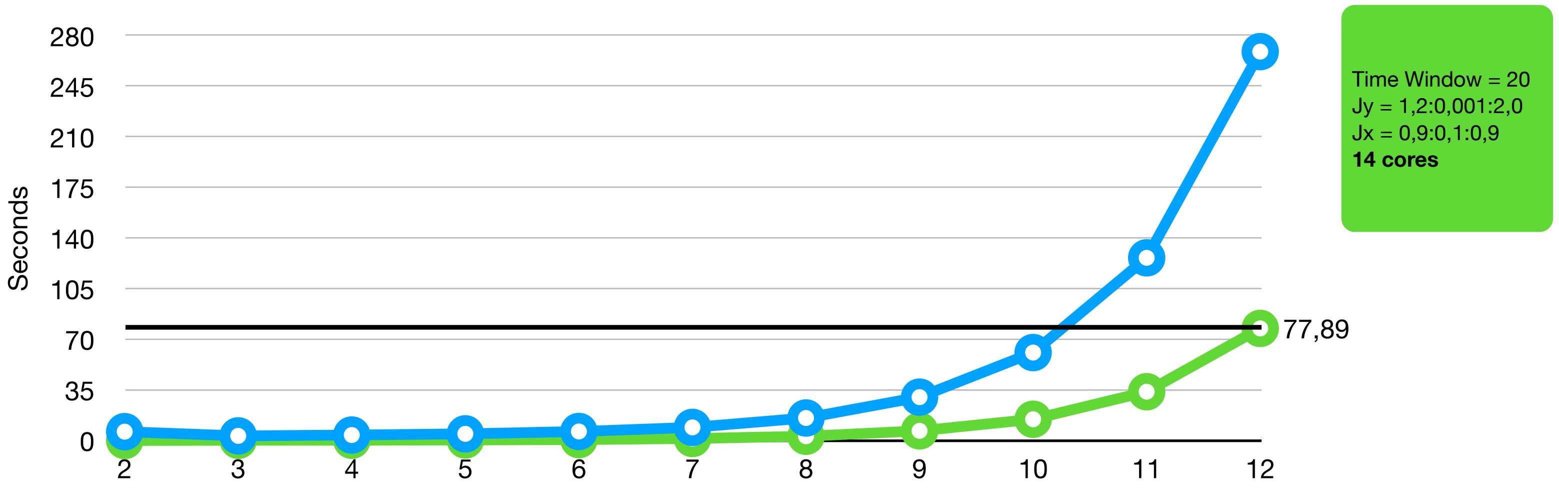


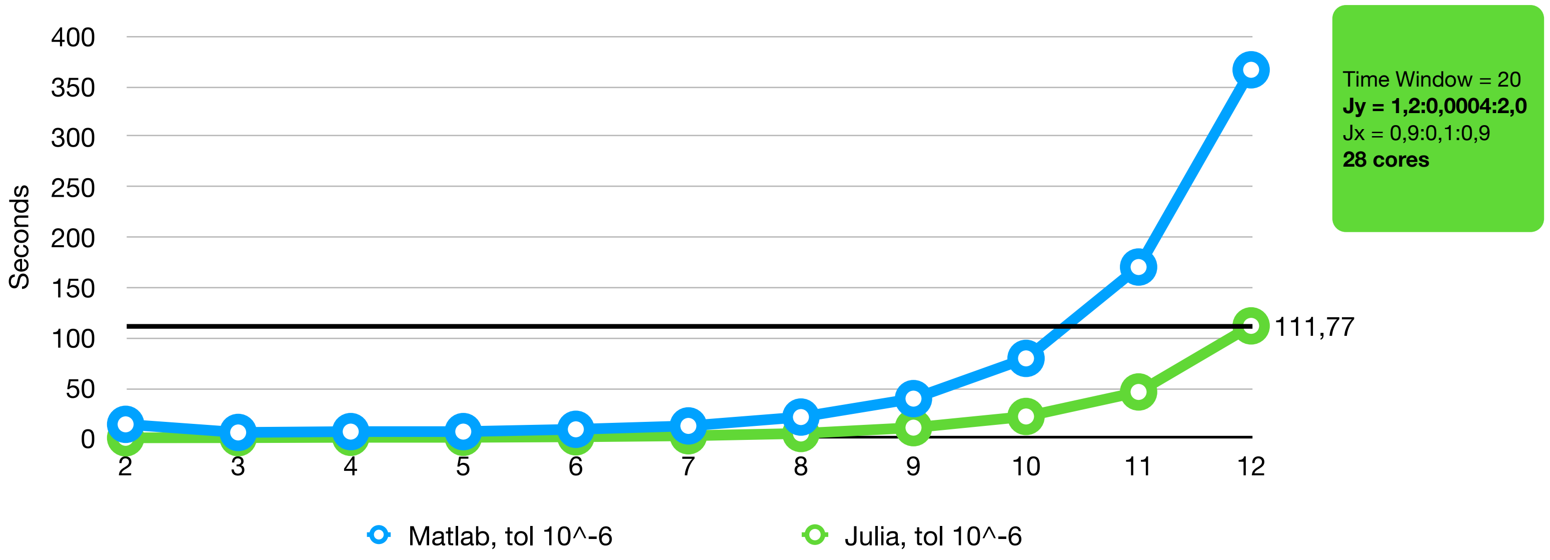


$Ny \in [5,12]$

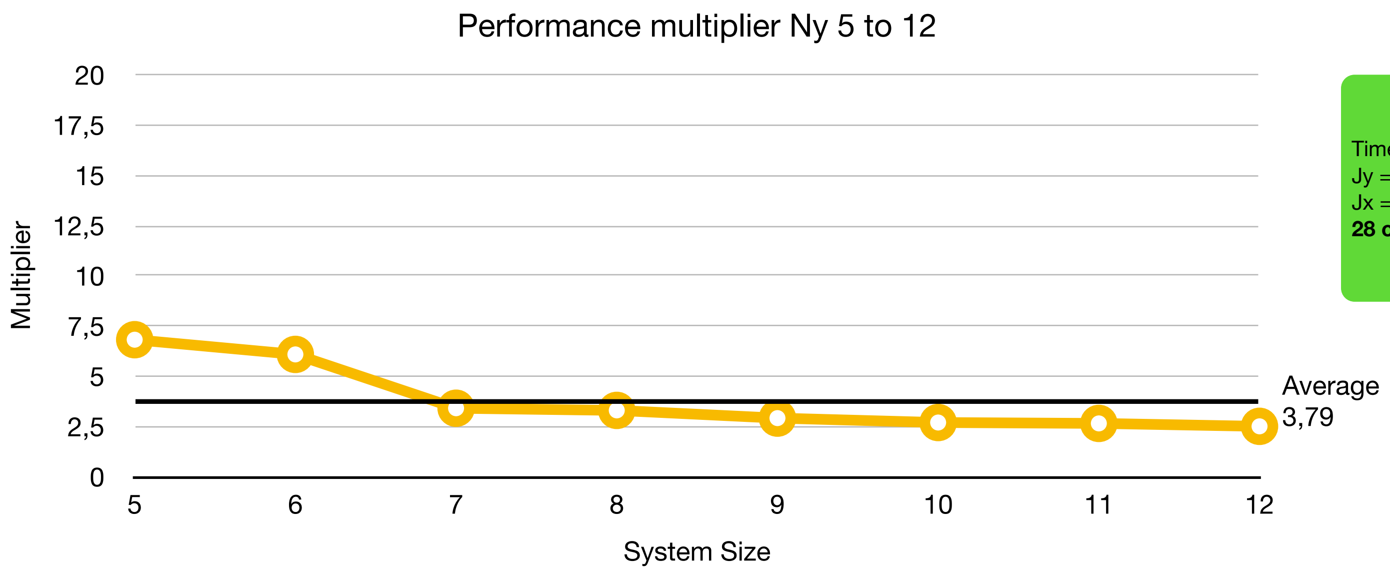
Benchmarks dealing with the limits of parallelisation on Machine 3. The minimum time is highlighted. Increasing the domain size reduced the performance degradation on the cluster node .



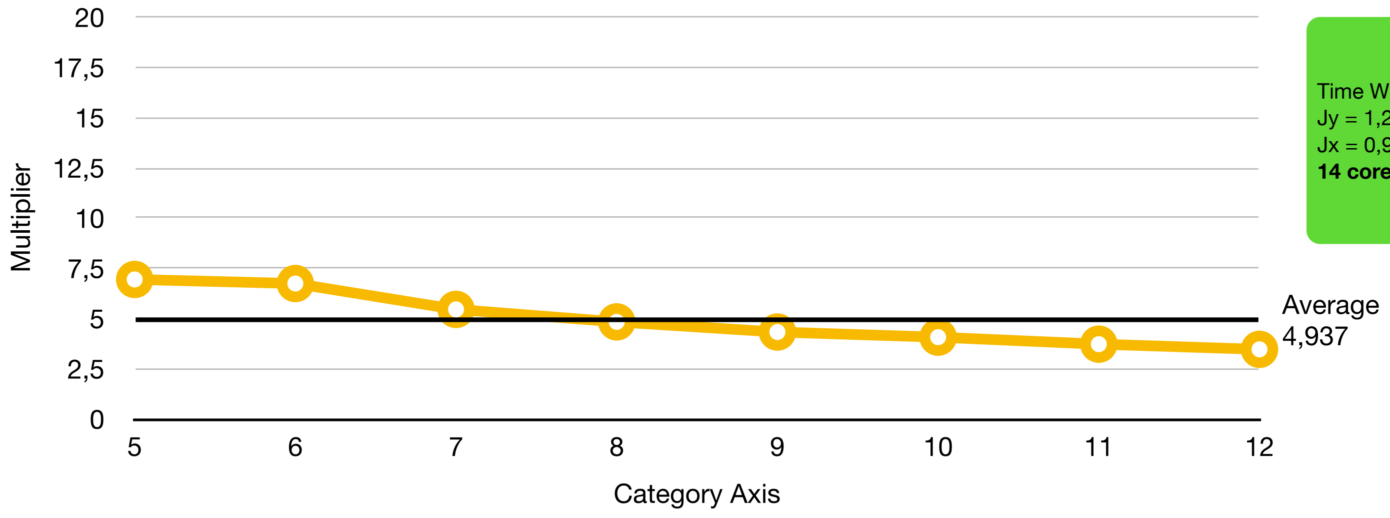




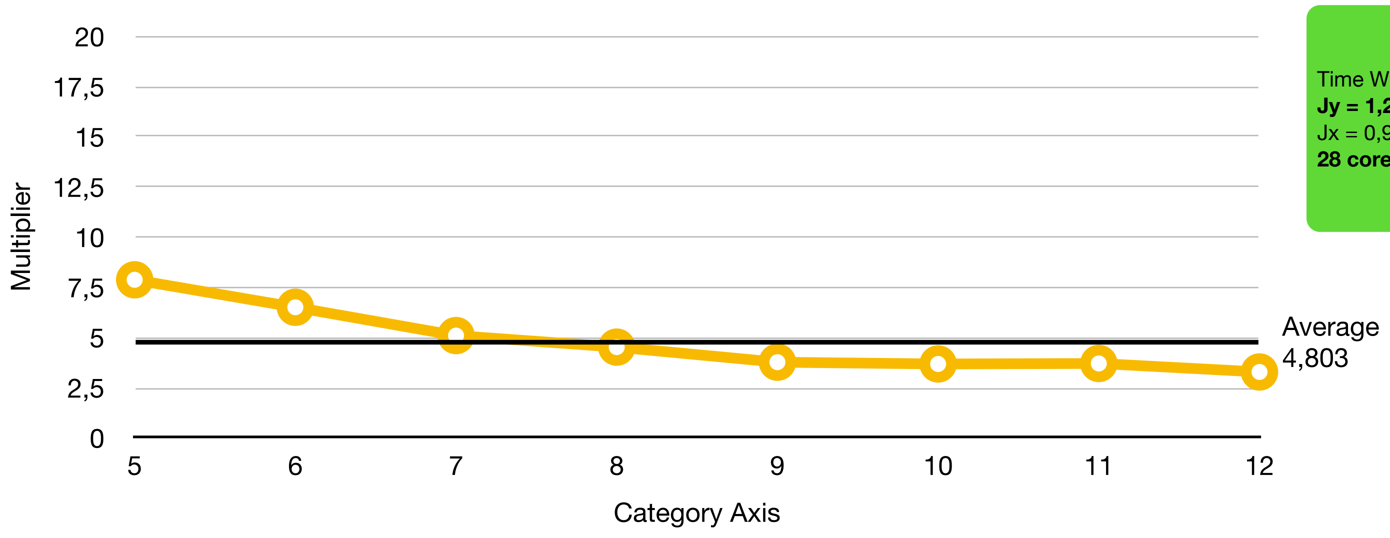
Study of the performance multiplier on machine3. Same configuration as in the previous pag.



Time Window = 20
Jy = 1,2:0,001:2,0
Jx = 0,9:0,1:0,9
28 cores



Time Window = 20
Jy = 1,2:0,001:2,0
Jx = 0,9:0,1:0,9
14 cores



Time Window = 20
Jy = 1,2:0,0004:2,0
Jx = 0,9:0,1:0,9
28 cores

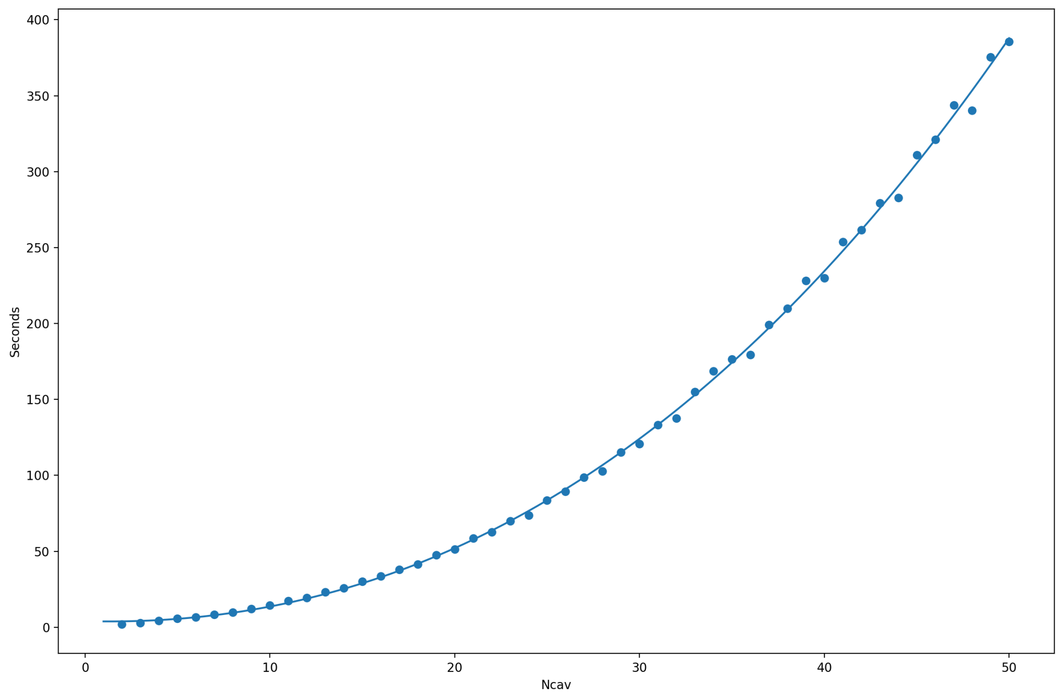
G. Appendix 2

This appendix is linked to section D

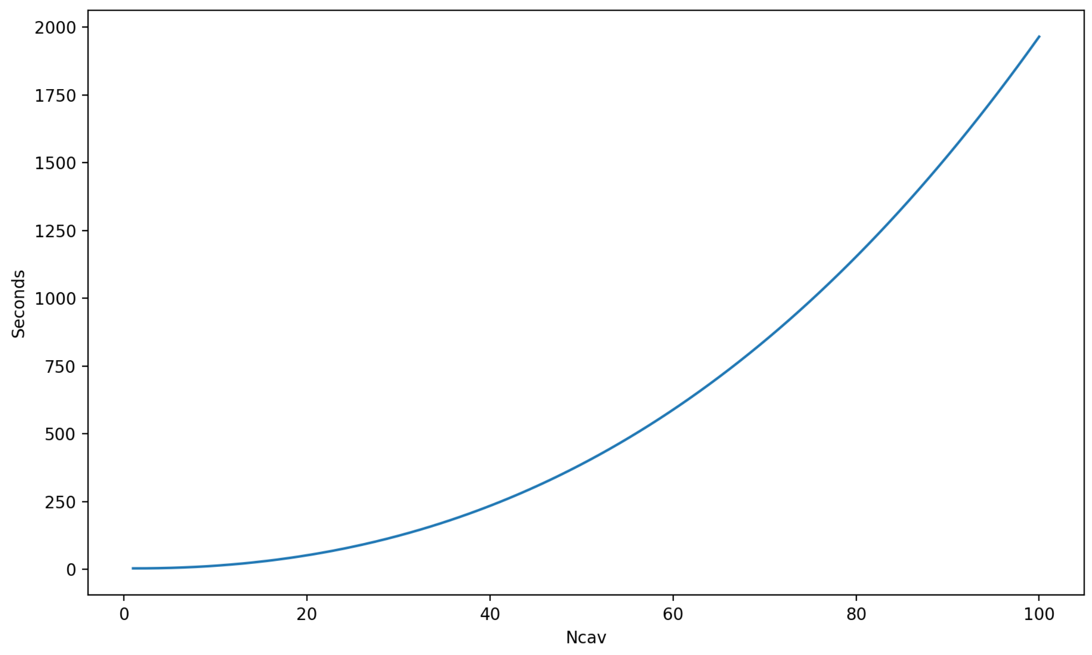
1. Benchmark

Executed on CalcUA Tier 2 Node

The following plot shows the benchmark results sample from single-threaded runs of a single trajectory with $dt = 2e^{-4}$:



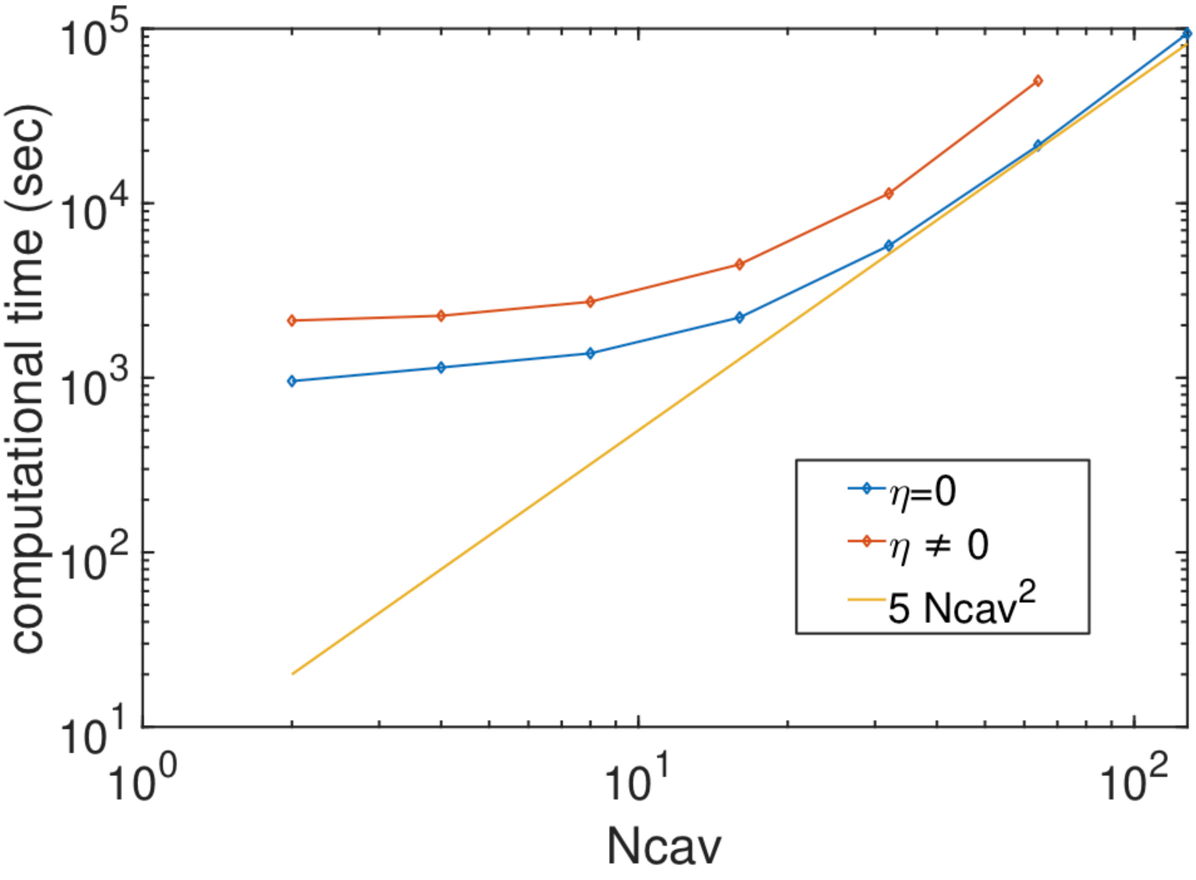
The following plot shows the predicted single-trajectory cost for a larger number of cavities (up to 100) with $dt = 2e^{-4}$:



Assigning one core per trajectory and knowing that cluster overhead is ~$2{,}5\ \%$ the absolute time cost of a single trajectory, the time-cost for a multi-node run can be deduced be the previous benchmark.

2. Paper Benchmark

The benchmark presented in the original Paper, executed on desktop computer with 6 cores and $dt = 1e^{-4}$.



H. References

- **“Julia: A Fresh Approach to Numerical Computing.”** Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral B. Shah. (2017) SIAM Review, 59: 65–98. doi: 10.1137/141000671[11]. pdf[12]
- "**Cluster methods for the description of a driven-dissipative spin model**"[13], D. Huybrechts & M. Wouters
- “**Differentialequations.jl--a performant and feature-rich ecosystem for solving differential equations in julia**”[14], Rackauckas Christopher & Nie Qing
- “**A family of embedded Runge-Kutta formulae**”[15], Dormand John R. & Prince Peter J.
- "**Gaussian trajectory approach to dissipative phase transition: the case of quadratically driven photonic lattices**"[16], W. Verstraelen, M. Wouters & collaborators
- “**Adaptive methods for stochastic differential equations via natural embeddings and rejection sampling with memory**”[17], Rackauckas Christopher & Nie Qing
- "**An Adaptive Euler-Maruyama Scheme For SDEs: Convergence and Stability**[18] by H. Lamba, J.C. Mattingly & A.M. Stuart

[11] https://dx.doi.org/10.1137/141000671

[12] https://julialang.org/assets/research/julia-fresh-approach-BEKS.pdf

[13] https://journals.aps.org/pra/abstract/10.1103/PhysRevA.99.043841

[14] http://doi.org/10.5334/jors.151

[15] https://doi.org/10.1016/0771-050X(80)90013-3

[16] https://journals.aps.org/prresearch/abstract/10.1103/PhysRevResearch.2.022037

[17] http://dx.doi.org/10.3934/dcdsb.2017133

[18] https://arxiv.org/abs/math/0601029