Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

Diana Almeida Barros

Evaluating Shared Memory Parallel Computing Mechanisms of the Julia Language

Rio de Janeiro
2023

Diana Almeida Barros

Evaluating Shared Memory Parallel Computing Mechanisms of the Julia Language

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Orientador: Prof.ª Dra. Cristiana Barbosa Bentes

Rio de Janeiro

2023

CATALOGAÇÃO NA FONTE

UERJ/REDE SIRIUS/BIBLIOTECA CTC/A

B277 Barros, Diana Almeida

Evaluating Shared Memory Parallel Computing Mechanisms of the Julia Language/Diana Almeida Barros. – 2023.

93 f.: il.

Orientadora: Cristina Barbosa Bentes

Dissertação (Mestrado em Ciências Computacionais) - Universidade do Estado do Rio de Janeiro, Instituto de Matemática e Estatística.

1. Julia (Linguagem de programação de computador) - Teses. I. Bentes, Cristina Barbosa. II. Universidade do Estado do Rio de Janeiro. Instituto de Matemática e Estatística. III Título.

CDU 004.43

Márcia França Ribeiro CRB7/3669 -Bibliotecária responsável pela elaboração da ficha catalográfica

Diana Almeida Barros

Evaluating Shared Memory Parallel Computing Mechanisms of the Julia Language

Dissertação apresentada como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Ciências Computacionais, da Universidade do Estado do Rio de Janeiro.

Aprovada em 26 de Abril de 2023
Banca Examinadora:

Prof.ª Dra. Cristiana Barbosa Bentes (Orientador)
Departamento de Engenharia de Sistemas e Computação - UERJ

Prof. Dr. Alexandre Sena
Instituto de Matemática e Estatística - UERJ

Prof. Dr. Júlio Hoffimann
Arpeggeo

Prof. Dr. Tiago Carneiro Pessoa
Interuniversity Microelectronics Centre (IMEC), Leuven, Belgium

Rio de Janeiro
2023

RESUMO

BARROS, Diana Almeida. *Avaliando os mecanismos de computação paralela com memória compartilhada da linguagem Julia.* 2023. 93 f. Dissertação (Mestrado em Ciências Computacionais) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2023.

Áreas de estudo como ciência de dados, aprendizado de máquina ou computação científica são áreas promissoras que estão recebendo muitos investimentos atualmente. Essas áreas geralmente são muito complexas e exigem um bom uso dos recursos de computação para um melhor desempenho. Nesse contexto, nasceu a linguagem Julia. Uma linguagem dinâmica que oferece um ambiente de alto desempenho com uma sintaxe amigável. Embora seu *design* seja voltado para a performance, a computação paralela com memória compartilhada ainda possui alguns recursos em desenvolvimento e os estudos nesta área até o momento são escassos. Neste trabalho, apresentamos um estudo do desempenho dos mecanismos de computação paralela de memória compartilhada da linguagem de programação Julia. Foram analisados o desempenho dos mecanismos Multithreading e SIMD. Na análise do Multithreading, comparamos as estratégias de paralelismo de dados e de tarefas disponíveis através das macros *built-in* `@threads` e `@spawn`, focando na forma como distribuem as iterações do `loop`. Além do mais, foram analisados os mecanismos de escalonamento de `loops` disponíveis na versão de Julia utilizada neste trabalho, que são o próprio escalonamento estático da macro `@threads` e os escalonamentos do pacote FLoops.jl, e foi observado o comportamento da performance de tais mecanismos num ambiente escalável. Na análise dos mecanismos SIMD, comparamos a autovetorização do compilador com a construção *built-in* `@simd` e dois pacotes para vetorização. Executamos nossos experimentos com kernels sintéticos, aplicações de benchmarks e em um framework de otimização do mundo real. Nossos resultados mostram que a macro `@spawn` apresentou melhor desempenho em cargas desbalanceadas e os diferentes tipos de escalonamento de *loops* oferecidos pelo FLoops.jl ajudam a melhorar a performance das aplicações com desbalanceamento de carga. Contudo, aplicações comuns em cenários reais se mostraram mais suscetíveis a *overhead* e perda de desempenho justamente pela natureza do problema influenciar na forma em que o código é implementado, sendo mais notáveis quando `@spawn` é utilizado ou quando o ambiente escala em número de threads. Para os mecanismos SIMD, mostramos que o pacote LoopVectorization.jl proporcionou os melhores resultados de desempenho com baixo esforço de programação.

Palavras-chave: Linguagem Julia. Memória Compartilhada. Multithreading. SIMD. Escalonamento de Loops

ABSTRACT

BARROS, Diana Almeida. *Evaluating shared memory parallel computing mechanisms of the Julia language*. 2023. 93 f. Dissertation (Masters in Computer Sciences) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2023.

Areas of study like data science, machine learning or scientific computing are promising areas that are currently receiving a lot of investment. These areas are usually very complex and demand optimal usage of computing resources for better performance. In this context, Julia language was born. A dynamic language that offers a high performance environment with a user friendly syntax. Although its design is focused on performance, shared memory parallel computing still has some features under development and the studies in this area are so far scarce. In this work, we propose a detailed performance study of the shared memory parallel computing mechanisms of Julia. It was analyzed the performance of the Multithreading and SIMD mechanisms. In the analysis of Multithreading, we compare the data and task parallelism strategies available through the language built-in macros `@threads` and `@spawn`, focusing on the way they distribute the loop iterations. Furthermore, the loop scheduling mechanisms available for the Julia version used in this work were analyzed, which are the one static scheduling provided by `@threads` and the ones provided by the package FLoops.jl, and it was observed their performance behavior when the environment scales. In the analysis of the SIMD mechanisms, the compiler auto-vectorization was compared to the built-in construction `@simd` and two packages for vectorization. Our experiments were run with synthetic kernels, benchmark applications and a real-world optimization framework. Our results show that the macro `@spawn` presented a better performance on unbalanced loads and the different loop schedulers offered by FLoops.jl help improving the performance of applications with load imbalance. However, we found that applications that are commonly found in real world scenarios are susceptible to overhead and loss of performance as the nature of the problem influences on code implementation, being more noticeable when `@spawn` was used or if the environment could scale with threads. For the SIMD mechanisms, we showed that the package LoopVectorization.jl provided the best performance results with low programming effort.

Keywords: Julia Language. Shared Memory. Multithreading. SIMD. Loop Scheduling

ACKNOWLEDGEMENTS

It is with much gratitude that we here express our thankfulness to everyone that helped and made this work possible.

Always a special thanks to my supervisor, Dr. Cristiana Bentes, who always encouraged me and believed that I was able accomplish this research, being always reachable and prompt to help.

Thanks to AtOptima company, which made our research possible, trusting on our work and contribution to their softwares.

Thanks to Dr. Tiago Carneiro, who gave us the opportunity to run part of the experiments on computers hosted by the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several Universities as well as other organizations.

Thanks to our parents, who have been supporting us throughout this Master title process. I am thankful to my friends and beloved one, that understood my dedication to this work, listened to my complaints when I was tired and even helped and gave advices on some problems.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

INTRODUCTION

The past two decades have witnessed the growth of scientific software developed by scientists without a specialized coding background. These scientists rely on higher level interfaces and programming languages, such as Python, R, Matlab, SciLab to build their application. The rise of these higher level languages comes from the fact that writing a scientific application in lower level languages like C, C++ or Fortran can be very time-consuming and requires expertise in programming. Higher level languages, on the other hand, offer significant advantages in terms of programmer productivity. They simplify the programming task by increasing abstraction, and are easy to learn and use for beginning programmers. This enlarges the community of scientific programmers and makes programming languages more accessible. There are a number of applications in different areas that adopted this higher level solution [1, 2, 3, 4].

All these advantages, however, typically come at a cost – performance. The higher level languages are interpreted and the features that make them appealing for the programmers, like dynamic typing and error checking, incur considerable runtime overheads. A program written in Python, R, Matlab or Scilab can be orders of magnitude slower than their counterparts written in compiled languages such as C, C++ or Fortran. This prevents their use for high performance computing.

Julia is a relatively new programming language that was designed to address this performance/productivity tradeoff, which is also referred as the "two-language problem" [5]. Julia focuses on providing the abstraction and syntax of dynamic languages with the performance of compiled languages. The idea is to "come for the syntax, stay for the speed" [6]. Features like a dataflow type inference algorithm allowing types of most expressions to be inferred, an aggressive code specialization against runtime types, and a fast Just-In-Time (JIT) compilation using the LLVM compiler framework provide the efficiency of Julia [5].

The continuously growing Julia community is another important feature of Julia. They provide a number of packages for problems already solved, optimizing the time spent during the development of the code. Not only does Julia allow those packages to be written in Julia itself (most of Julia standard libraries are actually written in Julia), but also as built-in libraries. This means that a piece of code will have the same performance as a built-in structure and it will also avoid any incompatibility that could arise from multiple libraries being used together.

Moreover, since several known scientific and mathematical algorithms are already implemented in languages like C and R, whenever a user wishes to reuse a known algorithm formerly written in one of those, Julia offers an easy way to import them.

In terms of parallel programming, Julia provides different models. It currently supports parallel models like shared memory, distributed memory and GPU computing. Although over the last 20 years, the high performance community has widely adopted well-stablished libraries such as OpenMP and MPI to exploit parallelism, Julia provides built-in support for parallel programming with an intuitive and simple syntax. For ex-

ample, we can exploit parallelism in Julia by placing a macro in front of a for loop, with `@threads` for multithreading, `@simd` for SIMD vectorization and `@distributed` for distributed parallelism.

Additionally, there is the option of using packages to offer more control over the parallel operations, to increase the abstraction and to improve performance. On distributed computing, there are the alternatives of MPI.jl [7] and Elemental.jl [8] for using the already existing MPI ecosystem of libraries. On GPU computing, there are CUDA.jl [9] for NVIDIA GPUs, AMDGPU.jl [10] for ROCm based AMD GPUs, Metal.jl [11] for the Mac devices with M-series chip and others. On shared memory models there are Floops.jl [12] for loop optimizations which includes multithreading with loop scheduling, and LoopVectorization.jl [13] also for loop optimizations, including SIMD operations and multithreading. Since shared memory is the topic of this research, these last two packages will be visited later in this work.

Overall, Julia shows an opportunity of growth for high performance computing not only for its own and designed way focused in performance but also for the simplicity in which it can be achieved.

MOTIVATION

Although parallel programming support has been present in Julia from the very beginning, the primitives for multithreading programming using the shared memory parallel programming paradigm were released as experimental for many years. Only in 2019 [14], multithreading was included as stable with the release of Julia version 1.3. Multithreading in Julia provides a powerful mechanism to create threads dynamically and also a mechanism for loop parallelism. Besides multithreading, Julia also provides SIMD mechanisms to exploit vectorization in the code.

Since these parallel programming mechanisms are quite new for Julia programmers, there are a number of research questions to be investigated in this concern:

- What are the different ways to achieve shared memory parallelism in Julia?
- How do these shared memory features work, what parallelism problems can they solve and how they perform under different scenarios?

OBJECTIVES

Taking into account the research questions proposed, the objective of this work is to perform a detailed study of the shared memory parallelism in Julia. More specifically, we intend to analyze the multithreading and SIMD vectorization mechanisms present in Julia packages and built-in constructions. Our idea is to study their performance, scalability and programming effort.

In terms of the multithreading mechanisms, we focus on Julia's built-in macros for multithreading in different scenarios. Our idea is to compare the data and task parallelism offered by the macros and analyze the different loop scheduling strategies and how they have impact in the load balancing. We also intend to understand the behavior of Julia multithreading when the number of available threads increases.

In terms of SIMD vectorization, we focus on understanding the different existing implementation alternatives along with the trade-off between performance gain and development investment.

METHODOLOGY

In order to achieve the proposed objectives, we performed the following steps. First, we study the multithreading mechanims by carrying out a detailed analysis of Julia's parallel approaches in order to evaluate the performance impacts of the internal scheduling strategy. We evaluate the two main loop parallelization primitives present in Julia: `@spawn` and `@threads`. The parallelization with `@spawn` employs a task based parallel mechanism where the scheduling of the loop iterations is dynamic performed by the runtime. The parallelization with `@threads` employs data based parallel mechanism with static scheduling. These two mechanisms were evaluated using scenarios of balanced and unbalanced computation, on synthetic and real-world applications. We also study other loop scheduling mechanism available and how they would scale regarding the number of threads. The last multithreading analysis studies the effect of it on a real-world practical scenario in an large-scale optimization framework. Along with our multithreading experiments we present some C + OpenMP results even though the comparison between the two languages is not our goal.

After the multithreading study, we perform a detailed study of the SIMD mechanisms proposed in Julia. We explore auto-vectorization, built-in features, packages and SIMD-intrinsics.

Our results show that the user can take benefit from multithreading in Julia with data parallelism or task parallelism, according to the problem studied. For problems where the load distribution is unbalanced, the task parallelism with dynamically loop scheduling usually provides the best performance results. Different loop scheduling techniques showed to help performance improvement as long as the right scheduler is chosen for the given problem algorithm. The nature of the problem algorithm showed to have an impact on the multithreading scalability. For a real-world practical optimization scenario, the use of multithreading in Julia provided performance gains. The problem studied, however, exhibits limited parallelism. In terms of SIMD parallelization, the package LoopVectorization.jl provided the best performance results with little programming effort.

CONTRIBUTIONS

This work makes the following scientific contributions:

- A throughout analysis of the data and task parallelism mechanisms proposed in the Julia language for multithreading programming;
- An analysis of the loop scheduling mechanisms available by FLoops.jl and the performance improvement they provide;
- A scalability study of the multithreading mechanisms;
- A study of the SIMD mechanisms provided by Julia: auto-vectorization, built-in macro, SIMD.jl and LoopVectorization.jl packages and intrinsics.
- A study of the impact of the loop scheduling in a real-world practical optimization application;

TEXT ORGANIZATION

This work is organized in the following way: Chapter 1 presents the related works in terms of exploiting multithreading and SIMD in Julia. Chapter 2 gives a brief introduction to Julia Language. Chapter 3 explains more details about the parallel computing mechanisms present in Julia. Chapter 4 evaluates the performance of data and task parallelism in multithreading. Chapter 5 evaluates the performance of different loop scheduling mechanisms. Chapter 6 provides a performance analysis on the different SIMD mechanisms available. Chapter 7 shows a case study where we exploit Julia parallelism in a real-world practical scenario. Finally, in Chapter 7.7, we present our conclusions.

1 RELATED WORKS

Parallel computing features in Julia are still under development and improvement, the built-in package *Threads* was considered experimental until version 1.3 and the built-in `@simd` macro is still labeled experimental by the official documentation. Therefore, studies related to these topics are scarce. For this reason, Julia community was very proactive in helping scientists improve the performance of their Julia code by providing packages that can exploit parallelism. One of these packages is LoopVectorization.jl [13, 15], a loop optimizer package that exploits SIMD vectorization. This package has been used in BLAS-like libraries, such as Gaius.jl [16] and Octavian.jl [17]. It was also used by SnpArrays.jl [18] for reading and manipulating genome data. The work by Ko *et al.* [19] uses LoopVectorization.jl to show the applicability of DistStat.jl on high-performance statistical applications. DistStat.jl is a package that implements an array structure compatible with distributed environments, offering abstraction for computations either on CPU or GPU. Nagy *et al.* [20] uses LoopVectorization.jl on the implementation of techniques to solve ordinary differential equation systems. Even though the above studies make use of the optimization package, there were no considerations about how it affects the performance of the applications. We can, however, find some interesting performance observations on Elrod C. *et al.* [21] regarding a small network machine learning on CPU. As a way to illustrate the performance gains, the comparison between the execution time of matrix multiplication using Julia's primitive optimization, broadcasting, BLAS and LoopVectorization.jl optimization showed advantages when using LoopVectorizationjl. This leads to the development of a specialized package for such problems, SimpleChains.jl [22], that makes heavy use of LoopVectorization.jl.

In terms of using multithreading in Julia, there are some works that studied its impact in the performance. The work by Summers *et al.* [23] takes advantage of Julia easy parallelization interface with threads to improve the performance of a robot control package. The results show that the package using Julia multithreading is often faster than other options and performs close to C implementations with OpenMP. The work by Novosel and Slivnik [24] provides a preliminary comparison between Julia distributed and shared memory implementations with the parallel language Chapel. They show the implementation differences and some performance results. The recent work by Stanitzki and Strube [25] uses Julia multithreading and Julia channels to accelerate data analysis workflows in high energy physics. They compare Julia, C++ and Python in terms of throughput.

There are some Julia packages that exploit multithreading in order to improve performance. PopGen.jl [26] is a package for population genetics analysis that aims to offer an ecosystem that is fast and user friendly. They provide results on the comparison of PopGen.jl with equivalent R packages. Taking a look over the source code available [27], we can see that they parallelize the loops with the `@spawn` macro, which uses task parallelism with dynamic loop scheduling. LombScargle.jl [28] is a package for spectral analysis of signals using the Lomb–Scargle periodogram. The documentation provides a comparison between the single threaded and multithreaded execution, as well as the Python equiva-

lent. They show gains in execution time when using Julia multithreading. This package uses the `@threads` macro, according to its source code [29], that uses data parallelism with static loop scheduling.

On Gmys *et al.* [30] work, a comparison between high-performance languages is presented on problem solution quality, productivity cost and parallel performance on a multithreading environment that scales up to 64 threads. Since the study was done with a Julia version where multithreading was still considered experimental, it leaves an opportunity for an updated study using a newer Julia version with stable multithreading implementation.

Even though we can find applications that provide analysis of the multithreading performance with Julia and compare it to single threaded versions and other languages versions, no studies were found on how these multithreading mechanisms, data and task parallelism, behave and how to take advantage of them depending on the use case.

The problem of the scheduling of loop iterations in multithreading has also been studied with other parallel APIs like OpenMP. Ayguadé *et al.* [31] show that the best schedule for a parallel loop depends on different elements such as architectural characteristics or data input. They propose a runtime scheduler that uses past executions in the same run to decide the best schedule strategy for the loop. Ciorba *et al.* [32] show that a single loop scheduling technique is not sufficient to balance the load of different types of application, and propose the incorporation of other schedules techniques, like trapezoid self-scheduling, factoring, weighted factoring, and random. Thoman *et al.* [33] present a loop scheduling technique that takes into account the program structure, the problem size and external system load by integrating compiler and runtime analysis. Zhang *et al.* [34] propose an adaptive loop scheduler for hyperthreaded SMPs, studying its performance by inter-thread data locality, instruction mix and SMT-related load imbalance. The proposed scheduler is a self tuning two-level scheduler, it decides what is the best scheduler at runtime and evaluates the best number of threads to be assigned to the loop. Durand *et al.* [35] also propose an adaptive loop scheduler implemented on runtime that provides ways of balancing the load irregular loops while respecting memory locality and presents a way to extend it to be used on NUMA machines. Kale *et al.* [36] go beyond and propose that the loop scheduling should be user-defined, and present an interface to support this scheme.

2 JULIA LANGUAGE

Julia is a high level dynamic programming language that has been gaining popularity in recent years. Julia not only provides a simple programming interface, but also provides high performance. Hence, scientific computation has been taking great advantage from it. The language was proposed to solve the problem that has been described in the literature as *the two language problem* [5]. This problem arises when a dynamic language, like Python for example, is used for productivity, but the code is very slow to execute. So, the implementation ends up with parts of the code written in another language like C or C++.

The way the language was designed makes its writing similar to usual mathematical expressions, an operator can behave differently depending on the operands of the expression. Functions can be declared just as they are written as in $f(x) = x^2 - 4$. Methods are not linked to their first parameter unlike Object Oriented Programming, as it would not make sense for mathematical expressions.

Moreover, Julia was designed for parallel programming, making complex parallel algorithms easy to write. It has built-in primitives for instruction level parallelism, multithreading and distributed computing and packages for GPU programming.

Julia is an open source project available under MIT license. Anyone can collaborate with it and it has an enlarging community. The number of packages provided has been increasing over time and some of the most popular are: *Flux* [37] for machine learning, *DifferentialEquations* [38] for solving differencial equations and *JuMP* [39] for mathematical optimization.

2.1 The Language

In this section we present the main features of Julia language.

2.1.1 Julia REPL

When Julia code is executed, it starts the Julia REPL. As the name says, it **R**eads what you type, **E**valuates it, **P**rints the results and loop back to do it again. It has helpful features and is a great environment for experimenting the language and for doing quick tests.

The REPL has a quick access to Julia's documentation, if there is any term the user would like to check the documentation, he/she can type "?" to activate the help mode (see Figure 1) and then write the term desired.

The REPL also has quick access to the shell commands, the shell mode (see Figure 2) can be activated by typing ";". It can be helpful to navigate through the directories and list the files in a path.

The other mode presented by Julia REPL is the Package mode, or Pkg module (see Figure 3). It can be activated either by pressing "]" or importing the package module with the command `import Pkg`. The package mode is the Julia package manager, where

Figure 1 – Julia REPL help mode showing part of the documentation of the `sort` function.

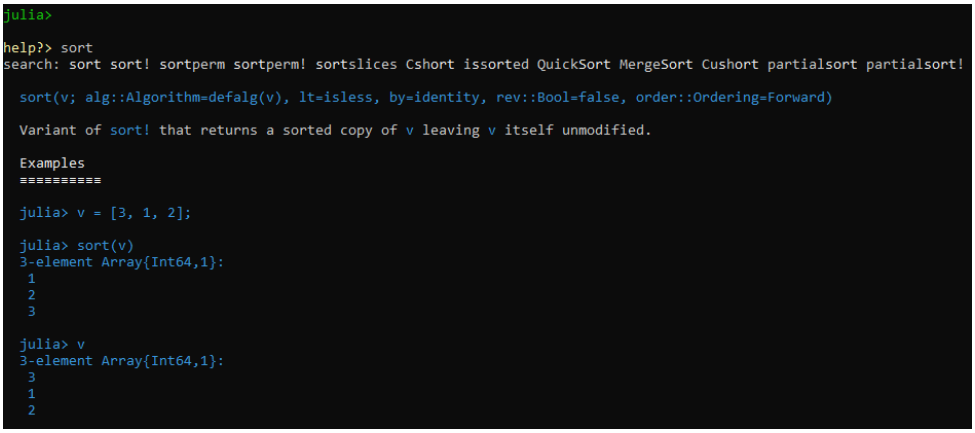
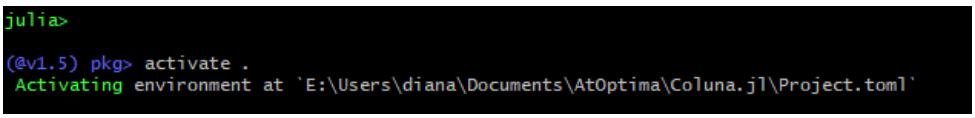


Figure 2 – Julia REPL shell mode showing the *ls* command.

the user can add or remove packages, manage package versions and environments. This is one advantage over the traditional package managers.

Figure 3 – Julia REPL Package mode activating the Coluna.jl project.



Besides the modes presented above, the REPL has other useful features like the `tab` key autocompletion, history of commands accessed by the `up` and `low` key arrows and the possibility to use special characters like some that are popular for mathematical expressions.

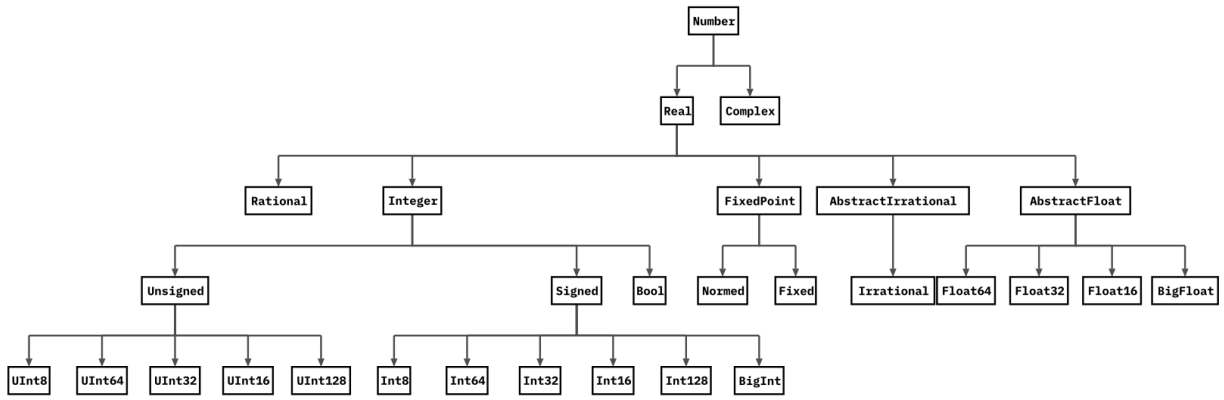
2.1.2 Types and Variables

As a dynamic language, variable types do not need to be defined in Julia, but it is left optional to the user if they want more performance.

Julia's type system is dynamic, nominal and parametric [40] and can be described as a hierarchy tree, where the type `Any` is the supertype, having all the other types as its children, directly or indirectly. Figure 4 shows the hierarchy of the type `Number`. It is a direct child of the type `Any` and all the types under it are its children. The advantage of having such hierarchy is to be able to generalize operations. A function defined as $f(x :: Number) = print(x)$ will work for any x of the type that can be found under the Number hierarchy. The "::" is used as an annotation, meaning that any expression on the left side of it is of the type on the right side of it.

Types can be abstracts, primitives or composites. Types that can be instantiated are final, so when looking at the hierarchy tree, only the types on the leafs can be instantiated, they are called concrete types. The parents of concrete types are abstract types

Figure 4 – Hierarchy of the type Number [41].



and are explicitly declared since inheriting behaviour is more important than inheriting structure [40].

In Figure 5, we show an example of the abstract types *Bird* and *Parrot* defined, being *Parrot* the child type of *Bird*. To be able to instantiate a type, it is necessary to create a constructor for it, as can be seen in line 5. The last line is the instantiation of the variable *myparrot* with its name.

Figure 5 – Defining abstract types.

```
1 abstract type Bird end
2 abstract type Parrot <: Bird end
3
4 # Constructor for Parrot type
5 Parrot(name::String) = name
6
7 myparrot = Parrot("Jules")
```

Composite types are known as structs, a type that can have multiple attributes. They are immutable unless the word “mutable” is written before the definition. Figure 6 shows the definition of the struct *Duck* and the instantiation of the variable *myduck* of a duck called Ben Affquack, who is 7 years old and has 4 ducklings. A constructor was not needed here but it could have been written.

Figure 6 – Defining a composite type.

```
1 struct Duck
2     name::String
3     years::Int64
4     ducklings::Int64
5 end
6
7 myduck = Duck("Ben Affquack", 7, 4)
```

Both abstract and composite types can be parametric to make them more generic. Parametric types can take parameters so that types with different parameters create different type that are part of the same family.

In Figure 7, we created a type to represent pets. As pets are animals that can be of different species, the field *animal* takes a parameter so that when we instantiate it passing a *Duck* type, the pet will be of type *Pet{Duck}*, and it would be possible to create other pets like *Pet{Dog}* for an example. These types can become handy when we use multiple dispatch methods, discussed in Section 2.3.

Figure 7 – Defining a parametric type.

```
1 struct Pet{T}
2     animal::T
3 end
4
5 mypet = Pet(myduck)
6 # mypet is of type Pet{Duck}(Duck("Ben Affquack", 7, 4))
```

Primitive types consist of bit types. They exist to allow Julia to bootstrap the standard primitive types that LLVM supports [40] and Julia recommends using primitive types that already exist, like `Int64`.

2.1.3 Arrays and Dictionaries

One remarkable feature of Julia arrays is that they are 1-based indexed. There have been many discussions around this topic, but the starting index as 1 has remained as a standard mostly because since Julia target is mathematical and scientific computation it can get similar syntax and better looking code.

Arrays work just as in any other language with array like types. In Figure 8, we show the basic ways to work with arrays, such as creating and accessing them, and the usage of the alias `Vector` that represents an one dimension array. We can notice that they are parametric types because they depend on the types of the elements, like `Vector{Float64}`, `Array{Int64, 1}` and `Vector{Any}`.

Multi-dimensional arrays work similarly as one dimension arrays and there is also an alias called `Matrix` for 2-D arrays, as can be seen in Figure 9.

Julia offers built-in operations for multi-dimensional arrays like multiplication by a scalar, or matrix multiplication. It also offers what is called broadcasting. With broadcasting the user can compute an operation over each element of an array. It is addressed by using the dot syntax before the operator or function to be broadcasted, as shown in Figure 10.

2.1.4 Control Flow

For conditionals, the `if...else` syntax is available, along with boolean switching expressions, ternary expressions and a function `ifelse`.

With boolean switching expressions, one can control what expressions to run with the help of the &&(and) and ||(or) operators. It is called "short-circuit evaluation". In Figure 11, there are a few examples of the usage of the boolean switch expressions. When the operator && is used, if the first operations returns `true`, the second operation must be executed to know the result of the boolean operation. Therefore, on line 11, the `println`

was executed and the entire expression returned "Hello world". On the other hand, if the first operation returned `false`, the `and` expression would be `false`, independently of the second operation. Hence, only `false` was returned. When the operator || is used, if the first operation returns `true` the entire expression will be `true` independently of the second operation. So, on line 17, will return `true`. When the opposite happens, the second operation must be executed and the expression then prints "Hello world".

Ternary expressions are there to help readability when the conditional is simple enough. The function `ifelse(op1,op2,op3)` does the work with the difference of evaluating both `op2` and `op3` no matter what is the result of `op1`. This can help improve performance as it will be discussed further in this chapter.

2.1.5 Functions and Methods

Functions in Julia can be just as simple as writing the keyword `function` in front of a function name and closing its body with the `end` keyword, but there are also other ways to declare them. One of them is the one expression function which can bring familiarity to the math writing as in line 5 of Figure 12.

Sometimes it is necessary to pass a function as an argument to another one, and this could be simplified by the usage of Anonymous functions, which are functions with no names. An example is the usage on the `map` function, where it is required to apply the function to each element of an array, as in 7 of Figure 12.

On Julia, a function is called a generic function and it can have several methods. This is due to the multiple dispatch mechanism, where a function can behave differently according to its arguments types.

Figure 8 – Using one dimension arrays.

```
# Creates an 1-D array of Int64 with five elements
julia> a1 = [1, 2, 3, 4, 5];

julia> a1[1] # Accessing first element
1

julia> a1[end] # Accessing last element
5

julia> a1[3:end] # Accessing a range from third element to the end
3-element Vector{Int64}:
 3
 4
 5

# Creates an 1-D array of Int64 with five undefined elements
julia> a2 = Array{Int64,1}(undef,5);

# Another way to create an 1-D array of Int64 with five undefined
    elements
julia> a3 = Vector{Int64}(undef,5);

# Creates an empty Float64 array
julia> a4 = Float64[];

# Creates an 1-D array of type Any
julia> a5 = ["Hello", 5, "world"]
3-element Vector{Any}:
  "Hello"
 5
  "world"

# Creates 1-D array using list comprehension
julia> a6 = [i for i in 1:5]
5-element Vector{Int64}:
 1
 2
 3
 4
 5
```

Figure 9 – Using multi-dimensional arrays.

```
# Creates an 2-D array of Int64 with five elements
julia> m1 = [1 2 3 4 5 6 7 8 9]
1x9 Matrix{Int64}:
 1  2  3  4  5  6  7  8  9

# Creates an 2x2 array of Int64
julia> m2 = [1 2;3 4]
2x2 Matrix{Int64}:
 1  2
 3  4

# Accessing 2-D array
julia> m2[1,2]
2

# Changing m1 from 1x9 Matrix to 3x3 Matrix
julia> reshape(m1,(3,3))
3x3 Matrix{Int64}:
 1  4  7
 2  5  8
 3  6  9

# Creating 2x2 undefined array
julia> m3 = Matrix(undef, 2, 2);

# Creating 3-D undefined array
julia> m4 = Array{Int64,3}(undef,3,3,3)
3x3x3 Array{Int64, 3}:
[:, :, 1] =
 0  0  0
 0  0  0
 0  0  0

[:, :, 2] =
 0  0  0
 0  0  0
 0  0  0

[:, :, 3] =
 0  0  0
 0  0  0
 0  0  0
```

Figure 10 – Broadcasting examples.

```
# Multiply array a1 by 2 and add 1 to each element
julia> a1 * 2 .+ 1
5-element Vector{Int64}:
  3
  5
  7
  9
 11

# Multiply matrix m2 by itself
julia> m2 * m2
2x2 Matrix{Int64}:
  7  10
 15  22
```

Figure 11 – Conditionals examples.

```
# If else conditional example
if <operation>
    # ...
elseif <operation>
    # ...
else
    # ...
end

# Using boolean switch expressions
julia> true && println("Hello world")
Hello world

julia> false && println("Hello world")
false

julia> true  println("Hello world")
true

julia> false  println("Hello world")
Hello world

# Ternary expressions
julia> iseven(2) ? "even" : "odd"
"even"

julia> iseven(3) ? "even" : "odd"
"odd"

# Ifelse function
julia> ifelse(iseven(2), "even", "odd")
"even"
```

Figure 12 – Function examples.

```
function foo()
    println("Hello world!")
end

f(x,y) = sqrt(x^2 + y^2)

map(x -> x % 2, [1,2,3,4,5,6])
```

2.2 Performance

The performance of Julia depends on some key aspects that basically rely on rich type information, code specialization and the JIT compilation with LLVM compiler framework [42].

When the source code is lowered to a Julia intermediate representation, most of the optimization happens. For every function, its body is specialized according to the tuples of argument types. These functions, called methods, when they are called for the first time, they are compiled and their results are cached. This way, even though the compilation is slow due to LLVM, next time the method is called it will run faster because its already cached and the argument types are hash-consed, a technique to share values that are structurally equivalent and that has shown to improve performance considerably [43].

For the specialization to happen, type inference must take part in it and it highly depends on type stability to improve performance. Following the example of Figure 13, we can see that **sumsofsins2** has lower execution time. This is because the variable r defined in this function does not change its type during execution, so the type inference can establish at first the right type for it. While **sumsofsins1** receives an integer value at first and later in the code it receives a float value. The compiler has to check every time for its current type and, indeed, makes it more complex and slower. By running the **code_llvm** built-in function we can see that the LLVM IR for **sumsofsins2**, Figure 14, is much simpler than for **sumsofsins1** (Listing 2.1). This example shows what should be fundamental to performance improvement when working with Julia, type stability.

Figure 13 – **sumofsins1** is an example of a type unstable function and **sumofsins2** is an example of a type stable function, followed by the execution time of each of them with 1000 iterations.

```
function sumofsins1(n::Int64)
    r = 0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end

function sumofsins2(n::Int64)
    r = 0.0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end

julia> @time [sumofsins1(100_000) for i in 1:1000];
  0.334276 seconds (112.10 k allocations: 6.189 MiB)

julia> @time [sumofsins2(100_000) for i in 1:1000];
  0.159626 seconds (107.16 k allocations: 5.970 MiB)
```

Listing 2.1 – LLVM representation of **sumofsins1**.

```
julia> code_llvm(sumofsins1, (Int,))
```

```
2
3  ;  @ <path>:1 within ‘sumofsins1’
4  ; Function Attrs: uwtable
5  define { %jl_value_t*, i8 } @julia_sumofsins1_1357([8 x i8]* noalias
       nocapture align 8 dereferenceable(8), i64) #0 {
6  top:
7  ;  @ <path>:3 within ‘sumofsins1’
8  ;   @ range.jl:5 within ‘Colon’
9  ;    @ range.jl:280 within ‘UnitRange’
10 ;     @ range.jl:285 within ‘unitrange_last’
11 ;      @ operators.jl:350 within ‘>=’
12 ;       @ int.jl:441 within ‘<=’
13        %2 = icmp sgt i64 %1, 0
14 ;
15       %3 = select i1 %2, i64 %1, i64 0
16 ;
17   br i1 %2, label %L11, label %union_move8
18
19 L11:                                             ; preds = %L49, %top
20 ;   @ range.jl:620 within ‘iterate’
21    %4 = phi double [ %value_phi3, %L49 ], [ 0.000000e+00, %top ]
22    %.sroa.014.0 = phi i64 [ %8, %L49 ], [ 0, %top ]
23    %tindex_phi = phi i8 [ 1, %L49 ], [ 2, %top ]
24    %value_phi2 = phi i64 [ %7, %L49 ], [ 1, %top ]
25 ;
26 ;  @ <path>:4 within ‘sumofsins1’
27   switch i8 %tindex_phi, label %L37 [
28     i8 1, label %L39
29     i8 2, label %L33
30   ]
31
32 L33:                                             ; preds = %L11
33 ;   @ promotion.jl:311 within ‘+’
34 ;    @ promotion.jl:282 within ‘promote’
35 ;     @ promotion.jl:259 within ‘_promote’
36 ;      @ number.jl:7 within ‘convert’
37 ;       @ float.jl:60 within ‘Float64’
38        %5 = sitofp i64 %.sroa.014.0 to double
39 ;
40 ;  @ <path>:3 within ‘sumofsins1’
41 ;   @ range.jl:620 within ‘iterate’
42    br label %L39
43
44 L37:                                             ; preds = %L11
45 ;
46 ;  @ <path>:4 within ‘sumofsins1’
47   call void @jl_throw(%jl_value_t* inttoptr (i64 176793136 to %
       jl_value_t*))
48   unreachable
49
50 L39:                                             ; preds = %L33, %L11
51   %value_phi3.in = phi double [ %5, %L33 ], [ %4, %L11 ]
52   %value_phi3 = fadd double %value_phi3.in, 0xBFD05AC910FF4C6C
53 ;  @ <path>:4 within ‘sumofsins1’
54 ;   @ range.jl:624 within ‘iterate’
55 ;    @ promotion.jl:398 within ‘==’
56    %6 = icmp eq i64 %value_phi2, %3
57 ;
```

```
58   br i1 %6, label %union_move, label %L49
59
```

```
60
61 L49:                                                ; preds = %L39
62 ;   @ range.jl:624 within ‘iterate’
63    %7 = add nuw i64 %value_phi2, 1
64 ;
65 ;  @ <path>:3 within ‘sumofsins1’
66 ;   @ range.jl:620 within ‘iterate’
67    %8 = bitcast double %value_phi3 to i64
68    br label %L11
69
70 post_union_move:                                    ; preds = %union_move8
       , %union_move
71    %9 = phi { %jl_value_t*, i8 } [ { %jl_value_t* null, i8 2 }, %
       union_move8 ], [ { %jl_value_t* null, i8 1 }, %union_move ]
72 ;
73 ;  @ <path>:6 within ‘sumofsins1’
74   ret { %jl_value_t*, i8 } %9
75
76 union_move:                                         ; preds = %L39
77   %value_phi3.lcssa = phi double [ %value_phi3, %L39 ]
78 ;  @ <path>:6 within ‘sumofsins1’
79   %10 = bitcast [8 x i8]* %0 to double*
80   store double %value_phi3.lcssa, double* %10, align 8
81   br label %post_union_move
82
83 union_move8:                                        ; preds = %top
84   %11 = bitcast [8 x i8]* %0 to i64*
85   store i64 0, i64* %11, align 8
86   br label %post_union_move
87 }
```

After having code specialized with its proper types inferred, devirtualization can be performed reducing dispatch overhead and enabling inlining. Inlining is an optimization where the function call is replaced by its body in the code. Sometimes it can have high memory cost and compilation time. In order to avoid it, heuristics are used to determine when this optimization [44] should be done. The next optimization step is unboxing, the process of retrieving a value of a previous allocated value with its type tag.

At last, the Julia Intermediate Representation (IR) is translated to LLVM IR and passes through its level O2 optimizations so it can finally generate an executable.

As a dynamic language, Julia lets the programmer write their code without worrying about type declaration of variables and memory management. However, to generate code with high performance, the user must adapt to the Julia way of programming, focusing on providing type stable code that can naturally comes from the usage of Julia features like Multiple Dispatch and Metaprogramming.

Figure 14 – LLVM representation of **sumofsins2**.

```
julia> code_llvm(sumofsins2, (Int, ))

;  @ <path>.jl:9 within ‘sumofsins2’
; Function Attrs: uwtable
define double @julia_sumofsins2_1358(i64) #0 {
top:
;  @ <path>:11 within ‘sumofsins2’
;   @ range.jl:5 within ‘Colon’
;    @ range.jl:280 within ‘UnitRange’
;     @ range.jl:285 within ‘unitrange_last’
;      @ operators.jl:350 within ‘>=’
;       @ int.jl:441 within ‘<=’
       %1 = icmp sgt i64 %0, 0
;
     %2 = select i1 %1, i64 %0, i64 0
;
  br i1 %1, label %L11, label %L37

L11:                                              ; preds = %L11, %top
  %value_phi2 = phi double [ %3, %L11 ], [ 0.000000e+00, %top ]
  %value_phi3 = phi i64 [ %5, %L11 ], [ 1, %top ]
;  @ <path>:12 within ‘sumofsins2’
;   @ float.jl:401 within ‘+’
   %3 = fadd double %value_phi2, 0xBFD05AC910FF4C6C
;
;   @ range.jl:624 within ‘iterate’
;    @ promotion.jl:398 within ‘==’
    %4 = icmp eq i64 %value_phi3, %2
;
   %5 = add nuw i64 %value_phi3, 1
;
  br i1 %4, label %L37, label %L11

L37:                                              ; preds = %L11, %top
  %value_phi6 = phi double [ 0.000000e+00, %top ], [ %3, %L11 ]
;  @ <path>:14 within ‘sumofsins2’
  ret double %value_phi6
}
```

2.3 **Multiple Dispatch**

Polymorphism is a concept where one same thing is able to assume different forms. In mathematics, this can be expressed as abstractions, for example, the multiplication being able to operate between different types of operands, such as real numbers, real and complex numbers, matrices, vectors and matrices and many others. In terms of mathematical implementations, it can lead the same operation to be executed with the best fitting algorithm regarding the operands, resulting in performance gains [45].

Julia makes this abstraction possible with its Dynamic Multiple Dispatch, where you have a function, called *generic function*, that can have multiple behaviors in its *methods*. These methods are defined by the different tuples of arguments that a generic function can have. It differs from overloading because it is resolved at runtime instead of compile time. It also differs from class based methods of object oriented programming because it takes into consideration every argument passed instead of only one.

Multiple Dispatch can be useful for scientific programming because it makes it easier to right mathematics expressions similar to as they really are. Figure 15 shows a multiple dispatch example of the operator * where it can be useful to represent different transformations and we can clearly see its similarities in the code:

1. $y = a \times g(x)$ - Scale operation of vertical stretch or compression
2. $y = f(t \times x)$ - Scale operation of horizontal stretch or compression
3. $y = f(g(x))$ - Function composition

Figure 15 – Example of multiple dispatch of the operator * [5]

```
1 *(a::Number, g::Function) = x->a*g(x)        # Scale output
2 *(f::Function, t::Number) = x->f(t*x)        # Scale argument
3 *(f::Function, g::Function) = x->f(g(x))     # Function composition
```

On the other hand, the Julia programmer is inherently writing code with better performance because of the synergy between multiple dispatch with code specialization and type inference.

2.4 **Metaprogramming**

Julia has strong LISP influences and it can be mostly seen at its metaprogramming strategy. The code can be accessed after it is parsed, but before it is executed, at the level of abstract syntax trees. This way, taking advantage of any compile time optimization.

The most common metaprogramming feature found in Julia is a macro. A macro is an expression that contains a piece of code and accepts arguments. It is evaluated at parse time, and returns an non-evaluated expression [46].

A simple example of this is the **timev** macro from Julia's source code. It prints the time an expression took to be executed and memory allocations as shown in Listing 2.2. The **quote** block delimits a piece of non-evaluated code that stores the time right before the expression **ex** is executed (line 4) and it is subtracted from the time right after it (line 6). The piece of code in this listing will then be expanded at runtime.

Listing 2.2 – Julia's timev macro

```
macro timev(ex)
    quote
        local stats = gc_num()
        local elapsedtime = time_ns()
        local val = $(esc(ex))
        elapsedtime = time_ns() - elapsedtime
        timev_print(elapsedtime, GC_Diff(gc_num(), stats))
        val
    end
end
```

3 PARALLEL COMPUTING IN JULIA

As Kristian Carlsson expresses very well in [47], the good old days of processors getting a higher clock-speed every year have been over for quite some time now. Therefore, nowadays, it is important to exploit parallel computing in the code in order to achieve higher performance. Julia language was developed with the power of parallelism as its goal. It provides built-in primitives for parallel computing at different levels, offering: distributed computing, multithreading, data vectorization and GPU programming.

3.1 Distributed Computing

Julia provides a parallel environment based on message passing to allow programs to run multiple processes in separate memory address spaces. An implementation of distributed memory parallel computing is provided by the module *Distributed* as part of the standard library shipped with Julia.

Although MPI is the popular standard for messaging passing, the message passing behaves differently in Julia. The communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process communication. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions. To control these processes, the standard library module *Distributed* is used. It provides several methods and features related with distributed programming. For example, the `addprocs` method that allows the creation a process dynamically, while the `procs` method gives the list of processes running in Julia environment as shown in Listing 3.1. The array returned by the `addprocs` method is different than the one returned by the `procs` method. This is because the first one references only the added processes that will be in charge of doing computations, those are called *workers*. The second one includes the process 1, which is called the *master*, it is the only one that can manage processes and it is where the Julia REPL runs.

Listing 3.1 – Methods to create processes

```
julia> using Distributed
julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4
julia> procs()
3-element Array{Int64,1}:
 1
 2
 3
 4
```

For the communication between processes, the *Distributed* package provides functionalities for data exchange, these are the remote references and remote calls. A remote reference is an object in one processor that references an object in another processor. Remote references can be either of type *Future* or *Remote Channel*. A remote call is a call to a function on the same or another process. When a remote call is made, it instantly returns the *Future* object and proceeds to execute the next instructions while the process that was designated by the remote call runs the called function. If the process that made the call needs to either wait for the remote call to be done or obtain the value returned, it can do it with the `wait` and the `fetch` method, respectively.

Listing 3.2 – Remote call example.

```
julia> @everywhere f(x) = x+3
#f (generic function with 1 method)
julia> r = remotecall(f, 2, 5)
Future(2, 1, 62, nothing)
julia> wait(r)
Future(2, 1, 62, nothing)
julia> fetch(r)
8
```

In Listing 3.2, we have a function f that receives a parameter and returns the value of this parameter plus 3. It is worth noting that the usage of the macro `@everywhere` is a way to load the function at every process. A remote call is then performed passing as parameters: the function f, the id of the process that will execute the function (in this case 2), and the parameters of that function (the value 5). The returned *Future* object contains the attributes to identify the worker id, the master process, the id of the remote call and a reserved attribute to keep the value returned (that at first has the type `nothing`). At last, we wait for the computation to be done with `wait` and get its result with `fetch`.

There are other ways, besides the `remotecall()` function, to send work to different processes, such as using the `@spawnat` macro, which as the name implies, spawns the function in a given process. The example in Listing 3.3 shows the use of `@spawnat` for the same computation done with `remotecall`.

Listing 3.3 – @Spawnat example

```
julia> @everywhere f(x) = x+3
#f (generic function with 1 method)
julia> s = @spawnat 2 f(5)
Future(2, 1, 67, nothing)
julia> fetch(r)
8
```

3.1.1 Channels

A channel in Julia works as a pipe, but it can only be accessed by a local process, another process does not have access to it. A remote channel is a channel that all the workers can access it. It can be useful to solve problems like the Producer-Consumer [48]. In this problem, the producer process produces something and puts in a buffer and another process, called consumer, consumes from this buffer.

Figure 16 shows an example of the producer-consumer problem. The package *Distributed* is loaded and one more process is added. We then make the buffer with the

remote channel, it has a channel that can keep 32 integers. Next, we define the produce method with the `@everywhere` macro so it can be run on another worker, we define it will produce n products, the sleep is used to simulate a computation and then it puts on the remote channel *products* the results. The consume method will try to consume n products, we used it as a parameter to indicate when to exit the program. The `remote_do` method is used to start the `produce` method on worker 2 and following we start the consume method. The output is on Figure 17 and we can note that right when an item is produced by the produce method it is consumed by the consume method.

Figure 16 – The producer-consumer problem using Julia's remote channels.

```
using Distributed

addprocs(1)

const products = RemoteChannel(() -> Channel{Int}(32))

@everywhere function produce(products, n)
    for _ in 1:n
        product = rand(1:10)
        sleep(product)
        println("produced ", product)
        put!(products, product)
    end
end

function consume(products, n)
    produced = 0
    while produced < n
        product = take!(products)
        println("Consumed product number ", product)
        produced += 1
    end
end

println("start producer")
remote_do(produce, 2, products, 10)

println("now consume")
consume(products, 10)
```

3.1.2 Reduction Operation

A very common operation found in parallel computing is the reduction operation. It reduces elements generated by different processes into a single result. The *Distributed* package offers a macro to deal with this operation. The syntax is simple: `@distributed` followed by the operator that represents the reduction operation, e.g., `@distributed` (+) to sum all the elements. To illustrate the use of reduction we show the Monte-Carlo method to compute the value of π in Figure 18.

The Monte-Carlo method uses the ratio between the area of a circumference, $A_c = \pi r^2$ and the area of a square $A_s = l^2$. If we take a square with a side of length 2 and a circumference of radius 1, the ratio between them would be $\frac{\pi}{4}$. So, to calculate the value

Figure 17 – Output of the producer-consumer problem script from Figure 16

```
start producer
now consume
      From worker 2:    produced 3
Consumed product number 3
      From worker 2:    produced 6
Consumed product number 6
      From worker 2:    produced 5
Consumed product number 5
      From worker 2:    produced 10
Consumed product number 10
      From worker 2:    produced 1
Consumed product number 1
      From worker 2:    produced 4
Consumed product number 4
      From worker 2:    produced 2
Consumed product number 2
      From worker 2:    produced 1
Consumed product number 1
      From worker 2:    produced 1
Consumed product number 1
      From worker 2:    produced 4
Consumed product number 4
```

of π based on this ratio, we suppose to have a 1 radius circumference inside a square of side 2. N random points are then generated and checked to see if they are inside the circumference or not. The approximated value of π is obtained by dividing the number of points inside the circumference by the total of points generated, as being the ratio between the areas, times 4.

In Figure 18, we show the sequential implementation of the Monte-Carlo method and a parallel implementation using the package *Distributed*. The reduction is used in this case to count the number of points that were generated inside the circumference. This is made by using `@distributed (+)`. At the end of each iteration of the for loop, the *n_landed_in_circle* will be either of value one or zero in each process, and the reduction provides the global count. To obtain the execution time we used the package BenchmarkTools [49] to run each Monte-Carlo method 1000 times with one evaluation each on a Intel Core i5-8250U CPU. The sequential method returned a value for π of 3.1404 within the mean time of 778.895μs and the parallel method returned a value of 3.13604 within the mean time of 639.761μs using 3 workers.

Figure 18 – Functions to compute π value with Monte-Carlo method sequentially and using *Distributed*. The execution time is measured with the *BenchmarkTools* package.

```
using Distributed, BenchmarkTools

addprocs(3)

evals=evals
samples = 1000
evals = 1
N = 100000

function compute_pi(N::Int)
    n_landed_in_circle = 0  # counts number of points that have radial coordinate < 1, i.e. in circle
    for i = 1:N
        x = rand() * 2 - 1  # uniformly distributed number on x-axis
        y = rand() * 2 - 1  # uniformly distributed number on y-axis

        r2 = x*x + y*y  # radius squared, in radial coordinates
        if r2 < 1.0
            n_landed_in_circle += 1
        end
    end

    return n_landed_in_circle / N * 4.0
end

function compute_pi_distributed(N::Int)
    n_landed_in_circle = @distributed (+) for i = 1:N
        x = rand() * 2 - 1
        y = rand() * 2 - 1

        r2 = x*x + y*y

        n_landed_in_circle = r2 < 1.0 ? 1 : 0
    end

    return n_landed_in_circle / N * 4.0
end

compute_pi(N)
compute_pi_distributed(N)

@benchmark compute_pi($N) samples=samples evals=evals
@benchmark compute_pi_distributed($N) samples=samples evals=evals
```

3.2 GPU Programming

GPU programming is one of the categories of parallel computing in Julia that has been showing increasing growth recently. It is a hot topic in lately JuliaCon (annual Julia Conference) editions. Julia supports GPU programming for several architectures: NVidia with CUDA.jl [9], Intel GPU with oneAPI.jl [50], AMD GPU with AMDGPU.jl [10] based on ROCm platform, and Apple GPU with Metal.jl [11]. Julia also offers other libraries, like OpenCL.jl [51].

Among these packages, the most mature one is NVidia CUDA.jl. That allows the development of kernels with the execution of operations directly on the GPU.

Figure 19 – Matrix multiplication using CUDA.jl.

```
1 using CUDA
2
3 M = rand(2^11, 2^11)
4
5 # Copying the matrix to the GPU
6 M_on_gpu = cu(M)
7
8 # Executes the matrix multiplication on GPU
9 CUDA.@sync M * M
```

In Figure 19, we can see how simple it is to execute a matrix multiplication using CUDA in Julia. To ensure that the GPU will be used and it has the data needed to perform the operation, it is necessary to copy the matrix structure to the GPU using the `cu(M)` function, which returns a `CuArray`. When the `*` operator is used on `CuArrays`, the execution is performed on the GPU. The macro `CUDA.@sync` is there to prevent the code execution to continue while the operation is executing on the GPU.

It is also possible to create your own kernels. CUDA.jl provides the functions `blockDim`, `gridDim`, `blockIdx` and `threadIdx` that provides the necessary information for the distribution of work among the GPU threads. Figure 20 shows a kernel implementation of the addition of two matrices on GPU using CUDA.jl. On CUDA, the threads are grouped in blocks, representing its dimention (or size) by `blockDim`, and the group of blocks are part of a grid, representing its dimention (or size) by `gridDim`. The `worker_gpu_add!` function will calculate the indexes of the matrix elements to be added using the dimension of the block and the current ID's of both block and thread, along the `x` dimension. In line 3, if the expression `index <= length(u)` is false, the result of the boolean expression with `&&` is false, and `@inbounds u[index] += v[index]` will not execute. If `index <= length(u)` is true, `@inbounds u[index] += v[index]` needs to execute to determine the result of the boolean expression. With this, `@inbounds u[index] += v[index]` can only execute if the current index is less than length of `u`. The `gpu_add!` function will calculate the necessary parameters for initializing the `worker_gpu_add!` and triggering its execution. The number of blocks is determined by the size of the matrix. Having the number of threads and blocks set, the `worker_gpu_add!` can be called by passing these parameters following the macro `@cuda` to execute it on the GPU. The `gpu_add!` is then called passing matrices that were previously copied to the GPU.

Figure 20 – Kernel to add two matrices using CUDA.jl.

```
function worker_gpu_add!(u, v)
    index = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    index <= length(u) && (@inbounds u[index] += v[index])
    return
end

function gpu_add!(u, v)
    numblocks = ceil(Int, length(u) / 256)
    @cuda threads=256 blocks=numblocks worker_gpu_add!(u, v)
    return u
end

u = rand(2^20)
v = rand(2^20)

u_on_gpu = cu(u)
v_on_gpu = cu(v)

gpu_add!(u_on_gpu, v_on_gpu)
```

3.3 **Multithreading**

Before elaborating on Julia multithreading, we shall introduce the concept of tasks. Tasks in Julia are a concept adopted in asynchronous programming and can also be known as coroutines or green threads. Basically, a task can be created with a given computational work and it needs to be scheduled to start running so that the computation can be switched between the main program and the task. To block the main program until the task finishes, the function `wait` can be called. If the return value of the task is needed, it can be obtained with the `fetch` function. An example is shown in Figure 21, where we can see that when a task is created it is marked as `runnable`. The task only waits for 10 seconds and then returns an array. When we check the task before it is done, it is still marked as `runnable` until it is finally marked as `done`. We then just call `fetch(t)` to obtain the array returned by the task.

Figure 21 – A task being created, scheduled and having its result fetched.

```
julia> t = @task begin; sleep(10); return rand(Int64,1000); end
Task (runnable) @0x00000000174d2850

julia> schedule(t)
Task (runnable) @0x00000000174d2850

julia> t
Task (runnable) @0x00000000175fab30

julia> t
Task (done) @0x00000000175fab30

julia> fetch(t)
1000-element Array{Int64,1}
...
```

Multithreading appeared for the first time in Julia in early 2017, at version 0.5 with the macro **@threads**. This macro, however, was experimental at that time. It handled simple parallel loops running on all cores and had two important contributions: Julia programmers could start taking advantage of multiple cores, and the use of the macro provided test cases to report thread-related bugs in runtime.

However, this version of the macro had some important limitations, the loops with **@threads** could not be nested. If the functions they called used **@threads** recursively, these internal loops would not be parallelized. Another limitation of the macro was that it was not able to perform I/O operations and it was incompatible with the task system.

In July 2019, in the version 1.3.0, Julia presented multithreading parallelism available with the macro **@spawn**, in the package **Threads**, allowing tasks to run simultaneously in a pool of threads and these tasks have a shared stack pool to allow nested calls.

In the first versions of Julia, the execution with multiple threads needed an environment variable, JULIA_NUM_THREADS, to be set with the number of threads to be created. From the version 1.5 on, there is an option to start Julia with a command line parameter indicating the number of threads to be created. This make it easier to change the number of threads according to the execution, with the option *-t*.

It is possible to check which thread is running by searching for its *id* using the method `threadid()`. In the latest version of Julia, I/O is fully supported. We can see it all in the example of Figure 22, where we parallelize the `for` loop with threads and for each iteration we print to which thread that iteration was assigned.

Figure 22 – Julia code for printing with threads

```
julia> import Base.Threads

# Shows total of threads available
julia> Threads.nthreads()
8

julia> Threads.@threads for i in 1:10
    println("i = $i on thread $(Threads.threadid())")
end
i = 1 on thread 1
i = 7 on thread 3
i = 2 on thread 1
i = 8 on thread 3
i = 3 on thread 1
i = 9 on thread 4
i = 10 on thread 4
i = 4 on thread 2
i = 5 on thread 2
i = 6 on thread 2
```

To avoid race conditions, the user can either use locks to isolate the code block or use atomic operations depending on what computation is needed. The atomic operations available are addition, subtraction, exchange of values and boolean operations.

Julia multithreading employs *data parallelism* and *task parallelism*. Data parallelism in Julia was first proposed using the `@threads` macro. This macro provides an easy parallelization of a for-loop and is the most common usage with multithreading. It is simply placed before a for loop and it distributes the loop iterations between threads as shown in Figure 22.

Figure 23 – Atomic operations

```
julia> x = Threads.Atomic{Int}(7)
Atomic{Int64}(7)

# Adds 3 ints to x and returns the previous value
julia> Threads.atomic_add!(x, 3)
7

# Accesses the value of x
julia> x[]
10
```

Task parallelism in Julia was inspired by parallel programming systems like Cilk [52] and Intel Threading Building [53]. The programmer creates tasks freely and the runtime scheduler is responsible to decide when and where the tasks will be executed.

The core built-in primitive for task parallelism in Julia is the macro `@spawn`. The `@spawn` macro specifies that a function (or an expression) will execute in parallel with the caller. To wait for the task to finish, a `wait` or `fetch` command has to be used. What is interesting about `@spawn` is that it allows the use of dynamic parallelism in a very simple way. Figure 24 shows a parallel code in Julia to compute the classic Fibonacci sequence [14]. In this code, the command `t = @spawn fib(n - 2)` creates a task to compute fib(n - 2), which will execute in parallel with `fib(n - 1)`. The command `fetch(t)` waits for task t to complete and gets its return value. That is a great example of its usage, a recursive function. Multithreading on Julia is composable and can make nested calls to multithreaded code without oversubscribing threads.

Figure 24 – Fibonacci computation with `@spawn` [14]

```
function fib(n::Int)
    if n < 2
        return n
    end
    t = @spawn fib(n - 2)
    return fib(n - 1) + fetch(t)
end
```

3.4 Loop Scheduling

Parallelizing independent iterations of a loop is one of the most common forms of data parallelism. In a parallel loop implementation, loop scheduling refers to how the iterations are distributed among the threads. The loop scheduling can be *static* or *dynamic*. In a static loop scheduling, the iterations of a loop are divided before the loop starts in a fixed manner. Each thread gets a predefined set of iterations to compute. In a dynamic loop scheduling, the iterations are assigned to the threads as the loop is being executed. Static scheduling may result in load imbalance when the computation of the iterations is uneven. Dynamic scheduling tries to overcome this problem by performing the assignment according to the execution time of the iterations. However, it can incur in overhead during the loop computation.

In Julia, there are two ways of parallelizing a loop: using the `@threads` macro or using tasks with `Threads.@spawn`. Figures 25 and 26 show a `for` loop that computes the sum of two vectors of size N parallelized using `@threads` and `Threads.@spawn`, respectively. The loop scheduling strategies for the two loop parallelization implementations are different. The parallelization with `@threads` splits the N iterations of the loop statically among the threads. It divides N by the number of threads and distribute each chunk of iterations to the threads in a round-robin fashion. This scheduling strategy resembles the OpenMP static loop scheduling.

The parallelization with `Threads.@spawn` creates N tasks that are dynamically assigned to threads by the runtime. To run spawned tasks simultaneously, Julia runtime incorporated the parallel task scheduler, called PARTR (https://github.com/kpamnany/partr). In PARTR, the tasks are scheduled according to the algorithm depth-first scheduling [54]. In the depth-first scheduling algorithm, when a core completes a task, the next task that will be assigned to this core is the task that is ready to execute and that would be the natural next task to be executed if the execution was sequential. For example, consider a nested parallel computation represented in Figure 27. In this Figure 27(a), we show a nested parallel computation where the arrows from the right to left represent forks (the parent thread is creating a child thread), the arrows from the left to the right represent synchronization of a child thread with its parent, and the vertical arrows represent sequential dependencies. The depth-first order of this computation is presented in Figure 27(b), where a forked child is executed before its parent, so that the depth-first order is the same as the unique serial execution order. The depth-first scheduling algorithm prioritizes ready tasks according to the depth-first order. In this way, the algorithm tends to make more effective use of cache resources [55].

Figure 25 – Parallelizing a loop using `@threads`

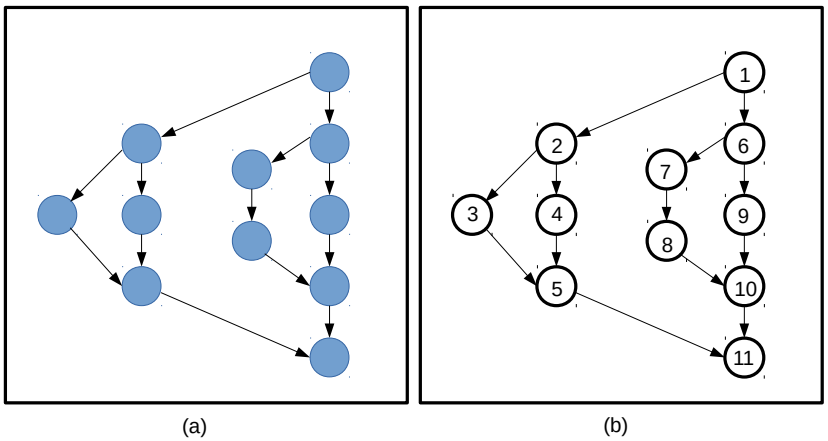
```
1        Threads.@threads for i = 1:N
2            a[i] = b[i] + c[i]
3        end
```

Besides Julia's built-in multithreading, there are community packages that aims in improving performance combining algorithms and parallelism. One of them is FLoops.jl [12], that is built on the idea of composable transformations, which often involve reducing operations, that are brought by the usage of Transducers [56]. FLoops.jl implements these transformations by converting `for` loops to `foldl` from Transducers.jl [57] and promises

Figure 26 – Parallelizing a loop using `@spawn`

```
        @sync for i = 1:N
            Threads.@spawn begin
               a[i] = b[i] + c[i]
            end
        end
```

Figure 27 – Parallelizing a loop using `@threads`



to improve them with the usage of the macro `@floop`, allowing their execution sequentially or in parallel.

`@floop` usage is similar to the macro `@threads`, as in its simplest form the user only needs to place it before the `for` loop, but broader because it is possible to iterate on different collection types, like dictionaries for an example. The way the loop preceded by this macro is written determines if it will run sequentially or parallel, Figure 28 shows an example of a sequential loop that performs a reduction operation and Figure 29 shows a parallel version of it where it is needed to combine it with the macro `@reduce`

Figure 28 – Sequential usage of the macro `@floop` on a reduction for loop [12]

```
@floop begin
   s = 0
   for x in 1:3
       s += x
   end
end
```

Figure 29 – Parallel usage of the macro `@floop` on a reduction for loop [12]

```
s = 0
@floop for x in 1:3
    @reduce s += x
end
```

Loops that do not involve reductions will also be parallelized and the parallel mechanism is defined by an executor. The executor is passed as a parameter to the `@floop` macro and will tell if it will be used distributed, GPU or mulithreading parallelism. If multithreading is used, the executor defines the loop scheduling it will execute and they are listed below, there are not many details about their implementation but more of considerations about best fitting scenarios for each of them [58]:

- **ThreadedEx**: The default executor used if none is defined. As the documentation says, it is implemented in a divide-and-conquer strategy, where the input is recursively halved until the parts, or base cases, are equal or less to a defined `basesize`. Each base case is assigned to a task and their results are combined pair-wise in distinct tasks [12];

- **WorkStealingEx**: Uses the continuation stealing [59], where if a task performs a function `x()` followed by the spawning of a task to execute a function `y()`, any other operation after the spawn is available for being stolen;

- **DepthFirstEx**: Distributes the chunks defined by the `basezise` among tasks in the order they are set in the collection but they do not wait the other chunks to be distributed to start running. The documentation declares that the best scenarios are the "findfirst" ones, where the result can be find early in executions and prevent further executions;

- **TaskPoolEx**: This strategy uses a pool of tasks to limit the resource usage, thus avoiding any latency generated by I/O operations or computer-intensive reductions;

- **NondeterministicEx**: The task distribution and combination for reduce operations in this executor is determined by the timing of the computations at runtime. It is recommended to execute reductions on non-deterministics operations, where the order of the operations in the reduction influences in the results.

3.5 SIMD Parallelization

Modern processors provide not only the opportunity to exploit thread parallelism with the growing number of cores, but also the opportunity to exploit fine-grained data parallelism with the vector units. Vector operations can process the same instruction on distinct vector elements at the same time (Single Instruction Multiple Data or SIMD). Recent Intel processors, for example, have a 512-bit vector unit that allows to process, simultaneously, 16 single-precision floating point operations or 8 double-precision floating point operations. This can be 8 or 16 times faster than a scalar (non-vector) operation. Combined with thread parallelism, vectorization can provide massive performance gains.

There are different alternatives to exploit vectorization in Julia code that vary in terms of complexity. The first alternative is to let the compiler exploit vectorization, called auto-vectorization. This alternative is simple and requires no additional programming effort, but it is also restrictive in terms of the parallelism opportunities to be exploited. The second alternative is to use the macro @simd. This alternative is also very simple but requires the programmer to indicate the opportunities for vectorization. The third alternative is to use a package for vectorizing the loops. This alternative requires more program effort than the other two, but it also provides optimizations on the memory usage and accesses. The fourth alternative is the more complex one, the use of low-level instructions provided by the vector processing unit, called intrinsic instructions.

3.5.1 Auto-Vectorization

Julia compiler has the auto-vectorization ability. The compiler can identify blocks of code that can be vectorized, exploiting vector registers and the vector arithmetic unit in a completely transparent way. The automatic vectorization, however, will only be applied if the compiler can tell whether a specific block of code would have the same behavior of a vectorized one and that it would be profitable. That way, the compiler will transform the program according to its hardware restrictions.

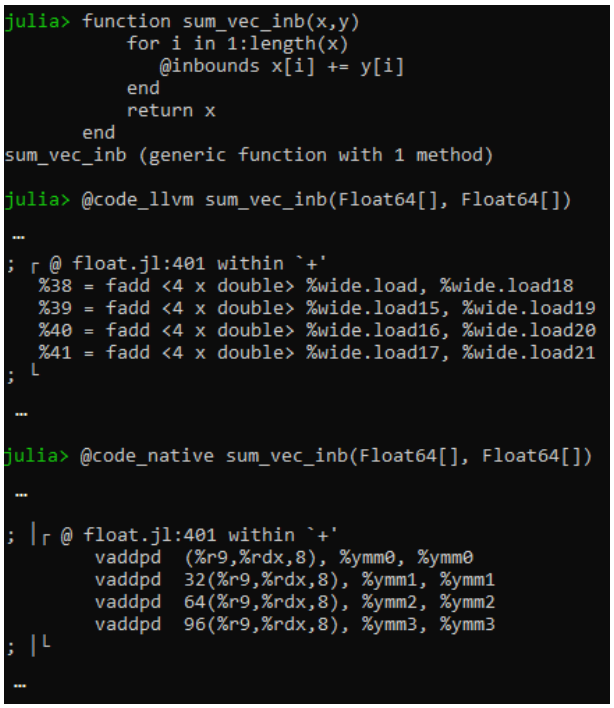
To check if the code was vectorized, the user can use either the `@code_llvm` or `@code_native` macros to look in the bitcode or the assembly instructions for expressions used for SIMD operations.

In Figure 30, we show an example with a simple summation of the elements of two arrays. When `@code_llvm` was used we got some hints that SIMD is being used. For example, the usage of the type *<4 x double>* represents a vector. The output of `@code_llvm` also shows that the compiler decided to unroll the loop in chunks of size four. When we look at the `@code_native` we can see that there are four vector instructions using the register `ymm` that fits 256 bits, having four double-precision floats [60].

The use of the @inbounds macro in Figure 30 needs further explanation. Julia uses bounds checking for array accesses. When the code has a loop over each element of an array, if the loop iterations gets outside of the array bounds, an error is thrown. Therefore, every time we try to access an element from an array with an index, there is an implicit `if` to check if the index actually exists in the array. This means that, at a lower level, there is a control flow in the piece of code we would like to vectorize, which would make it nonviable for vectorization. The macro `@inbounds` is used to avoid the bound checking, and vectorization can be done without problem. If another function would be created but without `@inbounds` we would not have the same output from `@code_llvm` or `@code_native`, no SIMD representations would be found.

The usage of the macro `@inbounds` illustrates that, although Julia has auto-vectorization,

Figure 30 – Example of the representation of the summation of two arrays in LLVM bitcode and assembly instructions showing usage of SIMD operations. Additional outputs hidden.



it is necessary to guarantee to the compiler that the code is safe to be vectorized. Other recommendations are the following [61]:

- No cross-iteration dependencies.
- Conditionals are not allowed unless they can be done using the `ifelse` method. With `ifelse`, both expressions that depend on the result of the condition would be evaluated.
- Make sure that all calls are inlined.
- Write type-stable code.
- Use unit-stride array subscripts inside loops. Accessing elements as `array[2i]` might not be possible.
- Reduction variables should be local variables.

In addition, reduction operations are not supported for Float32 or Float64 types. This is due to the fact that vectorization reorders the operations and that reordering can change the final results on floats.

3.5.2 @SIMD Macro

Sometimes, when the operations are too complex for the compiler to infer the legality of the vectorization, the user can explicitly tell the compiler that a piece of code is independent and there is no overlap between the structures being accessed. This can be done with the macro `@simd`.

The macro `@simd` can only be used prior to `for` loops and it is able to vectorize reductions. When it is used, it assumes that the code following the macro is safe to be vectorized. That means the user has to check for dependencies. If the loop has dependencies, they will be vectorized and it may behave unexpectedly.

By the time this Dissertation is written, the `@simd` macro is marked under Julia's documentation as an experimental feature and it is not encouraged to be used. There is also a list of considerations to be taken when using `@simd` in the documentation [62]:

- The loop must be an innermost loop;
- The loop body must be straight-line code. Therefore, `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short &&, ||, and ?: expressions into straight-line code if it is safe to evaluate all operands unconditionally. Consider using the `ifelse` function instead of ?: in the loop if it is safe to do so;
- Accesses must have a stride pattern and cannot be "gathers" (random-index reads) or "scatters" (random-index writes);
- The stride should be unit stride.

3.5.3 Vectorization Packages

As an alternative to the built-in macro `@simd`, there are a few packages available that can take advantage of the SIMD operations. The first one is the LoopVectorization.jl package [13].

The LoopVectorization package can optimize `for` loops and nested `for` loops as well as broadcasting operations [63], a way to perform element-by-element operations on arrays reducing memory usage and with a simple syntax. The package analyzes the cost of the instructions in the loops and the number of times those instructions should be executed. Based on latency and throughput of the instructions and register consumption measurements, it decides which strategy to apply to the loops, these involve vectorization of the loops, loop reordering and unrolling.

It provides for the user vectorized map, filter and reduce functions and the macros `@turbo` and `tturbo` to be used to optimize loops or broadcasting operations. The `@turbo` macro also takes a few arguments where the user can customize its usage, one of those arguments sets the usage of threads to true or false, enabling the optimization with multithreading. The `@tturbo` macro is just a simpler way to call `@turbo` with threads usage set to true.

When using LoopVectorization some limitations must be taken into consideration:

- it considers that the loop iterations are independent,
- it does not perform bound checks in arrays,
- it assumes that each loop iterates at least once and that there is only one loop per nest level.

Another package that can take advantage of the SIMD operations is the SIMD.jl [64]. The SIMD.jl package provides a vector type. With that type any usual arithmetic, logical and reduction operations will be executed using SIMD parallelism. The access to the

Figure 31 – Functions to sum two arrays using LoopVectorization.jl and SIMD.jl

```
using LoopVectorization

function turbo_sum_arrays(a::Array{T,N}, b::Array{T,N}, c::Array{T:N})
    where {T,N}
    @turbo for i in 1:length(a)
        c[i] = a[i] + b[i]
    end
    return c
end

using SIMD

function simd_sum_arrays(a::Array{T,N}, b::Array{T,N}) where {T,N}
    a_v = Vec(Tuple(a))
    b_v = Vec(Tuple(b))
    c = a_v + b_v
end
```

elements of an array is done by using the functions `load` and `store`. They allow the creation of arrays of scalars instead of array of vectors.

Figure 31 shows an example of the sum of two vectors using the packages LoopVectorization.jl and SIMD.jl. The function `turbo_sum_arrays` uses the LoopVectorization.jl package, for that it is necessary to perform the summation either including `@turbo` in front of the loop, as in the figure, or using broadcasting. The function `simd_sum_arrays` can perform the SIMD operation when using the type `Vec`, so for that we pass the arrays for the type `Vec`. We pass it first to a tuple because `Vec` is immutable and only accepts immutable types, like the Tuple is.

3.5.4 Intrinsics

The LLVM instructions can be called straight inside the code. This option may help if a specific instruction is needed or if none of the above options were helpful. However, it can make the code harder to read with and less portable since it depends directly on the LLVM instruction [60].

Figure 32 shows an example of a call to the LLVM `sin` intrinsic that returns the `sin` of a float or a vector of floats in radians. In order to Julia pass a data structure to the LLVM as a vector type, it needs to be declared in a special way using `VecElement`. Having the special vector type declared, the call to the intrinsic can be done by the `ccall` function that has as arguments the instruction, the library of the instruction, the return type, the argument types, and the argument of the function `llvm_sin` to be created. At last the function `llvm_sin` is called on a vector with $\pi/2$ and 0 values, to return 1 and 0 respectively.

Figure 32 – Declaration of special type for vectorization `fp128` and call for LLVM intrinsic instruction `sin`.

```
julia> const fp128 = NTuple{2, VecElement{Float64}};

julia> llvm_sin(vec) = ccall("llvm.sin.f128",llvmcall, fp128, (fp128,),
    vec)
llvm_sin (generic function with 1 method)

julia> llvm_sin(fp128((pi/2,0.0)))
(VecElement{Float64}(1.0), VecElement{Float64}(0.0))
```

4 EVALUATING DATA AND TASK PARALLELISM IN JULIA MULTI-THREADING

In this chapter, we perform a detailed analysis of Julia's data and task parallelism mechanisms. We evaluate the two main built-in loop parallelization macros present in Julia, `@spawn` and `@threads`.

4.1 Experimental settings

The experiments were conducted on a AMD Ryzen 7 2700 Eight-Core Processor with 16 threads (2 threads per core with Simultaneous Multi-Threading), 128 GB of RAM and Julia version 1.7. We used the speedup in our analysis, being calculated as the execution time of the sequential code block divided by the execution time of the parallel code block. To ensure statistical significance and amortize any possible random effect over task scheduling or distribution, each experiment was executed 10 times and the execution time used was the mean execution time of these 10 runs. For these measurements we used the macro `@timed` as it was easier to analyze different blocks of code.

We measured the execution time and the percentage of load imbalance as proposed in [65], that represents how unevenly the work is distributed. The percentage of load imbalance, λ is given by:

$$\lambda = (\frac{L_{max}}{\overline{L}}) - 1 \times 100\%, \tag{4.1}$$

where L_{max} is the maximum execution time of any thread and $\overline{L}$ is the mean execution time of all threads.

When the benchmarks have parallel loops inside another loop, the measurements were made for each iteration of the external loop and the load imbalance was calculated as the mean in regards to the number of the external loop iterations.

4.2 Benchmarks

We evaluate the performance of the Julia's data and task parallelism using synthetic kernels and applications from a well-known benchmark suite. The synthetic kernels fabricate situations of balanced and unbalanced scenarios, while the applications were selected from the suite Rodinia [66].

Three synthetic kernels were implemented: *Balanced*, *Unbalanced* and *Linked List*. The *Balanced* and *Unbalanced* kernels consist of a nested loop with its outer loop defining the parallel Julia task and an inner loop defining the iteration work. The work of one iteration of the Balanced kernel is constant (see Figure 33). While the work of iterations of the Unbalanced kernel varies depending on the task id (see Figure 34).

In the *Linked List* kernel, each node of a linked list has a value on which the Fibonacci number is computed. We expect an unbalanced computation given by the different cost

in computing the Fibonacci number of these different values. The implementation with dynamic scheduling is shown in Figure 35. The idea is to distribute the node computation among Julia tasks. In the main `while` loop of the algorithm, for each Fibonacci call, we spawn a new task. The implementation with the static scheduling required some modifications in the code. Each node of the list was assigned to an element of a vector and we iterated over it with a `for` loop. In this way, we used the macro `@threads` for the parallelization.

The benchmark applications from Rodinia that we used were: SRAD_v2, LUD, and BFS. SRAD_v2, Speckle Reducing Anisotropic Diffusion, is a method for removing noise in ultrasonic/radar imaging. LUD performs the LU decomposition by using upper and lower triangular products of a matrix. BFS is a breadth-first search graph traversal algorithm that, from the root, searches neighbor nodes before moving to the next level of tree. The sequential version of these three Rodinia applications written in Julia were obtained from [67]. We parallelized these three applications based on their OpenMP counterparts and make them available at GitHub for the Julia Community. [1]

For all the benchmarks, we used three different sizes of input data, named *small*, *medium* and *large*, which are specified in Table 1. The balanced and unbalanced application inputs correspond to the number of iterations of the outer loop N and the inner loop k. The linked list application size is determined by the number of nodes of the list. LUD inputs are matrix sizes, BFS inputs are graph files provided by Rodinia benchmark set and SRAD_v2 inputs are matrix sizes, position of the speckle, lambda value and number of iterations.

Table 1 – Input sizes used in the data and task parallelism benchmarks

Benchmarks	Input Size		
	small	*medium*	*large*
Balanced	N=10, k=500	N=100, k=500	N=500, k=500
Unbalanced	N=10, k=500	N=100, k=500	N=500, k=500
Linked List	1000	10000	100000
LUD	512x512	1024x1024	2048x2048
BFS	graph4096	graph65536	graph1M
SRAD_v2	128 128 0 31 0 31 0.5 2	512 512 0 67 0 67 0.5 2	2048 2048 0 127 0 127 0.5 2

Figure 33 – Balanced Synthetic Application to evaluate loop scheduling performance

```
1        for i = 1:N
2            for j = 1:k
3                ... (performs some computation)
4            end
5        end
```

[1] at https://github.com/dianabarros/rodinia

Figure 34 – Unbalanced Synthetic Application to evaluate loop scheduling performance

```
        for i = 1:N
            for j = 1:k/i
                ... (performs some computation)
            end
        end
```

Figure 35 – Linked list implementation to evaluate loop scheduling performance

```
        node = linked_list.head
        for i in 1:length(linked_list)
             node.value = c + 10*i
             node = node.next
        end
        node = linked_list.head
        while node.next != nothing
             fibonacci(node.value)
             node = node.next
        end
```

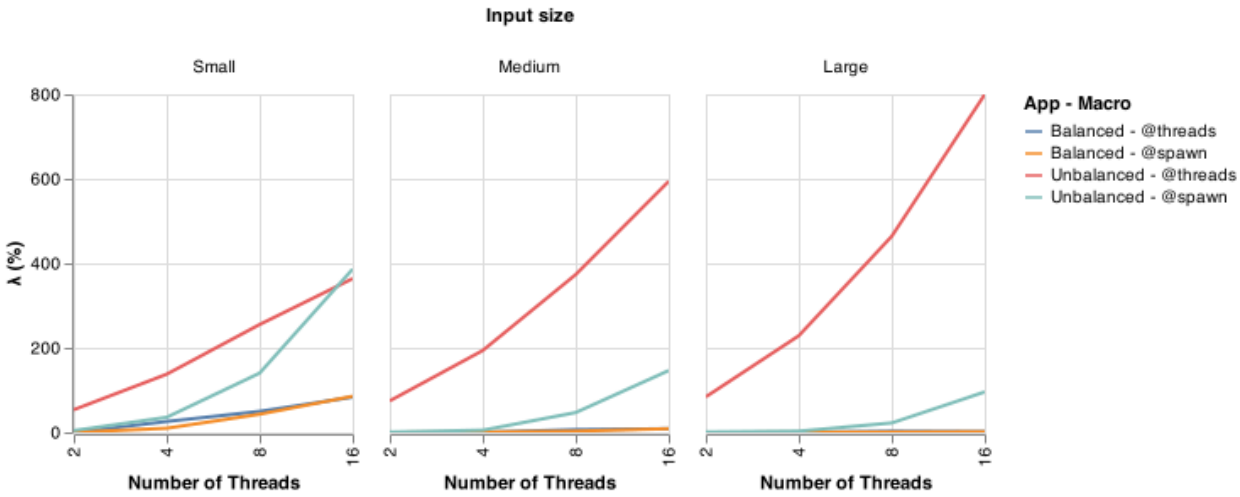
4.3 **Results**

The following subsections contain the results obtained for each experiment and the observations made.

4.3.1 Synthetic Kernels

The synthetic kernels test extreme scenarios of load balancing for `@spawn` and `@threads` macros. Figures 36 and 37 show respectively the percentage of load imbalance and speedup obtained for the balanced and unbalanced kernels for different input sizes and number of threads.

Figure 36 – Load imbalance (λ) of balanced and unbalanced kernels with macros `@threads` and `@spawn`.

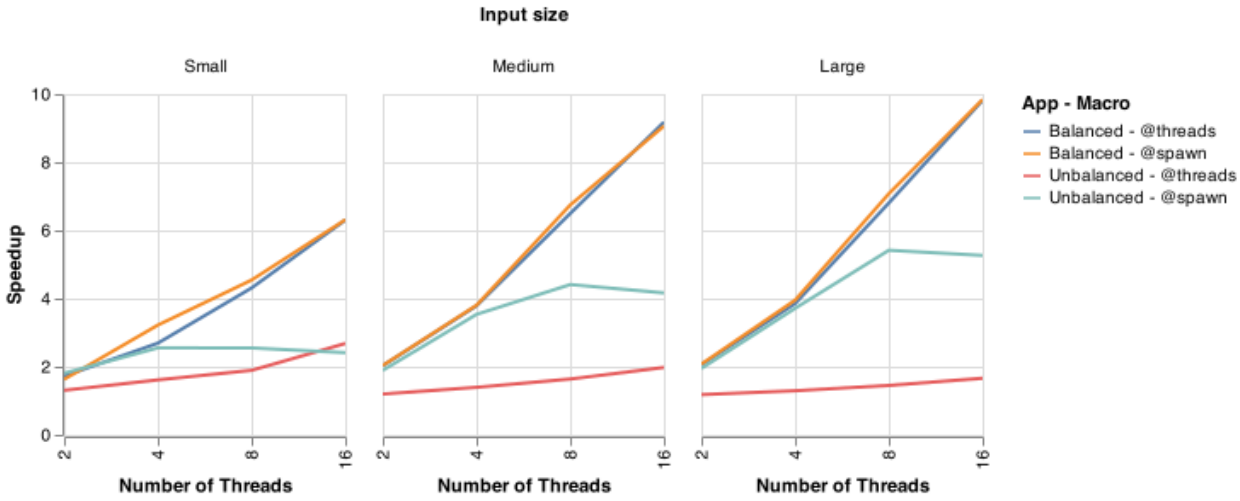


We can observe in Figure 36 that `@threads` and `@spawn` produced similar load bal-

ancing results for the balanced kernel. Since this kernel is regular and the iterations are identical, both macros are able to perform a good distribution, statically or dynamically. When the input size is small the load imbalance is high since the time to create the threads and tasks has great influence in the execution time of each thread. For the larger input sizes, the load imbalance is negligible, since the amount of work to be distributed is huge, and it is unlikely that a small difference in the computation of some iterations would affect the result.

With the unbalanced kernel, we observe a more pronounced load imbalance percentage and a considerable difference in the imbalance produced by `@spawn` and `@threads`. For this kernel, the dynamic nature of the `@spawn` loop scheduling with depth first scheduling is able to keep the load balance among the threads. The static scheduling of `@threads`, on the other hand, produced significant load imbalance in the application execution. Using `@threads`, the first thread will receive the first iterations that hold the majority of the computation. This thread execution will take a longer time to compute and produce the imbalance. We can also notice that, for larger inputs, the use of `@spawn` can help amortize the increase of load imbalance as the number of threads increases, while when using `@threads` the load imbalance increases much faster with the increase in the number of threads.

Figure 37 – Speedup of balanced and unbalanced kernels with with macros `@threads` and `@spawn`.



In the speedups result of Figure 37, we can observe that, for the balanced kernel, both macros obtain similar speedup. For the unbalanced kernel, the speedups are smaller, but the use of the macro `@spawn` provides a significant impact in the performance of the kernel.

Figures 38 and 39 show the load imbalance and speedup results, respectively, for the Linked List kernel with `@threads` and `@spawn`. In the Linked List kernel, for each node of the list there is an offset increase of 10 on the values of the nodes. Therefore, as the list is traversed, the amount of computation is considerably increased. For the small input size, the usage of `@spawn` showed more imbalance than the macro `@threads`. For the medium and large input sizes, however, the results are different, the use of `@spawn` provided great reductions in the load unbalancing. The load unbalancing reflected on the speedup as seen in Figure 39. Even though the usage of `@threads` had no impact in the load imbalance, we could notice speedup gains with increase of input size and number

of threads. However, the decrease in load imbalance with `@spawn` allowed an increase in performance that surpassed the ones obtained with `@threads`.

Figure 38 – Load imbalance (λ) of Linked List kernel of Julia implementations of data parallelism (`@threads`) and task parallelism (`@spawn`).

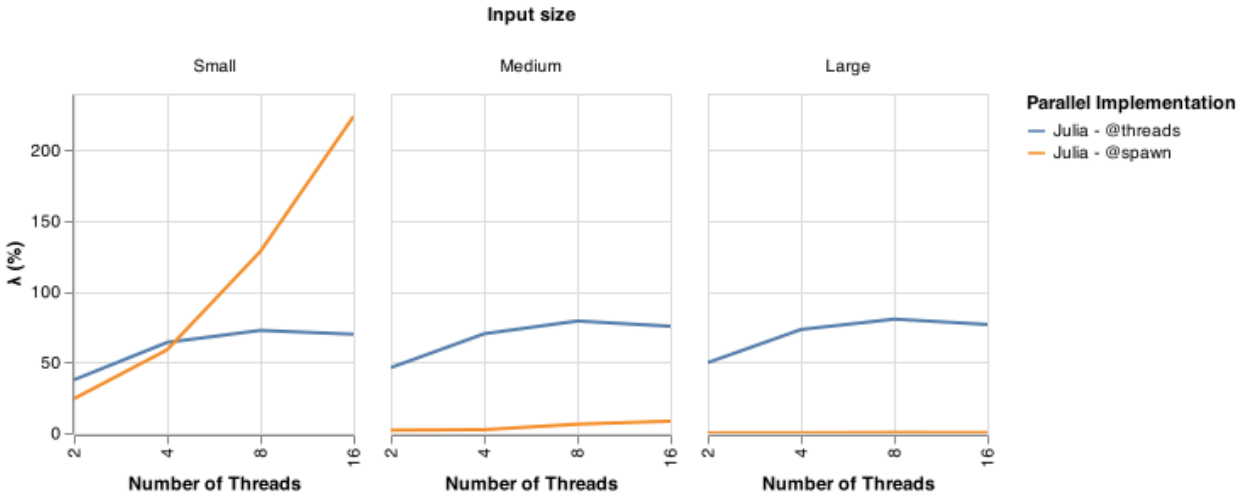
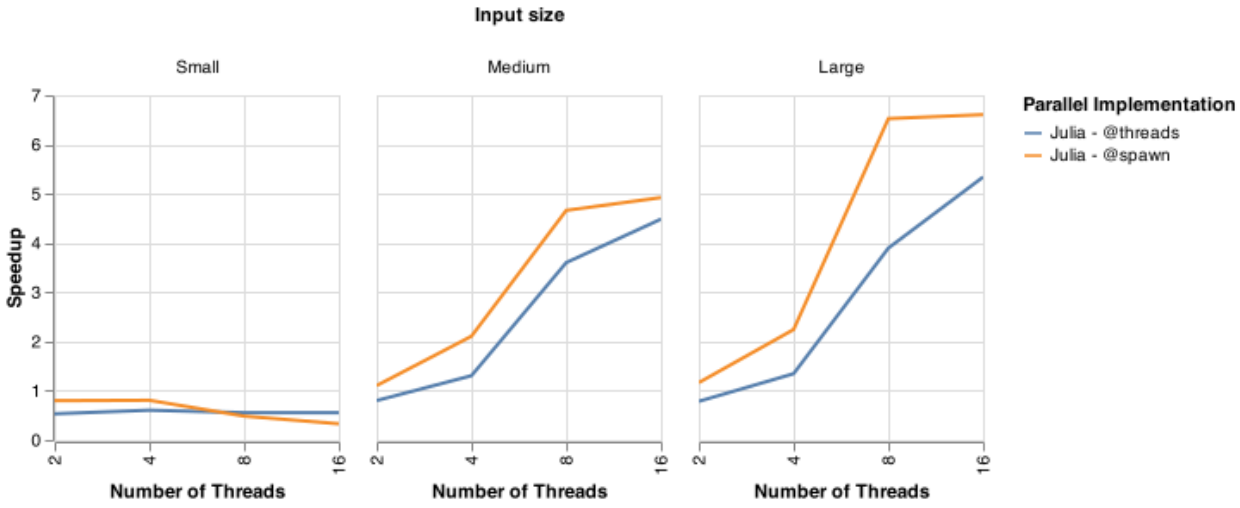


Figure 39 – Speedups of Linked List kernel of Julia implementations of data parallelism (`@threads`) and task parallelism (`@spawn`).



4.3.2 Benchmark Applications

SRAD_v2: This application has two different parallel loops and we analyse the one that produced more load imbalance. Figures 40 and 41 show the percentage of load imbalance and the speedup for the SRAD_v2 application, when the input sizes and the number of threads vary. In Figure 40, we observe that when `@threads` is used the imbalance is lower and shows a tendency to remain constant with the increase in the number of threads. For the large input, the load imbalance of `@spawn` reaches a peak at 8 threads and then decreases. What happens is that when the matrices are small, the number of spawned tasks is small. We observed that some threads finish rapidly without

any task assigned to them, by the time the the task is allocated and scheduled, most or all of the work has been done by the others. Figure 41 shows that the speedup of SRAD_v2 using the two macros are similar, with `@threads` obtaining more speedup as the difference in amount of speedup between them increases. This is due to the depth first scheduling overhead which is high for the larger inputs.

Figure 40 – Load imbalance (λ) of SRAD_v2 with macros `@threads` and `@spawn`.

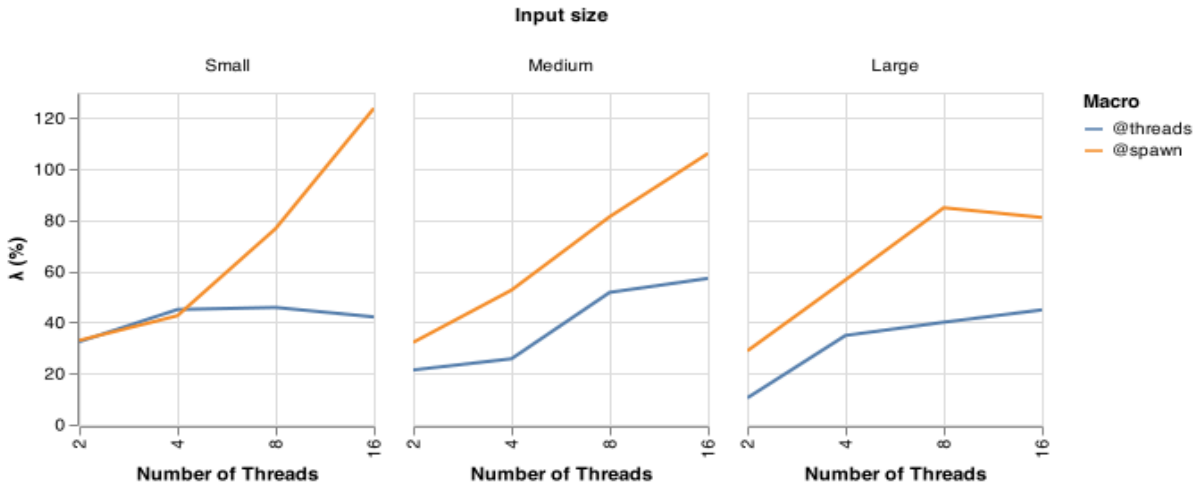
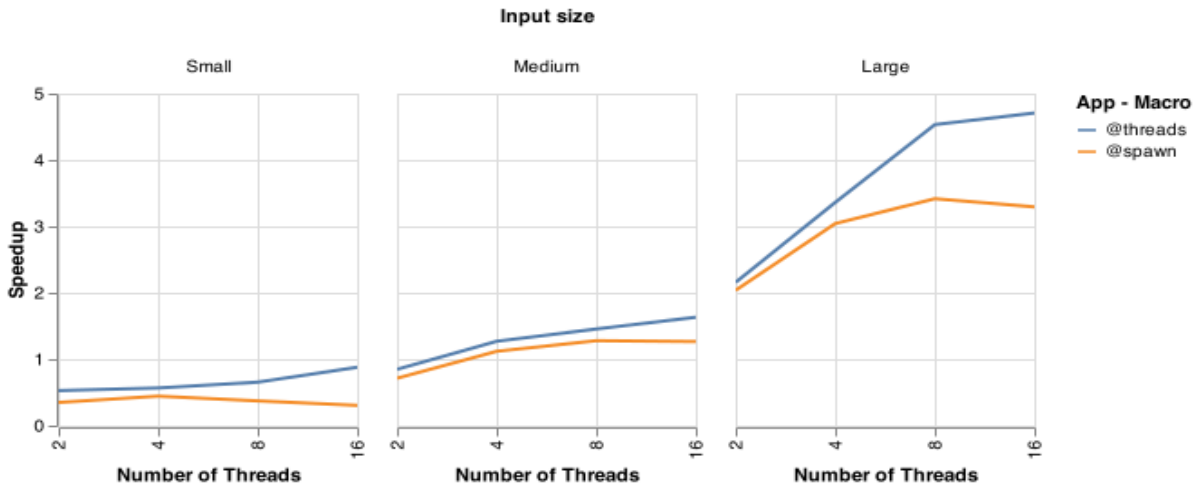


Figure 41 – Speedups of SRAD_v2 with macros `@threads` and `@spawn`.



LUD: This application has two independent parallel loops with dependency to an outer loop. As the outer loop advances, the number of iterations of the inner loops reduces. We analyse the loop that impacted more on the execution time of the application. In Figures 40 and 43, we show the percentage of load imbalance and speedups for the LUD application, when the input sizes vary. The load imbalance results show that the `@spawn` macro yields a worse load balancing for this application. This occurs because as the outer loop advances and the amount of work on the inner loops reduces, the depth first order can produce situations where a thread remains without work, even though when the input size and the number of thread increase, it does a better load balancing than with smaller inputs. Figure 43 shows that the speedup obtained by the two macros follows the load imbalance results, with `@threads` being the macro with best speedup with larger inputs.

Figure 42 – Load imbalance (λ) of LUD with macros `@threads` and `@spawn`.

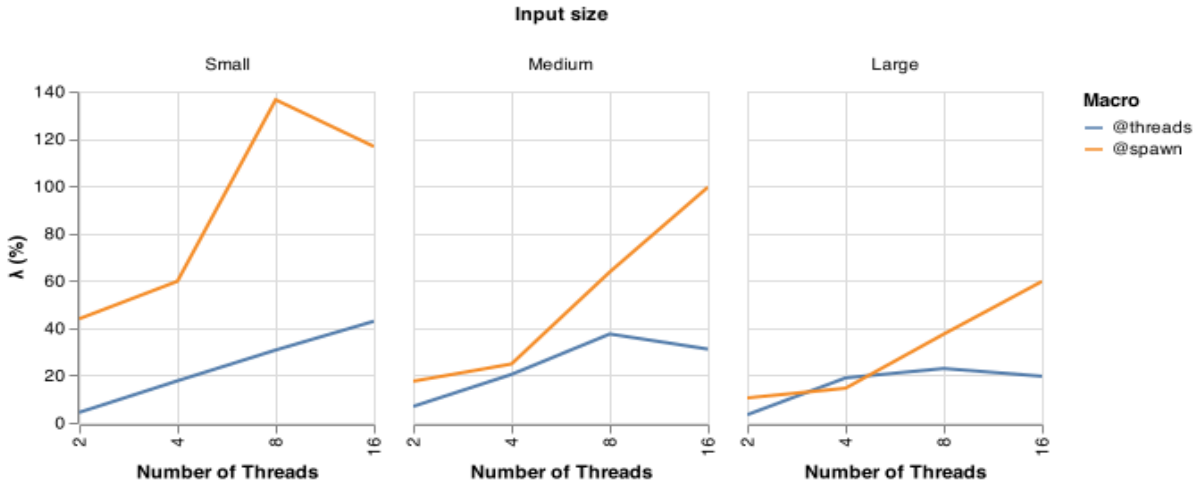
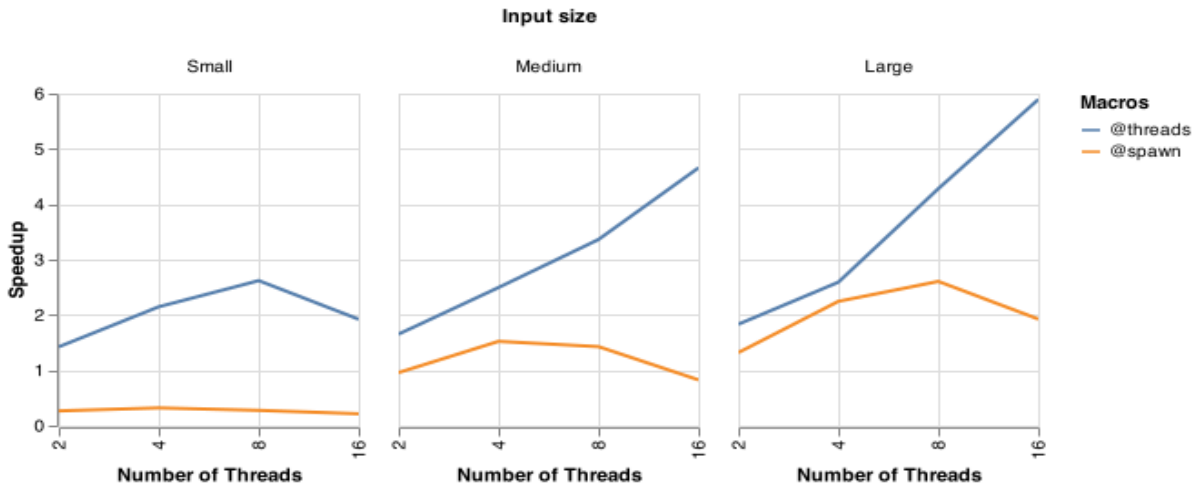


Figure 43 – Speedups of LUD with macros `@threads` and `@spawn`.



BFS: There is one parallel loop in this application with dependency to an outer loop. The parallel loop traverses all the nodes in the graph, it only performs computation in the node if the node is marked. Although the loop iterates over all the nodes in the graph, the amount of work that each iteration handles might be completely different. In addition, since the graph nodes are not contiguous in memory, the iteration might have high cache miss rates. Figures 44 and 45 show the load imbalance and the speedup of BFS using `@threads` and `@spawn` for different input sizes and number of threads. We can observe that `@spawn` generates worse load imbalance results and it oscillates as the number of threads increases. The irregular nature of BFS can lead to situations where the task assignment occurs, but no work is performed in the task. We observe in Figure 45 that the speedups of `@spawn` are very small, in fact, their performance was worse than the sequential version. This is mainly due to the overhead of the creating a number of tasks that do not perform any work.

Figure 44 – Load imbalance (λ) of BFS with macros `@threads` and `@spawn`.

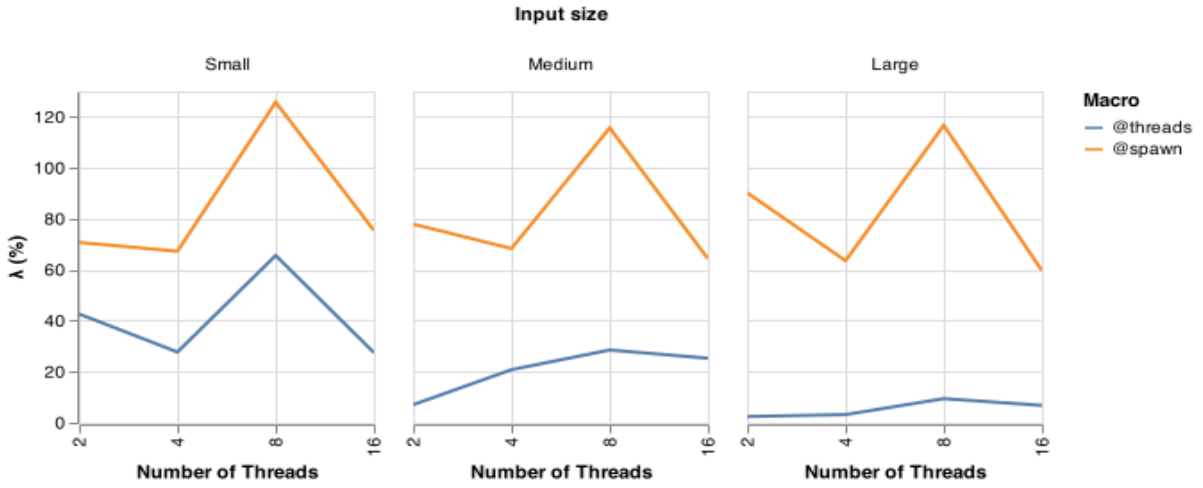
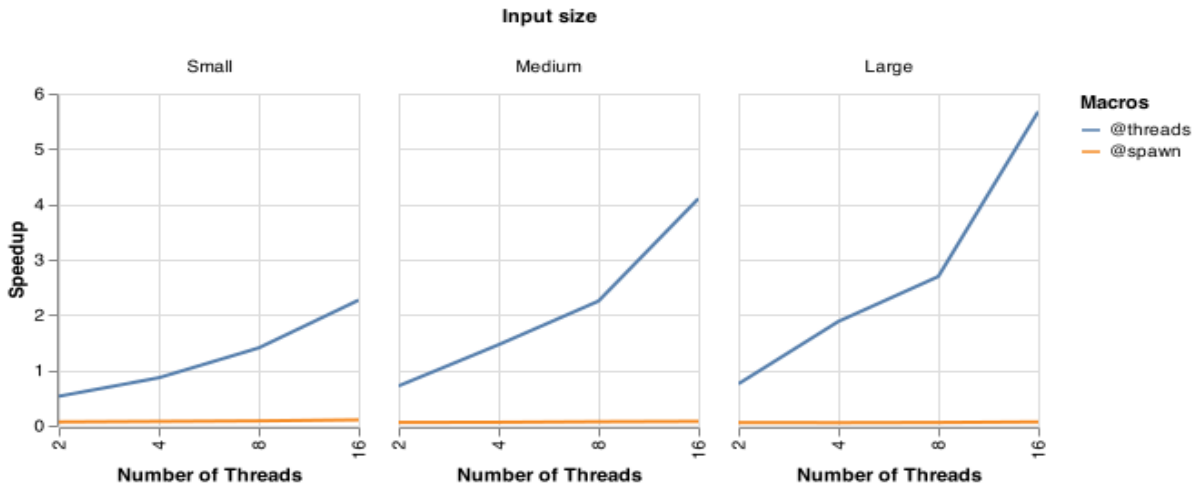


Figure 45 – Speedups of BFS with macros `@threads` and `@spawn`.



4.4 **Discussion**

Load imbalance is one of the main sources of performance degradation in parallel applications. Our comparison of a static and dynamic loop level scheduling mechanisms in Julia shows that there is no best solution for all the cases and that the load balancing is dependent on the input size. The static scheduling speedup can increase linearly along the number of threads for applications where the iterations have a similar amount of work, whereas when the iterations have different amount of work the maximum increase along the number of threads was of around 20% and it was observed on the small input, while also being the worse performance obtained compared to all the other implementations on any input size. The dynamic scheduling, on the other hand, was able to balance a very unbalanced loop on our synthetic applications, reducing about 4 times the load imbalance from the small input size to the large input size and allowing an increase of speedup about 2 times of the small input size on the large input size.

However, the nature of the own code implementation can lead to a significant runtime overhead generation from the depth first scheduling, as the Rodinia Benchmarks we analyzed. When the iterations have a small amount of work, the overhead of scheduling and

allocating a task can create scenarios where some threads do not receive tasks to compute. Not only can this overhead impact the speedup, but also produce a parallel version worse than its sequential one, as seen on the BFS benchmark.

Even though the macro `@threads` has a scheduling parameter in the latest version of Julia, only the static option is available. This is equivalent of when no parameter is set. We expect that the inclusion of this parameter in `@threads` represents an evidence that other scheduling strategies will be implemented in this macro in the future.

In addition, it is not possible to change the granularity of the static distribution and also it is not possible to configure the initial task queue for each thread in the dynamic scheduling. Creating a initial task queue composed by a predefined number of threads would prevent the situations of threads without any task assigned to them.

5 EVALUATING MULTITHREADING LOOP SCHEDULERS

In this chapter, we further evaluate the multithreading mechanisms presented in Julia. We provide a detailed performance analysis of the internal loop scheduling strategies on small and large scale environments.

5.1 Experimental settings

The experiments were executed in two different environments due to the access availability of small and large scale environments. The first set of experiments was performed on an AMD Ryzen 7 2700 Eight-Core Processor with 16 threads (2 threads per core with Simultaneous Multi-Threading) and 128 GB of RAM. The second set of experiments was performed on one node of a cluster from the Grid5000 [68]. The node is equipped with two Intel Xeon Gold 6130 @ 2.10 GHz with a total of 32 cores/64 threads per node and 192 GB RAM. We ran our experiments using Julia version 1.7 and GCC compiler version 9.4. For both of them we did not set any compilation flags, leaving them with their default values. For the comparisons we used the speedup calculated as the sequential execution time over the parallel execution time of the version in question, it being either Julia or C.

We used the macro `@timed` to measure the execution time in Julia because of the practicality it offers when isolating the measurement on specific blocks of codes. With the C implementations we measured the execution time with the function `gettimeofday` on sequential versions and `omp_get_wtime` on OpenMP versions.

The evaluation of Julia's internal loop scheduling strategies was done by comparing the default loop scheduling strategy of the macro `@threads` against different loop schedulers provided by the package FLoops.jl [12] executors: `ThreadedEx`, `DepthFirstEx`, `WorkStealingEx`, `TaskPoolEx` and `NondeterministicEx` as seen in Section 3.4, where the basesizes were the $input\_size/number\_of\_threads$. We also compared Julia loop scheduling against C + OpenMP loop scheduling, where we fixed the scheduling strategy as the default scheduler: static round-robin with the chunks being the number of the loop iterations divided by the number of threads.

5.2 Benchmarks

In order to test the different loop scheduling strategies of Julia, we used a different set applications. We tested with real-world applications, but we chose applications where the load imbalance is more pronounced than the load imbalance shown in Rodinia applications. The applications were: Mutually Friendly Numbers, Password Cracking using Brute Force, and Transitive Closure of a Graph.

In the Mutually Friendly Numbers application, two numbers `a` and `b` are said to be mutually friendly if the ratio between the sum of every divisor of `a` and `a` is equal to the ratio between the sum of every divisor of `b` and `b`. Our Mutually Friendly Numbers application calculates all the mutually friendly numbers between a range of numbers

specified in the input. The Password Cracking application using Brute Force takes in a string and, with a previously defined set of possible characters, guess character per character of the string until it finds the entire string. The longer the string the more work it will perform. Finally, the Transitive Closure problem relies on determining if for every pair of nodes in a graph there is a directed path. Our implementation of the problem follows the Warshall algorithm [69].

The input sizes of these benchmarks are shown in Table 2. The input size of the Mutually Friendly Numbers application corresponds to the range of numbers that will have their mutual friendly number calculated. The input size of the Password Cracking application indicates the length of the password string input. The input size of the Transitive Closure problem is the amount of nodes in the graph.

Table 2 – Input sizes used in the loop scheduling benchmarks

Benchmarks	Input Size		
	small	*medium*	*large*
Mutually Friendly Numbers	0 - 50000	0 - 200000	0 - 350000
Password Cracking/Brute Force	4	5	6
Transitive Closure	1280	2560	5120

5.3 **Evaluating the Loop Scheduling Strategies**

The first set of experiments evaluates the different strategies for scheduling the loop iterations among the threads in the AMD Ryzen single processor environment. We compare the loop scheduling strategies provided by default in Julia with the different executors of the package FLoops.jl, and the default loop scheduling strategies of C + OpenMP.

5.3.1 Mutually Friendly Numbers

The parallel loop in this application iterates between the range of numbers that was passed as input. For every iteration, it calculates the sum of the factors of the *i-th* number divided by *i*. It means that the work load depends on the size of the number of the current iteration and its characteristics that determine the amount of factors that it has. Given this unbalanced nature, we would expect either the `DepthFirstEx` or `WorkStealingEx` executors to have better performance.

Figure 46 shows the speedup of the FLoops.jl executors and we can notice that they behave similarly. For every input size observed, the speedup increased faster until 8 threads. `DepthFirstEx` and `ThreadedEx` executors had similar speedups, but we chose `ThreadedEx` for the next experiments because it had higher speedup than `DepthFirstEx` on small input sizes. The results of `ThreadedEx` are due to the divide-and-conquer strategy adopted by the scheduler where combining tasks with different workload could help balancing the loop.

The comparison with the C implementation is shown in Figure 47. This figure shows the speedups of Mutually Friendly Numbers for C + OpenMP, Julia with the macro `@threads`, and Julia with FLoops.jl using `ThreadedEx`. The speedup varies consistently with the increase in the input size, growing as the number of threads increases. The different implementations show similar speedup results, but in most of the cases, the implementation with FLoops managed to get slightly better speedup, specially when 4

and 8 threads are used. Figure 48 compares the execution time of the sequential version of this application in C and Julia. This plot shows that there was no significant difference between the sequential execution time in C and in Julia. This means that the speedups were not heightened by a slow sequential code.

Figure 46 – Mutually Friendly Numbers speedup with FLoops.jl executors.

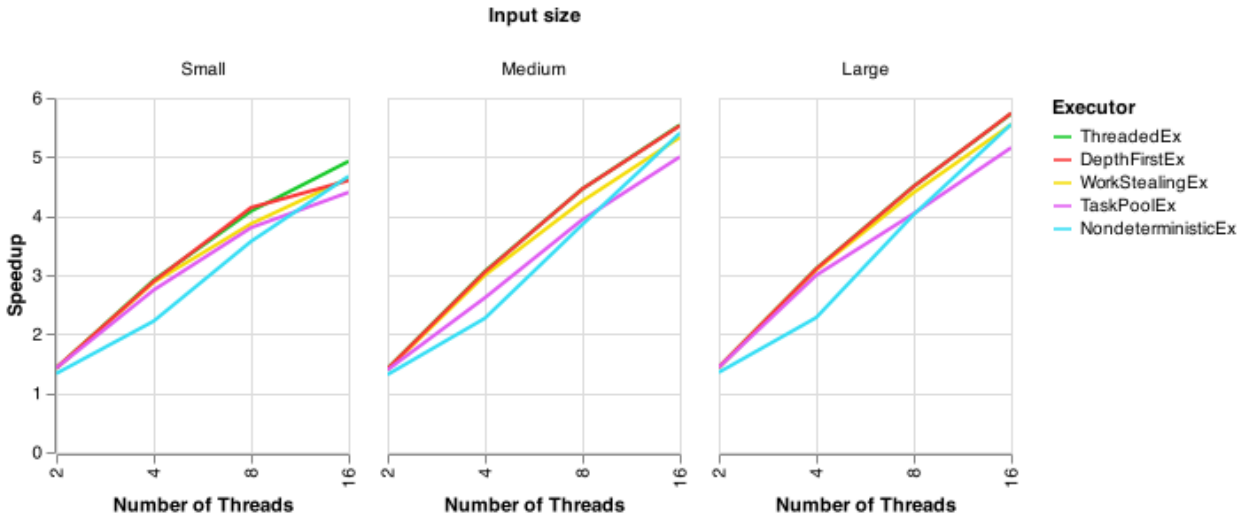
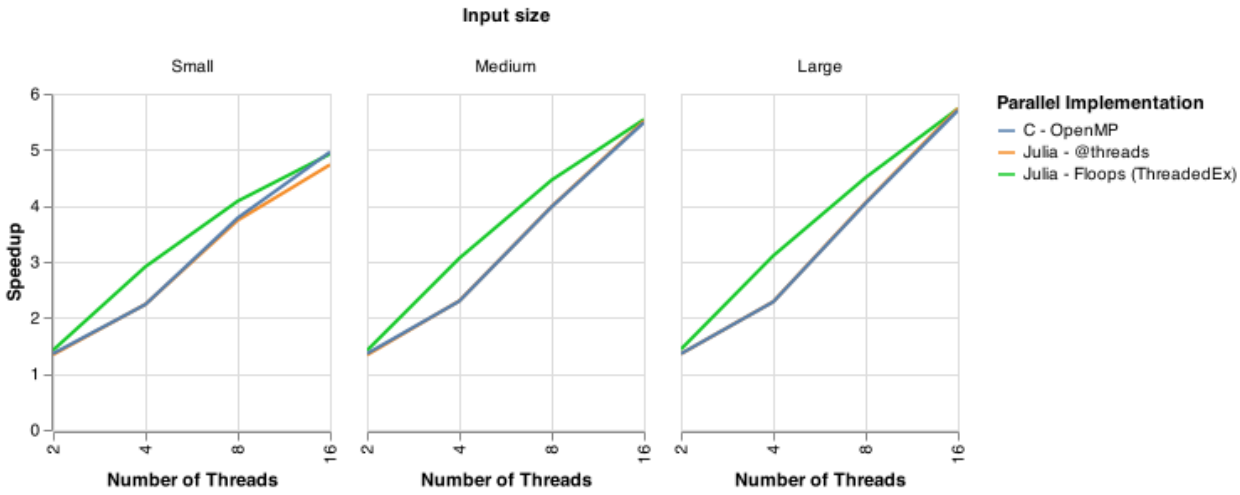


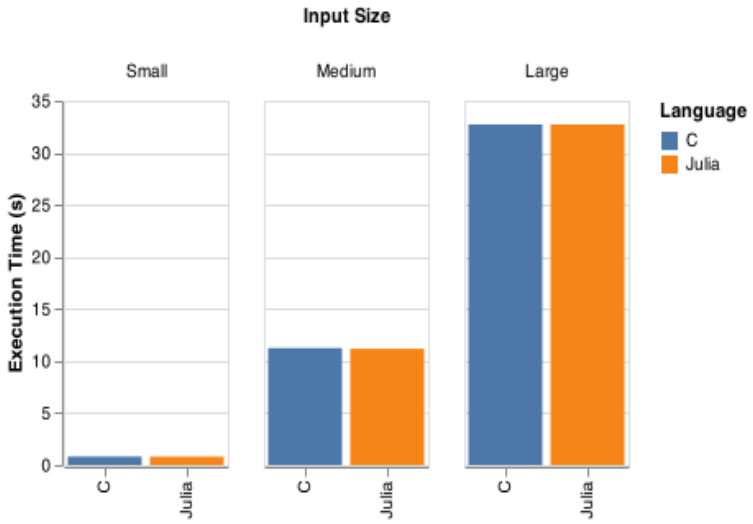
Figure 47 – Mutually Friendly Numbers speedup.



5.3.2 Password Cracking with Brute Force

This application goes through every possible combination of characters and compares it to the password to check if it matches. It starts with a loop in which the number of iterations is the length of the smallest possible password and continues to another loop in which the number of iterations is increased by one to test another length of password. The longer the password string is, the greater is the amount of work. We need a flag to exit the loop when the password is found. In the parallel versions, this flag needs to be protected by a lock to avoid race conditions. Since the characters are tested in a specific order, the password chosen has influence in the amount of computation performed. Since

Figure 48 – Mutually Friendly Numbers sequential execution time in seconds.



the loop is exited when the password is found, we expect the `DepthFirstEx` executor to obtain better speedups.

Figure 49 shows the speedup of the FLoops.jl executors. We can observe that the speedups are more pronounced with a tendency of grow for the small input size. For the large input size, we can observe that the results for the different executors oscillates. We chose the `DepthFirstEx` executor for the next experiments since it showed the highest speedups in average. Although we expected that `DepthFirstEx` would return better results, `ThreadedEx` and `TaskPoolEx` obtained close speedup results.

Figure 50 shows the comparison between C + OpenMP and Julia parallel implementations. For small and medium input sizes, we could not see an increase in speedup along with the increase of number of threads or imput size, even though we Julia implementations had higher speedup than C + OpenMP. For the large input size, a similar scenario occurs for up to 8 threads. For more than 8 threads the speedup of the C implementation escalates very quickly, reaching a speedup of over 20. During the execution of our experiments we could observed that, for larger inputs, the Julia implementation made large amounts of memory usage, resulting in a slower execution time. Figure 51 shows the execution time of the sequential versions. We can observe that the Julia version was around $4x$ slower than the C version on large input size. This behavior seen on Julia's version prevented the performance gains in its parallel implementation, while on C's version the amount of work load that starts taking advantage of multithreading parallelism is seen on large input size.

5.3.3 Transitive Closure

The Warshall algorithm that is used in this implementation traverses the adjacency matrix of the graph and accesses it both by row and column. It is done by three nested loops, where the second loop was parallelized. For this loop, the amount of work inside each iteration is determined by an if condition. Since these iterations might not obtain the same amount of work, we expect that either `DepthFirstEx` or `WorkStealingEx` present better speedups.

We can see in Figure 52 how the executors scale with the input size. We observe that the speedups are small with little increase. Among these executors, `NondeterministicEx`

Figure 49 – Password Cracking with Brute Force speedup with FLoops.jl executors.

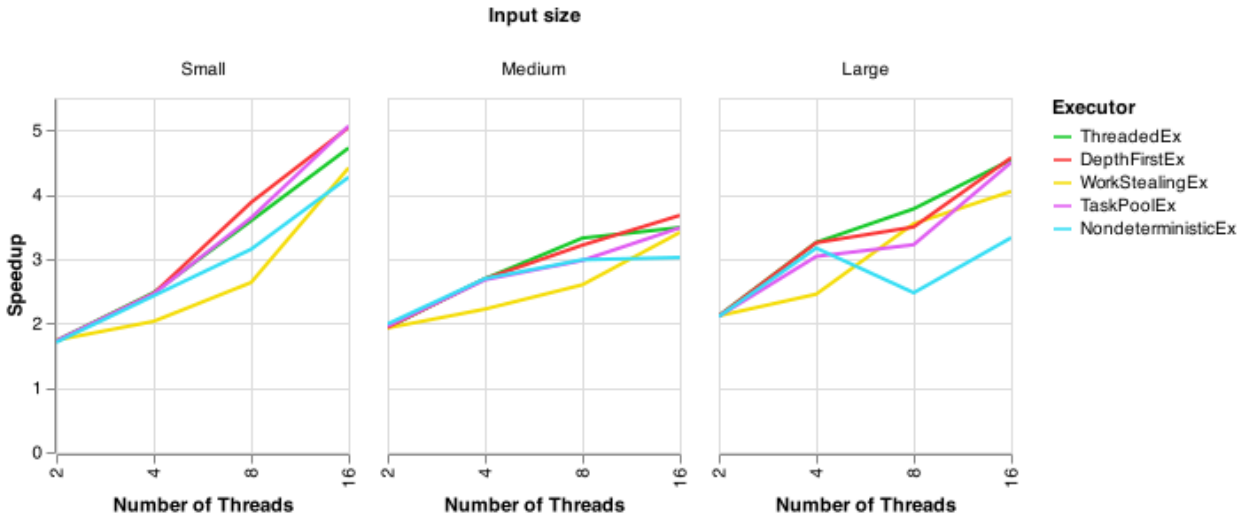
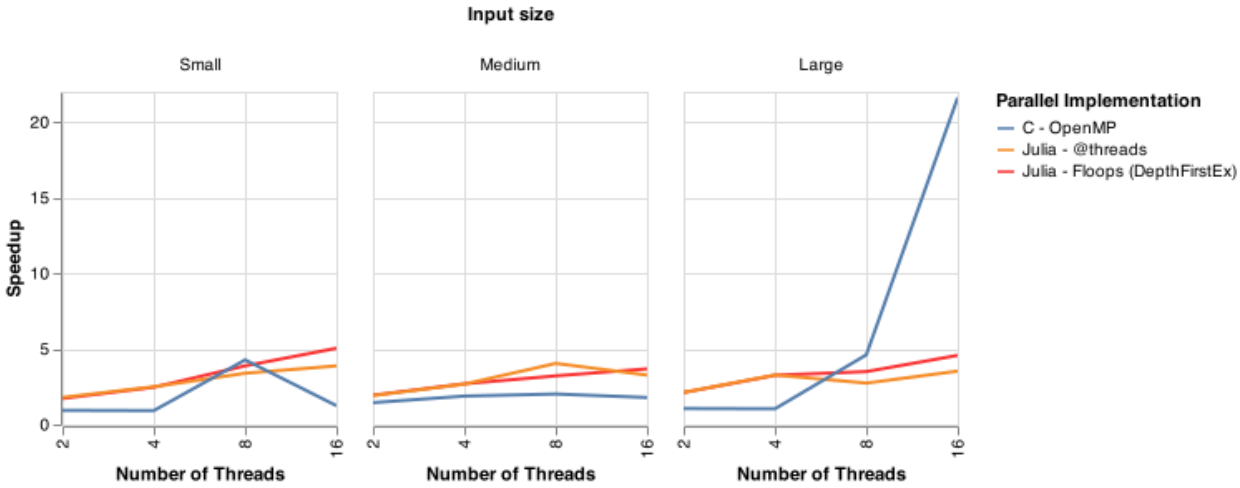


Figure 50 – Password Cracking with Brute Force speedup.



is the one that scales the least, `WorkStealingEx` shows more oscillation, `TaskPoolEx` scales until 8 threads but decreases after that, and `ThreadedEx` and `DepthFirstEx` obtained the best results. We chose `DepthFirstEx` for the next experiment because it provides better results on large input size.

The comparison of C and Julia implementations, depicted in Figure 53, shows that Julia versions do not seem to benefit enough from parallelism when compared to the C + OpenMP version, even though Julia's sequential version optimization could reduce its execution time as seen in Figure 54.

Figure 51 – Password Cracking with Brute Force sequential execution time in seconds.

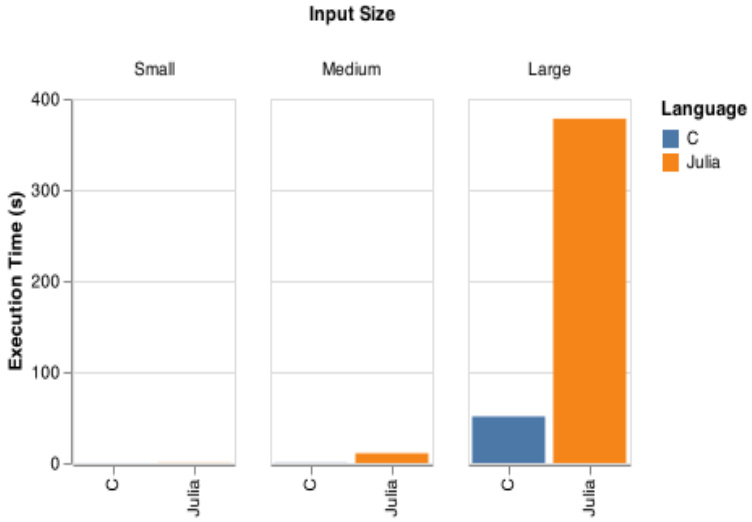


Figure 52 – Transitive Closure speedup with FLoops.jl executors.

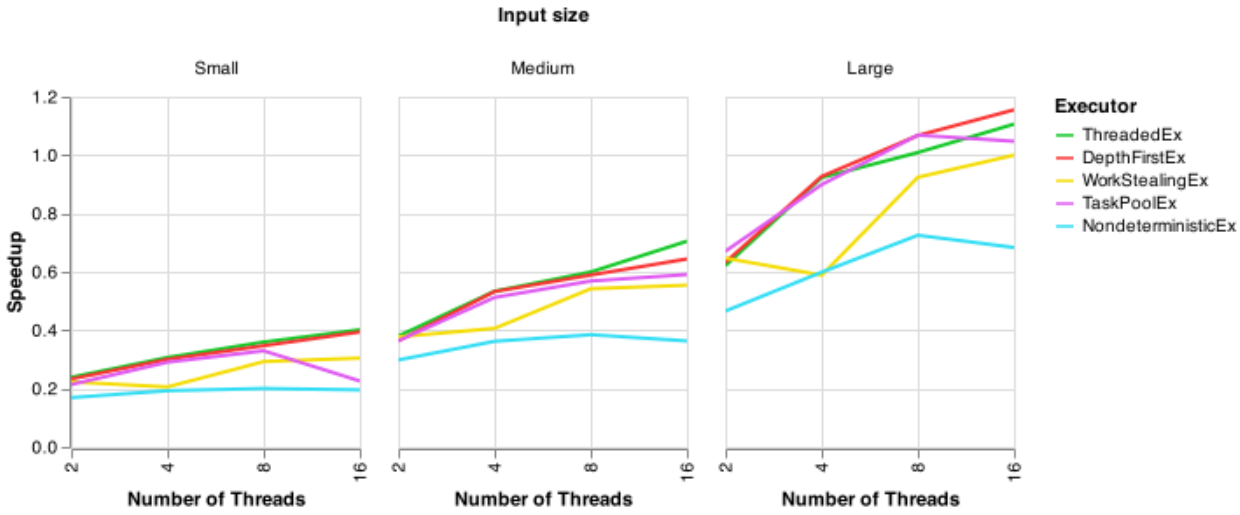


Figure 53 – Transitive Closure speedup.

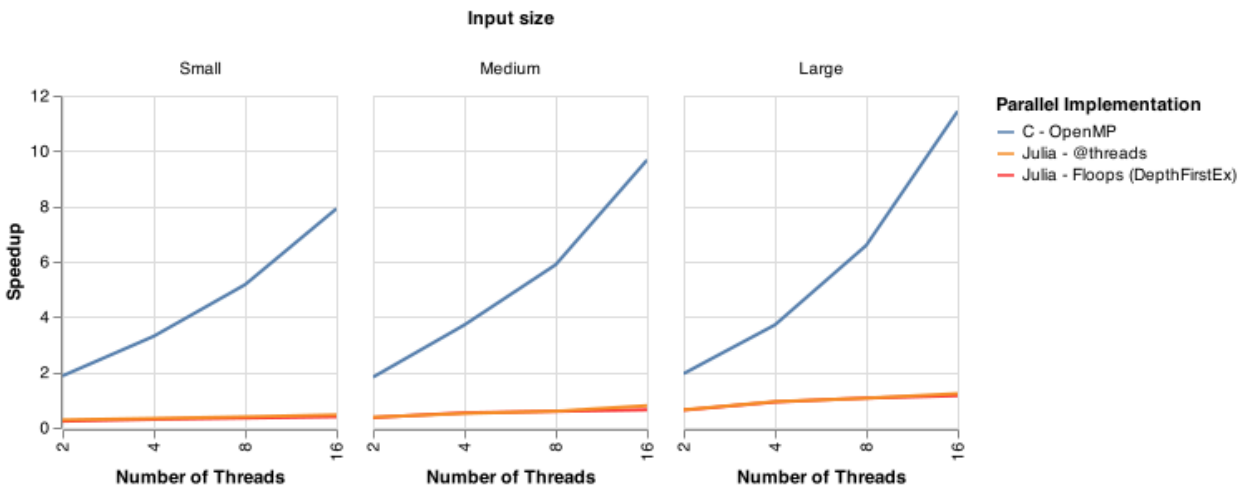
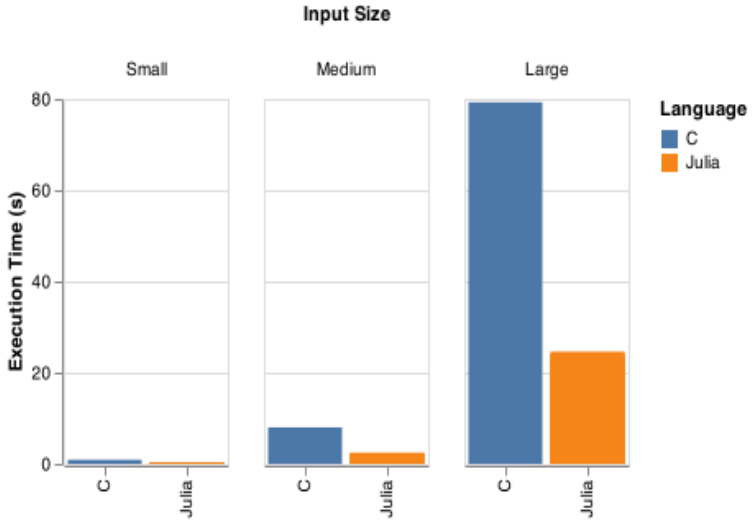


Figure 54 – Transitive Closure sequential execution time in seconds.



5.4 Evaluating the Scalability of the Loop Scheduling

In our second set of experiments, we evaluate the scalability of the Julia loop scheduling strategies. The experiments were performed on the 64-threads node of the University of Luxembourg cluster.

5.4.1 Mutually Friendly Numbers

Figure 55 shows the speedups of the Mutually Friendly Numbers application using the different executors of the FLoops.jl when the number threads ranges from 2 to 64. We can observe in this figure that the speedups show a growing trend, except for the `TaskPoolEx` executor with the small input size. Since the curves of the `ThreadedEx` executor show higher speedups, we will use it for the further comparisons.

Figure 55 – Mutually Friendly Numbers speedup of FLoops.jl executors on an environment ranging from 2 to 64 threads.

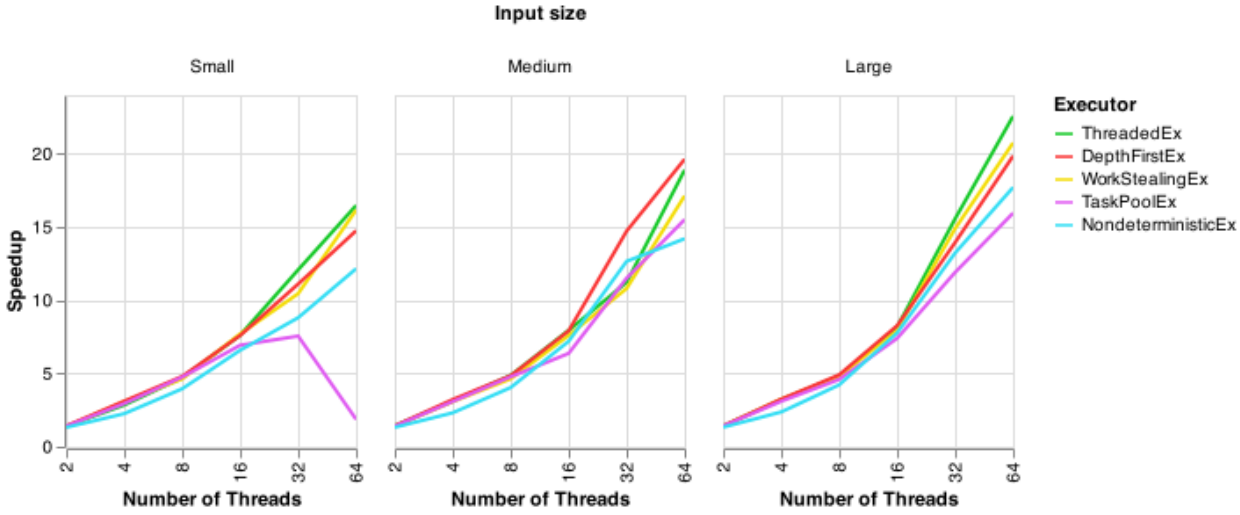


Figure 56 shows the speedups for the `@threads` and the the FLoops.jl with `ThreadedEx` executor. We can observe that the `@threads` only scales better on the small input size, even though the curves are very close.

5.4.2 Password Cracking with Brute Force

Figures 57, 58 and 59 show the speedups of the FLoops.jl executors on the Brute Force application. Since the results oscillated a lot with the different input sizes, we split them in separate plots. We can see in Figure 57 that `WorkStealingEx`, `DepthFirstEx` and `NondeterministicEx` increase their speedup with the increase on number of threads. `WorkStealingEx` is the one that scales better. The behavior of `ThreadedEx` and `TaskPoolEx` oscillates as the number of threads increases. However, when we look at Figures 58 and 59, we can notice a drop in the speedups, which range around 2 and 4 for medium and large input sizes. For these input sizes, the speedups oscillates a lot with the increase in the number of threads.

Based on these plots observations, we chose the executor `WorkStealingEx` to compare with the implementation with `@threads`, since it showed good speedups for higher number of threads, more precisely with 32 threads. Figure 60 shows for the small input size, the

Figure 56 – Mutually Friendly Numbers speedup of `@threads` and FLoops.jl with ThreadedEx executor on an environment ranging from 2 to 64 threads.

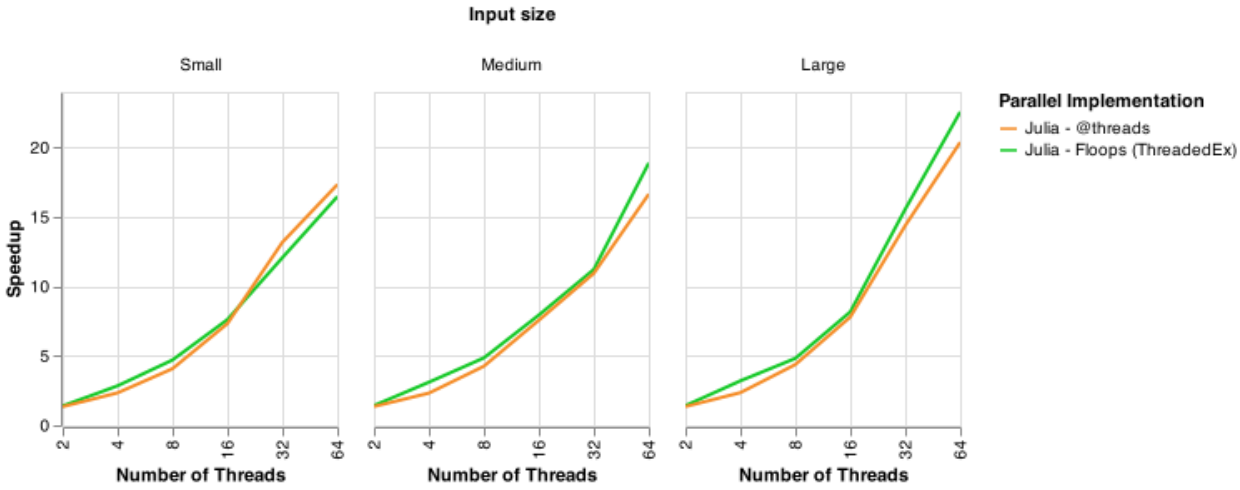
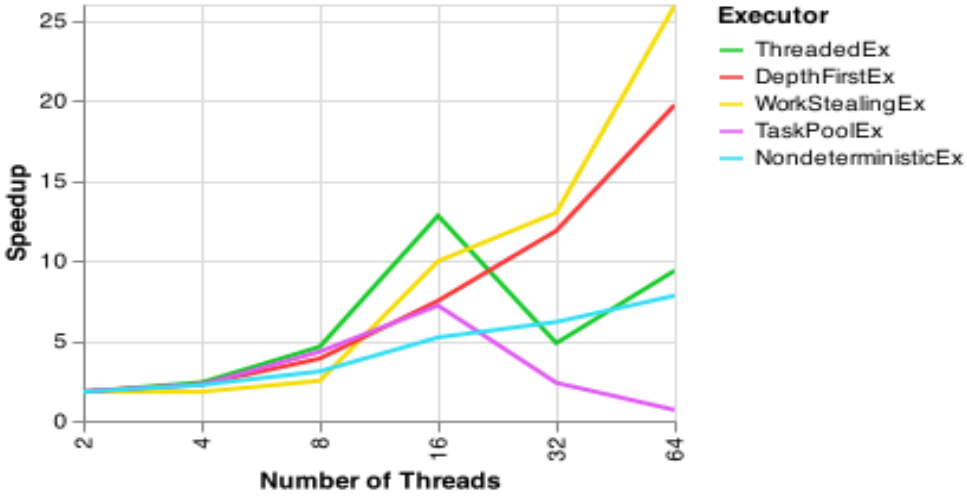


Figure 57 – Password Cracking with Brute Force speedup with small input size of FLoops.jl executors on an environment ranging from 2 to 64 threads.



speedups of `WorkStealingEx` and `@threads`. We can observe in this figure that the speedup of `@threads` grows close to FLoops.jl speedups until 32 threads, where `@threads` shows a decrease in the speedup. Figures 61 and 62 show the speedups for the medium and large input sizes respectively. We can observe in these figures that the speedups are small with an oscillation behavior. For the medium input size, the difference between them is that the peak for `@threads` is at 8 threads and is higher than the peak for `WorkStealingEx` at 32 threads. For the large input size, the peak of `WorkStealingEx` is at 8 threads and is higher than the peak of `@threads` at 4 threads. Both implementations show similar speedups from 32 to 64 threads.

As mentioned in Subsection 5.3.2, this application includes multiple parallel loops that are executed one after the other for guessing passwords of different string lengths. The application exits when the right password is found. As the first loops present smaller number of iterations, they provide less work to be distributed among the threads. When we test this application with a greater number of threads, some threads may become idle in these first loops computation, which have direct impact on the speedup achieved.

Figure 58 – Password Cracking with Brute Force speedup with medium input size of FLoops.jl executors on an environment ranging from 2 to 64 threads.

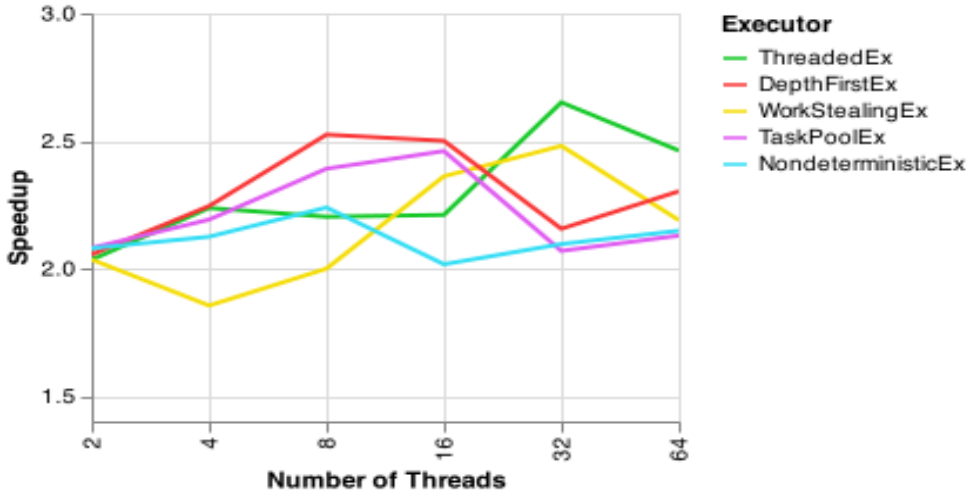
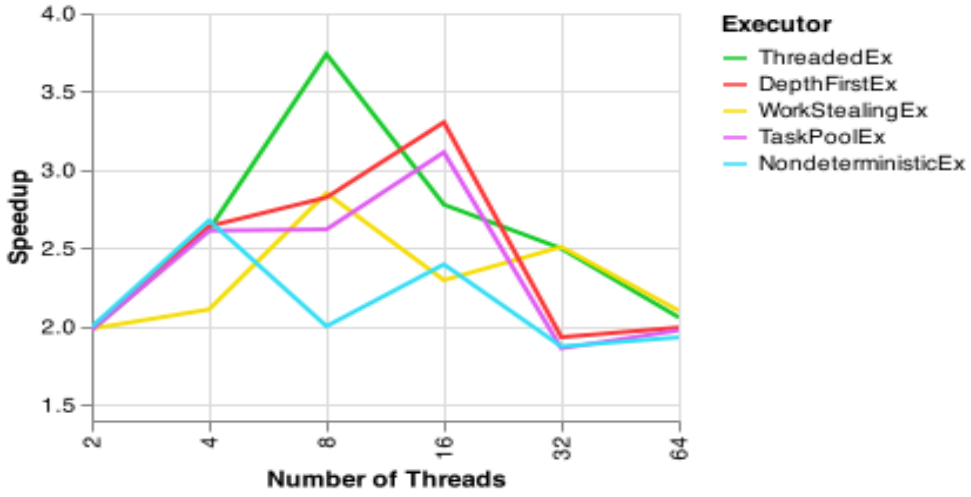


Figure 59 – Password Cracking with Brute Force speedup with large input size of FLoops.jl executors on an environment ranging from 2 to 64 threads.



5.4.3 Transitive Closure Problem

Figure 63 shows the speedups of Transitive Closure for the FLoops.jl executors. We can observe a decrease trend in the speedup values as the number of threads increase, though there is some oscillation in these values. From these results, we chose `DepthFirstEx` to compare with `@threads`, since it maintained the higher speedup results.

Figure 63 shows the speedups for `DepthFirstEx` and `@threads`. We can observe in this figure that they showed small speedups but with a similar behaviour. This drop tendency in the speedup could be a result from how the algorithm and the parallelism are implemented. This algorithm has an outer loop that iterates sequentially on the input size, each iteration has a parallel loop also iterating on the input size, this means the tasks are being created for each iteration of the outer loop, producing an overhead that could impact the overall performance.

Figure 60 – Password Cracking with Brute Force speedup with small input size of `@threads` and FLoops.jl with `WorkStealingEx` executor on an environment ranging from 2 to 64 threads.

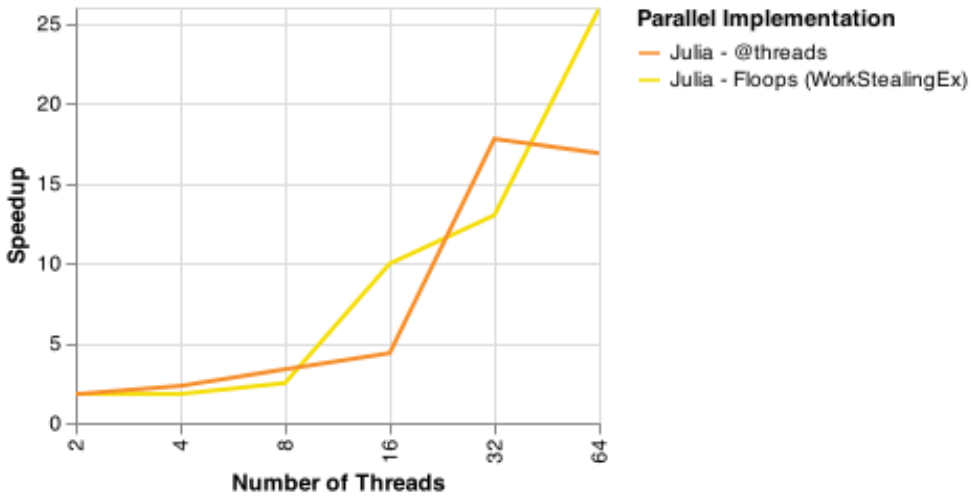


Figure 61 – Password Cracking with Brute Force speedup with medium input size of `@threads` and FLoops.jl with `WorkStealingEx` executor on an environment ranging from 2 to 64 threads.

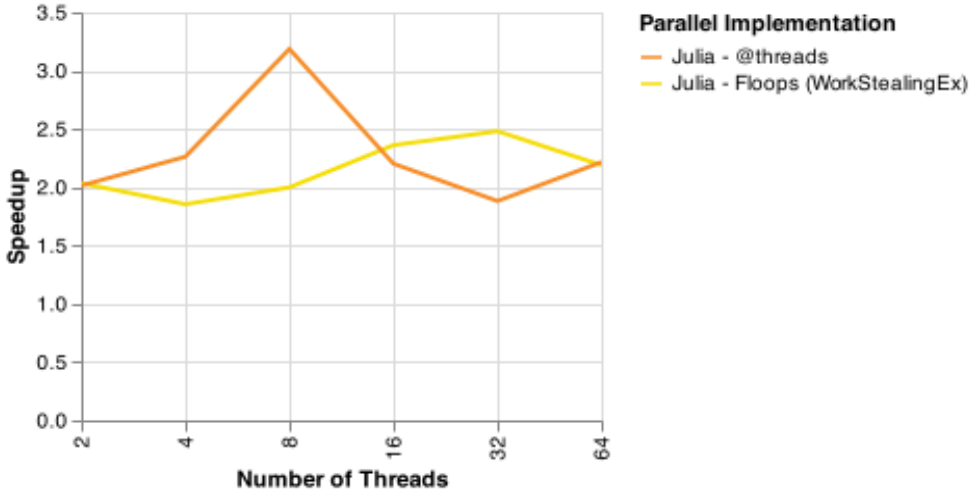


Figure 62 – Password Cracking with Brute Force speedup with large input size of `@threads` and FLoops.jl with `WorkStealingEx` executor on an environment ranging from 2 to 64 threads.

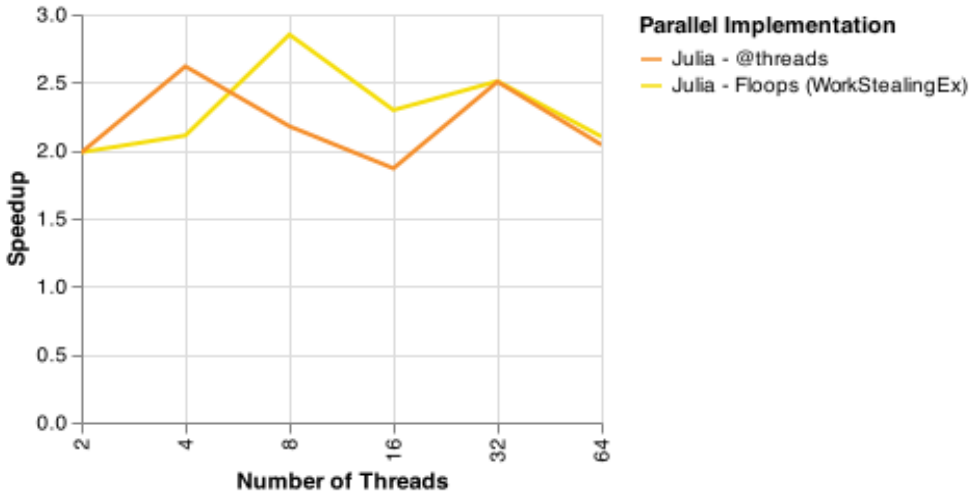


Figure 63 – Transitive Closure speedup of FLoops.jl executors on an environment ranging from 2 to 64 threads.

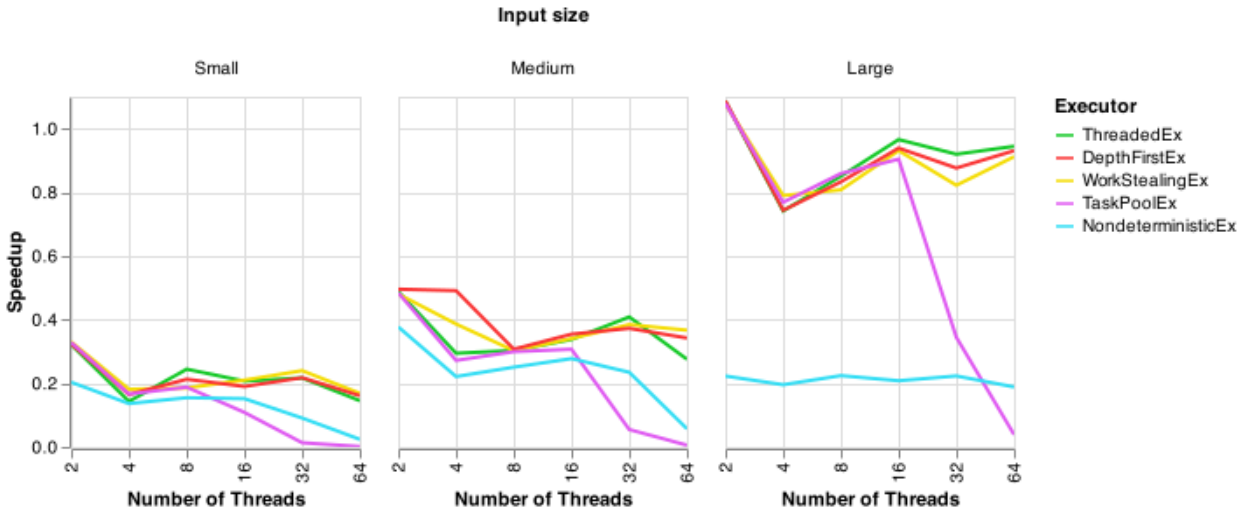
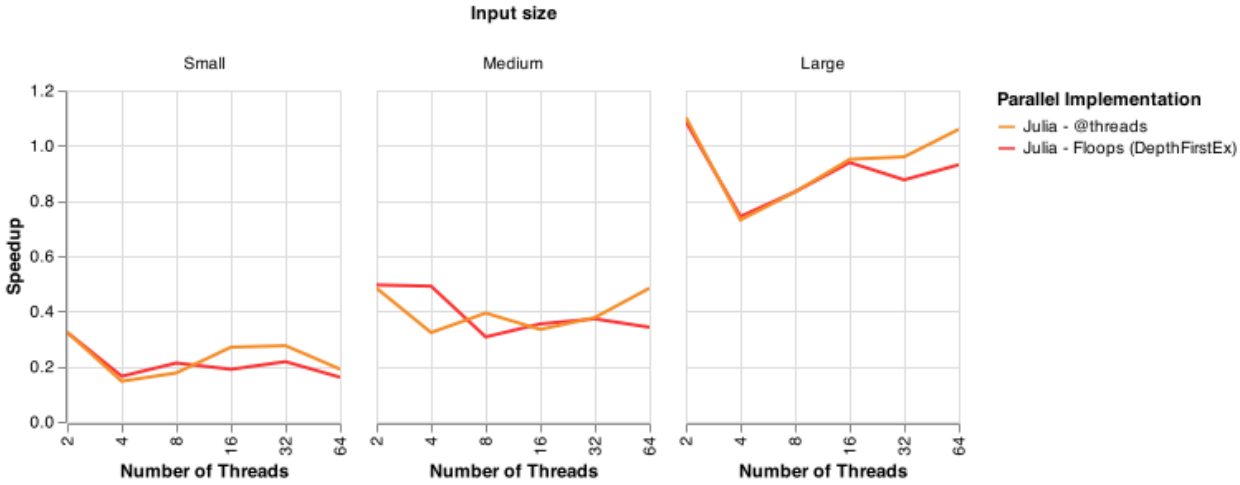


Figure 64 – Transitive Closure speedup of `@threads` and FLoops.jl with DepthFirstEx executor on an environment ranging from 2 to 64 threads.



5.5 Discussion

We observed different behaviors in speedup among the applications from the experiments as a result of the nature of each problem and how Julia was able to handle them. There were gains in performance and scalability with the Mutually Friendly Numbers application. With the Password Cracking with Brute Force and the Transitive Closure Problem applications, the C implementation outperformed Julia and was able to scale when larger work loads were in use. Therefore Julia's multithreading gains varies according with the code implementation.

The Julia package FLoops.jl shows to be an interesting option because the loop schedulers offered as executors can help improving performance. The Mutually Friendly Numbers showed to be an application that can gain from parallelism since the schedulers speedups showed a very consistent increasing behavior. The `ThreadedEx` executor outperformed C + OpenMP static scheduling and presented a similar behavior to the Julia macro `@threads`. For Password Cracking with Brute Force, we expect that the application would take more benefit with the `DepthFirstEx` executor, but `ThreadedEx` obtained similar results. However, the sequential version already showed a slower performance when comparing to C and when using multithreading we could not notice speedup increase with either a increase on number of threads or input size. Its implementation used parallelism when checking the characters of a password for each possible string length, but each different length of the password was tested sequentially. If we were able to also parallelize the testing for each string length and, for each of them, test the possible characters and traverse the possibility tree with the `DepthFirstEx`, we could prevent the additional work arisen from the sequential checking of the original implementation. On the Transitive Closure Problem, there is gain in performance with the increase of the input, but there is no scaling along with the increase of threads. This problem can deal with large matrices and the overhead from scheduling and task distribution might not compensate the time spent accessing the matrices elements on memory.

In our applications, the executors `ThreadedEx` and `DepthFirstEx` were the ones that presented better performance compared to the others available. The `DepthFirstEx` executor uses the Depth First algorithm for balancing the load. This algorithm is well-known to work well with unbalanced load [54]. The `ThreadedEx` executor uses a divide and conquer algorithm that recursively divides the load into halves until it reaches the basesize. Although this algorithm was designed to deal with a reduction style computation, it showed good performance on our unbalanced applications. Nevertheless, we did not experiment with different basesizes for the executors, the adjustment of this parameter could help tuning the scheduling.

Regarding the scalability of the parallel implementations of the loop scheduling mechanisms in Julia, we observed that Julia multithreading can scale with the environment. However, it is more noticeable in the application where there is only one loop as in the Mutually Friendly Numbers application. For Password Cracking with Brute Force and Transitive Closure, where there exists different parallel loops, and the amount of work that each loop generates is irregular. So, sometimes there is not enough work to be distributed when the number of threads increases.

6 EVALUATING SIMD MECHANISMS

In this chapter, we evaluate the performance of the SIMD mechanims offered by Julia. We compare the auto-vectorization ability of Julia with the auto-vectorization ability of the C compiler. We also compare the vectorized code generated by the macro `@simd`, the package SIMD.jl, and the package LoopVectorization.jl.

6.1 Experimental settings

The executions were performed on a AMD Ryzen 7 2700 Eight-Core Processor with 16 threads (2 threads per core with Simultaneous Multi-Threading), support to AVX/AVX2 instructions with 256 bits registers and 128 GB of RAM. According to Rosetta@Home's Whetstone benchmarks [70] this CPU peak speed is 4.36 GFLOPS/core and 69.78 GFLOPS/computer.

The experiments were run with the package BenchmarkTools.jl [49] to measure the execution time, with its parameters of number of samples equal to 10 and 1 evaluation. The execution times of each sample were logged in case a deeper investigation on each execution was needed and the mean execution time was calculated using Julia's built-in package Stastistics.jl. To measure the quantity of FLOPS reached by each experiment we used the number of instructions of a matrix multiplication. Multiplying two matrices $i \times k$ and $j \times k$, for each element of the resulting matrix $i \times j$ we would have k multiplications and $k-1$ additions, so $2k-1$ instructions. For every element of the resulting matrix we would then have $i \times j \times (2k-1)$ instructions. As the matrices we used on our experiments are $n \times n$, the number of operations on the matrix multiplication is $2n^3$. The FLOPS was then calculate as $2n^3/(execution\_time\_in\_seconds)$.

6.2 Benchmarks

Our first experiment was to test the auto-vectorization ability of Julia compiler on the Rodinia benchmarks applications: SRAD_V2, LUD and BFS. We checked whether the code was vectorized using `@code_llvm`. We observed though that the loops in these applications are complex, since they have conditionals and function calls inside them. So, the compiler was not able to vectorize any of these applications.

Our second experiment was to test the auto-vectorization ability of the gcc compiler for the same set of applications. We used the compiler flags *-ftree-vectorize* and *-fopt-info-vec-all* and the gcc compiler was also not able to vectorize the loops of SRAD_V2, LUD and BFS.

Therefore, we decided to perform the evaluation of the SIMD mechanisms present in Julia using a simple application: matrix multiplication. The sequential code is shown in Figure 65. The input matrices with elements of type `Int64` are square matrices of size $N \times N$, where N varies from 512 to 4096. A test with the use of `@threads` macro was also performed for comparison purposes. The SIMD mechanisms tested are:

- The macro `@simd`;
- The package SIMD.jl. This package provides a vector type for vectorization. Since the AVX vectors hold 4 double precision floating points numbers, the inner loop of the matrix multiplication was divided into chunks of 4 so that the SIMD.jl could handle the operations correctly. This package was tested both with and without the multithreading macro *@threads*;
- The package LoopVectorization.jl with the macros `@turbo` and `@tturbo`.

Figure 65 – Sequential and auto-vectorized versions of the matrix multiplication application used to evaluate the SIMD mechanisms.

```
function mm_sequential(A, B, C)
    for i in 1:size(A,1)
        for j in 1:size(B,2)
            for k in 1:size(A,2)
                C[i,j] += A[i,k] * B[k,j]
            end
        end
    end
    return C
end

function mm_auto_vectorized(A::Array{T,N}, B::Array{T,N}, C::Array{T,N})
    where {T,N}
    for i in 1:size(A,1)
        for j in 1:size(B,2)
            @inbounds for k in 1:size(A,2)
                C[i,j] += A[i,k] * B[k,j]
            end
        end
    end
    return C
end
```

To trigger auto-vectorization on the sequential code it was necessary to specify the data types on the function parameters and add `@inbounds` to prevent index checking inside the innermost loop. We need to ensure that the function is type stable so that the compiler is able to vectorize the code.

In the implementations that used *@simd* and the package LoopVectorization.jl, the only modification needed compared to the autovectorized version was the inclusion of the macros before the loops.

The package SIMD.jl, on the other hand, required more modifications in the code. This package was designed to give the programmer the ability to write explicit SIMD code. Still, the implementation using SIMD.jl required further modifications since it showed poor performance as explained in the next section.

6.3 **Results**

In our first SIMD.jl implementation, we had the rows of A and the columns of B divided into SIMD.jl vectors of size 4. This was done to fit into the SIMD registers, even

if the size of the matrices were not a power of two. The innermost loop iterates in steps of 4, so that the rows and columns could be loaded into the SIMD.jl vectors. In this way, they could be multiplied and stored in a temporary SIMD.jl vector and a reduction is made on the temporary vector after the innermost loop is finished. The issue with this implementation is similar to one pointed out at Julia's forum [71], and according to Elrod, C. [72], while the vectors of chunk 4 are computing the multiplication with SIMD parallelism, the innermost loop was not unrolled and the need to load and store the indexes turn the execution very slow.

We checked the assembly code of this implementation and we observed a similar structure as pointed by Elrod, C. Figure 66 shows the assembly code (obtained using *@code_native*). It corresponds to part of the innermost loop and we can see the instruction `vmovupd` performing the load and store operations for the index. This means that it needs to access the memory for these index operations. The last three instructions correspond to the increment of the loop counter, a comparison test, and a jump to the beginning of the loop.

After these findings, we decided to try another strategy for the matrix multiplication with SIMD.jl. We transferred the rows from A and the columns from B entirely to a SIMD.jl vector, so that the innermost loop of the previous implementations is no longer necessary. In this case, the loop was unrolled. In this implementation, if the size of the matrices is a power of two, it is possible to execute the entire matrix multiplication with SIMD instructions. If the size is not a power of two, the rest of the matrix is computed sequentially.

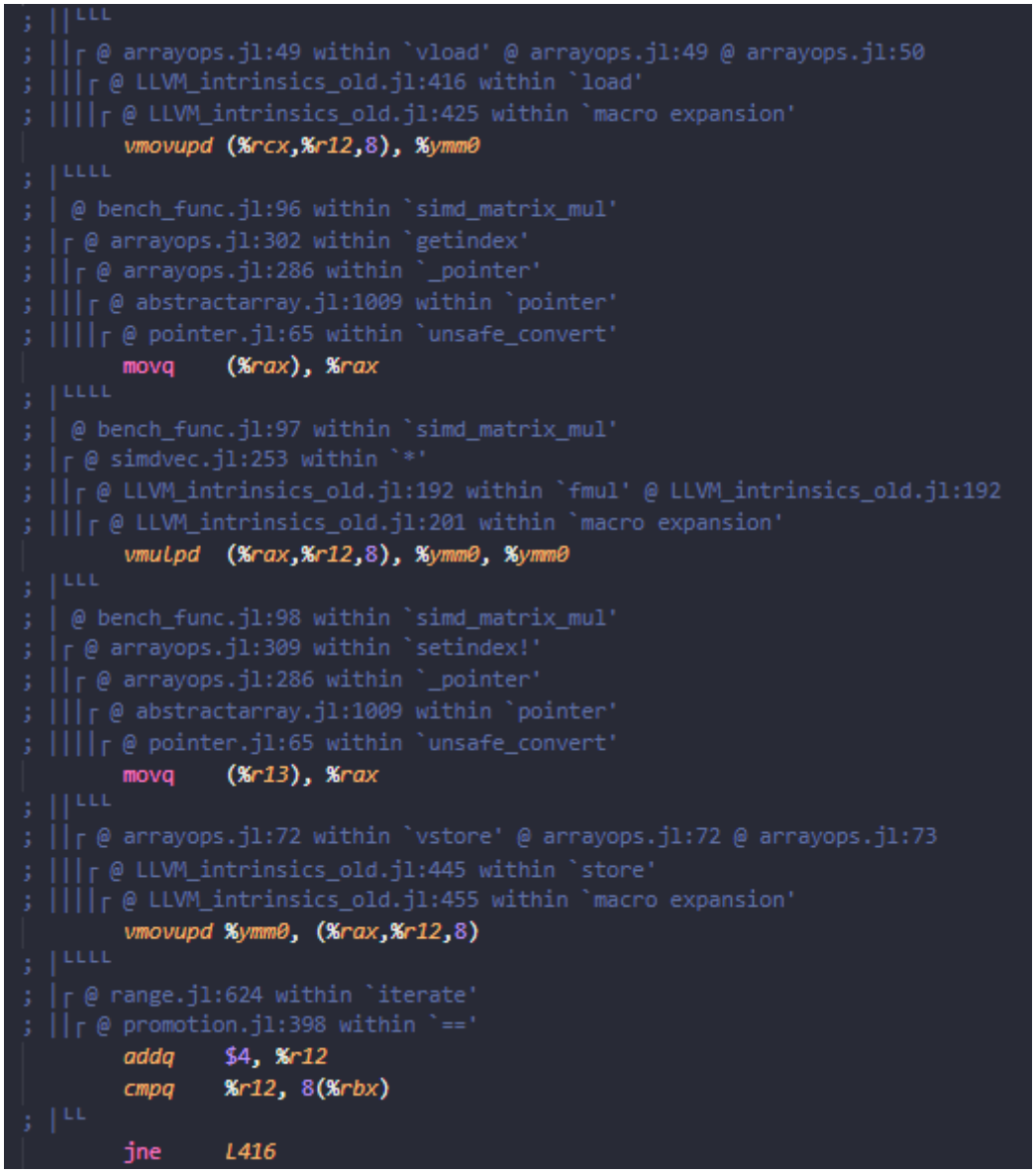
Table 3 – GFLOPS for matrix multiplication with different SIMD mechanisms and different matrices sizes.

Size ($N \times N$)	Sequential	Auto-Vectorized	Multithread	@simd	SIMD.jl		LoopVectorization.jl	
					without Multithreading	with Multithreading	Turbo	TTurbo
512	0.38	0.37	3.12	0.37	0.28	8.3	4.51	22.51
1024	0.30	0.37	2.6	0.37	0.33	10.91	4.34	31.11
2048	0.23	0.23	0.54	0.23	0.39	8.91	4.17	24.11
4096	0.10	0.10	0.47	0.10	0.42	8.17	4.16	20.46

Table 3 shows the GFLOPS obtained on the sequential, auto-vectorized, multithreaded, @simd, SIMD.jl (without and with multithreading), and LoopVectorization's Turbo and TTurbo mechanisms. The matrices are $N \times N$ and the column *Size* contains the value of N, we analyze input sizes of 512, 1024, 2048 and 4096. Looking at the single threaded implementations we can clearly see that the only mechanism that was able to exploit better the CPU was the `@turbo`, as its GFLOPS were close to the peak GFLOPS/core obtained by the CPU Whetstone benchmark. The sequential, auto-vectorized and `@simd` implementations obtained very similar results on GFLOPS, with 8% of the peak CPU speed on the smaller input size, lowering to 2% on the larger input size. The SIMD.jl implementation obtained a lower number of GFLOPS than the sequential version on the smaller input size but it increased with the input size, reachin 10% of the peak CPU speed. The `@tturbo` implementation could take more advantage of multithreading than SIMD.jl with multithreading, reaching around 35% of the peak GFLOPS/computer while SIMD.jl with multithreading reached around 15%.

For a better understanding of the behavior of each mechanism, we provide in Figure 67 a plot that shows the matrix multiplication execution times in seconds for all the SIMD mechanisms. The sequential, auto-vectorized, `@simd` and SIMD.jl implementations showed similar behavior. It is worth remembering that the use of the `@simd` macro is

Figure 66 – Assembly code obtained by *@code_ native* on thde first SIMD.jl matrix multiplication implementation. Corresponds to a part of the most inner loop.

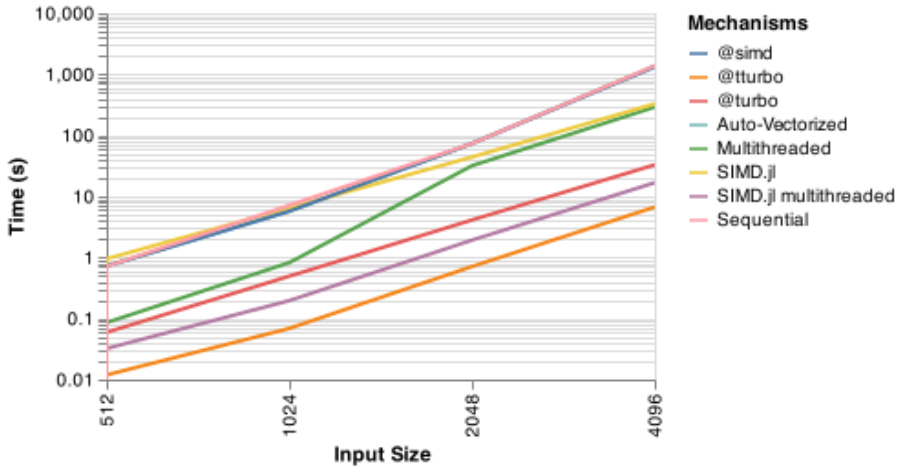


discouraged by Julia language, because it relies totally on the programmer skills to make the vectorization safe. There was not much difference between them until after matrix size 1024×1024 where SIMD.jl started to show better performance. However, as a specialized package for SIMD operations, one would expect an execution time closer to the other specialized package, what was not observed. Even though the innermost loop was unrolled in the SIMD.jl implementation, there is a high overhead of having to transfer the rows of A and the columns of B to the SIMD `Vec` type.

The implementations with faster execution time were `@tturbo`, SIMD.jl with multithreading and `@turbo`, in this order. Right after them there was the multithreading only implementation we displayed to serve as a reference for the SIMD.jl with multithreading and the `@tturbo` implementations.

In order to check what kind of optimization LoopVectorization.jl was performing, we used the macro `@turbo_debug`. This macro returns the `LoopSet` structure used by the package instead of evaluating the loop. With `LoopSet`, we are able to pass it as

Figure 67 – Execution time in seconds of each SIMD mechanism shown in logarithmic scale. Input size represents N for matrixes $N \times N$.



argument to the function `choose_order` and understand what LoopVectorization.jl was doing. Figure 68 shows the usage of `@turbo_debug` and the output from `choose_order`. Looking at the output, we can see six parameters. The first one is an array that represents the order that LoopVectorization evaluated each loop. The second and third arguments are the indexes that represent which loops were marked to be unrolled. The fourth argument is the loop chosen to be vectorized. The last two arguments correspond to the number of times the loop is replicated in the loop body for each unrolled loops.

Figure 68 – Usage of *@turbo_debug* on matrix multiplication and the output from *choose_order* showing what strategy LoopVectorization used.

```
julia> ls = LoopVectorization.@turbo_debug for i in 1:size(A,1)
                for j in 1:size(B,2)
                    for k in 1:size(A,2)
                        @inbounds C0[i,j] += A[i,k] * B[k,j]
                    end
                end
            end;

julia> LoopVectorization.choose_order(ls)
([:j, :i, :k], :i, :j, :i, 2, 6)
```

The last experiment was to compare our results with the C version of matrix multiplication that relies on the auto-vectorization of the gcc compiler. In this experiment, however, we faced some difficulties in trying to make the compiler vectorize the code.

First we had to modify the matrix multiplication code to a version where we could obtain contiguous memory accesses. This was done by swapping the inner loops, as shown in Figure 69. When we compile this code using `cmd` and the flags `-O2`, `-ftree-vectorize` and `-fopt-info-vec`, the compiler indicated that the matrix multiplication loop was *versioned*. Loop versioning happens when there is an unaligned access and the compiler creates two versions of the loop, one that is vectorized and the other that is not. Which version will be used is decided at runtime. So, even though there is a vectorized version, we cannot tell if this is the one that executed.

Figure 69 – Sequential version of C matrix multiplication.

```
void mm_sequential_C(int m, int n, int p, int **A, int **B, int **C){
    for(int i = 0; i < m; i++){
        for(int k = 0; k < p; k++){
            int a_row_i = A[i][k];
            for(int j = 0; j < n; j++){
                C[i][j] += a_row_i * B[k][j];
            }
        }
    }
}
```

6.4 **Discussion**

Since SIMD units are available in most of the current processors, exploiting this type of parallelism can be advantageous. Julia offers the user many alternatives to exploit SIMD parallelism. The simplest alternative, auto-vectorization, requires the programmer to follow the recommendations mentioned in section 3.5.1. When the programmer has to explicitly request the compiler to vectorize the code, they could add the built-in feature `@simd` in front of the `loop` , but it did not show any significant gains over the sequential and the auto-vectorized versions and it is still marked as experimental and discouraged by the developers since it can lead to unexpected results. We were not able to measure the reliability of this macro in our experiments.

Julia also provide some packages for SIMD parallelism. The SIMD.jl package started showing gains in execution time compared to the sequential, auto-vectorization and the `@simd` versions after the 1024×1024 matrix, but compared to the LoopVectorization.jl package it was 10 times slower. This occurred due to the need to transform the data type to the SIMD.jl `Vec` type. SIMD.jl seems to be a package in which the user interface is more similar to a lower level "vectorization language". The way it is programmed is similar to lower level vector instructions. In this case, the user should be more familiar with programming vectorized code and should be responsible for the correctness of the results. However, the user has the option to optimize it combining with multithreading, which led to an execution time 10 times faster than both SIMD.jl and multithreading only implementations on the larger matrix size.

The best results were obtained with the package LoopVectorization.jl, being the only implementation that could exploit the best the CPU speed. This package not only vectorizes the loop, but also implements loop reordering and unrolling, which provides further performance improvements. The `@turbo` results shows how the application can benefit from vectorization, and also shows how loop optimizations have influence in the execution time. The `@tturbo` version seems to be the best option. It combines vectorization, loop optimizations, and it also includes the benefits of using multithreading.

Our experience with the vectorization in C shows that for the users to benefit from it, they need to have enough knowledge of the code to make vectorization possible and reliable. GCC provides documentation about its auto-vectorization [73] with several examples of loops that can and cannot be vectorized. The documentation confirms that the possibilities are restricted. Even though it was not possible to compare the use of SIMD operations of C and Julia in the matrix multiplication, we verified that Julia can make the vectorization more user friendly and more accessible for the user that does not have

enough knowledge on SIMD programming.

7 EXPLOITING JULIA PARALLELISM IN A REAL-WORLD SCENARIO

Another contribution of this Dissertation is the study of the use of Julia multithreading in a real-world scenario, where we exploit parallelism to improve the performance of Coluna.jl. Coluna.jl is an open-source branch-cut-and-price framework code written in Julia that implements optimization methodologies based on decomposition and extended reformulation for mixed integer linear optimization programs (MIPs).

7.1 The Coluna.jl Framework

Coluna.jl is an open-source branch-cut-and-price framework code in Julia that implements optimization methodologies based on decomposition and extended reformulation for mixed integer linear optimization programs (MIPs). Combinatorial optimization problems with semi block-diagonal constraint coefficient matrix are well suited for Coluna.jl that reformulates their original MIP representation and optimizes the reformulation using specialized divide-and-conquer algorithms customized by the user. The underlying solution method is a branch-cut-and-price approach that combines a tree search in the solution space with online generation of the variables of the extended reformulation. The latter is at the core of the solver and often a bottleneck using significant computing time for complex applications.

The framework is developed on an open-source basis and hence can engage innovations and constant updates; it is made to serve as a solid socle to researchers building further algorithmic progress. Another important feature of Coluna.jl is that it is completely implemented in Julia, a recent dynamic language that provides high productivity and high performance. Choosing Julia to develop Coluna.jl was motivated by striking the best trade-off between efficiency and ease to share co-developments under a simple syntax to facilitate updates and extension of the framework.

7.1.1 Mixed-Integer Programming

Mixed-integer programming (MIP) is an optimization paradigm with a wide range of real world applications. In a MIP, the decision variables take either continuous or integer values, while the objective function and the constraints are linear expressions of the variables. MIP allows a fair description of reality and is also well-suited for global optimization. The solution of such models is essentially based on enumeration techniques and is notoriously difficult given the huge size of the solution space. Although, commercial solvers for MIP have made significant progress in the last decade, one can go further using decomposition techniques. In this context, Coluna.jl was designed to offer functionalities to implement a branch-and-price-and-cut approach to a decomposable MIP at the core of which is a modular implementation of the so-called column generation method. Coluna.jl has been launched by the startup AtOptima in a collaboration with the University of Bordeaux and Inria and with a kick-off grant from the Math Optimization Society.

It is openly available at https://github.com/atoptima/Coluna.jl under a Mozilla Public Licence.

7.1.2 Column generation

Coluna.jl is designed to facilitate the use of complex decomposition paradigms such as Benders and Dantzig-Wolfe, while supporting the JuMP modeling language interface. This "basic-user-mode" is completed by a researcher usage mode as one can take over any routine and develop advanced algorithmic strategies by combining the solver modules. To get started, the user writes the original MIP that models his problem using JuMP together with a package called BlockDecomposition.jl that extends JuMP to specify the problem decomposition. Coluna automatically reformulates the original MIP following the decomposition instructions. Coluna then calls the algorithm chosen by the user to optimize the reformulation.

In this work, we focus on column generation. Usually, column generation arises when the original MIP has a set of "tractable constraints" that define a collection of independent subproblems (these subproblems can be solved with efficient specific algorithms), and a set of "linking constraints", that make the subproblems dependent on each other. This original MIP gives rise to an extended reformulation where the variables represent decision to choose a specific feasible solution to a subproblem. As this reformulation involves an exponential number of variables, one uses a column generation algorithm that incrementally adds these variables and associated columns of constraint coefficients into the formulation.

Thus, the column generation algorithm consists in a successive exchange of information between the resolution of the linear relaxation of the formulation restricted to a subset of variables, also known as the master problem, and the pricing oracle used to identify a subproblem solution that could induce an improvement to the current master solution. For each generated subproblem solution, a new variable associated with it is introduced in the master program. The master is then re-optimized and the algorithm iterates in that way until the pricing oracle can no longer identify an improving subproblem solution. The branch-and-price algorithm is a branch-and-bound algorithm applied to the extended reformulation. As this integer program has an exponential number of variables, they are generated dynamically, using the above column generation procedure at each node of the branch-and-bound tree.

7.1.3 Opportunities for parallelism

Coluna.jl is one of very few contributions attempting to offer a framework to implement a branch-cut-and-price methodology, along GCG [74] and DIP [75] that are the best known alternatives. Coluna.jl is the most recent initiative that differentiates itself by an attempt to offer a more modular design with innovations for flexibility such as algorithmic strategies; it is also designed to be high level based on a modeling interface and the high-level Julia language, while being focus on performance; parallelism is a key asset in this regard. GCG offers a generic branch-and-price solver over the SCIP [76] mixed integer programming platform (which is an open-source project but it is restricted to academic use). CGC and SCIP are written in C/C++. CGC can perform an automated Dantzig-Wolfe decomposition, while Coluna.jl takes instructions from the user to define the decomposition. SCIP offers facilities to parallelize the branch-and-bound tree search. DIP is distributed via the COIN-OR initiative[77]. DIP does the decomposition on the

user instructions (like Coluna) and it embeds a column generation algorithm. Under DIP, the solution of individual subproblems can be parallelized as in the present work, multiple subproblems can be solved simultaneously. In addition, with DIP, one can rely on running several decomposition-based algorithms in parallel.

The continuous effort to develop features for parallelization in Julia seems to have not yet been exploited in the Julia-Optimization community. A number of packages have been created for optimization modeling, optimization infrastructure and for solving different types of optimization problems [78]. But none exploits the Julia multithreading which is admittedly a very recent feature. There are only a few studies outside the optimization area that exploit it to our best knowledge. Summers *et al.* [23] used Julia threads to improve the performance of a robot control package. Novosel and Slivnik [24] performed a preliminary comparison between Julia and Chapel on distributed and shared memory implementations. Stanitzki and Strube [25] used Julia threads and channels to accelerate data analysis workflows in high energy physics.

7.2 JuMP

For optimization problems, Julia offers JuMP [79], an embedded modeling language. JuMP translates mathematical expressions into internal representations. The idea is to maintain a simple syntax to the programmer that is near to the natural representation of an optimization problem. The variables of the problem are described by the macro `@variable`, the objective function by the macro `@objective` and the constraints by the macro `@constraint`.

Suppose we have an optimization problem described as:

$$\begin{aligned} &\text{minimize} \sum_{ij \in A} c_{ij} x_{ij} \\ &\text{subject to} \\ &\qquad \sum_{ij \in A} x_{ij} - \sum_{ji \in A} x_{ji} = b_i \qquad \forall i \in N \\ &\qquad 0 \leq x_{ij} \leq 1 \qquad \forall (i,j) \in A \end{aligned}$$

This mathematical expression can be described with JuMP in the following way:

```
@variable(m, 0<= x[links] <=1)
@objective(m, Min, sum(c[(i,j)] * x[(i,j)] for (i,j) in links) )
for i=1:no_node
    @constraint(m, sum(x[(ii,j)] for (ii,j) in links if ii==i ) - sum(x[(
    j,ii)] for (j,ii) in links if ii==i ) == b[i])
end
solve(m)
```

7.3 Parallel Implementation

Among the parallelization opportunities in the branch-cut-and-price method of Coluna.jl, our first step was to exploit multithreading on the pricing oracle which is usually the bottleneck of the column generation algorithm. In principle, the resolution of each subproblem does not require any communication. Therefore, a straightforward and efficient way to parallelize the pricing oracle is to distribute the subproblem resolutions

among the number of available threads. In the end of all subproblem resolutions, a new variable associated with each subproblem solution is added to the master problem.

An important point to consider is the load balancing among the threads. Each subproblem has a different computational cost, that is not known in advance. Thus, we implemented two different multithreading strategies taking advantage of Julia support for task and data parallelism, with the macros `@spawn` and `@threads` respectively. In the task parallelism approach, our implementation spawns one task for each subproblem computation, and relies on the runtime system for dynamically distributing the tasks to the threads. In the data parallelism approach, the for-loop that iterates over all subproblems is parallelized and the subproblem computations are equally divided among the threads.

7.4 Case Study: the Generalized Assignment

In the generalized assignment problem (GAP), we are given n tasks to be assigned to m machines (or agents), where each machine i has capacity u_i, and each task j, when assigned to machine i, uses d_{ij} units of resource and costs c_{ij}, for $j = 1,\ldots,n$, and $i = 1,\ldots,m$. The problem consists in assigning each task to exactly one machine, such that the total resource usage on each machine does not exceed its capacity, and the total assignment cost is minimized.

Let x_{ij} be a variable equals to 1 if the job j is assigned to machine i, 0 otherwise. The mathematical program that models this problem is :

$$\text{minimize} \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \tag{7.1}$$

$$\text{subject to} \tag{7.2}$$

$$\sum_{i \in I} x_{ij} = 1 \qquad \forall j \in J \tag{7.3}$$

$$\sum_{j \in J} d_{ij} x_{ij} \leq u_i \qquad \forall i \in I \tag{7.4}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in I,\ j \in J \tag{7.5}$$

Set-partitioning constraints (7.3) ensure that each task is assigned to one machine. Knapsack constraints (7.4) ensure that the total weight of the jobs assigned to a machine does not exceed the capacity of the machine. This application involves multiple distinct subproblems. A solution of the pricing subproblem consists of a set of tasks to be assigned to one of the machines, that satisfies the knapsack constraint (7.4).

This mathematical program can be described with JuMP in the following way :

```
@variable(m, x[i in I, j in J], Bin)
@objective(m, Min, sum(c[i,)] * x[i,j] for i in I, j in J))
@constraint(m, setpart[j in J], sum(x[i,j] for i in I ) == 1)
@constraint(m, knp[i in I], sum(d[i,j] * x[i,j] for j in J) <= u[i])
optimize!(m)
```

7.5 Experimental setting

The experiments were performed on a AMD Ryzen 3950X processor with 16 cores and 64GB of memory. Each experiment was executed 10 times. We tackle GAP instances from [80] : gapC10-100 (100 tasks and 10 machines) and gapC20-200 (200 tasks and 20

machines). Both the master program and the binary knapsack subproblems are solved using Gurobi 9.0 (as efficient as specialized knapsack solvers for the instance size considered here). To eliminate variability related to primal bounds, all variants were executed with the optimum solution value reported in the literature as the initial primal.

7.6 **Performance Evaluation**

Figure 70 shows the speedups of the parallel subproblem computation with an increasing number of threads compared to the sequential execution for both instances using `@threads` and `@spawn`. We can observe that the parallelization was able to provide speedups for the subproblem computation from 2.3 to 3.3 for gapC10-100 and 3.3 to 4.0 for gapC20-200. Comparing the different parallelization strategies, the dynamic thread distribution of `@spawn` was able to balance the load dynamically producing a slightly better performance than the static distribution of `@threads`. The increase in the number of threads does not impact significantly the speedups since the number of subproblems to compute at each node is not very large (with 10 or 20 subproblems in our tests). Table 4 shows the execution times of Coluna.jl with the time spent in subproblem computation considering the parallelization with `@spawn`. We can observe that, for GAP, the time spent in subproblem computation is not that much pronounced (the knapsack subproblem is relatively easy even for a general MIP solver); thus we expect more significant reductions of Coluna.jl execution time on more complex problems.

Figure 70 – Subproblem computation speedups.

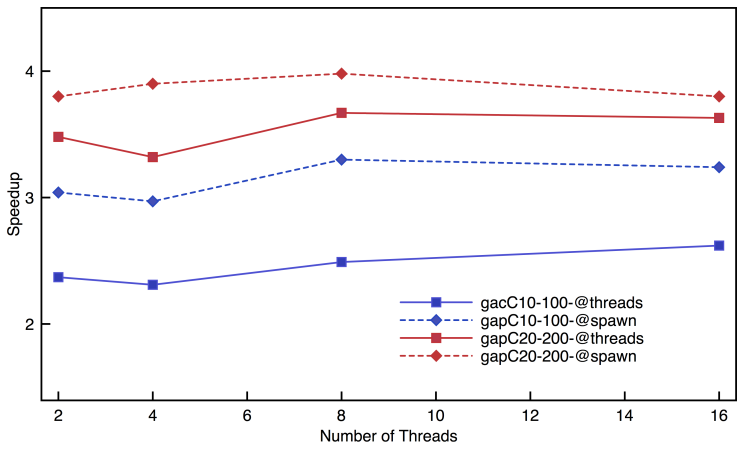


Table 4 – Execution times (in seconds).

	gapC10-100		gapC20-200	
threads	subproblem	Coluna	subproblem	Coluna
1	4.511	21.484	8.677	94.353
2	1.484	19.846	2.288	88.825
4	1.524	19.582	2.229	87.963
8	1.372	19.644	2.183	88.862
16	1.408	19.674	2.284	89.290

7.7 Discussion

In exploiting parallelism to improve the performance of an open-source branch-cut-and-price framework implemented in Julia language, we propose to exploit Julia multithreading in the generation of subproblem solutions at each branch-and-bound node. We implemented two thread distribution strategies and analyzed the performance on the Generalized Assignment Problem (GAP). The parallel strategies provided performance improvements in the subproblem computation, but, for GAP, the subproblem resolution does not represent a significant portion of the execution time. We intend in the future to evaluate Coluna.jl on other optimization problems like vehicle routing and also to explore other parallelization opportunities inside Coluna.jl, starting with the tree search.

CONCLUSIONS

Julia is a high level dynamic language that was developed with main focus on high performance computing. It is becoming widely used in scientific programming and, with the increase in its community, more packages are being developed to help the integration with various applications.

In this work, we studied the performance of shared memory parallel computing mechanisms present in Julia. We studied the multithreading strategies that apply data and task parallelism with static and dynamic scheduling of loop iterations, which are implicitly defined in the macros `@threads` and `@spawn`, respectively. We also studied the different loop scheduling approaches available for multithreading with the built-in macro `@threads` and the package FLoops.jl along with its schedulers available as executors from the package FoldsThreads.jl. We also investigated the performance of different mechanisms to exploit SIMD parallelization. The performance studies were carried out using synthetic kernels, benchmark applications, and a real-world framework for branch-and-cut-and-price, called Coluna.jl.

Our results showed that, for the synthetic kernels with high degrees of load imbalance, Julia's task parallelism with dynamic scheduling with the macro `@spawn` was able to better distribute the tasks spawned among threads and decrease the imbalance. However, when dealing with applications that are more likely to be found in real world scenarios, as the ones from Rodinia benchmark from our experiments, the depth first scheduling of `@spawn` included overhead in the loop computation and was not able to improve the load balancing. When applied to the Coluna.jl framework, multithreading did show gains in performance. The nature of the problem studied, though, presented limited parallelization opportunities.

Regarding the different loop scheduling strategies, Julia showed to be able to improve performance with multithreading with an increasing number of threads on the Mutually Friendly Numbers application. However, it could not scale with the others real-world applications Password Cracking with Brute Force and Transitive Closure Problem. Understanding the behavior of the program is of importance to determine the best scheduling approach. The FLoops.jl executors that presented best performance on unbalanced applications were mainly the `ThreadedEx` and the `DepthFirstEx`, even though their `basesize` parameter was not included in our experiments.

On the evaluation of the SIMD mechanisms, we concluded that the package LoopVectorization.jl provided the best performance results, being able to reach peak CPU performance and requiring only the addition of the macro `@turbo` in front of the loops. The package SIMD.jl has a "low level" type interface, since it gives the user the possibility to explicitly write SIMD code. In addition, it needs the data to be stored in a SIMD `Vec` type beforehand, which creates an overhead that can impair the performance. These two packages are recommended by developers over the built-in macro `@simd`, which can lead to unexpected results. At last, there is the auto-vectorization that, even though it does not perform as well as LoopVectorization.jl, it could help improve performance of codes

with smaller inputs without any programming intervention.

Overall, it is important to highlight that Julia provides shared memory mechanisms with a user-friendly interface. Moreover, the users can develop solutions to any kind of problems taking advantage of the abstraction provided by the usage of available packages. Julia's simplicity in developing parallel code could help promoting parallel processing for scientists without a specialized coding background, without falling on the "two-language problem", but it does show limitations on large scale environments.

The study of the performance of Julia loop scheduling mechanisms was published as:

- Diana A. Barros and Cristiana Bentes, "Analyzing the Loop Scheduling Mechanisms on Julia Multithreading," 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 257-264, doi: 10.1109/SBAC-PAD49847.2020.00043.

The study of the use of multithreading mechanisms in the framework Coluna.jl was published as:

- Diana A. Barros, Guillaume Marques, Vitor Nesello, Cristiana Bentes, François Vanderbeck. "Exploiting Parallelism in the Open-Source Coluna.jl Framework". The 2nd International Workshop on Parallel Optimization using/for Multi- and Many-core High Performance Computing (POMCO 2020). International Conference on High Performance Computing & Simulation, 2021.

Future Directions

This work is a first step in the analysis of the shared memory parallel programming features available in Julia. We intend to provide further outcomes in this subject by comparing the performance of the Julia applications presented in this work with equivalent C/C++ data and task parallelism implementations; by providing studies with the new `@threads` scheduler `:dynamic` available in Julia's current version; and by analyzing the impact of tunning the `basesize` parameter from the FLoops.jl loop scheduling executors on the performance.

Some of the features that make Julia an efficient language, such as dynamic typing or Just-in-time (JIT) compilation, also make Julia allocate more memory when compared to C implementation. Therefore, an interesting future direction of this work is the study of the memory footprint provided by Julia multithreading mechanisms.

Finally, another important future direction of this work is to Evaluate Coluna.jl with other optimization problems like vehicle routing.

REFERENCES

[1] Jose Juan Mijares Chan et al. "Parallel ant brood graph partitioning in julia". In: *Parallel Processing and Applied Mathematics*. Springer, 2016, pp. 176–185.

[2] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective extensible programming: unleashing Julia on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2018), pp. 827–841.

[3] J. Regier et al. "Cataloging the Visible Universe Through Bayesian Inference at Petascale". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 44–53.

[4] Raffaele Ferrari. *MIT News: New climate model to be built from the ground up*. <http://ferrari.mit.edu/news/new-climate-model-to-be-built-from-the-ground-up/>.

[5] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Rev.* 59.1 (Jan. 2017), pp. 65–98. ISSN: 0036-1445. DOI: <10.1137/141000671>. URL: <https://doi.org/10.1137/141000671>.

[6] Jeffrey M. Perkel. "Julia: come for the syntax, stay for the speed". In: *Nature* 572.7767 (2019), pp. 141–142.

[7] *MPI.jl*. <https://github.com/JuliaParallel/MPI.jl>. (Visited on 04/02/2023).

[8] *Elemental.jl*. <https://github.com/JuliaParallel/Elemental.jl>. (Visited on 04/02/2023).

[9] *CUDA.jl*. <https://github.com/JuliaGPU/CUDA.jl>. (Visited on 04/02/2023).

[10] *AMDGPU.jl*. <https://github.com/JuliaGPU/AMDGPU.jl>. (Visited on 04/02/2023).

[11] *Metal.jl*. <https://github.com/JuliaGPU/Metal.jl>. (Visited on 04/02/2023).

[12] *FLoops.jl*. <https://github.com/JuliaFolds/FLoops.jl>. (Visited on 01/25/2023).

[13] *LoopVectorization.jl*. <https://github.com/JuliaSIMD/LoopVectorization.jl>.

[14] Kiran Pamnany Jeff Bezanson Jameson Nash. *Announcing composable multi-threaded parallelism in Julia*. Retrieved from <https://julialang.org/blog/2019/07/multithreading/>.

[15] Chris Elrod. *ANN: LoopVectorization 0.12: multithreading and better handling of discontiguous memory accesses*. <https://gcg.or.rwth-aachen.de>. (Visited on 03/16/2021).

[16] *Gaius.jl*. <https://github.com/MasonProtter/Gaius.jl>. (Visited on 10/13/2021).

[17] *Octavian.jl*. <https://github.com/JuliaLinearAlgebra/Octavian.jl>. (Visited on 10/13/2021).

[18] *SnpArrays.jl*. <https://github.com/OpenMendel/SnpArrays.jl>. (Visited on 10/13/2021).

[19] Seyoon Ko et al. *DistStat.jl: Towards Unified Programming for High-Performance Statistical Computing Environments in Julia*. 2020. arXiv: <2010.16114> [stat.CO].

[20] Dániel Nagy, Lambert Plavecz, and Ferenc Hegedűs. *Solving large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs: performance comparisons of MPGOS, ODEINT and DifferentialEquations.jl*. 2020. arXiv: <2011.01740> [cs.DC].

[21] Chris Rackauckas Chris Elrod Niklas Korsbo. *Doing small network scientific machine learning in Julia 5x faster than PyTorch*. <https://julialang.org/blog/2022/04/simple-chains/>. 2022. (Visited on 12/18/2022).

[22] *SimpleChains.jl*. <https://github.com/PumasAI/SimpleChains.jl>. (Visited on 12/18/2022).

[23] Colin Summers et al. "Lyceum: An efficient and scalable ecosystem for robot learning". In: *arXiv preprint arXiv:2001.07343* (2020).

[24] Rok Novosel and Bostjan Slivnik. "Beyond Classical Parallel Programming Frameworks: Chapel vs Julia". In: *8th Symposium on Languages, Applications and Technologies (SLATE 2019)*. Vol. 74. OpenAccess Series in Informatics (OASIcs). 2019, 12:1–12:8. ISBN: 978-3-95977-114-6.

[25] Marcel Stanitzki and Jan Strube. "Performance of Julia for High Energy Physics Analyses". In: *arXiv preprint arXiv:2003.11952* (2020).

[26] Jason Selwyn Pavel V. Dimens. *BioJulia/PopGen.jl: v0.8.0 (v0.8.0)*. <https://zenodo.org/record/6450254>. 2022.

[27] *PopGen.jl @spawn implementation*. <https://github.com/BioJulia/PopGenCore.jl/blob/d3e954801430f4b4f68199e7e72567ca7fb14605/src/Permutations.jl#L8>. (Visited on 12/18/2022).

[28] *LombScargle.jl*. <http://juliaastro.org/LombScargle.jl/stable/>. (Visited on 12/18/2022).

[29] *LombScargle.jl @threads implementation*. <https://github.com/JuliaAstro/LombScargle.jl/blob/5d93ed93233972bd9bcfb3558abb916cf2a7f2ad/src/gls.jl#L53>. (Visited on 12/18/2022).

[30] Jan Gmys et al. "A comparative study of high-productivity high-performance programming languages for parallel metaheuristics". In: *Swarm Evol. Comput.* 57 (2020), p. 100720.

[31] Eduard Ayguadé et al. "Is the schedule clause really necessary in OpenMP?" In: *International workshop on OpenMP applications and tools*. Springer. 2003, pp. 147–159.

[32] Florina M Ciorba, Christian Iwainsky, and Patrick Buder. "OpenMP loop scheduling revisited: making a case for more schedules". In: *International Workshop on OpenMP*. Springer. 2018, pp. 21–36.

[33] Peter Thoman et al. "Automatic OpenMP loop scheduling: a combined compiler and runtime approach". In: *International Workshop on OpenMP*. Springer. 2012, pp. 88–101.

[34] Yun Zhang et al. "An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs." In: *ISCA PDCS*. Citeseer. 2004, pp. 256–263.

[35] Marie Durand et al. "An efficient openmp loop scheduler for irregular applications on large-scale numa machines". In: *International Workshop on OpenMP*. Springer. 2013, pp. 141–155.

[36] Vivek Kale et al. “Toward a Standard Interface for User-Defined Scheduling in OpenMP”. In: *International Workshop on OpenMP*. Springer. 2019, pp. 186–200.

[37] *Flux.jl*. URL: <https://github.com/FluxML/Flux.jl> (visited on 03/16/2021).

[38] *DifferentialEquations.jl*. URL: <https://github.com/SciML/DifferentialEquations.jl> (visited on 03/16/2021).

[39] *JuMP.jl*. URL: <https://github.com/JuliaOpt/JuMP.jl> (visited on 03/16/2021).

[40] *The Julia Language*. URL: <https://docs.julialang.org/> (visited on 03/23/2021).

[41] *Introducing Julia/Types*. <https://en.wikibooks.org/wiki/Introducing_Julia/Types>. (Visited on 04/02/2023).

[42] Jeff Bezanson et al. *Julia: A Fast Dynamic Language for Technical Computing*. 2012. arXiv: <1209.5145> [cs.PL].

[43] Jean Goubault. *Implementing Functional Languages with Fast Equality, Sets and Maps: an Exercise in Hash Consing*. Tech. rep. Bull S.A. Research Center, rue Jean-Jaur‘es, 78340 Les Clayes sous Bois, 1994.

[44] Jeff Bezanson et al. “Julia: Dynamism and Performance Reconciled by Design”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). URL: <https://doi.org/10.1145/3276490>.

[45] Jeff Bezanson, Jake Bolewski, and Jiahao Chen. *Fast Flexible Function Dispatch in Julia*. 2018. arXiv: <1808.03370> [cs.PL].

[46] *Introducing Julia Metaprogramming*. URL: <https://en.wikibooks.org/wiki/Introducing%5C_Julia/Metaprogramming> (visited on 03/09/2021).

[47] *SIMD and SIMD-intrinsics in Julia*. URL: <http://kristofferc.github.io/post/intrinsics/> (visited on 11/20/2019).

[48] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018. ISBN: 198508659X.

[49] *BenchmarkTools*. <https://juliaci.github.io/BenchmarkTools.jl/dev/>. (Visited on 09/27/2021).

[50] *oneAPI.jl*. <https://github.com/JuliaGPU/oneAPI.jl>. (Visited on 04/02/2023).

[51] *OpenCL.jl*. <https://github.com/JuliaGPU/OpenCL.jl>. (Visited on 04/02/2023).

[52] Robert D Blumofe et al. “Cilk: An efficient multithreaded runtime system”. In: *Journal of parallel and distributed computing* 37.1 (1996), pp. 55–69.

[53] Alexey Kukanov and Michael J Voss. “The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks.” In: *Intel Technology Journal* 11.4 (2007).

[54] Guy E Blelloch, Phillip B Gibbons, and Yossi Matias. “Provably efficient scheduling for languages with fine-grained parallelism”. In: *Journal of the ACM (JACM)* 46.2 (1999), pp. 281–321.

[55] Shimin Chen et al. “Scheduling threads for constructive cache sharing on CMPs”. In: *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. 2007, pp. 105–115.

[56] *Transducers*. <https://clojure.org/reference/transducers>. (Visited on 02/08/2023).

[57] *Transducers.jl*. <https://github.com/JuliaFolds/Transducers.jl>. (Visited on 02/08/2023).

[58] *FoldsThreads.jl*. <https://github.com/JuliaFolds/FoldsThreads.jl>. (Visited on 02/08/2023).

[59] Arch Robinson. *A Primer on Scheduling Fork-Join Parallelism with Work Stealing*. Tech. rep. N3872. Intel, Corp., Jan. 2014. URL: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf>.

[60] Kristoffer Carlsson. *SIMD and SIMD-intrinsics in Julia*. <http://kristofferc.github.io/post/intrinsics/>.

[61] Intel Articles. *Vectorization in Julia*. <https://software.intel.com/content/www/us/en/develop/articles/vectorization-in-julia.html>.

[62] *Julia Documentation: Essentials - @simd*. <https://docs.julialang.org/en/v1/base/base/\#Base.SimdLoop.@simd>.

[63] *Julia Documentation: Multi-dimensional Arrays - Broadcasting*. <https://docs.julialang.org/en/v1/manual/arrays/\#Broadcasting>.

[64] *SIMD.jl*. <https://github.com/eschnett/SIMD.jl>.

[65] Olga Pearce. "Load Balancing Scientific Applications". PhD thesis. Texas A&M University, 2014.

[66] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee. 2009, pp. 44–54.

[67] *GPU-Rodinia Repository*. <https://github.com/yuhc/gpu-rodinia/tree/master/openmp>. (Visited on 03/22/2023).

[68] Raphaël Bolze et al. "Grid'5000: a large scale and highly reconfigurable experimental grid testbed". In: *The International Journal of High Performance Computing Applications* 20.4 (2006), pp. 481–494.

[69] Stephen Warshall. "A theorem on Boolean matrices". In: *Journal of the ACM* 9 (1962), pp. 11–12.

[70] *Rosetta@home - CPU performance*. <https://boinc.bakerlab.org/rosetta/cpu_list.php>. Baker Lab Institute for Protein Design, University of Washington. (Visited on 05/13/2023).

[71] *A simple SIMD.jl loop that is slower than a vanilla '@inbounds @simd'*. <https://discourse.julialang.org/t/a-simple-simd-jl-loop-that-is-slower-than-a-vanilla-inbounds-simd/63655>. (Visited on 11/18/2021).

[72] *A simple SIMD.jl loop that is slower than a vanilla '@inbounds @simd'*. <https://discourse.julialang.org/t/a-simple-simd-jl-loop-that-is-slower-than-a-vanilla-inbounds-simd/63655/6>. (Visited on 11/18/2021).

[73] *Auto-vectorization in GCC*. <https://gcc.gnu.org/projects/tree-ssa/vectorization.html>. (Visited on 01/24/2023).

[74] *GCG*. <https://gcg.or.rwth-aachen.de>.

[75] *DIP*. <https://projects.coin-or.org/Dip>.

[76] *SCIP*. <https://scipopt.org>.

[77] *COIN-OR*. <https://github.com/coin-or/COIN-OR-OptimizationSuite>.

[78] *JuliaOpt*. <https://www.juliaopt.org/packages/>. (Visited on 08/28/2020).

[79] Iain Dunning, Joey Huchette, and Miles Lubin. "JuMP: A modeling language for mathematical optimization". In: *SIAM Review* 59.2 (2017), pp. 295–320.

[80] J. E. Beasley. *OR Lib*. <http://people.brunel.ac.uk/~mastjjb/jeb/info.html>. (Visited on 08/28/2020).