Impact of Programming Language Choice and Code Optimisation on Computational Efficiency: A Study on the NEH Heuristic

Manuel Canedo Tabares

***Abstract*— This paper scrutinises the impact of programming language selection and code optimisation on the execution speed of the Nawaz, Enscore, and Ham heuristic (NEH) for the Permutation Flow Shop Problem (PFSP). We start with a Python implementation of the NEH algorithm from the literature and translate it into Julia, a language providing enhanced control over low-level performance features, achieving a significant speedup. We further optimise this Julia version and translate the improved code into a finely-tuned C++ implementation, outperforming both previous versions. Our findings underscore the substantial benefits of informed language selection and meticulous code optimisation in computationally demanding fields such as Optimisation and Artificial Intelligence.**

I. INTRODUCTION

The Permutation Flow Shop Problem (PFSP) is a prevalent issue in various industrial contexts, such as manufacturing, logistics, and project scheduling. Given the problem's combinatorial nature and the escalating complexity with increasing number of jobs and machines, algorithms like the Nawaz, Enscore, and Ham (NEH) heuristic are essential for finding near-optimal solutions within reasonable timeframes.

In this paper, we examine the impact of programming language selection and code optimisation on the performance of the NEH PFSP algorithm. We optimise an implementation from existing literature, identifying and addressing performance bottlenecks using code optimisation techniques. These techniques include preallocating memory for data structures, avoiding conditional branches in computationally intensive code paths, utilising optimised built-in functionalities of the language, and eliminating unnecessary work.

We apply these optimisation techniques to two implementations: one in Julia and another in C++, allowing further fine-tuning by leveraging cache locality and vectorisation. We aim to demonstrate that these optimisation strategies can significantly boost the performance of the NEH heuristic for the PFSP.

This work's significance is further highlighted by the recent introduction of the Mojo programming language, a Python superset, designed for machine learning and AI applications. Mojo aims to combine the ease of use of Python with the low-level performance optimisation potential of languages like Julia and C++. This development suggests that the skills and knowledge required to extract the most from this emerging language extend beyond standard high-level programming practices. The implementation and optimisation techniques discussed in this paper are expected to directly apply to programming in Mojo.

The remainder of this paper is structured as follows:

- We first delve into the State of the Art, discussing the Julia and C++ languages, the NEH heuristic, and related work that combines these elements.
- Our Methodology section explains the study in detail, including the reference and optimised Julia implementations, the optimised C++ implementation, and the experimental setup.
- The Results section presents and analyses the performance comparison of the different implementations.
- Finally, we draw our conclusions, summarise the findings, and suggest future work in the Conclusion section.

II. STATE OF THE ART

Julia is a high-performance, dynamic programming language for numerical computing and data science [1]. One of its key features is its Just-in-Time (JIT) compiler based on the Low-Level Virtual Machine (LLVM) toolchain, which can compile and execute code at runtime, allowing for code optimisation and fast execution times. The JIT compiler uses type inference and whole-program analysis to generate efficient machine code. Also, it includes support for parallel and vectorised operations, making it well-suited for high-performance computing tasks. Julia has been gaining traction in AI and Optimisation, with libraries such as Flux.jl for machine learning and JuMP.jl for mathematical programming testifying its potential.

C++ is a high-performance, general-purpose language widely used in systems programming, scientific computing, and game development. One of its key features is its ability to achieve fine-grained governance over system resources through manual memory management and explicit control over hardware while offering zero-cost high-level abstractions. This makes it well-suited for applications that require maximum performance and extensive scalability. C++ has been a staple in performance-critical AI applications, from game development to on-device AI. Its use is exemplified by its use in the popular machine learning library TensorFlow. Additionally, many high-performance Optimisation solvers rely on C++ for its speed and control over hardware.

Recently, the introduction of the Mojo programming language, a high-performance JIT compiled language, has been a significant development in the AI and ML programming landscape [2]. Born out of a need for a scalable programming model for AI, Mojo aims to provide Python-like simplicity while addressing Python's limitations, including poor performance. Mojo is designed as a superset of Python, aiming for complete compatibility with the Python ecosystem.

The NEH heuristic is a well-established method for solving the PFSP. It is used across various fields, from manufacturing to cloud computing, because it can produce high-quality solutions within reasonable computational time. However, like any heuristic, it does not guarantee an optimal solution and may be sensitive to the initial ordering of the jobs. The key idea of the NEH heuristic is to start with an empty schedule and add jobs one by one in a non-increasing order of the sum of processing times on all machines while keeping track of the makespan (the total completion time of the last job) at each step [4]. One way to improve the NEH heuristic is by using Taillard's acceleration [5], a modification that allows for more efficient solution-space exploration, improving the performance of the algorithm.

We carried out a systematic literature review on the topic of implementing the NEH heuristic for the PFSP in various programming languages. Despite a wealth of literature on the NEH heuristic and its applications, papers have yet to directly compare its implementation in different programming languages or focus on the impact of code optimisation on performance. This literature gap underscores the current study's originality and significance.

III. METHODOLOGY

This section details the methodology used to investigate the influence of programming language selection and code optimisation on the performance of the NEH heuristic for the PFSP. The process begins with discussing the initial NEH heuristic implemented in Python, followed by a translation and subsequent optimisation in Julia. Next, the optimised Julia code is reimplemented in C++, further exploiting the capabilities of this language. Finally, we present the experimental setup devised to evaluate the performances of these implementations comparatively.

A. The reference Python implementation

The starting point of our study is an existing Python-based implementation of the NEH heuristic, which is part of a metaheuristic that combines the NEH with an Iterated Local Search procedure [6]. This Python implementation was chosen due to its straightforward and well-documented code, which offers a clear view of the algorithm's operation.

Python's high-level syntax and many libraries make it popular for prototyping and developing complex algorithms. However, these benefits can also be drawbacks when it comes to performance. While simplifying development, Python's dynamic typing, abstract object representation and garbage collection can create overhead during execution. This particular implementation of the NEH heuristic also has several potential performance bottlenecks. The following sections will detail how we addressed these performance issues when reimplementing and optimising the algorithm in Julia and C++.

B. The base Julia implementation

The base implementation of the NEH heuristic in Julia is a direct translation of the Python-based heuristic. Upon analysing and profiling this implementation, we identified several inefficiencies and bottlenecks:

1) Memory is dynamically allocated for data structures at every algorithm iteration.
2) Numerous conditional branches are created in the most computation-intensive loops of the core matrix calculation function. These include boundary checks to emulate Python's negative indexing, a feature that allows accessing array elements from the end using negative indices.
3) There is excessive use of matrix operators, leading to redundant calculations and memory allocation overhead.

To illustrate these points, consider the 'insertJobIntoSequence' function from the base Julia implementation below. This function calculates and updates the earliest, tail, and relative completion times, identifies the position of minimum makespan, and inserts a job into the sequence:

```
function insertJobIntoSequence(solution, inputs, k, kJob)
    # Create earliest, tail, and relative completion times structures
    n = length(solution.jobs)
    e = PythonLikeArray(n + 2, inputs.nMachines + 1)
    q = PythonLikeArray(n + 2, inputs.nMachines + 1)
    f = PythonLikeArray(n + 2, inputs.nMachines + 1)

    # Compute earliest, tail, and relative completion times values
    for j = 1:inputs.nMachines
        for i = 1:n+1
            if i < n + 1
                e[i, j] =
                    max(e[i, j-1], e[i-1, j]) +
                    inputs.times[solution.jobs[i], j]
            end
            if i > 1
                q[n+2-i, inputs.nMachines+1-j] =
                    max(
                        q[n+2-i, inputs.nMachines+2-j],
                        q[n+3-i, inputs.nMachines+1-j],
                    ) + inputs.times[solution.jobs[n+2-i], inputs.nMachines+1-j]
            end
            f[i, j] = max(f[i, j-1], e[i-1, j]) + inputs.times[kJob, j]
        end
    end

    # Find position of minimum makespan
    Mi = maximum(f.data + q.data, dims = 2)[1:end]
    index = argmin(Mi[1:min(k, end)])

    # Insert job in the sequence and update makespan
    insert!(solution.jobs, index, kJob)
    solution.makespan = Mi[index]
end
```

In this function, the memory allocations for the E, Q, and F matrices occurs at every call. All three matrices are populated in a single nested for-loop, specifying a different behaviour based on the iteration index. Finally, the built-in matrix operations used to calculate the index that minimises the makespan are performed in multiple whole matrix iterations, unnecessarily.

C. The optimised Julia implementation

To improve the performance of the base implementation, several optimisation techniques were applied:

- **Memory Allocation:** The matrices are preallocated and reused among iterations. With this approach, there is just a need to allocate two matrices, as the same memory used to calculate matrix E can be reused by matrix Q.
- **Function Decomposition:** The optimised implementation decomposes complex tasks into separate, well-defined functions ('populate_e¡, 'populate_q¡, 'populate_f¡). This approach increases code readability and maintainability and aids the JIT compiler perform its optimisations.

- **Matrix Iteration:** Many conditional branches in the core matrix iteration are avoided by populating the matrices section by section. This results in a better cache locality, an improved implicit vectorisation and facilitated work of the processor's branch predictor.
- **Arithmetic:** The reference implementation uses floating-point arithmetic. The profiling proved this to be slower than integer arithmetic for this current implementation.
- **Type Specification:** Explicitly specifying the type of the variables allows the compiler to make assumptions to speed up the execution.
- **Manual Matrix Operations:** Avoiding certain built-in matrix operations and managing matrix iterations manually resulted in performance gains. Built-in matrix operations may allocate memory to hold temporaries, slowing down the execution.

```
function populate_e!(
    jobs::Vector{Int},
    inputs::Inputs,
    index::Int,
    e::Array{Int},
)
    @inbounds begin
        e[1, 1] = inputs.times[jobs[1], 1]

        for j = 2:inputs.nMachines
            e[1, j] = inputs.times[jobs[1], j] + e[1, j-1]
        end
        for i = 2:index
            e[i, 1] = inputs.times[jobs[i], 1] + e[i-1, 1]
        end
        for j = 2:inputs.nMachines
            for i = 2:index
                e[i, j] = inputs.times[jobs[i], j] + max(e[i-1, j], e[i, j-1])
            end
        end
    end
end

function populate_q!(
    jobs::Vector{Int},
    inputs::Inputs,
    index::Int,
    q::Array{Int},
)
    @inbounds begin
        for j = inputs.nMachines:-1:1
            q[index, j] = 0
        end
        if index < 2
            return
        end
        q[index-1, inputs.nMachines] =
            inputs.times[jobs[index-1], inputs.nMachines]

        for j = inputs.nMachines-1:-1:1
            q[index-1, j] = inputs.times[jobs[index-1], j] + q[index-1, j+1]
        end
        for i = index-2:-1:1
            q[i, inputs.nMachines] =
                inputs.times[jobs[i], inputs.nMachines] +
                q[i+1, inputs.nMachines]
        end
        for j = inputs.nMachines-1:-1:1
            for i = index-2:-1:1
                q[i, j] = inputs.times[jobs[i], j] + max(q[i+1, j], q[i, j+1])
            end
        end
    end
end

function populate_f!(
    kJob::Int,
    inputs::Inputs,
    index::Int,
    e::Array{Int},
    f::Array{Int},
)
    @inbounds begin
        f[1, 1] = inputs.times[kJob, 1]

        for j = 2:inputs.nMachines
            f[1, j] = inputs.times[kJob, j] + f[1, j-1]
        end
        for i = 2:index
            f[i, 1] = inputs.times[kJob, 1] + e[i-1, 1]
        end
        for j = 2:inputs.nMachines
            for i = 2:index
                f[i, j] = inputs.times[kJob, j] + max(e[i-1, j], f[i, j-1])
            end
        end
    end
end
```

```
function insertJobIntoSequence(
    solution::Solution,
    inputs::Inputs,
    k::Int,
    kJob::Int,
    eq::Array{Int},
    f::Array{Int},
)
    # Compute earliest, tail, and relative completion times structures
    n = length(solution.jobs)
    populate_e!(solution.jobs, inputs, n, eq)
    populate_f!(kJob, inputs, n + 1, eq, f)
    populate_q!(solution.jobs, inputs, n + 1, eq)

    # Find position of minimum makespan
    index = k
    solution.makespan = typemax(Int)
    @inbounds for i = 1:k
        max_sum = 0
        for j = 1:inputs.nMachines
            max_sum = max(f[i, j] + eq[i, j], max_sum)
        end
        if max_sum < solution.makespan
            index = i
            solution.makespan = max_sum
        end
    end

    # Insert job in the sequence and update makespan
    insert!(solution.jobs, min(index, n + 1), kJob)
end
```

D. The optimised C++ implementation

The performance considerations used to optimise the base Julia implementation are not language dependent. These are general considerations that should apply to any language. These considerations are fundamental to building an efficient C++ implementation; without them, the Julia-optimised code would outperform C++. To make a C++ implementation at least as efficient as the optimised Julia implementation, several low-level factors were taken into account:

- **Data Layout:** Initially, a row-major memory distribution was used for the matrices. On the contrary, Julia uses a column-major layout, decreasing the number of cache fetches the processor has to perform while performing matrix multiplication. Implementing this layout in C++ yielded a performance increase. However, this layout was outperformed in C++ by using sparse memory: allocating an individual memory block for every row. This strategy aligns data to reduce cache fetches and improve cache utilisation.
- **Aiding the Compiler:** After improving the cache locality of data and not having conditional branches in the matrix iteration functions, the performance bottleneck became the intensive use of the Standard Template Library function std::max. The boilerplate code used inside this function to ensure type generality did not allow the compiler to optimise the call fully. This was solved by implementing a simple inline function to calculate the maximum. In addition, using equality operators instead of less/greater-than operators proved beneficial as they eliminated the need for branching, allowing the compiler to generate fewer instructions.
- **Vectorisation:** The heavily simplified layout allowed the compiler to apply better implicit vectorisation to loops. Compilation reports show that only two loops could not benefit from implicit vectorisation. Investigating how these loops can be optimised for vectorisation is a potential avenue for future work.
- **Compilation Flags:** Many compilation flags and configurations were evaluated, finally settling on using

GCC's safe performance optimisations (-O3) and performance tuning for the platform executing the compilation (-march=native), as well as support for C++20 (-std=c++20).

E. Experimental Setup

To compare the performance of the reference Python, the base Julia, the optimised Julia and the optimised C++ implementations, a set of six problem instances from Taillard's benchmark instances were selected. These instances are widely recognised in the literature for their comprehensive coverage of different facets of the PFSP problem, making them a suitable choice for thorough performance analysis. Each algorithm was executed a hundred times for each instance, and the following runtime statistics were collected for detailed performance comparison: mean execution time, maximum execution time, and minimum execution time.

The runtime for each instance was measured from the initiation of the algorithm to the moment the final solution was produced. To maintain a consistent testing environment, all tests were conducted on the same machine - an x86 AMD Ryzen™ 9 5950X - running Ubuntu 20.04. Python (version 3.9.13), Julia (version 1.9) and GCC (version 12) were used, ensuring accurate and comparable results across all tests.

IV. Results

A. Problem Instance Results

The instances tested, including the number of jobs and machines in their name, and the solutions obtained by each implementation are summarised in Table I. The 'Python' column represents the solutions found using the Python implementation, the 'Julia' column indicates the solutions found using the unoptimised Julia implementation, the 'Julia Opt.' column indicates the solutions found using the optimised Julia implementation, and the 'C++ Opt.' column indicates the solutions found using the optimised C++ implementation.

Instance	Python	Julia	Julia Opt.	C++ Opt.
tai031_50_5	2733	2733.0	2733.0	2733
tai071_100_10	5897	5846	5846.0	5881
tai081_100_20	6573	6541	6541.0	6594
tai091_200_10	10955	10942	10942	10942
tai101_200_20	11651	11594	11594	11684
tai117_500_20	26803	26797	26797	26854

TABLE I
TAILLARD INSTANCES AND BEST SOLUTIONS OBTAINED WITH EACH IMPLEMENTATION

The Python implementation calls NumPy's 'np.argsort', which uses a variation of Quick Sort. The Julia implementations call 'sortperm', which uses a variation of the Merge Sort algorithm. In contrast, C++ uses a variant of Muser's Introsort in its call to std::sort from the Standard Template Library's Algorithm header. The difference in the initial ordering of jobs, caused by the varying sorting algorithms, accounts for the slight variations in the results found between implementations.

B. Python Results

The execution times measured for the reference Python implementation are summarised in Table II. From the Python results, it is evident that as the complexity of the problem increases (i.e., more jobs and machines), the execution time also increases significantly. This sets the stage for comparing these times to those obtained from the Julia and C++ implementations.

Instance	Average (ms)	Minimum (ms)	Maximum (ms)
tai031_50_5	9.44	8.94	10.41
tai071_100_10	72.96	71.54	77.36
tai081_100_20	142.26	136.23	151.30
tai091_200_10	275.05	267.24	282.23
tai101_200_20	563.05	523.88	566.46
tai117_500_20	3321.79	3274.64	3387.21

TABLE II
TAILLARD INSTANCES AND EXECUTION TIMES FOR THE REFERENCE PYTHON IMPLEMENTATION

C. Julia Results

The execution times measured for the base Julia implementation are summarised in Table III. These results represent a massive improvement in execution times compared to the Python implementation, with the most demanding problem seeing an average execution speed-up of 113.41x.

Instance	Average (ms)	Minimum (ms)	Maximum (ms)
tai031_50_5	1.01	0.13	77.48
tai071_100_10	1.57	0.69	33.72
tai081_100_20	1.33	1.16	23.92
tai091_200_10	2.61	2.36	3.85
tai101_200_20	4.68	4.34	6.82
tai117_500_20	29.29	27.57	66.06

TABLE III
TAILLARD INSTANCES AND EXECUTION TIMES (MS) FOR THE BASE JULIA IMPLEMENTATION

The execution times measured for the optimised Julia implementation are shown in Table IV. These results represent a significant improvement in execution times, with the most demanding problem seeing an average execution speed-up of 5.45x and the least demanding one seeing a minimum execution time speed-up of 5.12x, compared to the base Julia implementation (618.58x and 364.94x compared to Python). These gains are primarily attributable to the optimisation strategies that focused on improving memory allocation, redesigning matrix iteration, implementing integer arithmetic, judicious use of built-in Julia functions, and reducing conditional branches.

This section's results highlight the potential of Julia as a high-performance language for technical computing, capable of delivering impressive speed-ups when optimised correctly. It is important to note that different problem instances benefited variably from the optimisations, indicating that the effectiveness of such optimisations may depend on the problem size and complexity.

Instance	Average (ms)	Minimum (ms)	Maximum (ms)
tai031_50_5	0.37	0.02	34.71
tai071_100_10	0.15	0.14	0.20
tai081_100_20	0.34	0.30	0.51
tai091_200_10	0.50	0.49	0.66
tai101_200_20	1.04	0.98	1.32
tai117_500_20	5.37	5.21	6.33

TABLE IV

TAILLARD INSTANCES AND EXECUTION TIMES (MS) FOR THE OPTIMISED JULIA IMPLEMENTATION

D. C++ Results

The execution times measured for the optimised C++ implementation are shown in Table V. These results represent a further improvement in execution times, with the most demanding problem seeing an average execution speed-up of 1.38x compared to the optimised Julia implementation (851.74x compared to Python).

Instance	Average	Minimum	Maximum
tai031_50_5	0.02	0.01	0.04
tai071_100_10	0.08	0.08	0.09
tai081_100_20	0.16	0.16	0.18
tai091_200_10	0.30	0.30	0.32
tai101_200_20	0.64	0.63	0.66
tai117_500_20	3.90	3.88	3.98

TABLE V

TAILLARD INSTANCES AND EXECUTION TIMES FOR THE OPTIMISED C++ IMPLEMENTATION (MS)

The further improvement in execution times observed in the optimised C++ implementation can be primarily attributed to two factors. First, unlike Python or Julia, C++ allows building for a specific platform, enabling better implicit vectorisation. Secondly, C++ provides greater control over the memory layout of data structures. In the case of the NEH heuristic, it was possible to modify the memory layout of the matrix implementation to a form more suited to the specific access patterns of the heuristic. An efficient memory layout can reduce cache misses and improve execution times, as data retrieval from main memory is orders of magnitude slower than from the different levels of cache. This level of control over hardware resources is a feature of low-level languages like C++. Although it requires a deeper understanding of the algorithm and the hardware, it can lead to significant performance enhancements.

The ability to perform such customised optimisations is a compelling reason to use lower-level languages like C++ for computationally intensive tasks, despite the steeper learning curve and increased code complexity compared to higher-level languages like Python and Julia.

Figure 1 represents each implementation's average completion time per instance, plotted over a logarithmic Y axis. It helps visualise how each graph is translated in the execution time axis due to the optimisations while maintaining the same asymptotic growth.

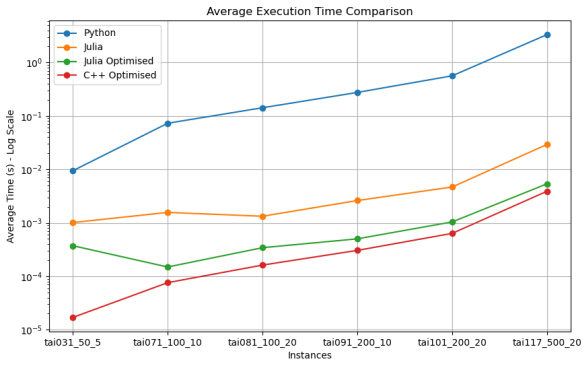

Fig. 1. Logarithmic execution times

V. CONCLUSION

This study re-implemented and optimised the NEH heuristic for solving the Permutation Flowshop Scheduling Problem (PFSP). The reference heuristic implemented in Python was translated to Julia and C++. The limitations of Python, particularly its poor low-level performance, underscore the need for more efficient languages in computationally intensive tasks such as the PFSP.

Our results demonstrated that Julia provided an impressive average speed-up of 109.75x on the most demanding problem instance compared to Python. Upon optimisation, Julia showed its true potential as a high-performance language, offering an additional average execution speed-up of 2.43x,

The optimised C++ implementation outperformed the optimised Julia implementation by providing an average execution speed-up of 2.17x on the most demanding problem instance. This underscores that while a lower-level language like C++ may require a steeper learning curve and longer development times, the performance gains can be substantial. The C++ implementation aimed to be consistent with the optimised Julia implementation. There is room for further tuning to the C++ implementation to achieve additional benefits.

The recent release of Mojo, a language that aims to offer Python simplicity with low-level control and high-performance capabilities, introduces exciting new opportunities for the field. This study's findings and the performance considerations applied to the Julia and C++ implementations are directly relevant to the new language. The lessons learned from this study could be applied to the use of Mojo, contributing to the maximisation of its performance potential.

In conclusion, this study emphasises the critical role of careful programming language selection and efficient coding practices in computational tasks. It sheds light on the potential of high-performance languages like Julia and C++ to significantly enhance the performance of AI and optimisation problems when leveraged correctly. Future research should explore the performance of the NEH heuristic implemented in Mojo, providing further insights into the impact of the choice of programming language and code optimisation on

computational efficiency.

APPENDIX

All the code developed for this study can be publicly found in the repository below:

`https://github.com/ManuCanedo/icso-neh`

The author encourages everyone interested in the topic to use and contribute to the repository.

REFERENCES

[1] Julia Language Documentation. (2023). Julia v1.9 Documentation. Available at: `https://docs.julialang.org/en/v1/` [Accessed: June 4, 2023]

[2] Modular. (2023). Why Mojo? A New Language for Machine Learning. Available at: `https://docs.modular.com/mojo/why-mojo.html` [Accessed: June 4, 2023]

[3] Fernandez-Viagas, V., Framinan, J. M. (2015). NEH-based heuristics for the permutation flowshop scheduling problem to minimize total tardiness. Computers & Operations Research, 60, 27-36.

[4] Nawaz, M., Enscore Jr, E. E., & Ham, I. (1983). A Heuristic Algorithm for the m-Machine, n-Job Flow-shop Sequencing Problem. Omega, The International Journal of Management Science, 11(1), 91-95.

[5] Taillard, E. (1990). Some efficient heuristic methods for the flow shop sequencing problem. European Journal of Operational Research, 47(1), 65-74.

[6] Juan, A. A., Lourenço, H. R., Mateo, M., Luo, R., & Castellà, Q. (2014). Using iterated local search for solving the flow-shop problem: Parallelization, parametrization, and randomization issues. International Transactions in Operational Research, 21, 103-126.