A comparative study of high-productivity high-performance programming languages for parallel metaheuristics

Jan Gmys, Tiago Carneiro, Nouredine Melab, El-Ghazali Talbi, Daniel Tuyttens

A comparative study of high-productivity high-performance programming languages for parallel metaheuristics

Jan Gmys[c,*], Tiago Carneiro[b], Nouredine Melab[a,b], El-Ghazali Talbi[a,b], Daniel Tuyttens[c]

[a] *Université de Lille, CNRS/CRIStAL, France*
[b] *INRIA Lille - Nord Europe, France*
[c] *Mathematics and Operational Research Department (MARO), University of Mons, Belgium*

Abstract

Parallel metaheuristics require programming languages that provide both, high performance and a high level of programmability. This paper aims at providing a useful data point to help practitioners gauge the difficult question of whether to invest time and effort into learning and using a new programming language. To accomplish this objective, three productivity-aware languages (Chapel, Julia, and Python) are compared in terms of performance, scalability and productivity. To the best of our knowledge, this is the first time such a comparison is performed in the context of parallel metaheuristics. As a test-case, we implement two parallel metaheuristics in three languages for solving the 3D Quadratic Assignment Problem (Q3AP), using thread-based parallelism on a multi-core shared-memory computer. We also evaluate and compare the performance of the three languages for a parallel fitness evaluation loop, using four different test-functions with different computational characteristics. Besides providing a comparative study, we give feedback on the implementation and parallelization process in each language.

Keywords: Metaheuristics, Parallel metaheuristics, High-productivity languages, Parallel computing

*Corresponding author
Email address: jan.gmys@umons.ac.be (Jan Gmys)

1. Introduction

Optimization problems in economic and industrial applications, for instance in logistics, telecommunications or bioinformatics are often complex, NP-hard and time-consuming. In contrast to exact methods, which usually require excessive computational efforts, metaheuristics aim at providing high-quality solutions in a reasonable amount of time. Nevertheless, the computation time needed to reach near-optimal solutions can still be high, and parallel computing has been recognized as an efficient means to improve the solution quality and/or reduce the running time of metaheuristics [1, 2].

Therefore, when choosing a programming language for the implementation of a (framework of) metaheuristic(s), performance and the availability of parallel computing tools are important criteria to consider. As high-performance computing (HPC) systems tend to be increasingly complex and heterogeneous, these systems become more and more difficult to program [3]. Therefore, HPC builders and researchers are designing new programming approaches (e.g. Chapel, UPC, X10) which aim at increasing programmability while maintaining the performance of classical environments (e.g. MPI+X) [4]. At the same time, metaheuristics are, per definition, destined to be applied to a variety of problems, which means that implementations should aim to be easy-to-use and extensible for users coming from different fields of application. In order to solve complex problems, a high level of interoperability with machine learning and statistical tools is needed. Moreover, adapting metaheuristics to a given problem often requires a considerable amount of parameter tuning and adjustment of algorithm components (genetic operators, neighborhood structures, selection criteria, etc.). It is commonly acknowledged that dynamic programming languages (e.g., Python, Julia, Matlab) greatly facilitate these tasks and enable fast prototyping [5]. However, these languages historically suffer from severe performance penalties and lack support for efficient parallel computing. In order to avoid the highly time-consuming process of re-programming all or parts

of an existing program in a more low-level language, several projects aim at increasing the performance of dynamic languages, either by proposing extensions (e.g., Numba for Python) or new languages (e.g., Julia).

To make an informed decision considering the trade-off between these two conflicting objectives, empirical evidence regarding the relative merits of different programming languages is needed. However, while the literature provides abundant experimental comparisons between different algorithms, it is very hard to find information regarding the performance of the same algorithm implemented in different languages. More generally, as noted in the perspectives of a 2013 survey on trends in parallel metaheuristics by Alba *et al.*, an "[...] interesting topic for future research is to explicitly address the balance between software usability and efficiency when dealing with parallel metaheuristics" [6].

In various domains, researchers have conducted empirical studies comparing the usefulness of different programming languages for a typical problem from their respective fields of research. For instance, in [7], a comparison of statically compiled languages (Fortran, C++, Java) and dynamically interpreted languages (Python, Matlab, Julia) applied to classical astrodynamics problems is proposed. In computational economics, [8] proposes a comparative study of C++, Fortran, Java, Julia, Python, Matlab, Mathematica and R which aims at providing "a measure of the "benefit" in a cost-benefit calculation for researchers who are considering learning a new language". The suitability of six programming languages for algorithms relevant to bioinformatics is studied in [9]. In [10], a library for mixed-integer algebraic modeling written in Julia (JuMP), is proposed and compared to similar libraries written in "highly expressive yet typically slow high-level languages" (Python and Matlab) and "highly efficient yet typically cumbersome low-level languages" (C++, Fortran). We have not found any works that compare programming languages in the context of parallel metaheuristics.

With this article, we aim at providing a useful data point to help practitioners in the field of parallel metaheuristics gauge the difficult question of whether to invest time and effort into learning and using a new programming

language. We compare the (parallel) performance, scalability, and productivity of three programming languages (Chapel, Python and Julia) for two metaheuristics (one trajectory-based and one population-based) applied to the quadratic 3-dimensional assignment problem (Q3AP). The Q3AP is used as a test-case because metaheuristic solution approaches are time-consuming [11] and parallel computing is highly recommended to tackle the problem [12, 13, 14]. Targeting shared-memory multi-core systems, two key building blocks of parallel metaheuristics, parallel neighborhood exploration and parallel population fitness evaluation are implemented using thread-based parallelism. In order to analyze the impact of the batch-size (neighborhood or population size) and computational characteristics of the fitness function on the performance of these building blocks, we report experimental results for four additional combinatorial optimization problems. Chapel, Julia and Python (using Numpy and Numba) are designed to provide high-performance and a high-level of expressiveness simultaneously, so we chose these languages as *a priori* candidates to lie on a "Pareto front" of optimal trade-offs between performance and programmability. For comparison with the "gold standard" in terms of performance, an implementation in C/OpenMP is used. All source codes for our experiments are made available through a github repository [15].

The main contributions and findings of this paper can be summarized as follows:

- We compare three high-performance and high-productivity languages in terms of scalability, performance, and productivity. To the best of our knowledge, this is the first time such a comparison is performed in the context of parallel metaheuristics.

- We provide feedback on the implementation experience in Python/Numba, Julia and Chapel, including encountered performance pitfalls and opportunities to improve execution speed. In particular, we describe how parallel neighborhood and parallel fitness evaluation, two key building blocks of parallel metaheuristics, can be implemented using thread-based paral-

lelism provided by the three languages.

- The reported experimental results show that, for sequential executions, the performance of the two interpreted languages can be in the same order of magnitude than C (2-3 times slower) while statically compiled one Chapel is equivalent to C in terms of performance.

- In terms of parallel performance, the multi-threaded loop-level parallelism provided by Python/Numba and Julia allows us to speed up computations, but their multi-threading support (experimental) is not yet mature enough to compete with OpenMP, especially for very fine-grained tasks. Chapel's task-based parallelism, on the other hand, scales nearly as well as optimized C/OpenMP.

- Two productivity models from the literature were applied. According to the results, the two interpreted languages present an interesting trade-off between implementation cost and performance only for sequential execution.

The remainder of this paper is organized as follows. In Section 2, we introduce the test-case used in this study, which consists of the Q3AP, an iterated local search (ILS) algorithm and a genetic algorithm (GA). In Section 3, we present the three productivity-aware languages used for the implementation of both algorithms. Section 4 provides information on the guidelines we followed to make the comparison as fair as possible, as well as details on the algorithm implementation, its parallelization, and the performance tuning of each implementation. Section 5 reports experimental results in three parts: Section 5.1 compares the different implementations of the ILS and GA algorithms in terms of performance and Section 5.2 in terms of productivity; in Section 5.3 we investigate the performance of multi-threaded parallel fitness evaluation, considering 8 test functions with different computational characteristics. A discussion of the results is provided in Section 6. Finally, in Section 7, we draw conclusions and outline possible future works.

2. Background

2.1. Problem formulation

The Quadratic 3-dimensional Assignment Problem (Q3AP) is a generalization of the well-known quadratic assignment problem (QAP) that was originally introduced in a technical memorandum in 1967 and was re-discovered by Hahn *et al.* in 2008 [11]. The Q3AP is defined by a 6-dimensional cost matrix and solutions can be represented as a pair of permutations (π, σ) of length n. The objective function to minimize is

$$f(s = (\pi, \sigma)) = \sum_{i=1}^{n} \sum_{j=1}^{n} C_{i\pi(i)\sigma(i)j\pi(j)\sigma(j)}$$

where C_{ijkpqr} designates the cost of assigning (i, p) to (j, q) and to (k, r). The search space for the Q3AP contains $n! \times n!$ feasible solutions and it is clear that the Q3AP, as a generalization of the QAP, is $\mathcal{NP}$-hard. For a more detailed description of the Q3AP and its application domain, we refer the reader to [11].

Following the proposal of [11], benchmark instances for the Q3AP used in the literature are derived from the QAP instances (QAPLIB, [16]) using the following relationship between the flow matrix F and distance matrix D:

$$C_{ijkpqr} = F_{ip}^2 \times D_{jq} \times D_{kr},\ i, j, k, p, q, r = 1, \ldots, n. \tag{1}$$

2.2. Review of metaheuristics for the Q3AP

Several approximate solution methods have been proposed for the Q3AP. In [11] simulated annealing (SA), robust tabu search (RoTS), fast ant colony algorithm (FANT) and iterated local search (ILS) algorithms for the Q3AP are introduced and compared to each other. All four stochastic local search algorithms are based on a 2-exchange neighborhood (in either π or σ) containing $\mathcal{O}(n^2)$ neighbors. Using instances up to size $n = 15$, experimental results indicate that ILS gives the overall best performance among the approximate solution methods. Furthermore, the pioneering experimental evaluation in [11]

shows that required computation times for reaching optimal or best-known solutions are orders of magnitude larger than for QAP instances of similar size. To address this issue, most subsequent works in the literature use parallel computing.

In [12] a GPU-based iterated tabu search (ITS) is proposed. The experimental results show that it is advantageous to use a very large neighborhood, containing $\mathcal{O}(n^4)$ neighbors obtained by *simultaneous* 2-exchange moves in both permutations π and σ. The fitness of neighbor solutions is evaluated incrementally and neighborhood evaluations are accelerated by offloading computations to a GPU. A parallel hybrid genetic algorithm (GA) including a SA local search in the mutation operator is presented in [14]. The parallelization model is hierarchical, using multiple GA islands across distributed computation nodes, and thread-based parallel neighborhood and population evaluations on the intra-node level. Similarly, the hybrid GA presented in [17] integrates local search in the evolution process; moreover it uses an auto-adaption mechanism to select suitable crossover and mutation operators and GPU-acceleration for the local search. For all these approaches the parallel evaluation of solutions, either full or incremental, is a key building block which can be efficiently implemented on top of shared-memory multi-core computers and GPUs. Therefore, we choose to implement both ILS and a hybrid GA-LS algorithm for the purpose of this comparative study. With some minor modifications (detailed in the following section) our implementations follow [11] for the ILS, and [14] for the hybrid GA.

2.3. Iterated local search (ILS) for the Q3AP

ILS is a simple yet powerful trajectory-based metaheuristic which generates a sequence of solutions by iterating through solution perturbations followed by a local improvement method [18]. An outline of the algorithm is shown in Algorithm 1. In what follows, we describe the perturbation and local search procedures used in our implementations.

Algorithm 1 Iterated Local Search (after [18])

```
1: procedure ILS(maxiter)
2:     s* ← generateInitialSolution()
3:     localSearch(s*)                                   ▷ improve s*
4:     for i ← 1, ..., maxiter do
5:         s_tmp ← perturb(s*)
6:         localSearch(s_tmp)
7:         s* ← keepBetter(s*, s_tmp)                    ▷ acceptanceCriterion
8:     end for
9: end procedure
```

Perturbation mechanism. In our ILS algorithm for the Q3AP, a perturbation consists in either perturbing π or σ (alternately) by randomly selecting k positions and shuffling the corresponding elements randomly. The perturbation strength k is initialized at $k \leftarrow 3$ and dynamically adjusted during the search. If a local search does not improve the previous local minimum, then the perturbation strength is increased ($k \leftarrow k + 1$). If $k = n$, then the perturbation corresponds to a random replacement of either π or σ and the perturbation strength is subsequently reset to the minimal strength ($k \leftarrow 3$).

Local Search (LS). The LS procedure uses a best improvement neighbor selection, *i.e.* for a solution all possible moves are tried to select the best neighboring solution. After the neighborhood evaluation the best move is applied and if no improving neighbor is found, the search stops. A pseudo-code of the local search procedure is shown in Algorithm 2.

Algorithm 2 Local Search

```
1: procedure LOCAL SEARCH(s)
2:     s.cost ← evalQ3AP(s)
3:     repeat
4:         for m ∈ moves do                              ▷ (in parallel)
5:             Δ ← evalDelta(s, m)
6:             if Δ > Δ_max then (Δ_max, m_best) ← (Δ, m)   ▷ (critical section)
7:         end for
8:         if Δ_max > 0 then applyMoveAndUpdateCost(s, m_best, Δ_max)
9:     until Δ_max ≤ 0
10:    return s
11: end procedure
```

We consider a local search that uses incremental evaluation of neighbor solutions and a large neighborhood that consists in jointly exchanging two positions in both permutation, as in [12]. Each move can be represented by four integers. We denote $\mathfrak{m}(s)$ the neighbor of s obtained by applying move and we denote $\mathfrak{m} = (a, b, c, d)$ the move that consists in exchanging $\pi(a)$ and $\pi(b)$ in the first permutation and $\sigma(c)$ and $\sigma(d)$ in the second permutation. The neighborhood of a solution $s = (\pi, \sigma)$ is defined by applying to s all possible moves

$$\mathcal{M} = \{(a, b, c, d) | 0 < a \leq b \leq n,\ 0 < c \leq d \leq n\} \tag{2}$$

Therefore, the size of the neighborhood is $\frac{n^2(n+1)^2}{4}$. It is possible to reduce the computational effort of neighborhood evaluations by computing costs incrementally. For a solution $s = (\pi, \sigma)$, the incremental fitness of a neighbor $\mathfrak{m}(s) = (\pi', \sigma')$ obtained by applying a move $\mathfrak{m} \in \mathcal{M}$ is given by

$$\Delta(s, \mathfrak{m}) = f(s) - f(\mathfrak{m}(s)) = \sum_{i=1}^{n} \sum_{j=1}^{n} (A_{ij} - B_{ij})$$

where we denote $A_{ij} = C_{i\pi(i)\sigma(i)j\pi(j)\sigma(j)}$ and $B_{ij} = C_{i\pi'(i)\sigma'(i)j\pi'(j)\sigma'(j)}$. We have

$$A_{ij} - B_{ij} = 0 \Leftrightarrow (i \notin \mathfrak{m} \quad \text{and} \quad j \notin \mathfrak{m}),$$

so $\Delta(s, \mathfrak{m})$ can be computed as follows:

$$\Delta(s, \mathfrak{m}) = \sum_{\substack{i=1 \\ i \in \mathfrak{m}}}^{n} \left(\sum_{j=1}^{n} (A_{ij} - B_{ij}) + \sum_{\substack{j=1 \\ j \notin \mathfrak{m}}}^{n} (A_{ji} - B_{ji}) \right) \tag{3}$$

In Equation 3 the subscripts $i \in \mathfrak{m}$ (resp. $j \notin \mathfrak{m}$) indicates that the sum is only computed for values of i (resp. j) that are (resp. are not) involved in the move. For instance, for move $(2, 3, 2, 4)$ the first sum ($i \in \mathfrak{m}$) is computed for $i =$ $2, 3, 4$ and the second inner sum ($j \in \mathfrak{m}$) is computed for $j = 1, \ldots, 5, 6, \ldots, n$.

In the worst case, one incremental fitness evaluation requires therefore

$$4 \times (2n + 2(n-4)) = 16n - 32$$

additions/subtractions and as many read-accesses to the 6D cost-matrix C. Moreover, the implementation of Equation 3 contains multiple `if-else` conditions with may lead to branch prediction misses.

Overall, the evaluation of a neighborhood requires $\mathcal{O}(n^5)$ steps. Even for small instances the computational cost of ILS is dominated by repeated call of the `evalDelta` function (Alg. 2, line 5). In order to speedup the LS, the neighborhood evaluation should be parallelized. If the neighborhood loop (Alg. 2, line 4) is performed in parallel by multiple threads, then the update of the most improving move and cost (Alg. 2, line 6) must be protected by some mutual exclusion mechanism. Depending on the programming language, it might be preferable to store the incremental costs in a temporary array which is subsequently searched for the maximum value.

2.4. Genetic algorithm (GA)

As a second test-case we consider a generational GA, hybridized with a local search in a similar way as proposed in [14], where a simulated annealing search is embedded in the mutation operator. An overview of the algorithm is shown in Algorithm 3.

The number of individuals is fixed to 100. Each iteration starts by passing the fittest individual to the next generation without undergoing mutation. This guarantees that the fitness of the best found solution is non-increasing from one generation to another. Parent individuals are selected according to fitness-proportionate random selection. A position-based crossover (POS) [19] is used. For two single-permutation parents p_1 and p_2, k positions are randomly selected in p_1 and copied to the corresponding positions in offspring q. The remaining positions of q are filled by taking the elements from parent p_2, in their order of appearance in p_2, but omitting the elements already present in q. For

Algorithm 3 GA-LS

```
1: procedure GA
2:     P_0 ← initialize Population                          ▷ Population size : 100
3:     for i ← 0, ..., #Generations do
4:         P_{i+1} ← P_{i+1} ∪ get-best-individual(P_i)                ▷ Elitism
5:         P_{i+1} ← P_{i+1} ∪ select-and-crossover(P_i)   ▷ Fitness proportionate/POS
6:         evaluate(P_{i+1})                                          ▷ (in parallel)
7:         for p ∈ P_{i+1} do
8:             if random(0, 1) < 0.3 then                             ▷ Mutate
9:                 if random(0, 1) < 0.7 then localSearch(p)
10:                else apply-random-move(p)
11:            end if
12:        end for
13:    end for
14: end procedure
```

Q3AP solutions, the POS-crossover is successively applied to the two permutations. The number of elements k directly copied from the first parent is chosen uniformly at random from in the integer interval $[3, N-3]$. Each generated offspring undergoes mutation with a probability 0.3. With a probability 0.7 individuals selected for mutation are mutated by performing a best-improvement local search as described in Section 2.3, a random transposition in both permutations is performed otherwise. The LS embedded in the mutation operator is parallelized as described in Section 2.3. In addition, the fitness evaluation (Alg. 3, line 6) is performed in parallel.

3. The Languages

In this section, we briefly introduce and compare the three productivity-aware languages used for implementing the metaheuristics: Chapel, Julia, and Python. A summarized comparison of these languages is provided in Table 1.

Table 1: Brief comparison of the four languages used in this work for programming the metaheuristics for solving instances of the Q3AP.

Language	Compiled/Interpreted	Type Checking
C-OpenMP	Compiled	static
Chapel	Compiled	static
Julia	Interpreted/JIT	dynamic
Python-Numba	Interpreted/JIT	dynamic

3.1. Chapel

Chapel (Cascade High Productivity Language) is an open-source parallel programming language designed to improve the productivity in highperformance computing [20]. It incorporates features from compiled languages such as Fortran, C, and C++, as well as high-level concepts related to Matlab and Python. In Chapel, the program is started with a single task, and parallelism is added through data or task-parallel features (incremental parallelism). The parallelism is expressed in terms of lightweight tasks, which can run on a single locale (multicore computing) or multiple locales (distributed computing). The term *locale* refers to a symmetric multiprocessing computer in a parallel system [21].

Previous versions of Chapel were not a suitable replacement for C or Fortran in terms of performance. Instead, they could be suitable replacements for Matlab and Python [22, 23]. The performance issues of Chapel were solved on release 1.18 (two releases ago), and nowadays, the language has become competitive to MPI+X, OpenMP and SHMEM in terms of performance, taking into account different benchmarks [24].

Chapel has been used for exact optimization [25, 26], in both multi-core and distributed settings. Concerning the latter, Chapel's performance is equivalent to C/OpenMP for parallel tree-based search algorithms. Moreover, it has the advantage of providing work stealing schemes, which are not implemented in OpenMP. Regarding the distributed scenario, the combination of incremental parallelism and global view of control flow and data structures makes it possible to code with low programming effort a distributed tree search algorithm that scales.

3.2. Julia

Julia is an open-source high-performance dynamically-typed programming language for technical computing. Its development started as a research project at the Massachusetts Institute of Technology (MIT) and the first public version was released in 2012. Julia is designed to be easy and fast: it aims at bridging the gap between statically typed languages like C, C++ and Fortran, the gold

standard languages for computationally-intensive problems, and dynamic languages like Python, Matlab or R, whose popularity in the scientific community has grown over the last years [27].

Julia has a high-level syntax that is easy to learn and can be used interactively from a console or an "interactive notebook" [28] thanks to the build-in read-eval-print-loops (REPL). The ecosystem provides several tools for visualization, machine learning, data science and other scientific domains. For performance, Julia relies on just-in-time (JIT) compilation using the LLVM compiler framework, on optional type annotations and on multiple dispatch—a technique that selects a specialized function implementation based on the function's arguments. As a modern high-performance language, Julia provides several facilities to support all levels of parallel computing: distributed computing, multi-threading, instruction-level parallelism and hardware accelerator devices such as GPUs. However, some of the parallel programming functionalities are still in an exploratory state. For instance, in the current version 1.2.0, the `Base.Threads` package used in this work is still experimental, according to the Julia homepage[1].

Experiences with micro-benchmarks [27] and more realistic applications [10] show that Julia can be as fast as code written in C, C++ or Fortran. However, as stated in the introduction of the Julia 1.2 documentation[2], to achieve this performance the programmer needs to "understand how Julia works".

3.3. Python

Python is an interpreted dynamic programming language that favors readability and a highly expressive syntax. First released in 1991, Python has become one of the most popular programming languages, according to available popularity indices[3]. It is considered to be highly productive, due to its clean and

[1] `https://julialang.org/`

[2] `https://docs.julialang.org/en/v1/index.html`

[3] TIOBE, `https://www.tiobe.com/tiobe-index/`; PYPL, `http://pypl.github.io/PYPL.html`

concise syntax and the large number of available libraries. Over the last years, Python has become increasingly popular in the scientific community [29], which is largely due to the availability of performance-oriented libraries like NumPy, SciPy, TensorFlow or scikit-learn.

However, for compute-intensive operations that cannot be accelerated using specialized libraries, pure interpreted Python can be very slow. Moreover, for CPU-bound tasks multi-threading is generally inefficient, because of Python's global interpreter lock (GIL). Therefore, several projects like PyPy, Cython, Numba and Nuitka aim at increasing Python's performance. PyPy is an alternative implementation of Python using JIT compilation. Cython is a superset of Python that is compiled to C or C++ and interfaced with Python. It allows to write C extensions of the most compute-intensive parts of a code. Similarly, Nuitka compiles Python to C or C++ source code. Similar to PyPy, Numba uses JIT compilation to translate subsets of Python and NumPy to fast machine code. However, instead of replacing the Python interpreter, Numba provides decorators that are inserted into the code to trigger the LLVM-based JIT compilation of selected functions. OpenMP-style loop-parallelism is supported via the insertion of decorators and Numba attempts to auto-parallelize several expressions and array operations (if the option is enabled). Numba allows to use either OpenMP, Intel Thread Building Blocks (TBB) or a build-in work-queue as threading layer. However, as of version 0.46, Numba's multi-threading support is still experimental.

In this work we chose Numba, because it offers options for accelerating and parallelizing Python code incrementally with only minor code changes. Moreover, the fact that Numba is supported by hardware manufacturers such as AMD, Intel and Nvidia is promising with regards to its future development. According to the official Numba website[4], Numba works best on code that uses NumPy arrays, functions and loops - which is the case of our ILS and GA-LS Python implementations.

[4] `http://numba.pydata.org`

4. Algorithm and Implementation

In this section we provide details about the implementation of the ILS and GA-LS algorithms in the three languages and describe the performance optimizations applied in each case. For all three languages and the C/OpenMP baseline we follow the same approach:

1. The task is to implement the ILS and GA-LS algorithms as described in Algorithms 1 and 3. Some design choices, detailed below, are made independently of the programming language.
2. A first version is developed and tested for correctness, without making any attempts to optimize for performance.
3. The code is profiled to detect the most time-consuming parts and issues like excessive memory allocations.
4. Based on the profiling results and best practices, possible performance optimizations are applied. The last two steps are repeated, until we see no more room for improvement with *reasonably small code changes*.
5. The neighborhood and population evaluation loops are parallelized and, if possible, tuned (chunk size, scheduling, threading layer, etc.).

Let us first provide some implementation details that are common to all implementations. In order to avoid the difficulty of parallelizing the quadruple nested loop that results from generating the set of moves (Equation 2) on-the-fly, the set $\mathcal{M}$ is generated only once and stored in memory. This allows to use a simple loop for Line 4, Algorithm 2. For convenience and readability, a user-defined structure (a class or a record, depending on the language's terminology) `solution` is defined in all implementations.

4.1. Python

The basic Python implementation uses NumPy for the 6D cost-matrix and solutions. In order to accelerate the code with Numba, we follow the performance tips from the Numba website whenever they are relevant. The workflow for accelerating existing Python code with Numba is completely incremental:

1. install Numba and include the package in the Python code
2. add decorators to compute-intensive functions to trigger JIT compilation with Numba
3. add decorators to enable parallelization

JIT compilation with Numba. The fundamental way to accelerate Python programs using Numba is to apply the `@jit` decorator to compute-intensive functions. This instructs Numba to JIT compile the decorated function to machine code at the first call of the function. For best performance, the `@jit` decorator should be used in `nopython` mode, meaning that the compiled function will run without involvement of the Python interpreter. The `@njit` decorator acts as a shortcut for `@jit(nopython=True)`.

The JIT compilation of a function with Numba may fail, for instance due to the use of unsupported features or failed type resolutions. Without the `nopython` option, Numba attempts to compile parts of the function to machine code and runs the rest in the interpreter. In `nopython` mode the execution will terminate with an error message, containing some hints on the reason of failure. In an incremental approach, such compiler feedback is very useful for finding spots that prevent faster code execution.

In our case, Numba refuses to compile the convenient conditional expression `if j not in move` (in the incremental fitness Equation 3), or iterating with the built-in Python function `enumerate`. Using the compiler-feedback, it is easy to rewrite those expressions in a more C-like way.

In order to be usable in JIT compiled functions, user-defined objects need special treatment. To allow Numba to recognize the user-defined class `solution`, one needs to specify the types composing the class and apply a `@jitclass` decorator to the class declaration. The resulting object, called a `jitclass` or a JIT-aware class, is a C-compatible structure to which compiled functions can have access, bypassing the interpreter. While this allows functions that use instances of `solution` to be successfully JIT compiled, it also triggers further issues. Indeed, copying a `jitclass` with the Python `copy` module fails with

a `TypeError: can't pickle solution objects` error message, so we were forced to implement a `copySolution` function by hand.

In the GA, further difficulties due to the use of a JIT-aware `solution` class arise when it comes to choosing a suitable data structure for the population of individuals. While Numba supports the use of Python `list`s in JIT compiled functions, several restrictions on the allowed types prevented us from using a list of JIT-classes. A NumPy array of solution objects in also rejected, as Numba 0.46 does not support arbitrary Python objects to be used as NumPy scalar types. Finally, Numba manages to JIT compile and parallelize the fitness evaluation of the population when the population is declared as a NumPy array of "structured scalars", which means that we had to replace the JIT-class `solution` by a custom NumPy data type object. This modification was also applied in the LS code embedded in the GA.

Table 2 shows the speed (in terms of neighborhood evaluations per second) of the sequential Python-based ILS implementation with and without Numba. This experiment only evaluates the JIT-compilation feature, as the (automatic) parallelization feature is not enabled. As one can see, Numba significantly accelerates the LS, providing an overall speedup for the ILS of 400× and more.

Table 2: Processing speed (in neighborhood evaluations/second) for the sequential Python-based ILS implementation with and without Numba `@jit`-compilation. All results are averages over 20 runs with 100 ILS iterations.

	nug12	**nug13**	**nug15**	**nug18**	**nug22**	**nug25**
Python	0.59	0.41	0.20	0.080	0.029	0.015
Python+Numba	282	189	91	36	11	5.9
ratio	**477**	**466**	**459**	**451**	**388**	**388**

Parallelization with Numba. The `@jit` decorator has a `parallel` option, which is set to `True` for the neighborhood and population evaluation functions. This causes Numba to attempt an automatic parallelization of several operations, like NumPy array operations, reductions or assignments. Moreover, Numba provides a `prange` function, which can be used instead of `range` to specify that

a loop can be parallelized. If the `parallel` option is set, the iterations of a `prange`-based loop are split between a number of threads, which is either detected automatically or specified by the `NUMBA_NUM_THREADS` environment variable. OpenMP-like critical sections are currently not supported, so the best move and the associated incremental cost cannot be determined "on the fly" as in Algorithm 2. Therefore, the incremental costs of all neighbors are stored in a temporary array and the best value is retrieved using NumPy's `argmax` function (for which Numba provides automatic parallelization support). In preliminary experiments we have tested both threading layers, TBB and OpenMP, without observing any significant difference in terms of performance.

4.2. Julia

Julia natively supports multi-dimensional arrays, so a 6D-array of integers is used for the matrix of cost coefficients. While the first version of our implementation in Julia was very easy to code, early preliminary experiments revealed it to be almost as slow as the pure Python implementation. Indeed, the documentation warns us that one may find Julia's performance "unintuitive at first" and highly recommends reading through the Performance Tips section[5]. The first paragraph of that section is entitled "Avoid global variables" and states that "Variables should be local, or passed as arguments to functions, whenever possible". We found that the latter recommendation should be taken very seriously if one is interested in performance.

For instance, we observed that *not* passing `dim` (a global integer that represents the size of the problem instance) as an argument to the incremental fitness evaluation function (the innermost function call) results in a 15-fold performance drop for ILS. However, we should note that this does not mean that it is necessary to manually specify the types of the variables—usually Julia will be able to infer these types.

Table 3 shows the speed (in neighborhood evaluations per second) of a first,

[5] `https://docs.julialang.org/en/v1/manual/performance-tips/index.html`

unoptimized ILS implementation in Julia, and a second version which was obtained after several optimization cycles, using Julia's profiling tools and following the "Performance Tips". One can see that the initial Julia implementation is about 4× faster than its Python-only counterpart and that the code optimizations allow to accelerate the initial version by a factor 80-90. This performance gain is mainly due to "passing arguments to functions".

Table 3: Processing speed (in neighborhood evaluations/second) for the sequential Julia-based ILS implementation before and after applying performance improvements. All results are averages over 20 runs with 100 ILS iterations.

	nug12	**nug13**	**nug15**	**nug18**	**nug22**	**nug25**
Julia - first	2.00	1.37	0.66	0.26	0.094	0.049
Julia - tuned	180	123	60	23	7.8	4.1
ratio	**87.9**	**89.6**	**90.4**	**88.5**	**82.6**	**82.2**

Parallelization of the fitness evaluation loop is performed adding a simple `@threads` macro (provided by the `Threads` module) to the for-loop. The `Threads` module is experimental and no equivalents to `omp critical` sections or `schedule` clauses are currently available. Although a mutex implementation is provided, we chose to store incremental costs in a temporary array on which a min-reduction is performed subsequently.

4.3. Chapel

An initial Chapel version was produced based on the original C implementation facing no major issues. Then, the code was analyzed to identify opportunities for applying Chapel's high-productivity features, such as zipped iterators, array initialization, reductions, multidimensional ranges and data structures. The final serial implementation was then parallelized using the task-parallel features provided by the language. The `--fast` compiler flag is used, enabling several optimizations.

As in OpenMP, Chapel provides five work-distribution schemes, which are implemented as built-in iterators used in `forall` statements[6]: *dynamic*, *guided*

[6] https://chapel-lang.org/docs/modules/standard/DynamicIters.html

and three work stealing strategies. The first two are Chapel implementations of OpenMP's scheduling policies of the same name. As for the C/OpenMP implementation (detailed in the following) using the static distribution instead of iterators results in the best overall performance.

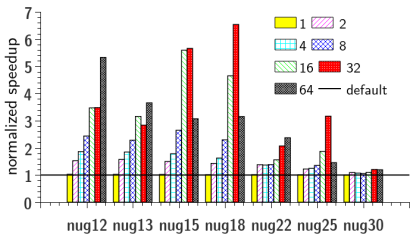
Chapel provides two task layer implementations [30]: *qthreads* (default) and POSIX Threads (Pthreads). A preliminary experiment was performed to verify which task layer implementation is the most advantageous in the context of this work. It is important to point out that the task layer is chosen in terms of environment variables and this action means no coding efforts. As for Numba, changing the task layer does not result in performance improvements.

Although several preliminary experiments for fine tuning were conducted, the best overall performance is obtained by using Chapel's *default* settings. In the context of the present work, this is an advantage of Chapel compared to C/OpenMP—as detailed in the following the latter requires fine tuning to scale on a Non-Uniform Memory Access (NUMA) architecture.

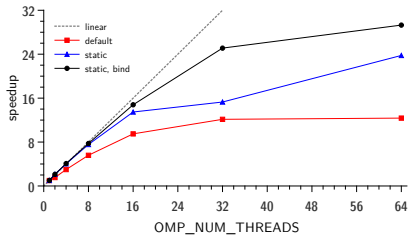
4.4. Baseline: C/OpenMP

For the reference C/OpenMP implementation no particular optimizations of the sequential code are performed. The code is compiled with the `-O3` option of the `gcc` compiler. The parallelization is achieved by inserting OpenMP directives into the code. Further optimization is achieved by tuning OpenMP environment variables.

As one can see in Figure 1a, using `static` scheduling for the distribution of loop-iterations and setting the `OMP_PROC_BIND` environment variable to `true` has a strong impact on the achieved processing speed. The relative speedup of the static-bind version over the default configuration depends on the number of threads and the problem instance. While there is no significant effect on the performance of single-threaded runs, the tuned OpenMP configuration is up to 6.5× faster than with the default setting (for *nug18* and 32 threads). Clearly, the tuning of OpenMP environment variables is particularly beneficial for the smaller instances (*nug12*-*18*), i.e. smaller neighborhoods and more fine-grained



(a) Effect of setting `OMP_SCHEDULE=static` and `OMP_PROC_BIND=true`, compared to default values



(b) Scalability of ILS for instance *nug22*, with different settings for OpenMP environment variables `OMP_SCHEDULE` and `OMP_PROC_BIND`

Figure 1: Effect of OpenMP environment variable settings on the performance of the ILS C/OpenMP implementation

parallelism.

For problem instance *nug22*, Figure 1b shows the speedup achieved with different OpenMP settings and a number of threads varying from 1 to 64. The neighborhood evaluation loop consists of a large number of fine-grained tasks of similar duration (Δ-evaluations). Therefore, dynamic scheduling is expected to incur significant overhead without bringing any benefit in terms of load balancing and a static distribution of loop iterations is more suitable.

When the number of OpenMP threads exceeds the number of physical cores within one socket, the application uses both sockets of the Non-Uniform Memory Access (NUMA) system. Especially in NUMA architectures, migration of threads between cores can considerably increase memory access times. As can be seen in Figure 1b, binding OpenMP threads to hardware threads (or to sockets) improves the scalability of the parallel ILS.

5. Experimental evaluation

This section presents the experimental evaluation, which is divided in three parts: Section 5.1 compares the different implementations in terms of performance and Section 5.2 in terms of productivity. In Section 5.3 we investigate the performance of multi-threaded parallel fitness evaluation, considering 4 problems and 8 test functions with different computational characteristics.

5.1. Performance Evaluation

In this evaluation, the implementations introduced in Section 4 are compared to the C/OpenMP reference. For each language, there are a GA and an ILS implementation. Concerning the ILS implementations, our objective is to analyze how the number of neighborhood evaluations/second scales as the number of used threads increases. In turn, for the GA, we intend to compare the quality of the solution obtained by Chapel, Julia and Python compared to ones obtained by the baseline with a fixed time budget of 5 minutes. Due to the large amount of collected data, some results are presented in a summarized way.

5.1.1. Parameters Settings

The benchmark Q3AP instances used in the experiments are derived (as explained in Section 2.2) from the *nug*-class of the QAPLIB library [16]. The instances chosen are those of size 12, 13, 15, 18, 22, 25 and 30 from the *nug* class. The parameters used in the ILS and GA implementations are the ones presented in Section 2.3 and Section 2.4 respectively. Moreover, each configuration $<$ threads, instance, implementation $>$ is run 20 times.

The testbed operates under Debian 4.9.0, 64 bits, and it is equipped with a dual-socket NUMA node composed of 2 Intel Xeon Gold 6130 CPUs (Skylake, @2.10GHz, 16 cores/CPU, hyperthreading enabled) and 192 GB of RAM. The C implementation was compiled with *gcc* 6.3.0 and OpenMP 4.5. The Chapel application was programmed for version 1.19 with the *default* task layer (qthreads). We use Julia version 1.2 and Python 3.6 with Numba 0.47.

5.1.2. Performance Results

ILS. Figure 2 shows the performance (measured in neighborhood evaluations per second) of the Python/Numba, Julia and Chapel implementations relative to the C/OpenMP baseline, for the number of threads varying from 1 to 64 and for instance *nug22*. The relative slowdown concerning the C/OpenMP reference is shown, so smaller values are better. Hereafter, we are going to refer

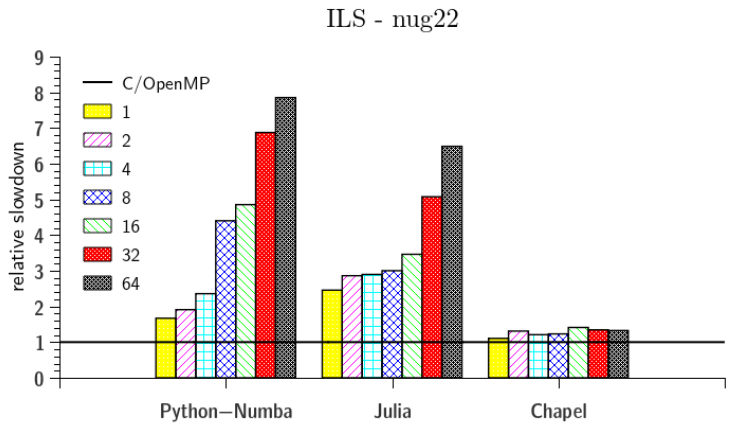


Figure 2: Relative slowdown of parallel ILS in Python, Julia and Chapel with respect to the baseline C/OpenMP implementation (smaller is better). Performance is measured in terms of neighborhoods/second, for 100 ILS iterations and instance *nug22*.

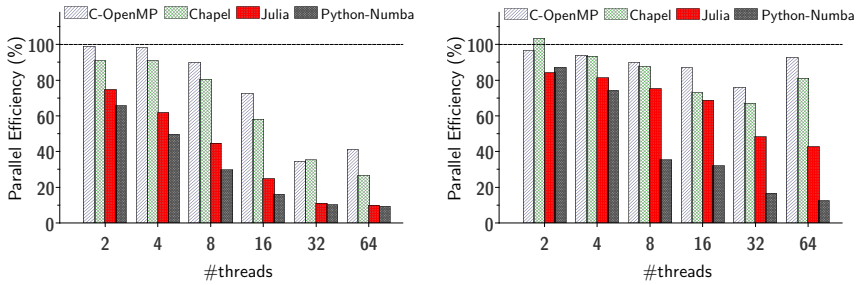
to the Chapel, Julia and Python/Numba implementations as Chpl, Jl and Py respectively.

One can see that the performance of ILS-Chpl is close to the C/OpenMP baseline (less than 1.4× slower) for up to 64 threads. The performance gap between ILS-Chpl and the baseline does not grow according to the number of threads, so the behaviour of ILS-Chpl in terms of scalability is similar to C/OpenMP. It should be noted that Chapel achieves this with default settings, while the baseline is obtained by tuning environment variables (see Figure 1).

Considering the two dynamic languages, the sequential versions of ILS-Py and ILS-Jl are 1.7 (resp. 2.5) times slower than the baseline. In turn, the corresponding parallel versions using all available hardware threads (64) are 7.9 (resp. 6.5) times slower than their C/OpenMP counterpart. While ILS-Py is faster than ILS-Jl for 1, 2 and 4 threads, the Julia-based parallel version outperforms the Python-based parallel ILS for 8 and more threads. Indeed, for ILS-Py one can notice a sharp increase in relative slowdown when the number of threads is increased from 4 to 8 - this is concerning, as it is hard to explain by the underlying hardware configuration (2×16 cores).

In turn, for ILS-Jl a significant drop in scalability occurs when increasing the

number of threads from 16 to 32, i.e. when using more threads than the number of cores on a single CPU. A likely explanation for this is found by analyzing the effects of different environment variables on the scalability of the baseline, shown in Figure 1. Indeed, the scaling of the C/OpenMP implementation with more than 16 threads is only achieved through the binding of OpenMP threads and by explicitly setting the distribution of loop iterations to `static`. To the best of our knowledge, neither Julia nor Numba currently provides user-control over these parameters.



(a) nug12, small (b) nug25, large

Figure 3: Parallel efficiency reached by all four ILS implementations compared to the respective sequential versions. Values are given in percent of the linear speedup (linear = 100%). Results are for (a) *nug12* (small) and (b) *nug25* (large).

Figure 3 depicts the parallel efficiency $\frac{\tau_1}{p\tau_p} \times 100\%$ where τ_p designates the processing speed (neighborhoods/sec) observed with p cores. In order to see whether an implementation can take advantage of hyper-threading, for 64 threads, p is set to 32.

On the left-hand side, Figure 3a shows the results for the smallest instance *nug12*. On the right-hand side (Figure 3b) shows results for the largest solved instance, *nug25* – we do not compare results for *nug30*, as internal errors occurred with both Julia and Python, likely due to the fact that the 6D array of cost coefficients requires more than 4 GB of memory. For *nug30*, the performance of the Chapel-based ILS is equivalent to C/OpenMP when using 1 to 16 threads and up to 15% slower with 64 threads.

Figure 3 shows that, as expected, the scalability of all implementations is

better for large instances. In Figure 3b, one can see that ILS-Jl is clearly more scalable than its Python-based counterpart. Indeed, for *nug25* and 8 threads, Julia's parallel efficiency, 75%, is more than twice as high as Python's, which drops to 35%. ILS-Chpl is the only implementation that scales equivalently to the baseline implementation for up to 8 threads. For 16 to 64 threads, ILS-Chpl reaches 78–88% of the efficiency achieved by the baseline implementation. Moreover, comparing the rates observed for 32 and 64 threads reveals that only the C/OpenMP baseline and ILS-Chpl can exploit the hyper-threading features of the testbed.

GA. In order to compare the different GA implementations we consider the solution quality reached after 5 minutes of execution. As a measure for solution quality we compute the relative percentage deviation (RPD), computed as $\frac{f^\star - f_{\text{best}}}{f_{\text{best}}} \times 100\%$, where f^* designates the objective value found after 5 minutes and f_{best} is the cost of the optimal or best known solution as shown in Table 4.

Table 4: Best known solutions for *nug*-derived Q3AP instances. For $n = 12, 13, 15$ optimal solutions are known due to [11, 31]. For $n = 18, 22, 25, 30$ we report the best solution found by all runs performed in this experiment.[7]

nug12	**nug13**	**nug15**	**nug18**	**nug22**	**nug25**	**nug30**
$580^\star$	$1912^\star$	$2230^\star$	5064	7910	9318	18602

The results of the GA experiment are shown in Figure 4. Clearly, for all four languages the use of parallelism (entries prefixed "64" in Figure 4) improves the quality of solutions, especially for the larger instances (shown in the lower part of the figure). For instances *nug18*, *22*, *25* the best overall solution is always found in parallel.

In turn, it is much more difficult to discriminate between languages. While Chapel and C appear to provide better results than Julia and sometimes Python

[7]The upper bounds for $n = 18, 20, 22, 25, 27, 30$ reported in [32] are inconsistent with our results. However, we do obtain consistent results with the literature by reading the QAPLIB files as $[F, D]$ for $n = 12, 13, 15$, and as $[D, F]$ for $n = 18, 22, 25, 30$ (F and D designate the flow and distance matrices used in Eq. 1). Indeed, even the QAPLIB home page is ambiguous concerning this order, as it is irrelevant for the QAP. This symmetry is not valid for the Q3AP when generating instances according to Eq. 1. We stick to the order $[F, D]$.

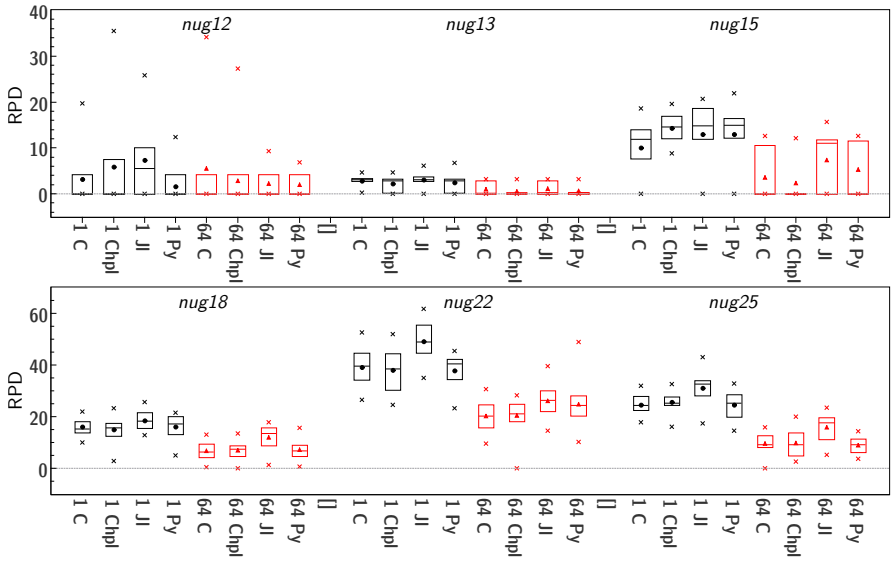


Figure 4: Relative Percent Deviation (RPD) from optimum/best-known solution achieved after 5 minutes by sequential and parallel implementations in all four languages. Each configuration is run 20 times. The boxes represent the 1st quartile, median and 3rd quartile. The filled dots/triangles represent the average and diagonal crosses min/max values.

(considering for example instance *nug22*), the results are not as clear as for ILS. Especially, for the small instances *nug12* and *nug13* the results do not allow to decide which implementation provides the best results. For *nug12*, all 8 sequential and parallel versions have a high success rate in finding an optimal solution—as shown on Figure 4, the median RPD is equal to 0 in all cases except for sequential Julia. At the same time, we observe strong worst-case outliers—for *nug12* and the observed sample of 20 runs, C and Chapel actually have the worst worst-case performance.

As the quality of solution quite strongly depends on random initialization and randomized genetic operators, the number of performed runs seems too small to make a sharp distinction between the four languages. Considering that it required 80h of computation time to produce the results shown in Figure 4, we limited this experiment to 20 runs per configuration. Two main conclusions can be drawn from this experiment: On the one hand, the parallel versions of the hybrid GA implemented in all four languages provide enough speedup to

improve the solution quality reached within a fixed time budget. On the other hand it puts the slowdown caused by the choice of a programming language into perspective, as the effects on the quality of solutions may actually become apparent only in the long run.

5.2. Productivity-oriented Evaluation

In this productivity-oriented evaluation, two models are applied for measuring *productivity in HPC*: Kennedy *et al.* [33] and Snir and Bader [34]. The first one provides a visual trade-off between relative implementation cost and performance, while the second one is closer to the industrial definition of productivity [35], expressing productivity as utility over a total cost. Both models are detailed in the following.

5.2.1. Visual Trade-off Model

The model by Kennedy *et al.* computes the relative implementation cost (ρ_l) and the relative performance (ε_l) of implementing a program P by using a language l. Both metrics are defined as follows:

- $\rho_l = \frac{I(P_0)}{I(P_l)}$ represents the cost of developing the program P in the control language 0 over the cost of implementing the same program using the language l. Details concerning the implementation cost are going to be given further.
- $\varepsilon_l = \frac{E(P_0)}{E(P_l)}$ represents the execution time of the program implemented in the control language over the execution time of the program implemented in language l.

Once those two metrics are computed, the values are plotted on a $\varepsilon \times \rho$ graph, providing a visual trade-off between relative implementation cost and relative performance. As one can see in Figure 5, the results of the reference implementation are plotted on the $(1, 1)$ point of the graph. Next, the plotted points are compared to the *desired productivity region* (DPR). For a high-level language l, the value ρ_l is usually greater than 1 and ε_l lower than 1. Therefore,

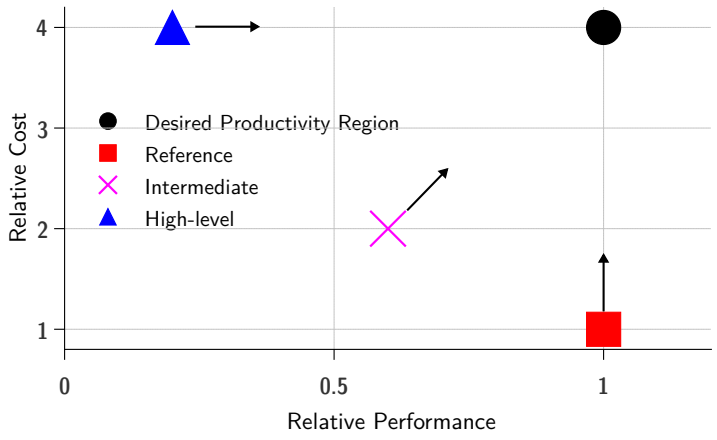


Figure 5: Illustration of the trade-off between relative cost and relative performance of three languages compared to the reference one. In the graph, the arrows point to the desired productivity region (DPR).

in the model by Kennedy *et al.*, the DPR means that an ideal high-productivity language is the one that achieves performance similar to the reference language and implementation cost equivalent to a high-level language. As all languages provide a similar solution quality, we are not going to use this model for the GA.

5.2.2. Utility Model

Initially, consider *Utility* as the value received on getting an answer to a problem in a certain time [36]. According to the model, *Productivity* (ψ) is utility over a total cost, and it is defined as follows.

$$\psi = \frac{S_p \times E \times A}{C_s + C_o + C_M}$$

where:

- S_p : is the operations/time peak that can be achieved on the system.
- E : efficiency achieved by the parallel program.
- A : availability of the system.

- C_s : software cost.
- C_M and C_o: cost of the machine and ownership, respectively. These metrics concern any cost related to energy, hardware maintenance, human resources, etc.

We adapted the model for calculating relative productivity based on the C/OpenMP baseline. In this variation, S_p is the performance of the reference application for a given < instance, #threads > configuration and E the efficiency achieved by ILS written in the language l given in % of the efficiency achieved by the baseline. The authors do not handle both monetary and ownership costs. This way, they are considered as equal to *zero*. Moreover, the availability of the system is 100%.

For the sake of simplicity, both implementation ($I(P_l)$) and software cost (C_s) are going to be based on the source lines of code (SLOC) count. Despite the criticism concerning the SLOC metric [34], it is a widely used indicator of programming effort [37] and it is expected that the implementation cost increase monotonically according to the program size [38].

5.2.3. Implementation Cost/Software Cost

One can see in Table 5 the SLOC count for ILS and GA, implemented in Julia, Chapel, Python, and C. For the GA, we isolate the crossover, mutation, evaluation, and selection operators. In turn, the whole ILS application is taken into account. Non-essential parts of the code, such as comments, includes, timers, and print functions are removed from the SLOC count.

As shown in Table 5, the Chapel-based implementation is the second largest, after the C-based one. As Chapel is also a compiled language, this is expected. Chapel's advantages in terms of SLOC come from the use of multidimensional data structures, which removes the need for using a function for returning an element of the 6D matrix. Moreover, built-in swap operations, high-level vector initialization, reductions and zipped iterators (`forall (a,b) in (A,B) do`) contributed considerably to shortening Chapel's code size.

Table 5: Relative implementation cost (ρ_l) and relative software cost (C_s) of Chapel, Julia, and Python/Numba compared to C/OpenMP. As C/OpenMP is the reference language, its relative implementation and software costs are equal to *one*.

Language	SLOC-ILS	ρ	C_s
C	247	1	1
Chapel	155	1.59	0.62
Julia	106	2.33	0.43
Python	137	1.80	0.55

Both Julia and Python-based implementations take advantage of built-in high-level functions and easily available libraries. For instance, using the `Random` and `numpy.random` packages, generating an initial population of random individuals requires a single line in both languages, such as `pop=[Sol([randperm(dim), randperm(dim)], 0) for i in 1:100 ]`. For both languages, the reduction in code size, with respect to C, is mainly due to built-in swap operations, list comprehensions and utility functions for sampling random variables or shuffling sub-arrays of permutations.

The relative implementation cost ρ_l has been already introduced in Section 5.2.1, whereas the software cost C_s is going to be considered as a relative software cost given by $SLOC^l/SLOC^C$. One can see in Table 5 the relative implementation (ρ_l) and software costs (C_s) for Chapel, Julia, and Python/Numba. For the same reason of the previous model, we are not going to apply this model to the GA.

5.2.4. Results

Figure 6 depicts the visual trade-off between relative implementation cost and relative performance observed for the three productivity-aware languages compared to the C/OpenMP baseline. The values of Chapel are the closest to the desired performance. Moreover, for Chapel the gap between serial and parallel relative performance is the smallest, reflecting its scalability. On the one hand, ILS-Chpl is almost 50% more costly to implement than ILS-Julia. On the other hand, Chapel's relative cost is close to the one observed for Python, a high-level interpreted language.

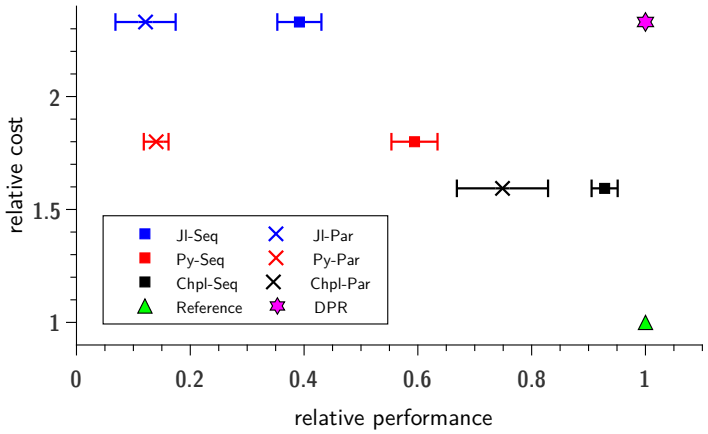


Figure 6: The trade-off between relative cost and relative performance of Chapel, Julia, and Python compared to the reference ILS implementation. In the graph, the desired productivity region (DPR) is on point $(1, 2.33)$.

The Python/Numba implementation achieves serial results towards DPR. However, as it does not scale, its relative parallel performance is far lower than the one observed for ILS-Chpl and the baseline. The ILS-Julia implementation faces a similar problem: despite Julia's relative cost, its sequential relative performance is the farthest from the performance observed for the baseline implementation and comparable to Python for the parallel version.

One can see in Figure 7 the relative parallel productivity results achieved by Chapel, Julia, and Python/Numba, taking into account the utility productivity model. In this model, a parallel programming language is only productive if it allows coding an application that scales [36]. This way, Python/Numba is from 2% to 7% more productive than C for serial execution. Taking into account the parallel execution on 64 threads, Python/Numba is on average 75% less productive than C. In turn, Julia is as productive as C only for the serial execution of *nug25* and it is up to 85% less productive than C on 64 threads. Due to the poor parallel efficiency achieved by ILS-Python and ILS-Julia, it is more productive to use C/OpenMP for programming the metaheuristic in question.

Differently from the two higher-level languages, Chapel is the only one that is more productive than C/OpenMP for *all* configurations. The reason

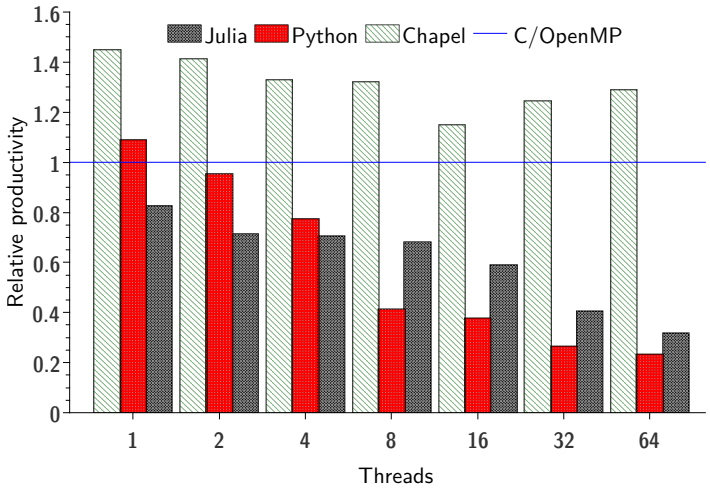


Figure 7: Relative productivity achieved by Chapel, Julia, and Python compared to the C/OpenMP reference. Results are given for the instance *nug22* and execution on 1 to 64 threads.

is that ILS-Chpl achieves similar performance to the base implementation, and the *relative software cost* of its ILS implementation is similar to the one of Python/Numba. As a consequence, Chapel is from 44% to 57% more productive than C/OpenMP taking into account serial execution, and up to 85% more productive than C/OpenMP for parallel execution (*nug13*, 32 threads).

5.3. Benchmark: Parallel Batch-Evaluation Loop

While we expect relative programming costs to be similar for other algorithms and problems, performance results may be significantly different. Considering the parallel fitness evaluation loop, its computational cost strongly depends on both, the batch-size (e.g., population or neighborhood size) and the computational characteristics of the evaluation function (e.g., granularity, arithmetic intensity). We have designed a simple benchmark to evaluate the impact of these two factors on the relative performance of parallel batch evaluations in the different languages.

The pseudo-code of this benchmark program is shown in Algorithm 4. A total number of 10^6 function evaluations are performed in batches of size `batchsize`$\in$

Algorithm 4 Benchmark: Parallel Batch Evaluation

```
1: procedure parallelBatchEvaluation(f,batchsize)
2:     for i ← 1, ..., 1e6/batchsize do
3:         A ← generateRandomSolutions()
4:         for j ← 1, ..., batchsize do                ▷ parallelize and time
5:             costs[j] ← f(A[j])
6:         end for
7:     end for
8: end procedure
```

Table 6: Summary of test-functions used in the benchmark experiment.

Problem	Objective function	Inst.
FSP	$f(\pi) = C_{\pi(n),m}$ with $\begin{cases} C_{\pi(i),j} = \max(C_{\pi(i-1),j}, C_{\pi(i),j-1}) + p_{\pi(i),j} \\ C_{0,j} = Cj, 0 = 0 \end{cases}$	$ta20/ta120$
QAP	$f(\pi) = \sum_{i=1}^{n} \sum_{i=1}^{n} F_{ij} D_{\pi(i)\pi(j)}$	$nug12/tho150$
Q3AP	$f(\pi, \sigma) = \sum_{i=1}^{n} \sum_{j=1}^{n} C_{i\pi(i)\sigma(i)j\pi(j)\sigma(j)}$	$nug12/nug25$
TSP	$f(\pi) = \sum_{i=1}^{n-1} d(\pi(i), \pi(i+1)) + d(\pi(n), \pi(1))$ with $d(x,y) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$	$berlin52$ $pr2392$

$\{10^2, 10^3, 10^4, 10^5\}$ (Algorithm 4, line 2 and line 4). As test-cases, we use makespan evaluation in the permutation flowshop scheduling problem (FSP) and the objective functions of Q3AP, QAP, traveling salesman problems (TSP). We consider a small and a large instance of each problem. A summary of the used test-functions is given in Table 6.

Only the inner batch-evaluation loop is parallelized and measured (the purpose of the outer loop and the random solution generation is to obtain a reliable average execution time). As for the ILS and GA-LS test-cases, the baseline and the three implementations are optimized before running the benchmark for $1, 2, 4, \ldots, 64$ threads. Out of these 7 runs the best execution time is retained and compared to the best performance obtained with the baseline implementation.

5.3.1. Results

Figure 8 reports the results of this benchmark. The y-axis (in log10-scale)

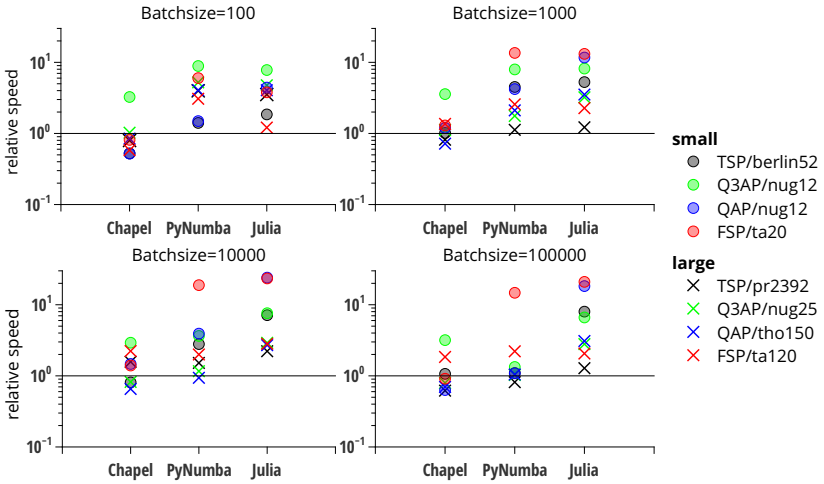


Figure 8: Performance of parallel evaluation loop relative to the C/OpenMP baseline. Out of 7 runs with $1, 2, \ldots, 64$ threads, the best-achieved performance is compared to the best performance obtained with the baseline.

shows the relative performance $\frac{\tau_L}{\tau_C}$, where $\tau_l = \min\ T_{L,p}\,(p = 1, 2, \ldots, 64)$ is the best execution time reached in language L and $\tau_C = \min\ T_{C,p}$ the best performance with the baseline C/OpenMP implementation.

A first observation that can be made is that the Chapel's performance is similar to the C/OpenMP baseline, being up to $2\times$ faster and in the best case and up to $3\times$ slower in the worst case. The only test function for which C is consistently better than Chapel is the small Q3AP instance *nug12*.

While the average performance of Julia and Python are roughly equivalent for small batch-sizes (100, 1000), Python/Numba outperforms Julia for larger batch-sizes. For both, Python and Julia, the performance for large problem instances is clearly better than for small ones. This is particularly visible for batch-sizes 1000 and 10000.

The type of performed computations also impacts the observed performance. For the large TSP instance *pr2392* and batch-size ≥ 1000 the performance of all 3 languages is very close or even better than the baseline. The TSP instances are given in coordinate form, so the computation time is dominated by the computation of square-roots, which is an arithmetically intensive operation.

In contrast, the achieved performance for both FSP instances *ta20* and *ta120* is clearly inferior. As shown in Table 6, the FSP makespan evaluation requires mn max operations/additions and as many memory accesses. The max operation is semantically equivalent to an `if-else` statement, so this test-function is characterized by low arithmetic intensity and divergent control flow. Especially for the small FSP instance *ta20*, both Julia and Numba have difficulties dealing with this type of workload. Computationally, the FSP objective function is the closest to the incremental cost-evaluation in the large $\mathcal{O}(n^4)$ neighborhood of our ILS implementation for the Q3AP. Indeed, as indicated in Section 2.3, the Δ-evaluation for the Q3AP contains a high number of `if`-conditions and the number of operations $(16n - 32)$ is lower than for the test-functions considered in this experiment.

6. Discussion

In this section, we discuss the presented results, stating the main insights, threats to validity and further aspects of comparison that one may take into account.

Insights. The main insights can be summarized as follows. For the Q3AP test-case and the two implemented algorithms, ILS and GA:

- None of the three productivity-aware languages, Python, Julia and Chapel, beats the reference C/OpenMP implementation in terms of sequential or parallel performance, but all three present a smaller implementation cost.
- Productivity results for the two dynamic JIT compiled languages Julia and Python/Numba are similar: both are clearly more expressive than C, resulting in codes that are about 2 times smaller. In terms of sequential performance slowdown factors of 2-3 are observed to C. According to the applied utility model, both languages are roughly as productive as C in a sequential setting, but less productive than Chapel.

- Numba's and Julia's (experimental) multi-threading support is not mature enough to compete with OpenMP or Chapel in terms of scalability. Using 64 threads, the implemented ILS in Julia and Python is 6-8 times slower than C. As a consequence, they face poor productivity results taking into account the utility model. In turn, Chapel's performance is very close to C, both sequential and parallel, while presenting a lower implementation cost, thanks to high-level functions. As a consequence, it is more productive than C/OpenMP according to the utility model.

The benchmark experiment presented in Section 5.3 investigates the performance of a parallel fitness evaluation loop for 8 different test-functions. The main insights from this experiment are:

- The performance results strongly depend on the algorithm and the problem being solved. The ILS/Q3AP and GA/Q3AP test-cases are very challenging for multi-threaded parallel computing.
- For large batch sizes and more regular fitness evaluation functions, it can be expected that Python/Numba (and to a lesser extent Julia) can achieve a similar parallel performance than C/OpenMP and Chapel.

The possibility to quickly prototype and test algorithm variants give to both interpreted languages an advantage in terms of *time to a solution from scratch*. While this advantage is hard to evaluate, it is one of the main reasons for the growing popularity of those languages and one of the main motivations for investigating them. While we obtained first serial versions with those languages rapidly, these initial versions performed poorly. In order to obtain satisfactory sequential performance and speed-up from multi-threading, a significant amount of code optimization was necessary. In this regard, Chapel has a clear advantage even when compared to C/OpenMP, as no tuning or code tweaking was required to obtain the final version, which performs nearly as good as C/OpenMP.

Threats to validity. There are several threats to the validity of these results and precautions to take when extrapolating them to different problems or algo-

rithms. We have chosen two algorithms, ILS and GA, applied to the Q3AP, as they contain certain *a priori* representative features of parallel metaheuristics and their application to combinatorial optimization problems (such as costly neighborhood and population evaluations, irregular memory access patterns). However, as indicated by the experimental results obtained with an isolated parallel evaluation loop and 8 test-functions, the achieved performance strongly depends on the computational characteristics of the considered problem/algorithm combination.

Besides the bias induced by the choice of a specific problem/algorithm, there is necessarily a bias introduced by the programmer(s). Both the program size and the attained performance may vary according to the level of expertise of the programmer. In our case, both programmers have strong prior experience with C and parallel computing and little to intermediate prior knowledge of Python, Julia and Chapel. As detailed in Section 4, we have followed a protocol that aims at making the comparison fair. However, we cannot completely exclude that some parts of the code could be written more efficiently or concisely.

Further aspects to consider. The presented comparative study focuses on performance and productivity, both defined by certain metrics. When it comes to evaluating the usefulness of a programming language in a particular domain of application, there are many important aspects that are context-dependent and difficult to quantify – some were mentioned in the presentation of the languages in Section 3.

For instance, the popularity of a language, available documentation and support, code portability, interoperability and extensibility are important criteria to consider, especially if one aims at producing reusable code. Each of the three languages is appealing in its own way and the following is a (somewhat subjective) account of this:

- A strong argument in favour of Python is its popularity and a large number of existing libraries. Python interpreters are installed on almost any machine. As confirmed by our experimental evaluation, pure Python is

slow and for acceleration or multi-threading support one currently has to choose between different available solutions (e.g. Numba). The fact that Numba allows to speed up existing Python code incrementally is a definite plus. The sequential performance of Numba-based JIT compiled Python is very promising and our experiments show that Numba's loop-based parallelism can be as powerful as OpenMP *for certain applications*. For other applications that are more challenging for multi-threaded parallel computing, Numba's multi-threading support still needs some improvement. Furthermore, as many Python features are currenty not available in JIT-mode, it can be challenging to work around those restrictions. If future versions can overcome those issues, the combination of Python/Numba seems like a good way to increase the (re)usability of parallel metaheuristics as well as their interaction with various domains, such as machine learning.

- Julia aims at bridging the programmability-performance gap, providing scientific programmers in various fields with one language for quick prototyping and high-performance computing. If the language can fulfil its ambitions, it might become the predominant scientific computing language of the future. As the language is still young, there were important changes between early versions. Consequently, much of the information one can find in online forums and documentation is no longer valid, which might confuse newcomers to the language. Also, they way how Julia has to be programmed to reach good performance results can actually be quite subtle – understanding the language, not only the syntax, seems to be a prerequisite to obtain code as fast as C. There is a large amount of documentation available, but it sometimes feels opaque—for instance we were unable to find information on the thread layer used for the multi-threading package. Concerning popularity, Julia is gaining momentum among scientific programmers and it remains to be seen how widely it will be adopted.

- Chapel is a language designed for high-performance computing, but one of its most compelling features has not been used in this work: Global-view distributed data structures (Partitioned Global Address Space - PGAS [39]). The Chapel codes written for this paper can actually run on distributed systems by performing straightforward modifications [26, 25]), which represents a potentially significant gain in productivity. Chapel is currently used by a portion of the HPC community, which is more familiar to lower-level languages. On the other hand, users from a different community may be reluctant to learn a compiled language, even if it is higher-level than C [4].

7. Conclusions and Future Works

In this paper, we have compared three high-performance high-productivity programming languages for the implementation of parallel metaheuristics: Julia, Python/Numba and Chapel. As a test-case, we have programmed an Iterated Local Search (ILS) algorithm, and a Genetic Algorithm (GA) hybridized with a local search. All languages studied are suitable options for programming parallel metaheuristics. They provide a feasible time-to-solution and the high-level features present in the three chosen languages can considerably shorten the code when comparing the implementation to the C/OpenMP baseline.

The main obstacle of using Python/Numba and Julia for programming parallel metaheuristics is that their multi-threading support is not yet mature enough to replace C/OpenMP. For instance, Python/Numba and Julia present a clear advantage in producing a first implementation from scratch. However, the two interpreted languages were also the most difficult to tune for scalability. This relatively poor scalability of the Python/Numba and Julia implementations results in lower productivity scores than the ones observed for C/OpenMP. In contrast, Chapel's parallel features and performance are competitive with C/OpenMP. The main limitation for its adoption is that it is another compiled language for HPC, which may require a learning curve bigger than the one necessary for

Julia or Python.

Python, Julia and Chapel are languages that support distributed programming. However, this feature was not studied in the present work. Therefore, we plan to investigate the use of these three languages for programming distributed metaheuristics. Another important aspect is GPU programming support, which is provided in Julia and Python/Numba and supported, but yet not mature in Chapel. Thus, we plan to investigate the use of Julia and Python/Numba for programming massively parallel GPU-based metaheuristics for solving big optimization problems.

Acknowledgments

The experiments presented in this paper were carried out on the Grid'5000 testbed [40], hosted by INRIA and including several other organizations [8]. We thank Bradford Chamberlain, Elliot Ronaghan (Cray inc.) and Louis Jenkins (University of Rochester) for their support on Chapel's Gitter platform [9].

References

[1] E. Alba, Parallel metaheuristics: a new class of algorithms, Vol. 47, John Wiley & Sons, 2005.

[2] E.-G. Talbi, Metaheuristics: from design to implementation, Vol. 74, John Wiley & Sons, 2009.

[3] G. Da Costa, T. Fahringer, J. A. R. Gallego, I. Grasso, A. Hristov, H. D. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. L. Stavrinides, et al., Exascale machines require new programming paradigms and runtimes, Supercomputing frontiers and innovations 2 (2) (2015) 6–27.

[8] http://www.grid5000.fr
[9] https://gitter.im/chapel-lang/chapel

[4] E. Lusk, K. Yelick, Languages for high-productivity computing: the DARPA HPCS language project, Parallel Processing Letters 17 (01) (2007) 89–102.

[5] P. Prabhu, H. Kim, T. Oh, T. B. Jablin, N. P. Johnson, M. Zoufaly, A. Raman, F. Liu, D. Walker, Y. Zhang, et al., A survey of the practice of computational science, in: SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2011, pp. 1–12.

[6] E. Alba, G. Luque, S. Nesmachnow, Parallel metaheuristics: recent advances and new trends, International Transactions in Operational Research 20 (1) (2013) 1–48.

[7] H. Eichhorn, J. L. Cano, F. McLean, R. Anderl, A comparative study of programming languages for next-generation astrodynamics systems, CEAS Space Journal 10 (1) (2018) 115–123.

[8] S. B. Aruoba, J. Fernández-Villaverde, A comparison of programming languages in economics, Tech. rep., National Bureau of Economic Research (2014).

[9] M. Fourment, M. R. Gillings, A comparison of common programming languages used in bioinformatics, BMC bioinformatics 9 (1) (2008) 82.

[10] M. Lubin, I. Dunning, Computing in operations research using julia, INFORMS Journal on Computing 27 (2) (2015) 238–248. doi:10.1287/ijoc.2014.0623.

[11] P. M. Hahn, B.-J. Kim, T. Stützle, S. Kanthak, W. L. Hightower, H. Samra, Z. Ding, M. Guignard, The quadratic three-dimensional assignment problem: Exact and approximate solution methods, European Journal of Operational Research 184 (2) (2008) 416 – 428. doi:https://doi.org/10.1016/j.ejor.2006.11.014.

[12] T. V. Luong, L. Loukil, N. Melab, E.-G. Talbi, A gpu-based iterated tabu search for solving the quadratic 3-dimensional assignment problem, ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010 (2010) 1–8.

[13] M. Mehdi, J. Charr, N. Melab, E.-G. Talbi, P. Bouvry, A Cooperative Tree-based Hybrid GA-B&B Approach for Solving Challenging Permutation-based Problems, in: GECCO 2011, 13th Conf. on Genetic and Evolutionary Computation Conference, Dublin, Ireland, 2011, pp. 513–520.

[14] L. Loukil, M. Mehdi, N. Melab, E.-G. Talbi, P. Bouvry, A parallel hybrid genetic algorithm-simulated annealing for solving q3ap on computational grid, in: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IPDPS '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–8. doi:10.1109/IPDPS.2009.5161126.

[15] J. Gmys, T. Carneiro, N. Melab, E. Talbi, D. Tuyttens, jangmys/benchparallelmeta: A comparative study of high-productivity high-performance programming languages for parallel metaheuristics (version v1.0). doi:10.5281/zenodo.3765373.

[16] R. E. Burkard, S. E. Karisch, F. Rendl, Qaplib–a quadratic assignment problem library, Journal of Global optimization 10 (4) (1997) 391–403.

[17] P. Lipinski, A hybrid evolutionary algorithm to quadratic three-dimensional assignment problem with local search for many-core graphics processors, in: C. Fyfe, P. Tino, D. Charles, C. Garcia-Osorio, H. Yin (Eds.), Intelligent Data Engineering and Automated Learning – IDEAL 2010, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 344–351.

[18] H. R. Lourenco, O. C. Martin, T. Stutzle, Iterated local search (2001). arXiv:math/0102188.

[19] G. Syswerda, Schedule Optimization Using Genetic Algorithms, Van Nostrand Reinhold, 1991, pp. 332 – 349.

[20] D. Callahan, B. L. Chamberlain, H. P. Zima, The cascade high productivity language, in: Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings., IEEE, 2004, pp. 52–60.

[21] Cray Inc., Chapel language specification v.986, Cray Inc.

[22] N. Dun, K. Taura, An empirical performance study of chapel programming language, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE, 2012, pp. 497–506.

[23] B. L. Chamberlain, S.-E. Choi, M. Dumler, T. Hildebrandt, D. Iten, V. Litvinov, G. Titus, The state of the chapel union, Proceedings of the Cray User Group (2013) 114.

[24] B. L. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, P. Sahabu, et al., Chapel comes of age: Making scalable programming productive, in: Cray User Group, 2018.

[25] T. Carneiro, N. Melab, An incremental parallel pgas-based tree search algorithm, in: The 2019 International Conference on High Performance Computing & Simulation (HPCS 2019), 2019.

[26] T. Carneiro, N. Melab, Productivity-aware design and implementation of distributed tree-based search algorithms, in: International Conference on Computational Science, Springer, 2019, pp. 253–266.

[27] J. Bezanson, A. Edelman, S. Karpinski, V. Shah, Julia: A fresh approach to numerical computing, SIAM Review 59 (1) (2017) 65–98. doi:10.1137/141000671.

[28] H. Shen, Interactive notebooks: Sharing the code, Nature 515 (7525) (2014) 151–152.

[29] T. E. Oliphant, Python for scientific computing, Computing in Science and Engineering 9 (3) (2007) 10–20. doi:10.1109/MCSE.2007.58.

[30] K. B. Wheeler, R. C. Murphy, D. Stark, B. L. Chamberlain, The chapel tasking layer over qthreads., Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2011).

[31] M. Mezmaz, M. Mehdi, P. Bouvry, N. Melab, E.-G. Talbi, D. Tuyttens, Solving the three dimensional quadratic assignment problem on a computational grid, Cluster Computing 17 (2) (2014) 205–217. doi:10.1007/s10586-013-0313-4.

[32] M. Mehdi, N. Melab, E. Talbi, P. Bouvry, Interval-based initialization method for permutation-based problems, in: IEEE Congress on Evolutionary Computation, 2010, pp. 1–8. doi:10.1109/CEC.2010.5586526.

[33] K. Kennedy, C. Koelbel, R. Schreiber, Defining and measuring the productivity of programming languages, The International Journal of High Performance Computing Applications 18 (4) (2004) 441–448.

[34] M. Snir, D. A. Bader, A framework for measuring supercomputer productivity, The International Journal of High Performance Computing Applications 18 (4) (2004) 417–432.

[35] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, L. Votta, Measuring high performance computing productivity, The International Journal of High Performance Computing Applications 18 (4) (2004) 459–473.

[36] J. Kepner, HPC productivity: An overarching view, The International Journal of High Performance Computing Applications 18 (4) (2004) 393–397.

[37] F. Cantonnet, Y. Yao, M. Zahran, T. El-Ghazawi, Productivity analysis of the upc language, in: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., IEEE, 2004, p. 254.

[38] J. Kepner, High performance computing productivity model synthesis, The International Journal of High Performance Computing Applications 18 (4) (2004) 505–516.

[39] G. Almasi, Pgas (partitioned global address space) languages, in: Encyclopedia of Parallel Computing, Springer, 2011, pp. 1539–1545.

[40] R. Bolze, F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E.-G. Talbi, I. Touche, Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed, International Journal of High Performance Computing Applications 20 (4) (2006) 481–494.