Unlocking Python Multithreading Capabilities using OpenMP-Based Programming with OMP4Py

César Piñeiro
Dept. of Electronics and Computer Science - CITIUS
University of Santiago de Compostela
Santiago de Compostela, Spain
cesaralfredo.pineiro@usc.es

Juan C. Pichel
CITIUS
University of Santiago de Compostela
Santiago de Compostela, Spain
juancarlos.pichel@usc.es

***Abstract*—Python exhibits inferior performance relative to traditional high performance computing (HPC) languages such as C, C++, and Fortran. This performance gap is largely due to Python's interpreted nature and the Global Interpreter Lock (GIL), which restricts multithreading efficiency. However, the introduction of a GIL-free variant in the Python interpreter opens the door to more effective exploitation of multithreading parallelism in Python. Based on this important new feature, we introduce OMP4Py with the aim of bringing OpenMP's familiar directive-based parallelization paradigm to Python. Its dual-runtime architecture design combines the benefits of a pure Python implementation with the performance and low-level capabilities required to maximize efficiency in compute-intensive tasks. In this way, OMP4Py offers both full Python support and the high performance required by HPC workloads.**

***Index Terms*—OpenMP, Python, Multithreading, Performance, Scalability**

I. Introduction

Python has become a leading programming language in recent years [1], valued for its ease of use, efficiency, and clear syntax. However, Python still lags behind low-level HPC (High Performance Computing) languages like C and Fortran when it comes to achieving high performance, mainly due to two key factors. First, since Python is an interpreted language, it incurs substantial overhead from translating source code into machine code at runtime. This drawback can be partially addressed with Just-In-Time (JIT) compilers such as Numba [2], which convert Python functions into optimized machine code on the fly using the LLVM compiler. This method seeks to deliver performance close to that of C or Fortran. However, Numba is particularly suited for numerical tasks, so programs that involve heavy string processing, complex data types, or significant input/output operations often do not benefit much. The second issue is related to how Python handles multithreading due to the existence of the Global Interpreter Lock (GIL). The GIL is a locking mechanism that restricts multiple threads from executing Python code at the same time, as it controls access to Python objects. It was initially introduced to simplify thread management and to prevent race conditions and memory corruption, thereby making concurrent programming safer and more manageable for developers. However, because only one thread can run Python code at any given moment, multithreading is ineffective for CPU-bound tasks, where performance benefits from true parallelism are expected. In this sense, the GIL stands as the primary barrier to efficiently using multi-core processors for parallel execution in Python.

Various efforts have been made to improve multithreading support in Python to mitigate the GIL issue, but key limitations remain [3]. However, a more definitive solution has been expected since the release of Python 3.13 in October 2024, which introduced thread-safety and allowed Python code to run without the GIL. In any case, this important feature does not close the significant performance gap between an interpreted language like Python and standard HPC compiled languages. Other approaches to exploit multithreading in Python are based on Numba, aiming to bypass the GIL while notably improving Python code performance [4], [5]. This comes at the expense of important restrictions on the use of many libraries, as well as certain Python objects and data structures.

To overcome these issues, we introduce OMP4Py[1], an implementation of OpenMP [6] for Python. OpenMP is a directive-based programming model widely recognized as the standard for exploiting multithreading parallelism in HPC. OMP4Py features a dual-runtime architecture, consisting of a pure Python runtime and a native C-based runtime generated using Cython [7]. This design combines the flexibility and ease of use of Python with the performance benefits of native execution, allowing OMP4Py to offer both full Python support and high computational efficiency. Experimental results indicate that OMP4Py performs well and scales effectively for both numerical and non-numerical tasks. Moreover, it shows promising potential for hybrid applications with mpi4py [8], addressing both intra- and inter-node parallelism.

II. Background & Related Work

A. OpenMP

Originally created for shared-memory computer systems, OpenMP [6] is a parallel programming model that aims to simplify the use of inherent concurrency in many algorithms.

This work was supported by MICINN [PID2022-137061OB-C22]; Xunta de Galicia [ED431C 2022/16, ED431G-2023/04]; and European Regional Development Fund (ERDF).

[1]It is publicly available at https://github.com/citiususc/omp4py

OpenMP primarily follows the fork-join model of parallel execution. The program begins with a single initial thread in this model. At parallel regions, the initial thread creates multiple parallel threads that concurrently execute the assigned tasks. The threads rejoin the original thread to continue running the program sequentially after the parallel tasks are finished. Programmers can avoid the manual handling of thread creation and task assignment necessary by low-level approaches such as pthreads in the POSIX library by using directives to instruct the compiler to generate multithreaded code at a higher level of abstraction. The OpenMP API is compatible with C/C++ and Fortran. Although initially designed for shared-memory architectures, OpenMP has expanded its capabilities to support heterogeneous computing as well. Since the introduction of the `target` directive family, OpenMP has enabled offloading computations to accelerators, such as GPUs, which often employ distributed-memory models internally. This extension allows OpenMP to be used in hybrid computing environments where both shared- and distributed-memory paradigms coexist. For this reason, OpenMP remains the standard for exploiting multithreading capabilities of modern multi-core CPUs while also enabling high-performance execution on heterogeneous architectures.

The OpenMP API standard specification[2] started in 1997 (version 1.0), and it continues to evolve, with new constructs and features being added over time. The latest release, version 6.0, was recently published in November 2024. However, most OpenMP programmers typically use only a subset of the OpenMP 3.0 specification released in 2008. This subset, referred to as the *OpenMP Common Core* [9], comprises the 21 most commonly utilized elements of OpenMP.

B. Advances in Python Multithreading

Python offers several parallelism approaches, each with notable limitations [3]. A common one is the `multiprocessing` library, which enables parallel execution by spawning subprocesses, each with its own Python interpreter and GIL. Frameworks like PyTorch [10], TensorFlow [11], and IgnisHPC [12] use this model to parallelize tasks. However, inter-process communication requires object serialization or shared memory, adding overhead and complicating API design. Subprocesses are also more resource-intensive to launch than threads, and many C/C++ libraries support multithreading but not multiprocessing.

Another way to leverage parallelism in Python is through multithreading in C extensions, where functions implemented in C can internally use multiple threads. For instance, Intel's NumPy distribution parallelizes operations this way. This method is effective for large computations but less so for many small ones or operations involving Python code. Since calling Python from C requires acquiring the GIL, even minimal Python involvement can limit scalability.

In any case, as previously discussed, the primary limitation to achieving true concurrency in multithreaded Python code is the GIL. Although there have been various attempts over the years to remove it [13], [14], none of them were seriously considered for integration into the official Python interpreter until recently [3]. With the release of Python 3.13 in October 2024, it is now possible to disable the GIL using the `--disable-gil` build configuration flag. This option introduces the required modifications to ensure that the interpreter is thread-safe. It marks the beginning of a gradual transition toward making the GIL disabled by default in future Python versions.

C. PyOMP, Numba and Cython

PyOMP [4], [5] is an approach for introducing parallel multithreading into Python, a prototype system that brings partial support for the OpenMP API to Python programs. PyOMP provides a set of compiler directives, runtime routines, and environment variables that instruct the compiler to generate multithreaded code. PyOMP leverages this model by integrating with Numba [2], a JIT compiler that translates a subset of Python code into efficient machine-level LLVM instructions. This translation allows PyOMP to bypass Python's GIL, thereby enabling true multithreaded execution. A key aspect of PyOMP's design is its reliance on NumPy arrays for data handling. Since Numba operates on statically typed data structures and compiles numerical functions to LLVM IR (Intermediate Representation), any array-based computations within a PyOMP-parallelized function must use NumPy arrays. This makes PyOMP particularly suitable for numerical and scientific computing workloads, where data is often already structured in this format.

However, PyOMP inherits some of the limitations of Numba. In particular, it struggles with Python's dynamic features, such as arbitrary function calls, object manipulation, and flexible data structures like dictionaries or lists. These dynamic behaviors are challenging to compile into efficient machine code, limiting PyOMP's applicability to programs that are more static and computation-intensive. Because Python lacks native syntax for compiler directives, PyOMP introduces a Pythonic interface using the `with` statement to define OpenMP constructs. For example, a parallel region in PyOMP is written as '`with openmp("parallel"):`' a direct analogue to the C/C++ directive '`#pragma omp parallel`'. This design choice makes PyOMP more accessible to Python developers while preserving the underlying OpenMP semantics. In its latest version (v0.2.0, released in April 2025), PyOMP supports approximately 90% of the OpenMP Common Core, with notable omissions such as the `nowait` clause and the `dynamic` scheduling policy in for loops[3]. This new version introduces support for some directives from more recent OpenMP standards, such as `teams` and `target`, with a focus on enabling GPU offloading.

Cython [7] is an optimizing static compiler that allows users to write Python code with C-like performance by optionally adding type declarations. It allows seamless interaction between Python and C/C++ code, making it ideal for speeding

[2]https://www.openmp.org/specifications [online, accessed Nov 3, 2025]

[3]https://pyomp.readthedocs.io/en/latest/openmp.html [online, accessed Nov 3, 2025]

up Python programs, wrapping C libraries, and building high-performance Python extensions. Cython presents a powerful alternative to Numba for cases where more compatibility is needed. While a JIT compiler like Numba can only provide significant performance improvements for certain workloads, Cython offers near-complete support for Python, with only minimal limitations. Cython optimizes the portions of code that can benefit from static typing and low-level optimizations, while leaving the rest of the code unaffected. In the worst case, the performance of Python code remains unchanged. This stands in contrast to Numba, where non-optimized code cannot be mixed with optimized code within the same function block. With Cython, however, developers have the flexibility to use any Python library or module without restricting the possibilities of the user, enabling the integration of both high-performance, Cython-optimized sections and pure Python code together. In the realm of numerical computing, Cython can further enhance performance by leveraging Pythran [15] as an additional backend for NumPy-based code. This allows Cython to compete with Numba in terms of raw performance while providing the user with the flexibility to fine-tune their results. By optimizing the compilation process, Pythran integration achieves even greater performance improvements and offers users more control to tweak and optimize their code for specific use cases, ensuring the best possible performance in numerical computations.

III. OMP4Py

OMP4Py is a novel implementation of the OpenMP standard, specifically designed for Python. It fully covers the specifications of version 3.0[4] and incorporates certain features from newer standards such as `declare reduction`. The OpenMP standard has historically provided support for C, C++, and Fortran, which are compiled, low-level languages that use directives for handling parallelism. These directives are interpreted by the compiler before the code is compiled. In C/C++, these commands begin with `#pragma`, while in Fortran, they are indicated by the marker `!$`. The main goal of OMP4Py is to introduce the well-known parallelization model of OpenMP to Python, allowing Python programmers to create parallel code with the same degree of control and flexibility as they have in C, C++, or Fortran. It is important to note that, as in standard OpenMP, incorrect usage of constructs (e.g., placing barriers inside work-sharing constructs) may lead to correctness issues such as deadlocks or data races, and it remains the programmer's responsibility to ensure conforming usage of the API. OMP4Py attempts to adapt the OpenMP model, ensuring that all code execution is handled natively using Python threads with a fully integration with Python's libraries. OMP4Py integrates the core functionalities of OpenMP into Python in the following ways:

- *Parser*: OMP4Py lets Python users incorporate parallel constructs into their code by adapting OpenMP's directive-based methodology. These directives instruct OMP4Py to transform the code for parallel execution.
- *Runtime*: The runtime is responsible for implementing the low-level routines generated by the parser, which define the behavior of the corresponding OpenMP directives and clauses. In addition, it provides a set of runtime library functions, similar to those in OpenMP, that allow users to obtain information about or control the parallel behavior of their Python scripts (e.g., `omp_get_num_threads()`).

[4]https://www.openmp.org/wp-content/uploads/spec30.pdf [online, accessed Nov 3, 2025]

```
1  from omp4py import *
2
3  @omp
4  def pi(n):
5      w = 1.0 / n
6      pi_value = 0.0
7      with omp("parallel for reduction(+:pi_value)"):
8          for i in range(n):
9              local = (i + 0.5) * w
10             pi_value += 4.0 / (1.0 + local * local)
11     return pi_value * w
12
13 print(pi(10000000))
```

Fig. 1. Example of a method for π calculation using OMP4Py.

A. Parser

Since Python lacks a preprocessor, we integrated OpenMP directives directly into the language while following Pythonic conventions. To do this, we defined an `omp` function that mirrors OpenMP directives in C/C++ in both syntax and behavior. This function itself does nothing at runtime—it simply serves as a container for OpenMP directives. For example, a directive like '`#pragma omp parallel num_threads(2)`' is written in Python with OMP4Py as '`with omp("parallel num_threads(2)"):`'. When used within structured blocks, the directive is placed inside a `with` statement (similar to PyOMP [5]); otherwise, it can appear as a standalone function call. Regardless of their placement, calls to the `omp` function alone do not produce any effect. Consequently, we must instruct the interpreter to transform the code according to each directive before execution. This is done using Python decorators. A decorator can modify the behavior of a function or class in a clear and elegant manner. To enable directive-based transformation, the target must be decorated with `@omp`. Figure 1 shows an example of Python code for the parallel calculation of π using OMP4Py. First, in line 3, the pi function is decorated with `@omp`, indicating that it contains OpenMP directives that need to be processed. Next, in line 7, a parallel region is started using '`omp("parallel for reduction(+:pi_value)")`'. This statement instructs OMP4Py to parallelize the following for loop, where each thread contributes to the reduction operation on `pi_value`. Finally, line 11 returns the computed value of π.

Since Python is an interpreted language, it does not include a compilation phase like lower-level languages that support OpenMP. Thus, OpenMP code generation occurs when the OMP4Py module is loaded. At this point, global definitions are evaluated, including imports, variables, and decorated functions or classes. When a decorator is applied, the interpreter

calls it with the target object as an argument, replacing the original with the transformed version. The `@omp` decorator processes all directives within a function or class, producing a new version with parallel behavior. It first uses the `inspect` module to extract the source code, then constructs an abstract syntax tree (AST) using Python's `ast` module. This tree, which is easy to analyze and modify, is traversed in order. Each directive is parsed, validated, and transformed. If any errors are detected, a `SyntaxError` is raised. Once processing is complete, the modified AST is compiled and executed with `exec`, and the decorated object replaces the original.

B. Runtime

The OMP4Py runtime, as previously mentioned, has two main responsibilities. The first is implementing the runtime API, which allows users to customize the execution environment and retrieve relevant information from it. This includes functions for controlling the number of threads, scheduling policies, and other execution details. The second objective is to implement the low-level routines generated by the parser to execute the corresponding OpenMP directives and clauses.

An early prototype of the OMP4Py runtime was developed in pure Python [16], implementing all features using only the language's standard modules and without any external dependencies. The experimental evaluation demonstrated that this implementation has significant potential for non-numerical workloads. However, the threading limitations of Python's interpreter (v3.13) hinder its scalability for numerical applications. It is important to consider that Python is still in the early stages of multithreading support and, for example, does not yet provide critical features such as atomic operations in its public API. These operations are essential for ensuring thread safety without relying on locks, and are therefore fundamental in OpenMP for efficiently managing shared data between threads. Although atomic operations are now part of the interpreter's internal implementation, they remain inaccessible at the Python language level. On the other hand, another well-known and important issue is the performance gap between standard Python code and the machine code generated by tools like Numba when using PyOMP. Although PyOMP restricts the use of important Python features, as we mentioned previously, it provides significantly reduced execution times in numerical computations, a level of performance that cannot be achieved with a runtime implemented purely in Python.

In this way, to overcome the above limitations, we introduce a completely new version of OMP4Py. It relies on Cython (see Section II-C) to generate a second native runtime, which we call `cruntime`. A new pure Python runtime was designed and implemented to enable efficient Cython code generation, ensuring that `cruntime` could be effectively produced using the Python runtime. From now on, we refer to the Python runtime simply as `runtime`. This design allows us to combine the benefits of pure Python code with the performance and low-level capabilities required to maximize efficiency in computational tasks. With both runtimes available, OMP4Py can be executed in three different modes:

- *Pure*: User's code makes calls only to the `runtime` written entirely in Python.
- *Hybrid*: Defined as the default option, this mode replaces the `runtime` with the `cruntime`, so that internal OpenMP API operations are implemented using native code instead of Python code.
- *Compiled*: This option uses the `cruntime` and also passes the user's code through Cython to generate native code. The resulting code links directly to the `cruntime`, eliminating Python interpreter overhead.

The execution mode can be selected passing different arguments to the `omp` decorator and importing the corresponding module. The execution flow can combine functions that use *Hybrid* and *Compiled* modes, as both rely on the `cruntime`, but they should not be mixed with the *Pure* mode. Each runtime operates independently, and they do not share information between them, which may lead to unexpected results.

As previously explained, the `runtime` module is a pure Python implementation, while the `cruntime` module is a native extension generated using Cython. The Python-based runtime implements the full logic of OMP4Py, structured into high-level modules that define the core logic and low-level modules that handle common basic operations. The `cruntime`, on the other hand, implements only the low-level modules to take advantage of native code performance, while the remaining logic modules are reused directly from the Python runtime. The `runtime` is written in `.py` files, like any typical Python module. In contrast, Cython introduces two special file types to enable integration between Python and C: `.pyx` and `.pxd`. The `.pyx` files contain the actual Cython implementation, similar to `.py` files but with a superset of Python syntax that enables easier static typing and direct calls to C-level functions. The `.pxd` files act as interface or header files, like C header files (`.h`), and are used to declare structures, types, and functions that can be shared between multiple Cython modules without Python overhead. These declarations help separate the interface from the implementation and allow the reuse and extension of C-level components across modules. To avoid duplicating logic and reduce maintenance complexity, as mentioned above, the `cruntime` reuses most of the code from the `runtime`. Its strategy is structured as follows: for each module, if the `cruntime` includes both a `.pxd` and a corresponding `.pyx` file, the native module is created by compiling the `.pyx` file. However, if only a `.pxd` file is available (declaring the expected interface) and no `.pyx` file exists, the system automatically falls back to copying and using the corresponding `.py` implementation from the `runtime`. In this case, the Cython compiler leverages the native types declared in the `.pxd` file to accelerate the Python code and generate a native module. With this methodology, it is possible to maintain a single implementation of the core logic and only duplicate the low-level modules.

C. Parallel Directive

At the start of execution, a Python program using OMP4Py begins with a single thread running the main program, just like

any other Python script. According to the OpenMP standard, this thread is known as the initial thread. OMP4Py initializes the context for this initial thread upon the first call to any function in its API. All threads in an OpenMP program are associated with either an implicit or explicit `parallel` directive. The initial thread, and any other threads created outside OpenMP constructs, such as those spawned using `threading`, `asyncio`, `concurrent.futures`, or the `multiprocessing` module, are not part of any OpenMP-created team by default. These threads are implicitly part of a single-thread `parallel` team that consists only of themselves. If one of these external threads subsequently calls any function from the OMP4Py API, a new OpenMP context will be created for that thread, treating it as a new and independent initial thread. Each of these threads can then use the full functionality of OMP4Py independently. However, it is the programmer's responsibility to manage any concurrency or synchronization issues that may arise when multiple initial threads are used in parallel. Threads created within OpenMP via a `parallel` directive, by contrast, are part of the team generated by that directive and share the OpenMP context derived from the original thread that executed the `parallel` directive. In OMP4Py, the context is implemented as a task stack, where the first task is the `parallel` region, and subsequent tasks are pushed onto the stack as the thread processes OpenMP directives, and popped as the directives complete. This context is stored locally for each thread. In the `runtime`, it is stored using Python's `threading.local`, while in the `cruntime`, it is stored in a variable marked with the C `thread_local` modifier, which ensures a separate instance of the variable for each thread, as handled by the compiler. Both runtimes use the same class structure to store task information. The `runtime` stores tasks using Python classes defined for each directive, while the `cruntime` uses C structs generated from those same classes via Cython. This ensures consistency in how tasks are represented across runtimes. However, since each runtime maintains its own separate context, threads from one runtime are treated as initial threads when used in the other.

The `parallel` directive allows a block of code to run concurrently across multiple threads. When used, it creates a team of threads, including the main thread, that execute the code simultaneously. OpenMP supports nested parallelism, which means threads can create new teams using nested `parallel` directives. This feature must be explicitly enabled with the `omp_set_nested` function. Even without explicitly defining parallel regions, all OpenMP programs start within an implicit single-threaded parallel context. This allows API functions such as `omp_get_num_threads` to operate globally. Inside a parallel block, variables declared within the block are local to each thread. Variables defined before the block are shared by default, which means all threads access the same value. To give each thread its own independent copy, a variable can be declared as private. These private copies start uninitialized and are discarded after the parallel block ends, leaving the original variable unchanged. To initialize private copies with the original value, use the firstprivate clause.

```
1   def pi(n):
2       w = 1.0 / n
3       pi_value = 0.0
4       def __omp_parallel():
5           nonlocal pi_value
6           __omp_pi_value = 0
7           for i in range(n):
8               local = (i + 0.5) * w
9               __omp_pi_value += 4.0 / (1.0 + local * local)
10          try:
11              __omp.mutex_lock()
12              pi_value += __omp_pi_value
13          except:
14              __omp.mutex_unlock()
15      __omp.parallel_run(__omp_parallel, ...)
```

Fig. 2. Code generated by OMP4Py for the `pi` function in Fig. 1, after processing the `parallel` directive.

The implementation of the `parallel` directive requires encapsulating the parallel code inside a function so that it can be executed by multiple threads. To preserve consistency with the original code, it is necessary to analyze how local variables are used within the parallel block. Since the original code is already inside a function, the generated parallel code will be placed inside an inner function, which by default has read access to all variables in the outer function. However, this access is limited to reading; if a variable is modified within the inner function, it must be explicitly declared as `nonlocal` so that Python updates the variable in the outer function instead of creating a new local instance.

Figure 2 shows the code generated by OMP4Py only for the `parallel` directive presented in the example of Figure 1. It is important to note that, in code generation examples, `__omp` refers to the `runtime/cruntime` module, and the prefix `__omp_` is used for all internal OMP4Py symbols. Generated names will include numerical suffixes to avoid naming collisions. Once transformed, the decorator and directives are removed, as they are no longer necessary and must be eliminated to prevent repeated processing. The example shows that the code originally inside the `parallel` directive has been moved into a function named `__omp_parallel`. The variable `w` (line 2) is only read and therefore does not require any special handling. In contrast, `pi_value` (line 3) is modified inside the block and must be declared as `nonlocal`. Furthermore, because `pi_value` is part of a reduction clause (line 7, Figure 1), the parser first creates a private copy of the variable, named `__omp_pi_value`, which replaces `pi_value` within the parallel function. Once the parallel code finishes, `pi_value` is updated using the private value `__omp_pi_value` (line 12) inside a critical section protected by a mutex (lines 11 and 14). Finally, the parallel execution by multiple threads is triggered by the call to `parallel_run`, which internally creates the threads, assigns them a context, and starts the parallel task by executing the `__omp_parallel` function. Both runtimes create threads using Python's `threading` module. Although creating threads in C might seem more efficient, each thread in both runtimes must call a Python API function to initialize its Python stack before interacting with the interpreter. Therefore, any potential benefit of creating them in C would be lost. The `parallel_run` function accepts additional arguments, which

```
1    __omp_bounds = __omp.for_bounds([0, n ,1])
2    __omp.for_init(__omp_bounds, ...)
3    while __omp.for_next(__omp_bounds):
4        for i in range(__omp_bounds[0], __omp_bounds[1]):
5            local = (i + 0.5) * w
6            __omp_pi_value += 4.0 / (1.0 + local * local)
```

Fig. 3. Code generated by OMP4Py for the loop inside the `pi` function in Fig. 1, after processing the `for` directive. The code replaces lines 7-9 in Fig. 2.

have been omitted here for simplicity. For example, if the user specifies the `num_threads` clause, the corresponding thread count will be passed as a parameter to the function. Finally, note that according to the OpenMP standard, runtimes do not propagate exceptions thrown within parallel regions; they must be caught and handled locally in each thread.

D. Worksharing Constructs

In OpenMP, efficient parallelism relies on how tasks are distributed among threads. Rather than having each thread execute the same code, OpenMP provides work-sharing constructs that allow developers to logically divide tasks. This mechanism ensures that different threads contribute to the overall workload without redundant computation. There are three primary forms of work-sharing: `for`, `sections`, and `single`. Each serves a distinct purpose: `for` distributes iterations across threads, `sections` divides the program into separate code blocks executed by different threads, and `single` ensures a particular segment is executed by only one thread, avoiding duplicated effort. The first thread to reach a work-sharing region can begin its portion of the task immediately. However, unless specified otherwise, it must wait until all other threads in the team have arrived at the same region before continuing. This synchronization is implicit and helps maintain program correctness, particularly when subsequent operations depend on the completion of shared tasks. If developers want to bypass this implicit barrier and allow threads to proceed independently once their task is done, they can use the `nowait` clause.

The `for` directive is widely used to parallelize loops by distributing iterations among threads according to a selected scheduling policy. These include `static`, which assigns chunks in advance in a round-robin fashion, `dynamic`, where threads request chunks as they complete previous ones, and `guided`, which uses decreasing chunk sizes to reduce scheduling overhead. Other options like `auto` and `runtime` allow the compiler or the execution environment to determine the scheduling policy. To improve flexibility, developers can also use additional clauses. For example, `collapse` enhances parallelism in nested loops by combining them into a single iteration space, while `ordered` allows specific sections within a loop to maintain a defined execution order when necessary.

Figure 3 illustrates, as example, the code generated by OMP4Py to implement the `for` directive shown in Figure 1. Recall that `__omp` refers to the `runtime/cruntime` module, and all internal OMP4Py identifiers use the `__omp_` prefix. The `for_bounds` (line 1) function initializes the loop boundaries by taking the `start`, `end`, and `step` values from the original loop's `range` function. Although Python's `range` allows these values to be omitted, all three are explicitly specified here. This is important because, when using the `collapse` clause, the triplets for all nested loops are also included. The function performs initial computations, such as determining the number of iterations, and generates the `__omp_bounds` array containing all relevant information. Note that this approach covers traditional `range`-based loops but does not support Python list comprehensions, as they do not align with the structured loop semantics required by OpenMP. The next call, `for_init` (line 2), is responsible for preparing the parallel loop execution environment, creating the loop task and adding it to each thread's context. This function determines how the iteration space is divided according to the scheduling policy and prepares the initial chunk(s) for each thread. It receives all relevant information from the clauses specified in the directive and updates `__omp_bounds` accordingly to reflect the parallel distribution. It is important to note that `__omp_bounds` is not a shared variable among threads and each thread maintains and updates its own independent copy. The core of the loop is driven by `for_next` (line 3), which acts as a condition to continue iteration within the thread. This function checks whether more work is available for the current thread and updates the bounds if necessary. Inside the while loop, the standard Python for loop iterates over the subrange assigned to the thread, from `__omp_bounds[0]` to `__omp_bounds[1]`, which represent the current chunk, allowing independent execution of loop iterations.

Figure 3 also illustrates how OMP4Py strategically selects the way it generates code for directives, highlighting the performance advantages of the `cruntime`. To maintain efficiency, the standard Python `range` function is preserved instead of being replaced by a custom iterator or a generator using `yield`, as range is a built-in function highly optimized at the C level. All mathematical computations related to workload distribution are offloaded to runtime-implemented functions. This design allows `cruntime` to execute these computations as native machine code. In particular, with the help of Cython, the `for_bounds` function generates a highly efficient numeric array that stores the iteration boundaries. This enables Python to simply read the required positions from memory, minimizing overhead and keeping the control logic lightweight and fast. On the other hand, unless the `static` scheduler is used, threads must coordinate dynamically to divide the work. The `dynamic` and `guided` scheduling policies require sharing a counter that keeps track of the last assigned iteration. The implementation of this mechanism is critical to the overall performance of the scheduling policy and presents several challenges. Thread communication is possible because threads created by the same parallel directive share a queue for exchanging information. Still, the threads must coordinate to determine who creates the shared counter and how its updates are managed. In contrast, `cruntime` uses atomic operations, where counter creation is done with an atomic swap,

```
1   @omp
2   def fibonacci(n):
3       if n <= 1:
4           return n
5       fib1 = 0
6       fib2 = 0
7       with omp("task"):
8          fib1 = fibonacci(n - 1)
9       with omp("task"):
10         fib2 = fibonacci(n - 2)
11      omp("taskwait")
12      return fib1 + fib2
13
14  @omp
15  def run(n):
16      with omp("parallel"):
17          with omp("single"):
18              fibonacci(n)
```

Fig. 4. Example of Fibonacci number calculation using OMP4Py tasks.

and updates are performed using a `fetch_add` operation. This significantly boosts performance by enabling lock-free, hardware-level synchronization, which is much more efficient.

The `sections` and `single` directives share a similar underlying mechanism, as both are designed to assign specific blocks of code to individual threads. Conceptually, `single` can be viewed as a special case of `sections` with only one `section`. In practice, the implementation of `sections` resembles that of a parallel for loop using a shared counter, similar to the one used in the `dynamic` scheduling policy, to manage work distribution. Each section is associated with a unique sequence ID, which is a fixed integer assigned in the order the sections appear in the code. At runtime, a shared counter is incremented by the threads. Each thread then compares the current counter value to the sequence ID of each section. If the values match, the thread executes that section. This approach ensures that each section is executed exactly once by a single thread, avoiding conflicts or duplication.

E. Tasking Directive

Tasking in OpenMP offers a powerful and adaptive approach for parallelizing workloads that are irregular, dynamic, or not easily divisible upfront. The `task` directive defines individual units of work, called tasks, which can be picked up and executed by any thread in the current team. When a thread encounters a `task` directive, it packages the enclosed code and its context into a task object. These tasks are not executed immediately; instead, they are usually placed in a shared task queue. Threads dynamically pull tasks from this queue as they become idle, ensuring efficient resource use. Task execution can be synchronized explicitly with the `taskwait` directive or implicitly through the runtime's scheduling. Regardless of timing, all tasks complete before the team of threads finishes its execution scope.

Figure 4 presents a recursive implementation of Fibonacci number calculation using the tasking model. In this example, each recursive call to `fibonacci(n-1)` and `fibonacci(n-2)` (lines 8 and 10) is wrapped in a task directive, allowing to create two separate tasks that can be executed in parallel by any available thread in the team. Because the function is recursive, this task creation process repeats at each level of the call stack, potentially generating a large number of fine-grained tasks. This pattern showcases how tasking can be used to exploit parallelism in divide-and-conquer algorithms, where the workload naturally branches out into smaller subproblems. The implementation of the `task` directive follows the same structural approach as `parallel`, encapsulating the task body inside an inner function so it can be scheduled for execution by any thread in the team. Variable scoping is handled in the same way, and in the example, `fib1` and `fib2` are shared variables used to store the results of the tasks. The generated code is functionally equivalent to that shown in Figure 2, with the call to `parallel_run` replaced by a call to `task_submit`. This function places the task into a shared queue, accessible to all threads within the team. Meanwhile, the `taskwait` directive is translated into a call to the `task_wait` function, which starts retrieving tasks from the queue. Each retrieved task can be in one of three possible states: `free`, `in-progress`, or `completed`. If the task is `free`, the thread marks it as `in-progress` and begins execution. Once the task finishes, it is marked as `completed`. If the task is already `in-progress`, another thread is executing it, so the current thread skips it and checks for other tasks. When no more tasks are available, the thread waits until all `in-progress` tasks finish, ensuring that all direct child tasks have been completed before continuing.

The shared task queue is created by the `parallel` directive and implemented as a linked list, where each node represents a task and stores its execution state, a completion event, the task function itself, and a next-reference. The completion event allows threads to wait for a task to finish, which is essential for implementing the `taskwait` directive. In the `runtime`, this is handled using Python's `threading.Event`. Although interpreters with free-threading support implement this object using atomic variables, the `cruntime` bypasses Python code entirely by interfacing directly with `PyEvent`, the internal implementation underlying `threading.Event`, in order to eliminate the overhead associated with the Python object. Another key difference lies in task enqueuing: the `runtime` uses a mutex to update the next-reference for thread safety, while the `cruntime` relies on the atomic `compare_exchange` operation to update the reference without locking.

Finally, the task scheduling policy in OpenMP is closely tied to its implicit barrier semantics, ensuring synchronization among threads. As discussed earlier, several directives introduce an implicit barrier at the end, requiring all threads to wait until the team reaches the same point. Rather than remaining idle, OpenMP allows threads to consume pending tasks from the shared queue during this wait. Internally, the barrier uses an event-based mechanism: each thread waits on an event, and the last thread to arrive signals it, allowing all to proceed. If new tasks are submitted while some threads wait at the barrier, they are reawakened to execute the work. Once the queue is empty and all tasks are complete, threads resume waiting if the team has not fully reached the barrier. In Figure 4, the `single` directive (line 17) introduces an implicit barrier. As the thread executing the block begins generating tasks, the other threads,

initially waiting for the directive to complete, are reactivated and start consuming the newly submitted tasks.

F. API

The OMP4Py API is composed of OpenMP-style runtime library functions and the `omp` decorator, which drives the code transformation process. To use the API, users should begin by importing the `omp4py` module. Upon import, the module attempts to load the compiled `cruntime` backend if available, *Hybrid* mode; otherwise, it defaults to the Python `runtime`, *Pure* mode. The `cruntime` is included when OMP4Py is installed via pip, but it may be unavailable if the library is obtained from source or run on platforms that lack a compatible compiled extension. If you explicitly wish to use the Python `runtime`, you can import `omp4py.pure`, which ensures that only the Python-level implementation is used.

Any function or class that uses directives must be annotated with the `omp` decorator. While the decorator automatically parses and transforms code, it also accepts optional arguments to customize its behavior. For example, `cache` enables reuse of transformed code by storing `.py` and `.pyc` files. `dump` outputs the transformed source for inspection. `debug` shows transformation-time debug information. `compile` uses Cython to generate native code, always caching the result. `force` disables caching by reprocessing on every run and, finally, `options` allows additional flags for the parser or compiler (e.g., `cython_cdivision = True` enables Cython's `cdivision` optimization). Default argument values can be set via environment variables using the prefix `OMP4PY_` followed by the parameter name in uppercase.

Finally, when we select the *Compiled* mode, `@omp(compile = True)`, we can improve performance by explicitly specifying the native types of numeric variables. In particular, annotating variable declarations with Python's built-in types `int` and `float` allows for straightforward type annotations without dealing directly with Cython's low-level details.

IV. Experimental Results

Next, we evaluate the performance and scalability of different Python applications parallelized using OMP4Py. We consider all four OMP4Py execution modes: *Pure*, *Hybrid*, *Compiled*, and compiled with data type annotations (i.e., explicitly specifying `int` or `float` for numeric variables). For simplicity, we refer to the latter as *CompiledDT*. For comparison, we also include performance results from PyOMP, a Numba-based OpenMP prototype. Experiments were conducted on a server with a 32-core Intel Xeon Ice Lake 8352Y @2.2GHz processor and 256 GB of RAM. Hardware threads (Intel's Hyper-Threading) are disabled on the server. The software used was Python 3.14b1 (without GIL)[5], Cython v3.1.0, NumPy v2.2.5, mpi4py v4.0.3, NetworkX v3.4.2, and PyOMP v0.2.0 (April 2025). Running times were averaged over 10 measurements for each test.

[5]According to PEP 779 (https://peps.python.org/pep-0779, June 2025), the no-GIL build of Python 3.14 is now a *supported* feature and is considered functionally correct. The PEP identifies remaining concerns primarily in performance and memory, not correctness. In our tests with Python 3.14b1, we found no issues, and results matched those from a GIL-enabled interpreter.

TABLE I
STATIC CHARACTERISTICS OF EVALUATED BENCHMARKS.

	OpenMP Features	Synchronization
fft	`parallel`, `for`	Implicit barriers
jacobi	`parallel`, `for reduction(+)`, `single`	Explicit `barrier`
lu	`parallel`, multiple `for` loops, `single`	Implicit barriers
md	`parallel reduction(+)` with inner `for`, `parallel for`	Implicit barriers
pi	`parallel for reduction(+)`	Implicit barriers
qsort	`parallel`, `single`, `task` with `if` clause	Implicit barriers
bfs	`parallel`, `single`, `task`	Implicit barriers

A. Numerical Algorithms

We have selected seven algorithms that represent different types of numerical application patterns to evaluate the performance and scalability of OMP4Py. In particular:

- *Fast Fourier Transform (fft).* The algorithm efficiently computes the Discrete Fourier Transform (DFT) of a sequence, converting a signal from the time domain to the frequency domain. Performance tests used a complex data vector of 16 million numbers.
- *Jacobi method (jacobi).* The iterative algorithm solves systems of linear equations $A \cdot x = b$, where A is a matrix and x, b are vectors. Each iteration updates the solution using values from the previous iteration. A $3k \times 3k$ square matrix A was used, with up to 1,000 iterations and a stopping criterion of 1×10^{-6} error tolerance.
- *LU decomposition (lu).* This method factors a matrix A into a lower triangular matrix L and an upper triangular matrix U, such that $A = L \cdot U$. LU decomposition simplifies solving linear systems, matrix inversion, and determinant calculation. We applied it to a $2k \times 2k$ square matrix.
- *Molecular dynamics simulation (md).* The simulation studied particle motion over time using the velocity Verlet integration scheme to update positions, velocities, and accelerations. We simulated 8,000 particles interacting via a central pair potential.
- *Riemann integration (pi).* The area under the curve $y = \frac{4}{1+x^2}$ from 0 to 1 approximates π. We estimated this integral using numerical summation with 20 billion intervals.
- *Quicksort (qsort).* This algorithm recursively partitions an array around a pivot element, sorting elements into lower and higher subarrays. We applied it to an array of 400 million floating-point numbers.
- *Pathfinding (bfs).* Solved via breadth-first search on a 2.1k$\times$ 2.1k grid (entrance at the top-left, exit at the bottom-right). Zeros are paths and ones are walls; moves are allowed only between 0-cells, and each feasible move spawns a task.

Table I displays a summary of the static characteristics of the benchmarks. All the source codes can be found in the OMP4Py repository. Figure 5 shows the execution times of the different parallel benchmarks using between 1 and 32 threads, considering all OMP4Py execution modes: *Pure*, *Hybrid*, *Compiled*, and *CompiledDT*. Results show that the scalability of the *Pure* Python implementation is limited. For

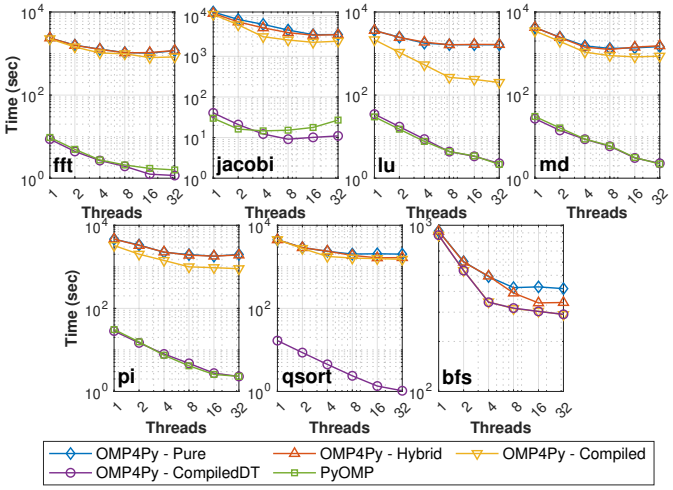

Fig. 5. Scalability of the parallel numerical applications. Axis in log scale.

instance, using 16 threads is beneficial in only a few cases. The maximum speedup, calculated as the ratio of sequential to parallel execution time, is 3.6× with the *jacobi* application. These results demonstrate that the current version of the Python interpreter (v3.14b1) still lacks mature support for multithreading, with several unresolved implementation issues that hinder its efficient use. In particular, as of September 2025, Python developers are dealing with more than 80 open issues labeled under the free-threading topic, half of which are bugs. While this new version is an important step forward compared to v3.13[6], it still limits the use of the OMP4Py *Pure* implementation to a few number of threads when running compute-intensive applications. However, it is important to highlight that as these issues in the Python interpreter are progressively resolved, the scalability limitations will gradually disappear without requiring any modifications to its implementation.

Performance differences between the *Pure* and *Hybrid* versions are generally very small but consistently favor the *Hybrid* mode. The advantage is more pronounced for *jacobi*, *qsort* and *bfs*, due to the specific characteristics of each benchmark: *jacobi* performs iterative calls to the `single` directive, while *qsort* and *bfs* exploits fine-grained task parallelism. In both cases, the *Hybrid* implementation benefits from the `cruntime` optimizations, which avoid mutex locks and instead rely on faster atomic operations for synchronization.

Regarding OMP4Py's *Compiled* mode, it outperforms the *Pure* and *Hybrid* versions in all cases considered. For example, when using 32 threads, it is on average 2.5× faster than the *Pure* implementation. Additionally, this version shows a clear improvement in scalability, with speedups of up to 10.6× observed at 32 threads. This improvement is due to optimizations introduced through Cython compilation of user code, which is executed as native C code rather than interpreted Python. However, the performance gains are not huge compared to the *Pure* and *Hybrid* versions. This is because Cython, by default, is conservative in type inference and assumes generic Python objects unless data types are manually annotated, resulting in additional overhead from Python's dynamic type system, including reference counting and boxed arithmetic.

On the other hand, execution times using the *CompiledDT* mode are significantly faster than those of the other OMP4Py modes. Notably, performance differences can reach up to three orders of magnitude. For instance, using 32 threads, the *CompiledDT* mode is on average 785× faster than the pure Python implementation across all applications. This dramatic improvement is due to the use of explicit native data types in performance-critical sections of the code. As previously commented, Cython is conservative with type inference and assumes generic Python objects unless instructed otherwise. This leads to additional overhead from Python's dynamic type system. In *CompiledDT* mode, data types such as integers and floating-point variables are explicitly declared using C-native types, allowing Cython to generate highly optimized machine code. This eliminates the cost of dynamic dispatch and enables tight, low-level numerical loops that rival the performance of hand-written C code. Scalability is also good, with execution times decreasing up to 32 threads, except for the *jacobi* application. The average speedup with 32 threads is 10.1×, with a maximum of 16.2× for the *qsort* application.

Finally, the performance comparison with PyOMP shows that OMP4Py (*CompiledDT* mode) offers slightly better overall performance in terms of both execution times and scalability. For example, excluding *qsort*, the average speedup with 32 threads is 9.9× for PyOMP, which means that OMP4Py achieves approximately 4.5% better performance on average. Note that the *qsort* application cannot be implemented in PyOMP, as it relies on a parallel recursive algorithm using OpenMP tasks with the `if` clause, which is not supported in the current version of PyOMP. For *bfs*, an error is raised during execution of the PyOMP code related to Numba.

B. Limitations of PyOMP

As previously mentioned, PyOMP is a fork of the Numba project. The `njit` decorator in Numba improves performance by compiling Python functions into machine code, but it also introduces important limitations. It restricts the use of functions from libraries that are not optimized for Numba, as well as certain Python objects and data structures. Only specific libraries, such as math and NumPy, are supported and their functions can be used within `@njit`-compiled functions. In contrast, OMP4Py is explicitly designed to fully support Python language and its libraries, including those that may not be compatible with Numba's restrictions. To demonstrate the benefits of this full Python support, we have implemented two applications as illustrative examples:

- *Clustering coefficient*. The clustering coefficient of a node in an unweighted, undirected graph is the fraction of possible triangles through that node that actually exist. This application computes the clustering coefficient for each node in the graph. We used a 300k-node graph with 100 edges

[6]Due to page limitations, results with Python v3.13 are not included; however, its scalability is worse than that of v3.14b1.

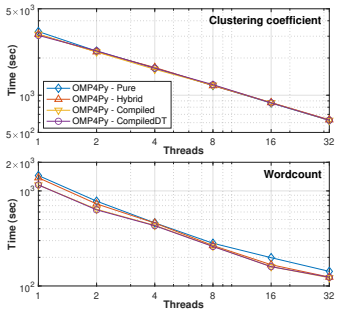

Fig. 6. Scalability of the *clustering coefficient* and *wordcount* applications. Axis in log scale.

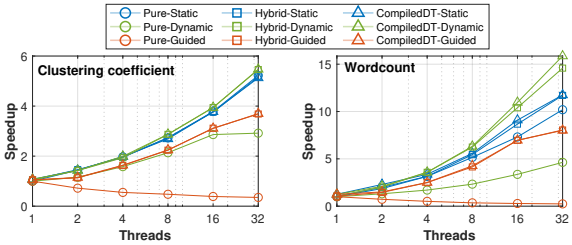

Fig. 7. Speedups using different scheduling policies of the *clustering coefficient* and *wordcount* applications. X axe in log scale.

per node, generated and processed using NetworkX [17]. PyOMP cannot run this benchmark because Numba cannot compile NetworkX's `Graph` object and related functions.

- *Wordcount*. It is performed using as input the Spanish Wikipedia dump as of May 2025 (21 GB). The latest version of PyOMP (April 2025) is still based on an older release of Numba and lacks support for compiling Python dictionaries, which are required for this benchmark.

Figure 6 shows the execution times for the above applications. First, observe that the scalability of the *clustering coefficient* calculation is similar across all OMP4Py modes. Compiled modes offer no significant advantage because Cython's optimization capabilities are limited to the portions of the code interacting directly with the NetworkX external library. The remaining computations are handled by NetworkX itself, which means optimization beyond the function call is not possible. On the other hand, the *wordcount* benchmark shows slight improvements with the `cruntime`-based modes. However, these gains are limited because *wordcount* primarily involves string and dictionary operations, which Cython cannot optimize effectively. In any case, both applications scale with the number of threads, achieving an average speedup of approximately 5× and 10× with 32 threads, respectively.

Unlike OMP4Py, PyOMP only supports the `static` scheduling policy. To broaden the evaluation, we include results with the `dynamic` and `guided` policies in Figure 7 (see Section III-D). Chunk size is 300. Speedups are calculated with respect to the *Pure* execution time using one thread and static scheduling. The results show the good performance of dynamic scheduling, particularly for *wordcount*, where load imbalance is more pronounced. In contrast, guided scheduling performs worse than the other policies, especially in the *Pure* mode, which does not scale effectively. We also vary the chunk size. With a chunk half the original size (150), we observe a slight time reduction for *wordcount* in *Pure* mode, while the other cases become slightly slower compared to Figure 7. With a doubled chunk (600), there are no noticeable changes, except for *wordcount* in *Pure* mode, where times worsen.

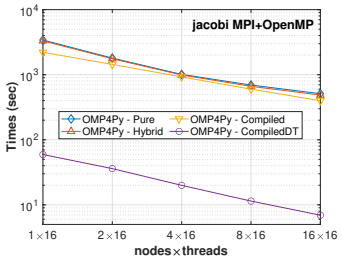

Fig. 8. Scalability of the *jacobi* hybrid MPI/OpenMP application using different number of computing nodes (16 threads per node). Axis in log scale.

C. Hybrid MPI/OpenMP Parallel Applications

A key feature of OMP4Py is its ability to combine with mpi4py [8] to implement hybrid parallel applications exploiting intra- and inter-node parallelism. The mpi4py package provides Python bindings for the MPI standard [18], the dominant HPC programming model. Although mpi4py is a Python interface to the MPI C library, Numba cannot use MPI within its functions because it treats mpi4py as an external library. Numba compiles only Python code and does not integrate external libraries like MPI, so it cannot translate mpi4py calls into native code. As a result, PyOMP cannot be combined with mpi4py.

As a case study, we implemented a hybrid MPI/OpenMP version of the Jacobi method for solving $Ax = b$ as described in Section IV-A. MPI distributes the matrix A and vector b across processors, each responsible for a subset of rows and elements. Within each iteration, processors update x using OpenMP based on their local data. The updated vector x is exchanged using `MPI_Allgather` to maintain consistency across processors. Convergence is monitored by computing the global error, with `MPI_Allreduce` used to evaluate the stopping criterion.

Figure 8 illustrates the scalability of the hybrid *jacobi* application across different numbers of nodes. Note that the *CompiledDT* version used a $20k \times 20k$ matrix as input, while the other modes used a $3k \times 3k$ matrix, to demonstrate scalability with increasing node counts. The application scales in all OMP4Py modes, with the *Compiled* modes achieving the highest performance. For example, speedups over single-node performance of 1.6×, 3×, 5.2×, and 8.6× were observed with *CompiledDT* on 2, 4, 8, and 16 nodes, respectively. Note that there is some overhead in the *Compiled* modes, as MPI functions must be invoked from Python after each iteration to synchronize the result array. However, although mpi4py is a Python library, it interfaces with a C-based MPI implementation that can directly access NumPy arrays at the memory level, thereby limiting the impact of this overhead.

V. Challenges in Extending OpenMP Support

Although OMP4Py primarily targets OpenMP 3.0, its development was guided by later specifications, including the 6.0 draft. Accordingly, it selectively incorporates improvements that enhanced functionality or were straightforward to implement. For example, the `default` clause supports all variants (shared, none, private, firstprivate), even though only the first two were in 3.0. Custom reductions via `declare reduction` (introduced in 4.0) are included, as are minor extensions such as the optional argument in `nowait`. Finally, while full OpenMP 6.0 functionality is not implemented, the OMP4Py parser supports its complete syntax (e.g., spaces or underscores in combined directives, semicolons separating clauses, directive names within clauses), easing future extensions.

While OMP4Py already incorporates selected improvements beyond OpenMP 3.0, extending full support to the latest standard introduces non-trivial challenges due to both the increased complexity of directives and clauses and the larger number of constructs. Some directives, such as `teams` or `taskloop`, are relatively straightforward since their semantics build on existing constructs and do not introduce new conceptual or architectural difficulties. The main effort in their inclusion lies in the development time required to implement the necessary transformations and runtime support.

On the other hand, other constructs introduce more significant challenges, particularly task dependencies added in OpenMP 4.0. In version 3.0, tasks are managed with an optimized queue based on the producer–consumer pattern; supporting dependencies would require replacing this queue with a dependency graph and implementing a scheduling policy to ensure tasks execute only after their dependencies are satisfied. Although this problem is well studied in other OpenMP implementations, the main difficulty in OMP4Py lies in how dependencies are defined: the standard relies on variables marked as `in` or `out`. In low-level languages this can be handled through memory addresses, but in Python this approach is infeasible. Variable names might work within a single function if no duplication occurs, yet full compliance with the standard demands a more robust mechanism. Using Python's `id` function is a possible first step, but it fails for immutable objects with identical values, leaving support for task dependencies as a major challenge for future work.

The second major challenge concerns the features introduced in recent OpenMP standards to support accelerators and heterogeneous computing. These standards define the concept of a device—a computational unit separate from the host CPU, such as a GPU, FPGA, or other specialized processor. Device programming requires mechanisms to offload code and data, manage memory transfers efficiently, and ensure synchronization between host and device. Extending these capabilities to Python raises additional difficulties. The runtime must be extended to detect and manage devices: whereas current functions execute only on the CPU, they must behave consistently across devices and maintain device-specific information, with parts of the runtime able to run on the accelerator. User code presents further challenges, since Python must be translated or compiled into code executable on the target accelerator, much like C requires a CUDA-capable compiler to run on NVIDIA GPUs. This can be achieved either through tools that generate device-compatible binaries or through interpreters capable of executing directly on the accelerator. In the current implementation, Cython is used to compile both the runtime and user code, and when combined with Numba, these tools provide a promising foundation for robust device support while preserving Python's high-level abstractions.

Finally, new constructs for memory allocation and vectorization are of low importance to most Python users, who prefer abstraction from low-level memory and vectorization details. Python already offers high-level libraries like NumPy with efficient internal vectorization. If OMP4Py supports these constructs, the practical path is to delegate them to such libraries by translating user operations into library calls, allowing compliance with the OpenMP standard while leveraging optimized Python implementations and minimizing direct low-level handling.

VI. Conclusions

We have presented OMP4Py, which brings OpenMP's directive-based parallelization paradigm to Python. It fully supports the complete OpenMP 3.0 API and includes selected directives from later versions, but it does not provide features such as accelerator offloading, full tasking, or advanced synchronization from newer standards. Its goal is two-fold: to reduce the performance gap caused by Python's interpreted nature, and to take advantage of GIL-free Python interpreters for efficient multithreading. To this end, OMP4Py employs a dual-runtime architecture, consisting of a pure Python runtime and a native C-based runtime generated using Cython. As a result, OMP4Py supports three modes of operation: *Pure*, using the Python-only runtime; *Hybrid*, using the native C-based runtime; and *Compiled*, which also uses the C-based runtime but compiles user code with Cython for maximum performance. There is also a variant of the *Compiled* mode that includes explicit numerical data type annotations.

Experimental results show that the scalability of the *Pure* implementation is limited when running numerical applications. *Hybrid* mode consistently outperforms *Pure*, though with modest gains, due to native runtime optimizations. The *Compiled* mode offers further improvements in execution time and scalability through Cython-based compilation, but its benefits are limited by default type inference. In contrast, the *Compiled* mode with explicit native data types achieves significantly higher performance and scalability, reducing Python runtime overhead by up to three orders of magnitude. On the other hand, this version delivers slightly better overall execution times and scalability than the Numba-based PyOMP tool. Additionally, unlike PyOMP, OMP4Py fully supports the Python language and its libraries, allowing it, for example, to be combined with mpi4py to build hybrid applications that exploit both intra- and inter-node parallelism.

Additional data related to this paper may be found in [19].

ARTIFACT APPENDIX

This artifact contains the source code of OMP4Py, all benchmark programs used in the evaluation, and the scripts to reproduce the results presented in the paper. We provide an automated script to run each experiment and generate the corresponding results for both OMP4Py and PyOMP. Each framework is preconfigured in a dedicated Docker image, ensuring full reproducibility and simplifying the setup across different systems.

A. Artifact Check-List (Meta-Information)

- **Algorithm:** OpenMP-style parallel loop scheduling, work-sharing, and reduction mechanisms.
- **Program:** OMP4Py (Python package).
- **Compilation:** Python bytecode (no native compilation required).
- **Transformations:** Source-to-source parallel transformation using decorators.
- **Binary:** Not applicable.
- **Model:** Shared-memory parallel execution model.
- **Data set:** Synthetic data generated from a fixed seed and Spanish Wikipedia for Wordcount (https://dumps.wikimedia.org/eswiki/latest/eswiki-latest-pages-articles.xml.bz2)
- **Run-time environment:** Python 3.14b1 free-threading
- **Hardware:** Multi-core CPU (reference hardware: 32-core Intel Xeon Ice Lake 8352Y @ 2.2GHz)
- **Run-time state:** Deterministic with configurable random seed
- **Execution:** Command-line with Python benchmark main script
- **Metrics:** Execution time and speedup
- **Output:** Execution time and benchmark result
- **Experiments:** Numerical algorithms : Fast Fourier Transform (fft), Jacobi method (jacobi), LU decomposition (lu), Molecular dynamics simulation (md), Riemann integration (pi), Quicksort (qsort) and Pathfinding (bfs). Non-Numerical algorithms: Clustering coefficient and Wordcount
- **How much disk space required (approximately)?:** Minimal disk usage; computations are performed entirely in memory.
- **How much time is needed to prepare workflow (approximately)?:** 1–2 minutes.
- **How much time is needed to complete experiments (approximately)?:** Note that with the default sizes, all the experiments together may take many hours to complete. For example, the sequential run for one execution mode of the longest benchmark alone takes about two hours on the reference hardware.
- **Publicly available?:** Yes, https://github.com/citiususc/omp4py
- **Code licenses (if publicly available)?:** GPL-3.0 license
- **Data licenses (if publicly available)?:** Not applicable.
- **Workflow automation framework used?:** Not applicable.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.17987713

B. Description

1) How to Access: OMP4Py is publicly available and can be installed on any Python 3.12 and later, which include the Global Interpreter Lock (GIL). However, to fully exploit multithreading for scaling applications, it is necessary to use Python 3.13 (free-threading) or later, which offers a no-GIL option. The source code, all benchmarks, and installation instructions are hosted in the GitHub repository:

https://github.com/citiususc/omp4py

In particular, all benchmark scripts used in the paper are provided in the `examples/` directory.

In addition, we provide two preconfigured Docker images that include all dependencies and benchmarks, allowing users to run experiments without additional setup.

2) Software Dependencies: As mentioned, when using the Docker images, all software dependencies required to run OMP4Py and every benchmark included in the artifact are already installed and preconfigured inside the container. This ensures a fully isolated and reproducible environment without requiring any additional setup from the user. For users wishing to run OMP4Py directly on a local machine (outside Docker), it is important to remember that a free-threading Python interpreter (Python 3.13+ with no-GIL support) must be used; otherwise, the code will execute sequentially regardless of the number of threads specified. The core package does not require any mandatory dependencies. However, to enable the *Compiled* and *CompiledDT* execution modes, it is necessary to have both `Cython` and `setuptools` installed in the system, since these modes rely on the generation and compilation of C extensions.

C. Installation

To install only OMP4Py, users can clone the repository and install it manually:

```
git clone https://github.com/citiususc/omp4py.git
cd omp4py
pip install .
```

However, the simplest approach is to use the `Poetry` package manager from the `examples/` folder that automatically install locally OMP4Py, the benchmarks and all dependencies required:

```
pip install poetry
poetry env use <python_binary>
poetry install --no-root
eval $(poetry env activate)
```

In addition, we provide two preconfigured Docker images to simplify reproducibility:

- **OMP4Py:** Python 3.14b1 free-threading, available as `cesarpomar/omp4py`
- **PyOMP:** Python 3.10 (maximum supported version), available as `cesarpomar/pyomp`

D. Experiment Workflow

All tests can be executed locally using the following command syntax:

```
python3 examples/main.py <mode> <test> <threads> [args...]
```

Where the parameters are defined as follows:

- **mode:** execution mode: 0 – Pure (OMP4Py), 1 – Hybrid (OMP4Py), 2 – Compiled (OMP4Py), 3 – Compiled with data types (OMP4Py), and -1 – PyOMP.
- **test:** Benchmark to run. Available tests include numerical applications—`fft` (Fast Fourier Transform), `jacobi` (Jacobi method), `lud` (LU decomposition), `maze` (pathfinding, BFS), `md` (molecular dynamics simulation), `pi` (Riemann integration), and `qsort` (Quicksort)—and non-numerical applications—`wordcount` (word count) and `graphc` (clustering coefficient).

- **threads:** number of threads to use for the execution
- **args:** optional arguments to modify the problem size; by default, the problem size corresponds to the values used in the article for reproducibility.

Using the provided Docker images, the corresponding commands are:

- **OMP4Py:** `docker run --rm cesarpomar/omp4py python3 /omp4py/examples/main.py [0-3] <test> <threads>`
- **PyOMP:** `docker run --rm cesarpomar/pyomp python3 /omp4py/examples/main.py -1 <test> <threads>`

E. Evaluation and Expected Results

The performance evaluation consists of running each benchmark while varying the number of threads within the same execution mode. For example, to illustrate how to obtain execution times and compare performance between OMP4Py in *Compiled with data types* (*CompiledDT*) mode (mode 3) and PyOMP, we provide the following example using the `pi` benchmark. The commands below execute the benchmark with 2 and 4 threads for both frameworks using the Docker images:

```
docker run --rm cesarpomar/omp4py python3 /omp4py/examples
     /main.py 3 pi 2
docker run --rm cesarpomar/omp4py python3 /omp4py/examples
     /main.py 3 pi 4

docker run --rm cesarpomar/pyomp python3 /omp4py/examples/
     main.py -1 pi 2
docker run --rm cesarpomar/pyomp python3 /omp4py/examples/
     main.py -1 pi 4
```

To simplify the reproduction of the full experimental evaluation presented in the paper, both Docker images include an **automated script** that runs all thread configurations for a given benchmark (1, 2, 4, 8, 16, and 32 threads) across all execution modes supported by each framework. The script can be invoked as follows:

```
docker run --rm cesarpomar/omp4py omp4py-test <test>
docker run --rm cesarpomar/pyomp  omp4py-test <test>
```

To reproduce the experiments in Figure 5, run both scripts for the following benchmarks: `fft`, `jacobi`, `lud`, `md`, `pi`, `qsort`, and `maze`. Note that with the default sizes, all the experiments may take many hours to complete. It is also possible to pass additional arguments to `<test>` when calling the automated scripts, for example to set the problem sizes, seeds, and other parameters.

The experiments in Figure 6 correspond to running only the OMP4Py test script with `graphc` and `wordcount`. In the case of the `wordcount` benchmark, both the single benchmark execution and the batch evaluation require passing a text file as an argument. If no input file is provided, the benchmark will automatically generate a synthetic dataset from a fixed seed. To reproduce the results reported in Figure 6, users should supply the Spanish Wikipedia dump as the input dataset (21 GB). The dataset can be downloaded as follows:

```
wget https://dumps.wikimedia.org/eswiki/latest/eswiki-
     latest-pages-articles.xml.bz2
bzip2 -d eswiki-latest-pages-articles.xml.bz2
```

Once downloaded and decompressed, the file can be passed as the final argument to the benchmark:

```
docker run --rm -v $(pwd)/eswiki-latest-pages-articles.xml
     :/eswiki cesarpomar/omp4py python3 /omp4py/examples/
     main.py <mode> wordcount <threads> \"/eswiki\"

docker run --rm -v $(pwd)/eswiki-latest-pages-articles.xml
     :/eswiki cesarpomar/omp4py omp4py-test wordcount \"/
     eswiki\"
```

Regarding the hybrid OpenMP/MPI applications, Docker provides an isolated runtime environment, which prevents programs running inside a container from using software installed on the host system. This includes the host's MPI implementation, making it impossible to execute MPI-based applications such as the `jacobi` benchmark from within the provided Docker images. Since `mpi4py` depends on an existing local MPI installation, both MPI (for example, MPICH or OpenMPI) and `mpi4py` must be installed directly on the host machine. Once MPI is installed, the recommended approach is to use a Poetry-managed environment and install the Python bindings by running `poetry install mpi4py`. Finally, the command required to run the `jacobi` benchmark with MPI in a local (non-Docker) installation is:

```
mpirun -n <procs> python3 <omp4py-folder>/examples/main.py
      <mode> jacobi <threads>
```

For numerical benchmarks, the expected performance ordering is consistent and correlated with the OMP4Py execution mode. The *CompiledDT* mode is expected to achieve the highest performance, since the generated code is closest to C and benefits from type-specialized optimizations. In contrast, the *Pure* mode yields the lowest performance, as execution remains entirely in Python and is subject to higher interpretation overhead. In the case of PyOMP, its performance is expected to be similar to the *CompiledDT* mode of OMP4Py for the benchmarks that the library supports. For non-numerical benchmarks, performance across the OMP4Py modes is generally similar, with a small advantage for the Cython-based modes, since native-code optimization is not feasible and PyOMP cannot run these benchmarks. Results should follow the trends reported in the paper's figures. Execution time decreases as thread count increases, but scalability depends on the execution mode. The *Pure* and *Hybrid* modes are limited by the scalability constraints of the Python 3.14b1 interpreter, so their speedup is not expected to improve beyond 8 threads. In contrast, the *Compiled* and *CompiledDT* modes rely on Cython-generated native code and scale with higher thread counts, with *CompiledDT* achieving the best performance.

Note that when using problem sizes smaller than those employed in the evaluation, it is important to ensure that the chosen sizes are large enough so that Python's overhead does not dominate the total execution time. Although the computation runs in native code in the *Compiled* and *CompiledDT* modes, extremely small workloads may complete so quickly that the measured time primarily reflects Python. To obtain meaningful and comparable results, users should therefore select input sizes where the native compute phase clearly outweighs Python's inherent overhead.

REFERENCES

[1] TIOBE Software. (2025) TIOBE index for April 2025. [Online]. Available: https://www.tiobe.com/tiobe-index

[2] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a LLVM-based Python JIT compiler," in *Proc. of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015, pp. 1–6, doi: 10.1145/2833157.2833162.

[3] Python Enhancement Proposals. (2023) PEP 703 – Making the Global Interpreter Lock Optional in CPython. [Online]. Available: https://peps.python.org/pep-0703

[4] T. A. Anderson and T. Mattson, "Multithreaded parallel Python through OpenMP support in Numba," in *SciPy*, 2021, pp. 140–147.

[5] T. G. Mattson, T. A. Anderson, and G. Georgakoudis, "PyOMP: Multithreaded parallel programming in Python," *Computing in Science & Engineering*, vol. 23, no. 6, pp. 77–80, 2021, doi: 10.1109/MCSE.2021.3128806.

[6] D. Padua, *Encyclopedia of Parallel Computing*. Springer Science & Business Media, 2011, doi: 10.1007/978-0-387-09766-4.

[7] S. Behnel, R. W. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011, doi: 10.1109/MCSE.2010.118.

[8] L. Dalcin and Y.-L. L. Fang, "mpi4py: Status update after 12 years of development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021, doi: 10.1109/MCSE.2021.3083216.

[9] T. G. Mattson, Y. H. He, and A. E. Koniges, *The OpenMP Common Core: Making OpenMP Simple Again*, ser. Scientific and Engineering Computation. MIT Press, 2019.

[10] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proc. of the 29th ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2024, pp. 929–947, doi: 10.1145/3620665.3640366.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a system for large-scale machine learning," in *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2016, p. 265–283.

[12] C. Piñeiro and J. C. Pichel, "A unified framework to improve the interoperability between HPC and Big Data languages and programming models," *Future Generation Computer Systems*, vol. 134, pp. 123–139, 2022, doi: 10.1016/j.future.2022.04.002.

[13] L. Hastings. (2016) Gilectomy. [Online]. Available: https://github.com/larryhastings/gilectomy

[14] S. Gross. (2022) Python Multithreading without GIL. [Online]. Available: https://github.com/colesbury/nogil

[15] S. Guelton, P. Brunet, M. Amini, A. Merlini, X. Corbillon, and A. Raynaud, "Pythran: Enabling static optimization of scientific python programs," *Computational Science & Discovery*, vol. 8, no. 1, p. 014001, 2015, doi: 10.1088/1749-4680/8/1/014001.

[16] C. Piñeiro and J. C. Pichel, "OMP4Py: A pure Python implementation of openMP," *Future Generation Computer Systems*, vol. 175, p. 108035, 2026, doi: 10.1016/j.future.2025.108035.

[17] A. A. Hagberg, D. A. Schult, P. Swart, and J. Hagberg, "Exploring Network Structure, Dynamics, and Function using NetworkX," *Proc. of the Python in Science Conference*, 2008, doi: 10.25080/TCWV9851.

[18] Message Passing Interface Forum. (2023) MPI: A message-passing interface standard version 4.1. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[19] C. Piñeiro and J. C. Pichel. (2025) Material for OMP4Py. [Online]. Available: https://doi.org/10.5281/zenodo.17987713