Compiling Python Modules to Native Parallel Modules Using Pythran and OpenMP Annotations

Serge Guelton*, †
Pierrick Brunet†
Mehdi Amini‡
* École Normale Supérieure, Département d'Informatique, Paris, France
† Télécom Bretagne, Plouzané, France
‡ SILKAN Inc., Los-Altos, USA

***Abstract*—High Performance Computing users traditionally rely on low-level, compiled language such as C or FORTRAN to perform compute-intensive tasks. As a consequence, it is a common situation to have High Performance Computing application written in a high-level language such as Python, calling native routines for compute-intensive tasks. To improve development speed and reduce maintenance costs, using a higher-level language like Python seems attractive. While it is usually associated with low performance, several solutions such as Cython, Numba, Parakeet or Pythran offer to automatically or semi-automatically turn Python functions into native ones.**

One of the key points required to match the performance of native applications is the ability to write parallel applications. This paper studies the addition of OpenMP directives, a popular model to describe parallelism in C/C++/FORTRAN applications, to Pythran, an automatic compiler from a subset of Python to C++. It shows that scientific Python applications annotated with OpenMP directives can be turned by an automatic compiler into native applications that run within the same order of magnitude than manually-written ones.

I. Introduction

Since its birth in December 1989, the Python language [21] has proved to be useful in various domains, ranging from system administration to web services, thanks to its dynamicity, expressiveness, rich ecosystem and "batteries included" standard library. It is also widely used in scientific computing [14], thanks to the `numpy` module and the SciPy [10] project which provide a set of scientific and numeric tools, ranging from linear algebra to ordinary differential equation solvers or generic algorithms.

It is now heading toward High Performance Computing (HPC), either as a glue language used to bind several native libraries together, or to support whole applications in Pure Python. However, the prohibitive overhead of the language implied by its interpreted and highly dynamic nature prevents its usage for the most performance-critical code sections, where native code is generally used. The SciPy modules partly overcome this issue through the usage of low-level routines written in C or Fortran and encapsulated in Python *native* modules. Moreover, its core data structure, the multi-dimensional array [20], has been designed so that the underlying data are available to both native modules and the Python interpreter without conversion cost. Yet when a new function not already part of an existing module is required, one has to write it in C or FORTRAN.

The hybrid approach, where an application contains a mix of interpreted and native code, is getting widespread in the Python landscape. It is also the recommended approach to write applications that make use of fine grain parallelism, as Python parallelism support is only suitable for coarse-grain parallelism. To allow the user to write native functions without having to write the boilerplate glue code that turns Python object into native structures back and forth, the SciPy package provides the `weave` module. It makes it possible to bundle C code snippets into Python code. They are then compiled and loaded at runtime. Other project like PyCuda or PyOpenCL offers the same convenience to target hardware accelerators.

This paper presents the addition of OpenMP, a popular standard to turn sequential C, C++ and FORTRAN application into parallel applications, to the Pythran [9] compiler, a compiler that turns Python modules into C++ meta-programs. Section II gives an overview of existing approaches to compile Python functions or modules and shows a critical lack of parallelism support. It also emphasises on the need for a backward-compatible approach. Section III presents the Pythran compiler and the underlying runtime library. Section IV studies the validity of using OpenMP directives within the Python language in order to add fine-grain parallelism support to Python in the context of Pythran. Finally, the Pythran compiler is benchmarked on several scientific kernels and compared to Cython, Numba and Parakeet in Section V.

II. Parallel Computations in Python

In the many-cores era, it is mandatory to exhibit parallelism to balance the performance limitations of scripting languages, as described in [7] in the context of the MATLAB language, or in [13] in the context of the R language. In the context of Python, most approaches have focused on fork-based parallelism.

A. Python and Parallelism

Parallel computations are supported by the Python standard library through the `multiprocessing` module. It spawns several interpreters that communicate through Inter Process Communication (IPC), using Python built-in object serialization. This approach is only viable for compute-intensive independent tasks, since the communication and synchronization overheads are much greater than a light-weight threaded approach.

The standard `threading` module also makes it possible to start several light-weight threads within the same interpreter, but this lower-level approach is not applicable to HPC, because of a specificity of CPython, the *Global Interpreter Lock* [23].[1] This lock ensures that only one thread is active at a time in the interpreter. While it enables the possibility of cooperative threads, say for a GUI, it does not take advantage of multiple cores. However, there are two notable exceptions: the GIL is released on I/O, and the GIL does not prevent the use of threads inside native modules, where the user has full control.

To illustrate the limitations of these two approaches, we used the CPU-bound Buffon's needle algorithm to estimate the value of π. A sequential version parallelized using the two approaches is illustrated in Listing 1. When comparing their respective execution time using 4 cores, it appears that the version that uses the `threading` module runs $\times 1.46$ the execution time of the sequential version. GIL contention actually *increases* the execution time. The version that uses `multiprocessing` provides a speedup of 3 over the sequential version while a speedup of 4 would be expected for this kind of application.

Another widely used solution is the module `IPython.parallel`, which supports many different styles of parallelism including: single program, multiple data (SPMD) parallelism, multiple program, multiple data (MPMD) parallelism, message passing using MPI, task farming, and data parallel.

There have also been several approaches to replace the GIL by Transactional Memories [17], [19] but none of them made its way to the mainstream interpreters. As a consequence, Python developers need to write multi-threaded native modules in order to fully benefit from multiple cores. This leads to a kind of computations referred as *hybrid computations*.

B. Hybrid Computations

In the context of interpreted languages, [12] defines a computation as *hybrid* when part of the code is interpreted, and part of it is executed natively.

It is now common for scripting language to have C bindings. To take advantage of compiled code, and to overcome the GIL limitations, Python developers have to write parallel C/C++ functions and the associated boilerplate based on the Python C API [22]. Tools have been developed to relieve the user from this cumbersome task, notably SWIG [2] that relies on an interface specification supplied by the programmer to generate the glue, or `boost::python` [1] that relies on C++ templates and type overloading facilities to guide translation.

An opposite approach consists in using the host language —herein Python— to describe both parts of the system, i.e. the hybrid and the native. An automated tool performs the translation to native code of a specific part of the application, generally the compute-intensive one where parallelism has been expressed in some ways. Therefore, developers not familiar with lower level languages or not eager to invest the additional development time can still benefit from a fair performance boost. This approach is the subject of many studies that can be classified according to their compatibility with the host language.

[1]Other Python interpreters, such as IronPython or Jython, do not have a GIL.

```
def buffon(darts, _ = 0):
  hits = 0
  for i in xrange (0, darts):
    x, y = random(), random()
    dist = sqrt(pow(x, 2) + pow(y, 2))
    if dist <= 1.0:
      hits += 1.0
  # hits / throws = 1/4 Pi
  pi = 4 * (hits / darts)
  return pi

from threading import Thread
from multiprocessing import Queue
def threaded_buffon(d):
  def work(darts, queue):
    queue.put(buffon(darts))
  n = 4
  q = Queue()
  threads = [
    Thread(target=work, args=(d//n, q)),
    Thread(target=work, args=(d//n, q)),
    Thread(target=work, args=(d//n, q)),
    Thread(target=work, args=(d//n, q)),
  ]
  map(Thread.start, threads)
  map(Thread.join, threads)
  return sum(q.get() for _ in threads)/n

import functools
from multiprocessing import Pool
def multi_buffon(darts):
  n = 4
  p = Pool(n)
  return sum(p.map(functools.partial(buffon,darts
```

Listing 1. Implementation of sequential and parallel version of the Buffon algorithm in Python.

1) Backward-Incompatible Approaches: A constraining (from the performance point of view) aspect of the Python language is its type system. It implies that each method call is resolved dynamically, even a simple add operation. It comes at no surprise that many approaches restrain the Python language to add a static typing overlay. Also, only two types of integers (64-bits integers and multi-precision integers) and one type of floating point type (double-precision floats) are available in Python, so using a type with the appropriate size may lead to significant performance boost.

Cython [3] is an hybrid Python/C dialect. It extends the Python syntax with typing information, calls to native functions from third party libraries, and a limited set of parallelism constructs, such as the possibility to define parallel loops, but no task parallelism. When possible, it unboxes Python variables to improve performance. Without type annotations, the performance improvement is not terrific, but given enough type annotation, Cython can generates code that runs as fast as its C equivalent, while maintaining a syntax close to Python.

PLW [12] and `Scipy.weave` both propose another approach that directly mixes Python with C, using raw strings to hold the C code. PLW also supports parallel directives that are limited to parallel for loops. PyCUDA and PyOpenCL [11] also target accelerators by mixing Python with kernels embedded as raw strings containing accelerator code.

The main drawback of these approaches is that they require to modify in a backward-incompatible way the original code. They require to learn a new dialect, and the long-term preservation of this investment is not ensured. Moreover, some of these approaches have no fallback if the code were to be deployed in an environment where the parallelizing tool/module is not available. For instance, once a code has been ported to PyCUDA and transformed into CUDA code embedded in Python strings, there is no way back and the developer has to maintain two versions of the algorithm.

2) Backward-Compatible Approaches: Most backward-compatible approaches also require to modify the input program. They do not extend the Python language, but restrict it. As a consequence, they remain compatible with the original language and do not suffer from the drawbacks of the previous approaches. They also benefit from existing tools associated to the language.

Copperhead [6] is a functional, data parallel language embedded in Python. It uses n-uplets, NumPy arrays and lists as its core data structure and prohibits usage of many control-flow operators such as loops, enforcing the use of the `map`, `filter` or `reduce` intrinsics to exhibit parallelism. But it can be efficiently compiled to either CUDA or C++ with calls to the Thrust[2] library. Python decorators are used to identify hot-spots that are JIT-compiled to native code.

The numba [3] compiler uses additional type information to generate sequential LLVM bytecode. Parakeet [18] follows an approach similar to numba, that is Python to LLVM bytecode translation, but limits its scope to the numpy package, only supporting a small subset of the Python language. Additionally, it supports implicit parallelism using an implicit mapping between numpy functions and a set of parallel primitives including `maps` `scans` and `reduces`. These two approaches use Just-In-Time(JIT) compilation and do not suffer from backward-incompatibility issues. When meeting an unsupported construct, numba falls back to Python C-API calls, while parakeet raises an exception.

Tools such as PyPy [5], a Python interpreter with a tracing JIT, or Shed Skin [8], a Python to C++ compiler are also viable ways to enhance Python performance. However Shed Skin does not provide support for fine-grained parallelism beyond what the standard library proposes. PyPy is heading toward STM for parallelism support.

To be completely backward compatible, it has to be possible to run the input code in an environment that is not aware of the existence of the parallelizing solution. This principle is not respected by parallel libraries, but code annotations partially satisfy it. This is where parallel annotations shine: the original code remains mostly compatible with a compiler that is not aware of the annotations, while taking the annotation into account turns the sequential code into parallel code.

[2] *cf.* `http://thrust.github.com/`

[3] *cf.* `https://github.com/numba/numba`

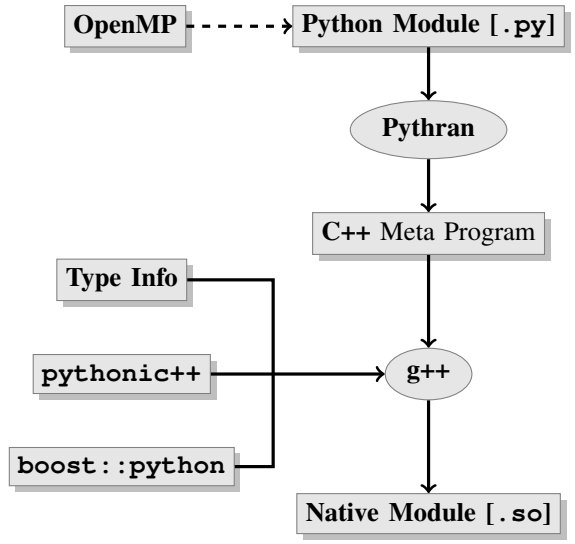


Fig. 1. Pythran compiler workflow.

This article proposes to combine OpenMP [15] parallel annotations with the Python subset supported by the Pythran compiler to make it possible to write fine-grain parallel applications in Python, while being fully backward compatible with both the Python language and the sequential algorithm. It implies that:

1) any Pythran code can be run (sequentially), with no module dependency or code change, by any Python interpreter;
2) parallelism is explicit and incrementally added to the original code through directives.

III. Static Translation of Python Programs

Pythran [9] is a subset of Python designed for scientific computing. It is implicitly statically typed and supports most Python constructs except those that involve introspection (e.g. `getattr`) or runtime compilation (e.g. `eval`). A few standard modules are supported in addition to the core language (e.g. `math`, `random`). The `numpy` module is currently only partially supported. User classes are not supported. The associated compiler turns Pythran code, possibly annotated with OpenMP and a few function type annotations, into C++ code.

Figure 1 summarizes its processing: type inference helps removing most dynamic behavior, and is combined with a high-level runtime library —`Pythonic++`— to allow a one-to-one mapping between Python and C++ constructs. Pythonic++ is a template library, i.e. it consists exclusively of a set of headers. Therefore the generated C++ file is easily distributable and does not depend on Pythran. One could even preprocess this C++ file to avoid the need of distributing along the Pythonic++ headers.

The purpose of this section is not to dig into the internals of the compiler, but rather to focus on the impact of the parallelism layer, especially on the runtime library.

A. Runtime Library Support

Pythran runtime library is based on the C++11 standard library which, unlike its C++03 counterpart, is thread-safe. It implies that no data dependencies are added to the original code.

A critical point of the design of the `pythonic++` runtime library is memory management, the very same aspect that led to the use of a GIL in CPython. Memory management is implemented in Shed Skin through the general-purpose Boehm garbage collector [4], and Cython forbids usage of Python-managed objects inside parallel regions, thus making memory management explicit for these parts.

Pythran handles the problem by refusing recursive types, which makes it possible to solely rely on reference counting for memory management. It can be implemented through the thread-safe shared pointer mechanism provided by the C++11 standard library.

Using shared references simplifies memory management, but the counterpart is an extra atomic operation for each copy. As it limits parallelism, reducing the number of copies becomes an important goal. The move semantic introduced in C++11 avoids a few copies when working on temporary objects, but argument passing still implies copies. To avoid a reference increment, one can pass parameters by reference. Using an inter-procedural memory effect analysis not presented in the paper, Pythran determines for each argument whether it has to be passed using reference or const reference to prevent this overhead.

B. Directive Oblivious Translation

During Python to C++ translation, Pythran adopts a blind strategy: it does not understand the semantics of the annotations. Instead, it just splits each annotation into a context-sensitive part —the variable names— and a context-insensitive part —the clauses—, and attaches them to the proper construct in the Abstract Syntax tree (AST).

The Pythran compiler ensures a bijective translation between Python constructs and generated C++ constructs so that the OpenMP directive is regenerated on the proper construct. The same approach is used at the expression level, to be compatible with the `if` clause.

At the AST level, it means that the Python AST is first reduced to a tree where all nodes not available in C++ are transformed into a compatible representation. For instance, list comprehension expressions are turned into function calls or tuple unpacking is turned in multiple assignments enclosed in a dummy `if`. During this transformation process, an expression is always transformed into a single expression and a statement is always transformed into a single statement. Then this AST is converted into a C++ AST meant to be pretty-printed.

C. Transfer Costs

When passing containers from Python to C back and forth, a copy of the whole container is made to turn the type agnostic, non-contiguous Python data into dense typed ones. This extra copy implies an extra cost that is not negligible: Passing two lists of float from Python to C++ requires as much as half the time to compute the dot product of the same lists directly in Python. Following Amdahl's law, this copy overhead greatly hinders the benefits of parallelization, and is a well known issue.

```
def pi(nsteps):
  sum, step = 0., 1. / nsteps
  for i in range(nsteps):
      x = (i - 0.5) * step
      sum += 4. / (1. + x**2)
  return step * sum
```

Listing 2. Motivating example: computing π in Python.

```
double pi(size_t nsteps) {
  double sum = 0., step = 1. / nsteps;
  #pragma omp parallel for reduction(+:sum)
  for(size_t i = 0; i < nsteps ; ++i)
  {
      double x = (i - 0.5) * step;
      sum += 4. / (1. + x * x);
  }
  return step * sum;
}
```

Listing 3. Motivating example: computing π in C with OpenMP.

The traditional solution is to use a native type that exposes a Python interface and has a constant translation cost. NumPy's `ndarray` is a typical implementation of this concept and is the keystone of the Scipy and NumPy packages. Basically, a NumPy `ndarray` is a raw C pointer that is exposed at the Python level. The PEP 3118 defined the API that allows to share efficiently the embedded data between Python and the native world without involving any copy. As a tool for scientific computing, Pythran supports such a structure, implemented as a class wrapping a raw pointer.

IV. OpenMP Semantical Adaptations

OpenMP is a standard API for parallel programming for Fortran, C and C++. It consists of a set of parallelizing directives and a few runtime library calls. If OpenMP is not activated, the directives are ignored, thus enabling incremental parallelization of the original source code while keeping the original code structure. The languages targeted by OpenMP are statically compiled. This section studies the semantical adaptation required to use the same API for a scripting language such as Python. Listing 2 introduces a motivating example through the computation of π, featuring a parallel reduction. The C equivalent with OpenMP directives is given in Listing 3 for reference.

A. Directives

OpenMP directives are held by C/C++ `#pragma`, or by Fortran comments. Most of them apply to structured blocks in C/C++ and delimited by comments in Fortran. A few directives (e.g. `threadprivate`) are not attached to a specific instruction.

```
def pi(nsteps):
  sum, step = 0., 1. / nsteps
#omp parallel for reduction(+:sum) private(x)
  for i in range(nsteps):
      x = (i - 0.5) * step
      sum += 4. / (1. + x**2)
  return step * sum
```

Listing 4. Motivating example: computing π in Python with OpenMP.

```
def pi(nsteps):
  sum, step = 0., 1. / nsteps
  #omp parallel for reduction(+:sum)
  for i in range(nsteps):
    x = (i - 0.5) * step
    sum += 4. / (1. + x**2)
  return step * sum
```

Listing 5. Motivating example: computing π in Python with OpenMP and automatic scoping.

While Python has a decorator mechanism[4], it only applies to functions, methods and classes and does not allow to attach decorations to other statements. PLW [12], uses string instructions to hold such decorations. As Python does not have anonymous block,[5] one has to create a dummy `if 1:` instruction to apply an annotation to a whole block. Pythran uses comments to hold OpenMP directives, and internally stores them as string instructions. Alternatively, the syntax `if 'my annotation':` is also supported.

Many OpenMP annotations are parametrized by clauses that list variables, specifying their behavior with respect to parallel regions, e.g. `private`, `shared`, `copyin`. They can only refer to variables that have already been declared. However, there is no variable declaration in Python, and all variables assigned in a function have the function scope, called local scope and available through the `local()` built-in. As a consequence, all variables that are referenced in a function are considered when building such variable lists: there is no concept of variable local to a block. Using the `default(none)` clause is possible to ensure no variable gets forgotten when building such lists.

The `reduction(`*`operator`*`:` *`list`*`)` directive is used to characterize some data dependencies when performing a parallel reduction. The list of supported operators depend on the input and backend languages: Python has `min`/`max` operators but C/C++ does not. C/C++ have `&&` or `||` while Python does not.[6] The latest OpenMP specifications [16] describe a way to declare user-defined reduction but is not implemented in any compiler yet.

The annotated motivating example is given in Listing 4. Note that unlike in C, `x` has to be listed in a `private` clause.

B. Automatic Scoping Computation

Using function scope for all variables may lead to very long `private` clauses. This section describes an algorithm to automatically limit the scope of a variable declaration so that it can be automatically marked as private, in a similar manner to C/C++

The whole idea is to compute the minimal nesting depth of a variable usage. To do so, a *nesting depth* is first associated to each expression. It lexically corresponds to the indentation level of its instruction, to the notable exception of the iterator declaration in a for loop, that holds a nesting depth equals to 1 plus the loop declaration indentation level.

Tracking the nesting depth of all variable use, for instance using def-use chains, makes it possible to build a list of nesting depths, one per variable reference. Computing the min of this list yields the scope of the variable.

For instance, in the motivating example, the `nsteps` variable is first defined as argument, then read twice, which gives a def-use chain of {DEF, USE, USE}. The associated nesting depths is {0, 1, 1} so the variable scope is 0. `i` is defined as a loop iterator then used in the loop body, which gives a chain of {DEF, USE}, with nesting depths of {2, 2} and a scope of 2. Likewise `x` has a chain of {DEF, USE}, nesting depths of {2, 2} and a scope of 2, which means it can be automatically declared local to the loop, thus not needing to be listed as `private`.

To be able to list a variable marked as local, for instance in a `lastprivate` clause, the directive accesses are taken into account in the def-use computations. For instance in Listing 4, the chain for `x` is {DEF, DEF, USE} and the associated nested depths are {1, 2, 2} so its scope is 1 and the variable is not declared local to the loop.

Thanks to this automatic scoping computation, it is possible to write shorter OpenMP directive, as illustrated in Listing 5.

1) Parallel For and Iterators: The core directive to handle data parallelism is the `for` directive that distributes the iteration space of the associated loop among the existing threads. To be compatible with OpenMP, the loop iteration space has to be described by a random access iterator with a total order. Integers used as loop indices in Fortran and C satisfy this conditions, as well as C++ iterators with the `random_access_tag` trait. But a Python iterator only advances by a step of one until it is exhausted, in which case it raises an exception: it behaves as a forward iterator. To be compatible with OpenMP, these iterators has to be turned into random access iterators.

The extension of Python iterator to random access iterators is direct for the standard containers: `list`,[7] `set` or `dict`. Other iterators require more care.

Generators, Python objects that behave like iterators, are commonly used in Python. The simplest one, `xrange(`*`start`*`,` *`stop`*`,` *`step`*`)`, successively yields value starting from *`start`* to *`stop`* by a step of *`step`*. It is easily extended to support random access, but it generally

[4] *cf.* PEP 318, http://www.python.org/dev/peps/pep-0318/ for a more detail explanation of Python decorators.

[5] *cf.* PEP340, http://www.python.org/dev/peps/pep-0340/ for a discussion concerning anonymous block support in Python.

[6] Although similar, the `and` and `or` keywords are not boolean operators in Python.

[7] Python `lists` behave as C++ `vectors`.

```
#omp parallel for
for i, v in enumerate([2, 3, 5, 7, 11]):
    print i, ':', v
```

Listing 6. Parallel loop in Pythran with tuple unpacking.

does not make sense to use a generator as a loop iterator, as the relation between two random states of the iterator may be of an arbitrary complexity.

Generator expressions are generators whose content is built from another iterator. For instance `(x*x for x in l)` successively yields the square of each element in `l`. They behave like adaptors: they apply a particular expression on each value of the iterator. It is also the case of the `enumerate` builtin, that yields each element of the enumerated iterator associated with its index. These generators are random access iterators only if their input iterator is a random iterator itself.

Finally, if the iterator yields a tuple, it is possible to unpack it inside the `for` construct, as shown in Listing 6. In that case all the unpacked variables are considered as iterators, especially with respect to default privatization rules: in the given example, `i` and `v` are private, and the parallel iteration is valid because the input of `enumerate` is a list, which allows random access iteration.

Pythran's runtime library is aware of these three kinds of iterators and supports parallel iteration over random access iterators and iterators adapting random ones.

C. OpenMP Runtime Library

OpenMP provides a small runtime library that, for instance, makes it possible to retrieve the active thread id. All the functions are declared in the `<omp.h>` header, and have a default behavior when OpenMP is not activated.

Providing a binding to these libraries in Python as an `omp` module does not raise particular problems, as the signature of these functions only involves integers, except for the mutex manipulation. In that case an opaque type is used to represent the native type.

The `_OPENMP` macro definition is always provided when OpenMP is activated, and can be used to detect when OpenMP is not available. Python does not have a preprocessor, but it is possible to catch the import exception if the `omp` module is not found. Listing 7 showcases such a call using an example converted from the OpenMP validation suite presented in next section. The code starts a parallel section using the `parallel` directive, spawning several threads. Then each started thread increases the `nthreads` variable that has unspecified visibility, thus is `shared`. A guard protects the incrementation using the `critical` directive. Then one of the thread retrieves the number of active threads in the parallel region through the `get_num_threads` function from the `omp` module. The function should always return `True`.

1) Interaction With Pythran Runtime Library: Pythran already uses OpenMP to parallelize some Python functions. For instance the `sum` function from the `__builtin__` is implemented using the OpenMP `reduction` clause. When its first argument proves to be a *pure* function, it can also call

```
import omp
def omp_get_num_threads():
  nthreads, nthreads_lib = 0, -1
  #omp parallel
  if 1:
    #omp critical
    nthreads += 1
    #omp single
    nthreads_lib = omp.get_num_threads()
  return nthreads == nthreads_lib
```

Listing 7. Example of OpenMP API usage from Python.

```
def omp_parallel_for_if(loop_count):
  import omp
  using = sum = 0
  #omp parallel for if(using == 1)
  for i in range(loop_count + 1):
    num_threads = omp.get_num_threads()
    sum += i
  known_sum = (loop_count *
               (loop_count + 1)) / 2
  return known_sum == sum and num_threads == 1
```

Listing 8. Example of Python OpenMP validation test case.

a parallel version of the `map` function that internally spawns a parallel region. This leads to a well known situation with OpenMP, where too many threads may be spawned: nested parallel regions. OpenMP 4 provides several ways to handle this situation:

1) Disabling nested parallelism, through the `OMP_NESTED` environment variable or the `omp_set_nested` routine.
2) Setting the maximum number of active levels, through the `OMP_MAX_ACTIVE_LEVELS` environment variable or the `omp_set_max_active_levels` routine.
3) Conditionally activating regions or conditionally setting the number of threads in a parallel region using the `omp_in_parallel`

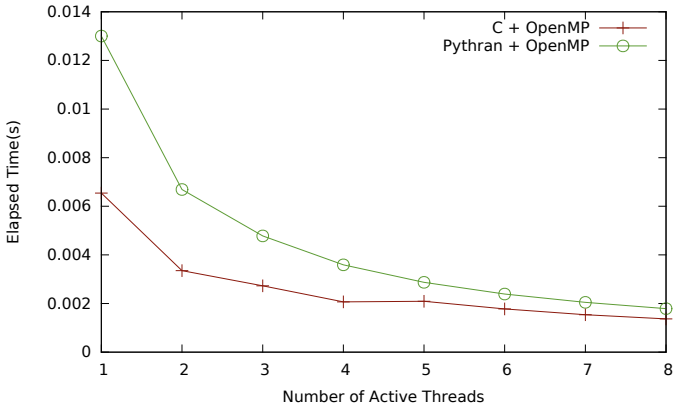
The decision is left to the user, depending on its application and OpenMP implementation.

D. Validation

A validation suite for OpenMP is proposed in [24] for C and Fortran. We ported it to Python, and also extended it to validate the corner cases specific to Python described in this section. A typical test case is given in Listing 8.

We have used the Pythran tool described in the following section to turn each Python test function into a C++ function with the same directives and runtime library calls. Apart from the `threadprivate` directive and the `collapse(`*n*`)` clause, all tests were successful. `threadprivate` directives were held by global variables not supported in Pythran yet ; and the C++ code generated by Pythran does not preserve the perfect loop nesting required by the `collapse` clause.

Fig. 2. Comparison of the scaling of π computation function for C and Python with OpenMP, depending on the number of active threads.



V. Validation

In addition to the experimental validation done using the OpenMP validation suite presented in Section IV, the performance of parallel module generated by Pythran from Python module with explicit parallelization using OpenMP directives has been studied on several situations: first on the motivating example, then on a synthetic geomatic application. Python and Cython versions are then compared and finally the effect of OpenMP directives on Pythran generated code are illustrated in the context of the Python Benchmarks suite.

The target machine has 8 AMD Opteron 6176 SE cores and 8 GB of RAM. It is running a Linux kernel 3.10-2-amd64, x86_64 GNU/Linux. The C++ compiler used is `g++` version 4.8.1 and all applications are compiled using the `-Ofast` flag. Pythran generated programs are linked with Boost.Python version 1.54. The Pythran version is extracted from the `sc2013` branch of the git repository https://github.com/serge-sans-paille/pythran. Python programs are run using the CPython implementation, version 2.7.5. Cython programs are generated using version 0.19.1. Numba and Parakeet are installed from the `master` branch of their respective repositories.

A. Motivating Example

To validate the approach proposed in this paper, let first consider the motivating example. Figure 2 shows how the C and Python version of the program scale when changing the number of core used. As a reference, the original Python code runs 41 times slower than the equivalent C code. As the program is not memory bound and exhibit trivial parallelism, the C + OpenMP version scales almost linearly. The Pythran + OpenMP version follows the same pattern.

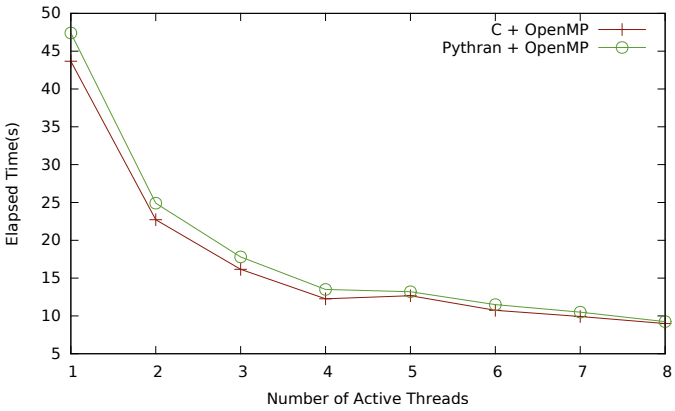
B. Hyantes: a Geomatic Application

To get closer to a realistic example, we have considered the code of a small geomatics application, Hyantes. Starting from the C code, we successively parallelized it using OpenMP, turned it into Python with a Pythran compatible kernel and parallelized the Python version using the approach described in this paper. We also measure the number of Source Lines Of Code (SLOC) of the two versions. Table I summarizes the result of this experimentation. It not only exhibit the use of a high level language for prototyping, but also shows that it is possible to turn this prototype into reasonably efficient code through Pythran. It is also possible to prototype the parallel version while remaining at the Python level.

TABLE I. Comparison of several versions of the Hyantes prototype.

Language	Source Lines Of Code (SLOC)
C	102
Python	30
Pythran	30
Pythran+OMP	31

Fig. 3. Comparison of the scaling of the hyantes application for C and Python with OpenMP, depending on the number of active threads.



The Hyantes programs scales relatively well. Figure 3 shows the relative speedup of the application when increasing the number of OpenMP threads using up to 8 cores.

C. Comparison with Cython

We then compare Pythran with Cython. These approaches share some similarities: they both generate code written in a lower level language, with OpenMP directives. However, the inputs differ as Cython requires more typing information than Pythran to generate efficient code, and Cython is not backward-compatible with Python (outside of *Pure mode*). They translate explicit fine-grained parallelism through code annotations or specific Python constructs, respectively.

Pythran exposes the full OpenMP interface to the user, thus enabling both data and task parallelism, as described in Section IV. It is not the case in Cython:

- Only loops can be made parallel, using a new `prange` generator.
- Reductions and variable privacy are inferred, but it chokes on reduction on private variables.
- The user is responsible from releasing the GIL inside parallel regions.
- It is impossible to use a function imported from a Python module in a parallel region, but it is still possible to use native C functions.

Listings 9 and 10 illustrate the difference between the two and illustrates the intrusive behavior of Cython.

```
from libc.math cimport sqrt
from cython.parallel import parallel, prange
def sum_sqrt(double r):
    cdef int i
    for i in prange(10000000, nogil=True):
        r += sqrt(i)
    return r
```

Listing 9. Cython implementation of a parallel reduction.

```
#pythran export sum_sqrt(float)
import math
def sum_sqrt(r):
    #omp parallel for reduction(+:r)
    for i in xrange(10000000):
        r += math.sqrt(i)
    return r
```

Listing 10. Pythran implementation of a parallel reduction.

The performance of the two approaches is shown in Figure 4. The benchmarked codes are typical mathematical, image-processing or linear algebra kernels. All these kernels have been written in Cython and Python —compatible with Pythran— and annotated through the mechanism of each language to exhibit parallelism, then compiled into native code. Their execution time when called from the Python interpreter is measured through the `timeit` module. All results are normalized against Cython sequential execution time. They show that while handling code at a higher level than Cython, Pythran achieves comparable results.

The source codes used for these benchmarks are available on the Pythran repository.[8]

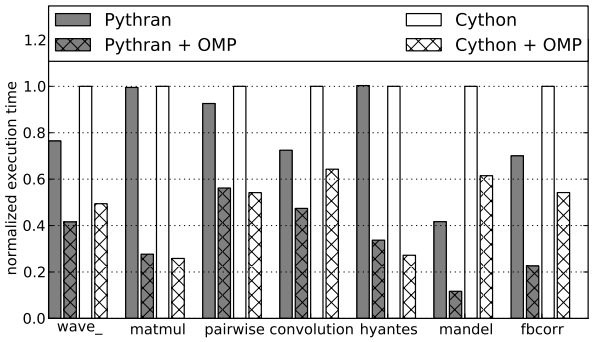


Fig. 4. Comparison of Cython and Pythran generated code performance.

D. Python Benchmarks Test Bed

The Python Benchmark test bed (https://github.com/numfocus/python-benchmarks) is an initiative to gather relevant numerical python benchmarks and compare several implementations and compilers. The original benchmarks are slightly adapted to match each compiler restriction (e.g. lack of support for a particular construct), or completely rewritten in the target's paradigm (e.g. using PyOpenCL or Theano). We picked up the targets that did not imply a full rewrite of their input: Cython, Numba, Pythran and Parakeet, when available and supported. Additionally, the original Python code is run.

The best execution time out of five run is shown for each code version. If the code does not compile with a particular compiler, the column is left blank. Two code versions are shown for Pythran: one compiler without the OpenMP flag set and one with the OpenMP flag set. The code itself is not modified, only the compilation flag and the directives. A logarithmic scale is used due to the generally poor performance of the pure Python version.

Figure 5 shows the execution time of the computation of the julia fractal. It's a highly parallel, compute-intensive code. Pythran and Cython's version have almost the same execution time, but activating OpenMP yields an extra $\times 3.95$ speedup when compared to Pythran alone.

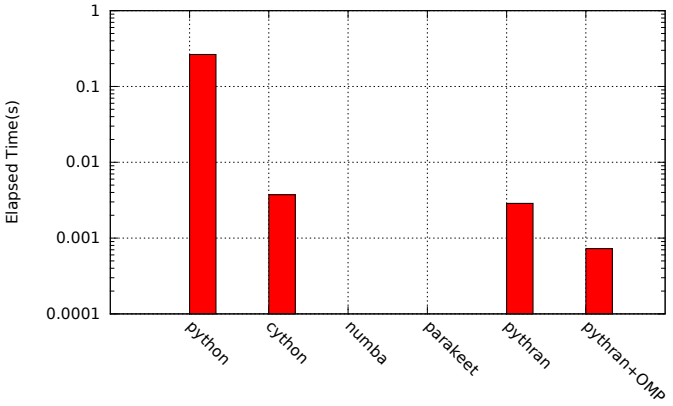


Fig. 5. Comparison of Various python Compilers on the Julia benchmark

Figure 6 shows the execution time of the computation of a distance matrix between two random vectors, using a parallel nested loop. On that example, Pythran slightly outperforms Cython, parakeet and Numba. Taking into account the OpenMP annotation that flags the outermost loop as parallel yields an $\times 4.27$ speedup to Pythran.

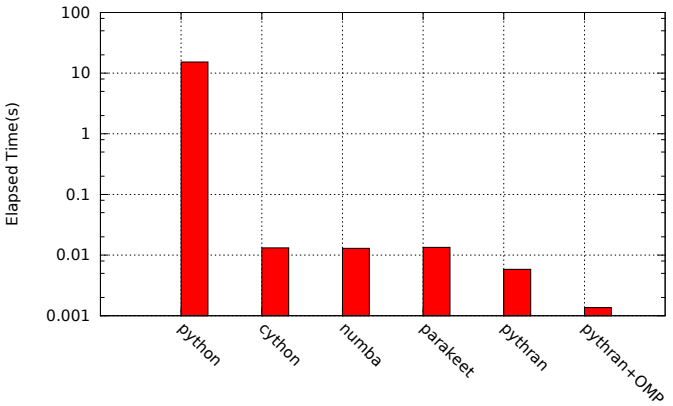


Fig. 6. Comparison of Various python Compilers on the pairwise benchmark

Figure 7 shows the execution time of the computation of the third derivative of the Rosenbrock function for a random array input, using Numpy's vector operations. There is no explicit parallelism in this function but the vector operation

[8] *cf.* the `sc2013` branch of the git repository.

is implicitly parallel. On this example, Cython is on par with Pythran and slightly faster than Parakeet, but turning OpenMP on makes the Pythran version run $\times 1.78$ faster than sequential Pythran.

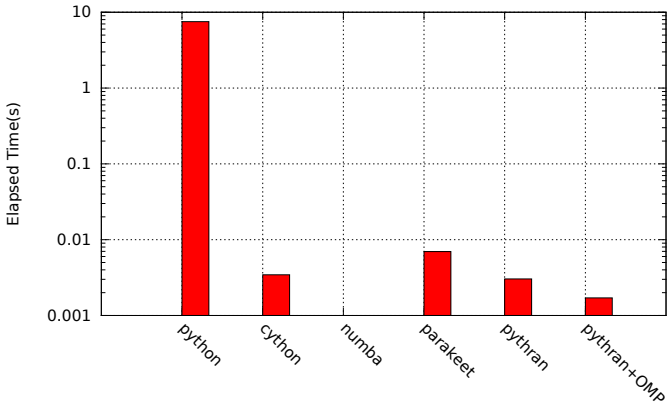


Fig. 7. Comparison of Various python Compilers on the rosen_der benchmark

Figure 8 shows the execution time of the computation of one step of the GrowCut algorithm used in image processing. On this example, Parakeet performs better than Cython and Pythran and way better than Numba, but parallelizing the outermost loop with OpenMP gives a $\times 2.65$ speedup to Pythran that helps it outperform Parakeet.

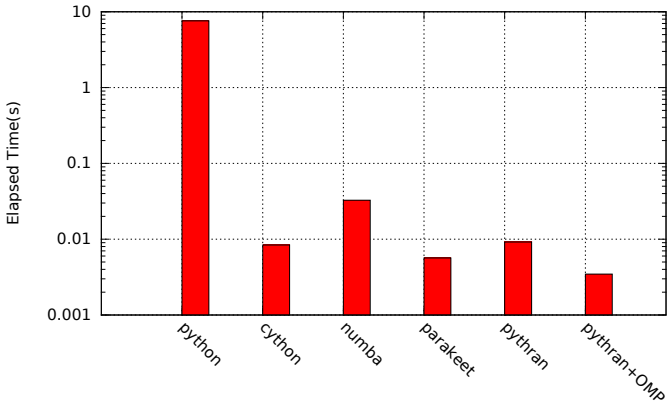


Fig. 8. Comparison of Various python Compilers on the Growcut benchmark

It appears that all Python compilers manage to yield significant speedups over the Python interpreter, each compiler managing some cases better than others. All the benchmarks exhibit some level of parallelism that are easy to capture using OpenMP without changing the original algorithm and while maintaining backward compatibility with the Python interpreter. Some even have a degree of implicit parallelism that makes parallelization fully transparent to the user. Taking advantage of this parallelism appears to be the key to reach an additional performance level.

VI. Conclusion and Future Work

This paper studies the addition of OpenMP annotations to Pythran. It shows that providing minor semantic adaptations, these annotations enable Python code to benefit from multi-cores while retaining backward compatibility and without worrying about the Global Interpreter Lock.

To achieve this goal, we designed Pythran, a translator from a subset of the Python language to C++. Pythran turns regular Python modules annotated with OpenMP directives and a few type annotations into native parallel module. The input module remains compatible with the standard interpreter and the underlying runtime library is compatible with parallel constructs. While Pythran accepts a limited input, we extend it progressively and with a great care on its stability. As such, even if its development is community-based, it is more than a research prototype and should be usable on real world code.

The approach is compared with Cython, an extension of the Python language used to generate native module with an hybrid Python-C syntax that also provides means to exhibit fine grained parallelism. It shows that retaining Python compatibility does not prevent the achievement of comparable performances. Comparisons to Numba and Parakeet shows the benefit of using parallelism on the benchmarks from the Python Benchmark project.

Future work will focus on the extension of the approach to OpenMP 4 and the `target` clause that should allow to target accelerator like Intel MIC from Python. The `simd` clause also brings interesting vectorization capabilities that could be exposed at the python level. There are also several implicit vectorization opportunities in Python, especially in the list comprehension construction, that need to be explored.

Acknowledgment

This work has received fundings from SILKAN and the French ANR through the CARP project. The authors thank Adrien Merlini and Alan Raynaud for their work on the runtime library, Albert Cohen, Béatrice Creusillet, Fabien Dagnat, Fran cois Irigoin and Ronan Keryell for their valuable reviews.

References

[1] D. Abrahams and R. W. Grosse-Kunstleve, "Building hybrid systems with Boost. Python," *C/C++ Users Journal*, vol. 21, no. 7, Jul. 2003.

[2] D. M. Beazley, "Automated scientific software scripting with SWIG," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 599–609, Jul. 2003.

[3] S. Behnel, R. 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.

[4] H.-J. Boehm, A. J. Demers, and S. Shenker, "Mostly parallel garbage collection," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '91. New York, NY, USA: ACM, 1991, pp. 157–164.

[5] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: PyPy's tracing JIT compiler," in *Proceedings of the 4th ICOOOLPS workshop*. New York, NY, USA: ACM, 2009, pp. 18–25.

[6] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2011, pp. 47–56.

[7] R. Choy, A. Edelman, and C. M. Of, "Parallel MATLAB: Doing it right," in *Proceedings of the IEEE*, 2005, pp. 331–341.

[8] M. Dufour, "Shed skin: An optimizing python-to-c++ compiler," Master's thesis, Delft University of Technology, 2006.

[9] S. Guelton, P. Brunet, A. Raynaud, A. Merlini, and M. Amini, "Pythran: Enabling static optimization of scientific python programs," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2013.

[10] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," http://www.scipy.org/, 2001.

[11] A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[12] P. Luszczek and J. Dongarra, "High performance development for high end computing with Python language wrapper (PLW)," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 360–369, Aug. 2007.

[13] X. Ma, J. Li, and N. F. Samatova, "Automatic parallelization of scripting languages: Toward transparent desktop parallel computing," in *IPDPS*. IEEE, 2007, pp. 1–6.

[14] T. E. Oliphant, "Python for scientific computing," *Computing in Science and Engineering*, vol. 9, no. 3, pp. 10–20, May 2007.

[15] "OpenMP application program interface," http://www.openmp.org/mp-documents/OpenMP3.1.pdf, Jul. 2011.

[16] "OpenMP application program interface," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, Jul. 2013.

[17] N. Riley and C. Zilles, "Hardware tansactional memory support for lightweight dynamic language evolution," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 998–1008.

[18] A. Rubinsteyn, E. Hielscher, N. Weinman, and D. Shasha, "Parakeet: a just-in-time parallel accelerator for python," in *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, ser. HotPar'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 14–14.

[19] F. Tabba, "Adding concurrency in python using a commercial processor's hardware transactional memory support," *SIGARCH Computer Architecture News*, vol. 38, no. 5, pp. 12–19, Apr. 2010.

[20] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: a structure for efficient numerical computation," *CoRR*, vol. abs/1102.1523, 2011.

[21] G. van Rossum, "A tour of the Python language," in *TOOLS (23)*, 1997, p. 370.

[22] G. van Rossum and F. L. J. Drake, Eds., *Python/C API Reference Manual*. Python Software Foundation, Sep. 2012.

[23] ——, *Thread State and the Global Interpreter Lock*. Python Software Foundation, Sep. 2012.

[24] C. Wang, S. Chandrasekaran, and B. M. Chapman, "An OpenMP 3.1 validation testsuite," in *8th International Workshop on OpenMP (IWOMP)*, ser. Lecture Notes in Computer Science, vol. 7312. Rome, Italy: Springer, Jun. 2012, pp. 237–249.