Department of Electronics and Computing
Higher Technical School of Engineering

MASTER'S THESIS
INTERUNIVERSITY MASTER IN
HIGH PERFORMANCE COMPUTING

Native Implementation of OpenMP For Python

Student: Dorian Ouakli
Supervisor/s: Juan Carlos Pichel Campos
Cesar Alfredo Piñeiro Pomar

Santiago de Compostela, July 29, 2024.

Dedication: to Sha.

Acknowledgements

Thank you to my tutors, Juan Carlos Pichel Campos and Cesar Alfredo Piñeiro Pomar. Thank you also to Juan Angel Lorenzo Del Castillo and Karima El Ganaoui, the heads of my specialization at CY Tech.

Abstract

It was previously impossible to obtain a speedup using multithreading in Python due to the "Global Interpreter Lock" in the main Python Implementation, CPython. However, in October 2024, Python 3.13 will be released, including PEP 703 making the Global Interpreter Lock optional in CPython.

Up to now, the only way to parallelize Python programs was to use multiple processes and Inter Process Communication. This context made the implementation of the OpenMP API pointless. With the Global Interpreter Lock removal in Python 3.13, providing simple multithreading APIs becomes relevant and useful.

The concrete goals of this thesis are to design and create a Python library implementing part of the OpenMP API, and asses the performance of this library. This has been achieved by setting up Python without GIL, researching existing multithreading Python APIs, creating a library design similar to OpenMP, implementing the most important features, and running example code using our library.

Keywords:

Global Interpreter Lock

Python

CPython

PEP 703

Multithreading

OpenMP

Parallelization

Concurrency

Abstract Syntax Tree

Variable scope

Shared Memory

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In HPC architectures[1], the parallelization of computations is crucial to make the best use of available resources. Indeed, a sequential program can only use a single CPU core. However, in a modern computer and especially in a node of a supercomputer, a program has multiple CPU cores at its disposal.

OpenMP is one of the various tools that allow developers to parallelize their C, C++, and Fortran programs. An interesting feature of OpenMP is that users can achieve very good efficiency[2] by adding just one line of code at the appropriate place. OpenMP is an approach to parallelization that uses Threads[3] for their lightness and shared memory.

Python, unlike C, C++, or Fortran, is a high-level programming language[4]. It is therefore quick to learn and is notably used for prototyping and data analysis. However, there is no *native*[5] implementation of OpenMP in Python.

Multithreading has limited interest in Python because of the GIL[6], which prevents the parallel execution of Threads. [1] This makes a native implementation of OpenMP in Python pointless since OpenMP is primarily aimed at parallelizing computations.

For several years, Sam Gross, a researcher at Meta, has been working on removing the GIL from CPython, the main implementation of the Python language. In October 2024, PEP[7] 703, which aims to make the GIL optional, will be included in Python 3.13. [2] This new context makes an implementation of OpenMP in Python interesting and useful.

This Master's Thesis aims to provide a first implementation of the most important OpenMP features and to create a foundation for a library that could implement a significant part of the OpenMP API.

[1]HPC: High Performance Computing
[2]Efficiency: Ratio between resources used and available resources
[3]Thread: Execution thread
[4]High level of abstraction: Close to natural language and problem-solving formalisms
[5]Native: Using the same language
[6]GIL: Global Interpreter Lock
[7]PEP: Python Enhancement Proposal

In the first chapter, ”Context,” we will discuss the limitations of multithreading in Python due to the Global Interpreter Lock (GIL). The chapter discusses the current state of parallelization in Python. It explores existing libraries and techniques such as `threading`, `multiprocessing`, and `concurrent.futures`. The chapter also discusses the ongoing work to remove the GIL, led by Sam Gross, and the implications of this change for Python developers. This chapter provides the necessary background and motivation for the development of our new parallelization library.

The second chapter, ”Library,” discusses the design and implementation of our Python package, which has the goal of bringing OpenMP-like capabilities to Python. This chapter is organized into three sections. The first section, ”Objectives,” sets the goals of the project. It explains the need for a user-friendly syntax that closely mimics OpenMP in C while using Python's standard libraries. The second section, ”Design,” describes the initial implementation, including the use of decorators and generators to manage parallel execution and task distribution. It uses code examples to demonstrate these concepts. The third section, ”On-the-Fly Modifications,” discusses the usability challenges of the initial implementation. It introduces techniques for dynamically transforming user code.

The third chapter, ”Results,” addresses the usage and performance of our Python package by introducing two examples of user code. This chapter is organized in three sections. The first section, ”Tools,” describes the set-up, specifications, and versions used in the tests of user code. The second section, ”Simple Sum,” discusses the challenges introduced with a lightweight parallelized loop body, as well as the importance of fine-tuning the library's parameters through a practical experiment using CESGA's FinisTerrae III supercomputer showing the efficiency of the parallelization. In the third section, ”Counting Primes,” we take a similar approach of a practical experiment to show the potential this Python package could have when used incrementally on more forgiving structures.

Chapter 2

Context

In this chapter we explore different methods of achieving parallelization in Python, looking into both the challenges and current solutions available within the language. We will discuss the limitations of multithreading in Python and move on to alternative approaches and ongoing developments aimed at improving Python's parallel computing capabilities. We will see the current state and future prospects of parallelization in Python.

The first section, ”Multithreading in Python,” discusses the existing support for multithreading provided by the `threading` library. It covers the primitives introduced by the library, such as locks, semaphores, events, and barriers, which aid in thread synchronization. Despite the comprehensive support, the section discusses the limitations imposed by the Global Interpreter Lock (GIL). The GIL restricts the concurrent execution of threads, making multithreading less effective in Python. This section also discusses non-native implementations of OpenMP for Python, such as PyOMP and Pythran. We note their constraints and limited usability.

The second section, ”Parallelization in Python,” discusses the `multiprocessing` library. The `multiprocessing` library creates isolated processes instead of threads, allowing simultaneous execution of Python bytecode. This approach avoids the GIL limitation by using process-based parallelism. This section explains the use of shared Queues and Pipes for communication between processes and compares the synchronization methods available in `multiprocessing` to those in `threading`. Then, it introduces the `concurrent.futures` library, which offers a higher-level interface for asynchronous execution using either threads or processes. It also introduces the `asyncio` library, which uses coroutines and an event loop for managing asynchronous tasks.

The final section, ”Ongoing Work,” discusses the efforts led by Sam Gross to remove the GIL from CPython. It details the creation of the `nogil` fork and the development of PEP 703. PEP 703 outlines the strategy for making the GIL optional in Python. This section discusses the potential performance benefits of a no-GIL Python and the necessity for the Python community to adapt their programs to the new paradigm. It discusses the importance of providing a native implementation of OpenMP in Python, because this would increase both the participation in the transition to a no-GIL environment and the parallel computing capabilities in Python.

2.1 Multithreading In Python

Python already has support for thread management through its standard library `threading`. [3] This support is quite comprehensive. The library introduces primitives for retrieving information about active threads and abstractions for using thread-local memory. Of course, the library introduces a `Thread` class, representing a thread that can be started and whose execution can be awaited. Finally, this library offers a set of common synchronization methods:

- Locks: Ensures that only one thread accesses a given resource at a time.
- Semaphores: Protects a resource with a limited number of concurrent accesses.
- Events: Allows one thread to block or unblock other threads.
- Barriers: Allows a team of threads to wait until all members reach the same point.

Despite comprehensive multithreading support in Python, it is not widely used in practice. Indeed, most Python operations are not *thread-safe*[1], including a simple assignment of an object to a variable. [4] To address this issue and for historical reasons, CPython uses a global lock, which must be acquired to execute Python bytecode[2]: the Global Interpreter Lock (GIL). The GIL has significant consequences on the performance of a multithreaded Python program since **only one thread** can execute Python code at a time.

When a thread in a Python program calls an external library, this library can potentially release the GIL during its operations. The famous scientific computing library Numpy, for example, releases the GIL during its operations. Another thread can execute Python code during this time. Input/output operations such as reading and writing to files or network communications also release the GIL. Implementing graphical interfaces and client/server architectures can benefit from using threads in Python.

There are non-native implementations of OpenMP for Python. PyOMP, an extension of Numba, allows writing OpenMP directives directly in a Python function compiled with the `@njit` decorator. [5] Numba is a library that compiles functions written in Python into LLVM IR[3] and then into machine code to take advantage of the performance and optimizations of low-level languages in Python. Numba can compile Python programs, but when a program uses a function from a library implemented in another language, this function must be reimplemented in Numba specifically to be compiled. A large part of the functions of major scientific libraries has already been reimplemented in Numba, but not all, which limits the usability of PyOMP.

[1]Thread-safe: Can be executed in the same memory space by multiple threads simultaneously.
[2]Python bytecode: Compiled form of Python code, using an instruction set specific to Python.
[3]LLVM IR: Intermediate Representation of the LLVM compiler allowing automatic transformations

Pythran is another way to use OpenMP syntax in Python. Pythran, unlike Numba which compiles on the fly, is an ahead-of-time compiler. Pythran is even more limited than Numba in its support of Python. It aims to prototype mathematical functions. Important features such as class definitions are simply not supported. [6]

2.2 Parallelization In Python

Given the significant limitations of multithreading in Python, the common approach to parallelizing a program is to use the built-in `multiprocessing` library. This approach involves creating processes rather than threads. Each process is isolated and uses its own interpreter. Therefore, processes can execute Python bytecode simultaneously. Unlike `threading`, variables are private by default with `multiprocessing`. Sharing is done explicitly through communication tools such as shared Queues or Pipes that allow bidirectional exchange. Besides this difference, synchronization between processes is done similarly to synchronization between threads. The `multiprocessing` library provides Locks, Semaphores, Events, Barriers, and other synchronization methods, which are implemented using IPC[4]. [7]

Python also has the standard library `concurrent.futures`, which can use `threading` or `multiprocessing`. This library introduces the concept of an executor, allowing independent tasks to be launched. Unlike `threading` and `multiprocessing`, the difference is that `concurrent.futures` does not provide synchronization methods or shared memory management. The functions called must operate independently of other threads. [8]

A final approach to parallelization in Python uses coroutines[5] to switch tasks at opportune moments. The standard library `asyncio` organizes this architecture with an event loop. Some operations take time without performing calculations. The `await` keyword is introduced, allowing control to be given to another coroutine while waiting for the result of a call. [9]

[4]IPC: Inter Process Communication

[5]Coroutine: Function whose execution can be "paused" and then resumed later.

2.3 Ongoing Work

Sam Gross, an engineer at Meta and co-author of the machine-learning library PyTorch, is familiar with the challenges of parallelization in Python within the context of PyTorch development. An RNN model playing the board game Hanabi against itself requires parallelization through threads, using shared memory to update coefficients in real time. To achieve performance gains from this parallelization, the team had to implement most of the model in C++. If the GIL did not exist, they could have written a much larger portion of the model in Python. Therefore, he devised a strategy to remove the GIL in CPython. [10]

After creating the `nogil` fork of Python 3.9, he ported his work to the latest version of Python at the time, Python 3.12. He opened PEP 703, which defines the approach to remove the GIL in Python. This removal is not without consequences. Some Python programs already use threads and implicitly rely on the GIL to protect operations. These operations must be manually protected when the GIL is removed. This is why the removal of the GIL is optional. Python must first be compiled with the `--disable-gil` option, and then when running the program, the `PYTHON_GIL` environment variable must be set to 0. The changes made cause a slowdown of about 5 to 8% for programs not using threads. These programs represent the vast majority of Python programs. [2]

In this context, a concerted effort from the Python community to convert their programs to no-GIL is necessary to truly benefit from the removal of the GIL in CPython. This project is part of that effort. It is now timely to provide a native implementation of OpenMP in Python, and such an implementation will allow developers to participate in this collective effort in a simple manner.

Chapter 3

Library

In this chapter we provide a comprehensive guide to the development of a native OpenMP-like library in Python, specifically focusing on the `parallel` and `for` constructs. We discuss the initial goals, design decisions, and the on-the-fly modifications necessary to create a practical and user-friendly library.

The first section, ”Objectives,” discusses the goal to develop a Python package that mimics the OpenMP syntax in C while adapting to Python’s needs. The proposed package, named `omp`, should ideally use only Python’s standard libraries. The section presents an example of how the library should be used. We mention the use of context managers for calling OpenMP directives to make the syntax intuitive and easy to learn.

The second section, ”Design,” discusses the initial implementation of the library using Python’s threading capabilities. It describes extending the Thread class to include ranks and teams for task distribution. The section explains the use of decorators, specifically the `run_parallel` decorator, to create and manage threads. It also introduces the concept of generators for the `for` directive, discussing how to distribute an iterator across multiple threads.

The final section, ”On-the-Fly Modifications,” discusses the limitations of the initial implementation, which requires users to wrap their code in decorated functions and generators. To improve usability, the section discusses transforming user code from the proposed syntax to the less convenient syntax automatically. It explains the use of the `inspect` and `ast` libraries for source code manipulation and conversion to ASTs, and the `compile` and `exec` functions for dynamic code execution. The section notes the challenges in managing local and shared variables and provides solutions using nonlocal variables and dummy declarations to ensure correct variable scope and access.

3.1 Objectives

We aim to create a Python package that implements the main OpenMP directives. We will specifically focus on implementing the `parallel` and `for` constructs, and reuse the same approach when implementing other directives and clauses. We wish to closely mimic the C syntax of OpenMP while adapting it to the peculiarities of Python. Ideally, this library should only use Python's standard libraries. We have chosen to name this library `omp`. The name evokes the OpenMP library and is short.

Unfortunately, the `omp` package name has been claimed on PyPI in the past by security researchers who wanted to exhibit supply chain attacks through LLM[1]s, such as ChatGPT, hallucinating python package names. Hopefully, our request to reclaim this package will be accepted and we will be able to distribute our library under this convenient name.

We envision a syntax using context managers to call OpenMP directives. The advantage of the context manager is twofold. The `with` keyword defines a block and thus allows delimiting an "action zone." The `with` keyword does not create a new scope. Variables introduced in the block still exist after it. The primitives defined by the OpenMP API will be available in the `omp` namespace via `omp.<primitive>`.

Here is an example of usage.

Example in C:

```c
int array[10];
#pragma omp parallel
{
    #pragma omp parallel for
    for (int i=0; i<10; i++) {
        array[i] = i;
    }
}
```

Example in Python:

```python
from omp import OpenMP
array = [None]*10
with OpenMP('parallel'):
    with OpenMP('for'):
        for i in range(10):
            array[i] = i
```

[1]LLM: A Large Language Model is an Artificial Intelligence technique designed to handle human natural language.

3.2 Design

Support for threads exists in Python through the `threading` library, which provides all the synchronization tools we might need. However, there is a catch: a thread necessarily starts by calling a function. Our first implementation will use a syntax that diverges from what we initially envisioned.

The library defines a Thread class. We extend this class to record a rank and a team of threads. These data are important for task distribution later on.

For the `parallel` directive, we use decorator notation. In Python, a decorator is a function that takes a function as a parameter and returns a new function. A decorator can simply note the function elsewhere, change its behavior by wrapping it, or even modify its source code. Decorators are used by prefixing a function definition with `@decorator`. [11]

We define a `run_parallel` decorator that returns a function which creates a team of threads to run the decorated function, then starts the threads.

run_parallel:

```python
def run_parallel(func):
    """
    When the new function is called, creates a team of threads
   that will each run the given function concurrently.
    Decorates the given function.
    """
    def wrapped(*args, **kwargs):
        team = Team(size=None, target=func, args=args,
   kwargs=kwargs)

        team.start()
        team.join()
    return wrapped
```

We can then use the decorator to launch a block of code on multiple threads.

Using run_parallel:

```python
@run_parallel
def main():
    print("Hello, world!")
main()
```

When the program is executed, we get a series of `Hello, world!` outputs.

Output:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

For the `for` directive, we turned to the concept of generators in Python. A generator is a function that uses the `yield` keyword. Calling a generator does not immediately execute the code; instead, it returns an **iterable** that can be used in a for loop. When iterating over this iterable, the function executes and then *pauses* at the `yield` keyword, yielding an element of the iterable. When the next element is needed, the function resumes execution until it encounters `yield` again or reaches the end of the function. [11]

Here is a minimalist example of `range` implemented with a generator.

my_range:

```python
def my_range(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Between each iteration, the value of `i` is not lost. We can use this concept to distribute an existing iterator across various threads of a team.

Generator for:

```python
def generator(it):
    """
    When called within a thread of a team, yields the
   iterations for the current thread.

    Overall, when all the threads of the team call this
   generator, all the elements of the iterator are yielded.
    """
    thread: Thread = threading.current_thread()
    for i, el in enumerate(it):
        if i % thread.team.size == thread.rank:
            yield el
```

We can then use our generator in a parallel construction.

Using generator:

```python
import threading
@run_parallel
def main():
     for i in generator(range(10)):
         print(f"Thread {threading.current_thread().rank} runs {i}.")
main()
```

When the program is executed, the iterations are distributed across the different threads.

Output:

```
Thread 0 runs 0.
Thread 0 runs 4.
Thread 0 runs 8.
Thread 1 runs 1.
Thread 1 runs 5.
Thread 1 runs 9.
Thread 2 runs 2.
Thread 2 runs 6.
Thread 3 runs 3.
Thread 3 runs 7.
```

Finally, we can implement the example given in the chapter 3: Objectives.

Implementation of the example:

```python
array = [None]*10
@run_parallel
def main():
     for i in generator(range(10)):
         array[i] = i

main()
print(array)
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3.3 On-The-Fly Modifications

This initial implementation works but remains rudimentary and inconvenient for the end user, who must wrap their code in decorated functions and their iterators in generators. Our solution is to transform the user code from the proposed syntax to this less convenient syntax.

The standard `inspect` library allows retrieving information about the current program execution. Notably, the `inspect.getsource` function can retrieve the source code of a function as a string. [12]

Manipulating the source code directly can be complicated and risky, as it would require understanding all the peculiarities of Python syntax and anticipating special cases. Instead, we transform the source code into an AST[2], a tree-like representation of the source code that is simple to manipulate algorithmically. The standard `ast` library allows transforming source code into an AST, manipulating it, and eventually converting it back into source code with the `ast.parse` and `ast.unparse` functions. [13]

Finally, we can compile Python code on the fly using the `compile` and `exec` functions. These functions can directly use the modified AST. We can use them to create the new function, combining the previous steps within a decorator. [14]

A major difficulty with this approach is managing local and shared variables. Let's take an example.

Example variable:

```python
import omp
from omp import OpenMP

@omp.enable
def main():
    i = 0
    with OpenMP("parallel"):
        i += 1
    print(i)

main()
```

The variable `i` is local to the `main` function. The `with` keyword does not create a new scope, so it should always be accessible in the parallel construction. However, our transformation wraps the block in a function.

[2]AST: Abstract Syntax Tree

Transformed variable example:

```python
import omp
from omp import OpenMP

def main():
    i = 0
    @omp.directives.parallel_construct.run_parallel
    def _inner_func():
        i += 1
    _inner_func()
    print(i)

main()
```

Output:

```
File "<stdin>", line 5, in _inner_func
UnboundLocalError: cannot access local variable 'i' where it is
    not associated with a value
```

We try to modify the variable `i` defined in the parent function. To update the variable `i`, we first considered using the `globals()` and `locals()` functions, which are accessible for both reading and **writing**. [14] However, we realized that it is not possible to define a new local variable in a function this way. Indeed, the list of local variables is defined at the `compilation` of the function. We can access the list of local variables of a function `f` through `f.__code__.co_varnames`. [15] We can use the `nonlocal` keyword to make the variable `i` free, i.e., non-local. This keyword is also resolved at compilation, meaning the variable must be detected as local by the compiler. [16] In some cases, the variable is introduced before the function, and we have no issues. But in certain situations, we may have variables introduced within the auxiliary function.

Late variable example:

```python
import omp
from omp import OpenMP

@omp.enable
def main():
    with OpenMP("parallel"):
        i = 1
    print(i)

main()
```

Transformed late variable example:

```
import omp
from omp import OpenMP

def main():
    @omp.directives.parallel_construct.run_parallel
    def _inner_func():
        nonlocal i
        i = 1
    _inner_func()
    print(i)

main()
```

Output:

```
File "<stdin>", line 4
SyntaxError: no binding for nonlocal 'i' found
```

To solve this problem, we use a dummy declaration in a branch that will never be executed. The compiler interprets the potential declaration as a use, and the variable is defined as local.

Newly transformed late variable example:

```
import omp
from omp import OpenMP

def main():
    if False:
        i, = [None]*1
    @omp.directives.parallel_construct.run_parallel
    def _inner_func():
        nonlocal i
        i = 1
    _inner_func()
    print(i)
```

```

main()
```

Output:

```
1
```

In order to allow for these transformation, we introduce an internal `Directive` type. A `Directive` child class is the implementation of the transformation of an OpenMP directive, in the user code, to the less friendly internal syntax. In practice, a `Directive` will work with AST directly, the transformations from source code to AST will be handled elsewhere. A `Directive` must implement a `parse` method, that takes an AST `With` object and returns a modified `With` object.

In our ”late variable example”, the `parallel` directive’s `parse` method would receive the AST representation of the following snippet.

Parse method input:

```
with OpenMP("parallel"):
    i = 1
```

Of this `With` object, we mostly are interested in the `body`. The `body` of a `With` object is a list of AST nodes composing the indented code block.

For the purposes of the `parallel` directive, we need to wrap the user code in a decorated function call. We can reduce the amount of AST nodes we have to construct ourselves by using a template.

Parallel’s directive template:

```
with _omp_internal.core.openmp.OpenMP():
    if False:
        pass # Replaced by shared variables declarations
    @_omp_internal.directives.parallel_construct.run_parallel
    def _omp_internal_inner_func():
        pass # Replaced by user code
    _omp_internal_inner_func()
```

The relevant AST nodes of the template are replaced with the user code, or our dummy declarations. This avoid the creation of an AST tree manually which can be bothersome.

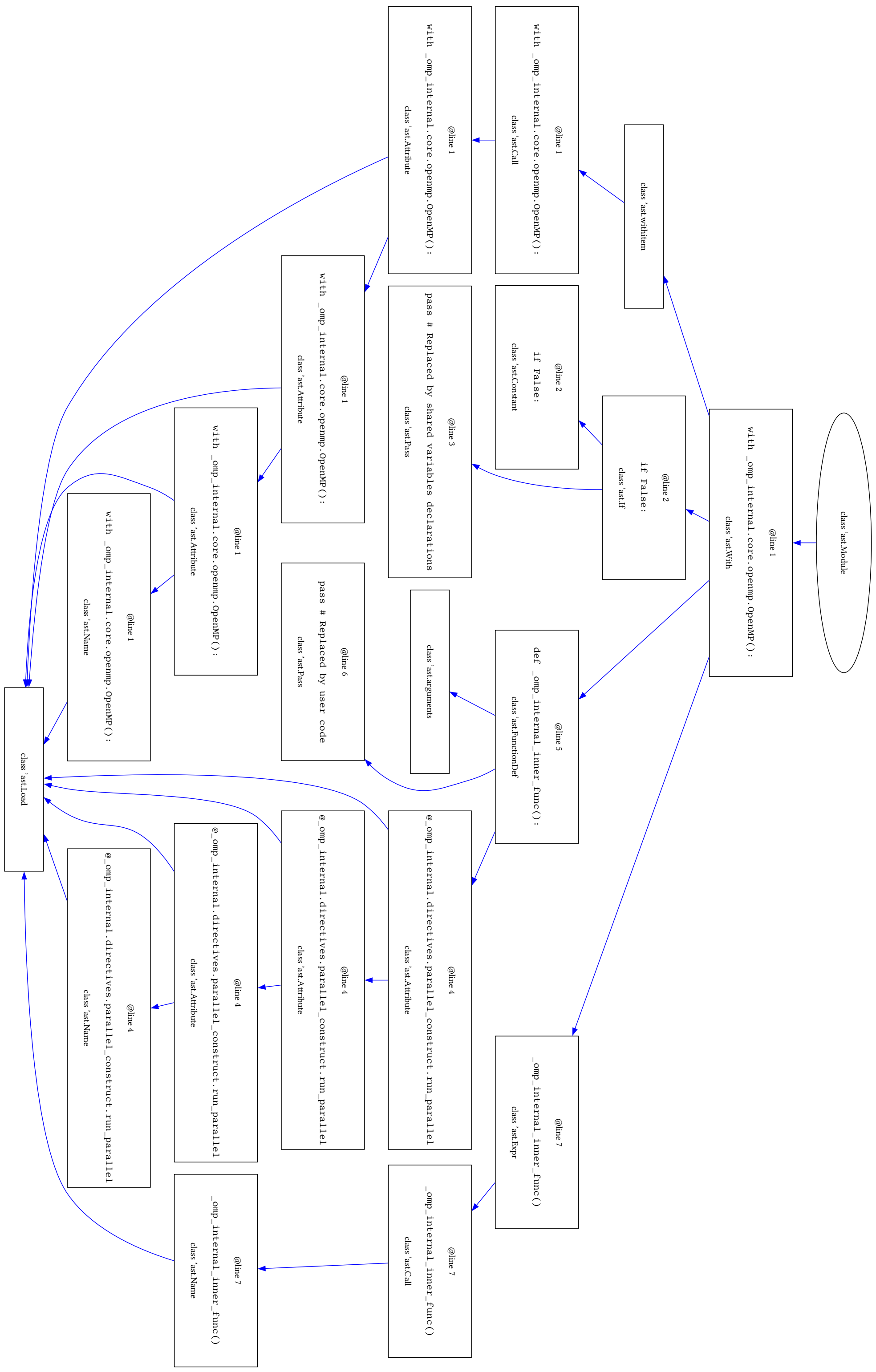

class 'ast.Module'
@line 1 with _omp_internal.core.openmp.OpenMP(): class 'ast.With'
class 'ast.withitem'
@line 1 with _omp_internal.core.openmp.OpenMP(): class 'ast.Call'
@line 1 with _omp_internal.core.openmp.OpenMP(): class 'ast.Attribute'
@line 1 with _omp_internal.core.openmp.OpenMP(): class 'ast.Attribute'
@line 1 with _omp_internal.core.openmp.OpenMP(): class 'ast.Attribute'
@line 1 with _omp_internal.core.openmp.OpenMP(): class 'ast.Name'
class 'ast.Load'
@line 2 if False: class 'ast.If'
@line 2 if False: class 'ast.Constant'
@line 3 pass # Replaced by shared variables declarations class 'ast.Pass'
@line 5 def _omp_internal_inner_func(): class 'ast.FunctionDef'
class 'ast.arguments'
@line 6 pass # Replaced by user code class 'ast.Pass'
@line 4 @_omp_internal.directives.parallel_construct.run_parallel class 'ast.Attribute'
@line 4 @_omp_internal.directives.parallel_construct.run_parallel class 'ast.Attribute'
@line 4 @_omp_internal.directives.parallel_construct.run_parallel class 'ast.Attribute'
@line 4 @_omp_internal.directives.parallel_construct.run_parallel class 'ast.Name'
@line 7 _omp_internal_inner_func() class 'ast.Expr'
@line 7 _omp_internal_inner_func() class 'ast.Call'
@line 7 _omp_internal_inner_func class 'ast.Name'

Figure 3.1: Graph of the AST representation of the parallel directive's template

Parallel's directive template AST extract:

```
Module(
body=[
  With(
    items=[
      withitem(
        context_expr=Call(
          func=Attribute(
            value=Attribute(
              value=Attribute(
                value=Name(id='_omp_internal', ctx=Load()),
                attr='core',
                ctx=Load()),
              attr='openmp',
              ctx=Load()),
            attr='OpenMP',
            ctx=Load()),
          args=[],
          keywords=[]))],
    body=[
      If(
        test=Constant(value=False),
        body=[Pass()], # Insert our dummy declarations here.
        orelse=[]),
        ... # More nodes here, removed for readability.
        ]
  )
],
type_ignores=[]
)
```

Notice the nested `Attribute` nodes, and their specific parameters. This AST extract is only part of the complete AST representation of the given source code, see the full graph in Figure 3.1. This representation is much more difficult to construct manually than source code. By using a template, we can reuse our knowledge of the Python language to get a valid AST representation.

Inserting the user code in the body of an AST node can be done easily. The body of an AST Node is a list of AST Nodes. We iterate through the nodes until we find our target node, here of type `ast.Pass`

User code insertion:

```python
def replace(target: list[ast.AST], content: list[ast.AST],
    match: ast.AST = ast.Pass) -> list[ast.AST]:
    """
    Return a new list where the elements of content are where
    there is a Pass in target.
    """
    res = []
    for el in target:
        if isinstance(el, match):
            res.extend(content)
        else:
            res.append(el)
    return res
```

The insertion of the dummy declaration uses the same function, but we still need to generate the AST of the declaration itself. In order to simplify the generation, we decided to go for a tuple assignment. In order to know how to generate a tuple assignment, we wrote one and inspected its AST representation.

Tuple assignment:

```python
a, b = [None]*2
```

Tuple assigment AST:

```
body=[
  Assign(
    targets=[
      Tuple(
        elts=[
          Name(id='a', ctx=Store()),
          Name(id='b', ctx=Store())],
        ctx=Store())],
    value=BinOp(
      left=List(
        elts=[
          Constant(value=None)],
        ctx=Load()),
      op=Mult(),
      right=Constant(value=2)))],
type_ignores=[])
```

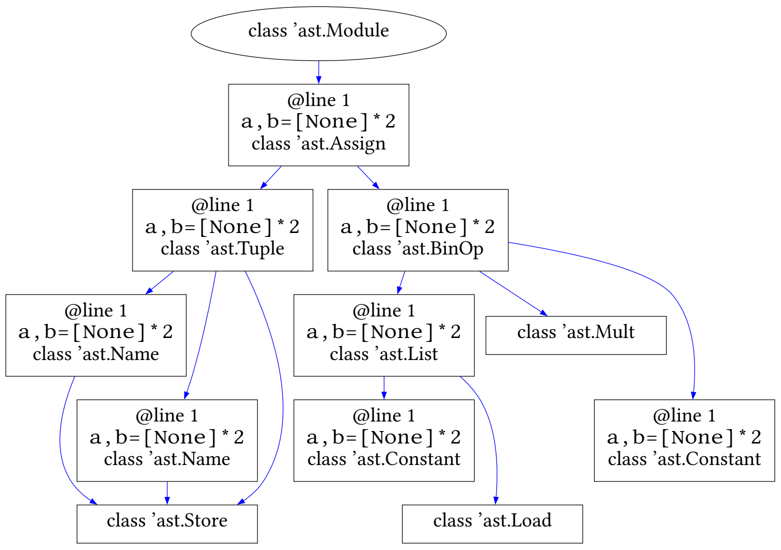


Figure 3.2: Graph of the AST representation of a tuple assignment

As we can see in Figure 3.2, the graph for this assignment is much simpler than for a complete template. Here, we can see that the only parts to modify are the `elts` attribute of the `Tuple` node as well as the `value` attribute of the `Constant` node. The `Name` nodes we add have a single changing parameter, which will be the name of the **shared** variable we want a dummy declaration of. The `Constant`'s value will simply be the number of shared variables we need a dummy declaration of. Something that makes the definition of that AST easier, is that when there are no shared variables, this assignment does not do anything, but remains valid Python code.

Empty tuple assignment:

```
()=[None]*0
```

We defined a helper function to generate the appropriate AST given a list of shared variable names.

Helper function:

```
def assign_shared(shared: list[str], value: ast.AST = None) ->
    ast.Assign:
```

```python
    """
    Return a tuple assignment to the shared variables listed.
    """

    return [ast.Assign(
        targets=[ast.Tuple(elts=[ast.Name(id=name, ctx=ast.Store()) for name in shared],
                           ctx=ast.Store())],
        value=ast.BinOp(
            left=ast.List(elts=[ast.Constant(value=value)], ctx=ast.Load()),
            op=ast.Mult(),
            right=ast.Constant(value=len(shared))
            )
        )]
```

Our `OpenMP` context manager we have been using everywhere to call OpenMP directives is actually a dummy context manager. If the code is not *enabled*, i.e. decorated with `@omp.enable`, then the OpenMP context manager has no effect. It is only during the preprocessing that `omp.enable` performs that the `OpenMP` call is interpreted. When `omp.enable` stumbles upon a `with` statement, it checks if the provided context manager is `omp.OpenMP` and then calls its `_parse_With` method with the AST `With` node that contains the `with` statement and the associated block. The `OpenMP` class *has* to be a context manager in order to be able to be used in a `with` statement. The fact it does nothing by default allows the user to quickly switch from OpenMP enabled to OpenMP disabled code while leaving the directives there. For the OpenMP class to keep a link between a directive's name and its implementation, we use a class dictionary. In Python, a class attribute can be seen as a Singleton. The dictionary in the instances of our class will be shared. When implementing a directive, we register the implementation in the `OpenMP.directives` dictionary.

Registering the Parallel construct:

```python
OpenMP.directives.update({'parallel': ParallelConstruct()})
```

Once the directives are registered, the `_parse_With` method can simply call the appropriate directive's parser.

OpenMP class:

```python
class OpenMP:

    """
    This class represents a call to an OpenMP directive.

    When used in omp-enabled user code, this class calls an OpenMP directive.

    When this class is used as a context manager, the with statement is replaced
    by the implementation of the construct.
    """

    # All the directives supported by the library. This variable is global.
    # To register a new directive, update omp.openmp.OpenMP.directives.
    directives: dict[str, Directive] = {}

    def __init__(self, directive: str = ''):
        self.directive = directive

    def __enter__(self):
        pass

    def __exit__(self, exc_type, exc_value, traceback):
        return False

    def _parse_With(self, node: ast.With) -> ast.With:
        """
        This method is meant for internal use.

        Replace this OpenMP context manager by its implementation.
        """

        directive: str = self.directive.split()[0]

        if directive in OpenMP.directives:
            return OpenMP.directives[directive].parse(node)

        return node
```

Some of the functions we showed earlier in this report take AST nodes as input. In order to find the correct AST nodes in the original source tree, we need to perform a graph search. The built-in `ast` library conveniently provides a `NodeTransformer` class that performs a traversal of an Abstract Syntax Tree, while creating a new tree. To use the `NodeTransformer` class, it is necessary to create a child class that implements custom *visitors*. A visitor is a method that handles a given *kind* of AST Node. For instance, in our case, we can write a `visit_With` visitor to handle AST With nodes in the tree. A visitor method of a `NodeTransformer` returns an AST Node. This new AST Node will be inserted in the new tree in place of the node that was visited. The `NodeTransformer` class also provides a `generic_visit` method, that leaves a node unchanged, and calls the appropriate visitors for the child nodes. In practice, when defining a new visitor, we always call the generic visitor on the newly created Node to ensure that all the subtree is visited.

With that knowledge, we defined an `OpenMPTransformer` class, that parses OpenMP directives in the tree and gives a new, OpenMP enabled, tree. This `OpenMPTransformer` has a single custom visiter, handling `With` AST nodes. Before making any changes, it performs many checks to ensure that the Node is indeed an OpenMP directive. We make sure the with statement has only one context manager. We also make sure the context manager is referenced as a `Call` to a `Name`. Once this is done, we **compile** and **evaluate** the given name in the context in which it was initially introduced. The resulting object **should** be our `OpenMP` class. We check that it is indeed exactly this class. Since we made sure that the `Name` was referencing the correct class, we can **compile** and **evaluate** the complete `Call` instanciating the `OpenMP` class. The result of this evaluation gives us an **instanciated** `OpenMP` object. We can call its `_parse_With` method with the `With` Node that we just visited, and return the result of this parsing.

OpenMPTransformer:

```
class OpenMPTransformer(ast.NodeTransformer):
    """
    Recursively find the OpenMP constructs and replace them
   with their implementations.
    """

    def __init__(self, locs=None, globs=None, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.locs = locs
        self.globs = globs

    def visit_With(self, node: ast.With) -> ast.With:
```

```python
        # We need to make sure that this is an OpenMP construct.

        # The with statement should use only one context manager.
        if len(node.items) != 1:
            return self.generic_visit(node)

        # Bulletproofing.
        if not isinstance(node.items[0], ast.withitem):
            return self.generic_visit(node)

        # We are now sure we have a withitem.
        item: ast.withitem = node.items[0]

        # The context manager should be an OpenMP instanciation.
        # This means calling the constructor.

        if not isinstance(item.context_expr, ast.Call):
            return self.generic_visit(node)

        call: ast.Call = item.context_expr

        if not isinstance(call.func, ast.Name):
            return self.generic_visit(node)

        name: ast.Name = call.func

        # Bulletproofing.

        if not isinstance(name.ctx, ast.Load):
            return self.generic_visit(node)

        # In order to check that this call indeed an OpenMP instanciation,
        # we will evaluate the name being called in the function's definition namespace.

        # ALERT: Catch the possible Name Exceptions when there are other context managers...
        expr: ast.Expression = ast.Expression(name)
        called = eval(compile(expr, filename='<OMP Parser>', mode='eval'), self.globs, self.locs)

        if called is not omp.core.openmp.OpenMP:
            return self.generic_visit(node)

        # We are now sure this is an OpenMP construct. (Not necessarily a valid one.)
        # We run the found instanciation and run its logic on the found construct.

        # ALERT: Catch possible Name Exceptions when there are other context managers...
        instruction = eval(compile(ast.Expression(call), filename='<OMP Parser>', mode='eval'), self.globs, self.locs)

        # Parsing this node **after** its children allows us to know the exhaustive list of local variables that will be
        # involved in the inner function definitions.
        return instruction._parse_With(
          ast.fix_missing_locations(self.generic_visit(node))
                                      )
```

Finally, we need to provide the user with an easy way to perform these transformations to their source code. We decided that the user should wrap their whole code in a decorated function. We called this decorator `enable`. This decorator should receive the user's function as a parameter. We use the built-in library `inspect`'s `getsource` to recover the source code of that function which we convert to an Abstract Syntax Tree. We could directly modify this AST directly using our `OpenMPTransformer`, but the source code of the function we just recovered actually **includes all the decorators** of the function definition. This means that after using our `OpenMPTransformer`, compiling the new function would run the `enable` decorator **again**.

This is why we also introduce an `EnableFunction` transformer class. This transformer finds the `FunctionDef` Node and removes its `enable` decorator. `EnableFunction` also adds an import to our library at the beginning of the body of the function. The library is imported with a custom `_omp_internal` name. This name should not be used by the user directly, and is used when directives need to refer to the library from the modified user code. After making those changes, `EnableFunction` calls `OpenMPTransformer` on the modified function definition.

EnableFunction:

```python
class EnableFunction(ast.NodeTransformer):
    """
    Transforms a enable-decorated function definition into an
enabled function
    definition without the enable decorator.
    """

    def __init__(self, locs=None, globs=None, varnames=None,
*args, **kwargs):
        super().__init__(*args, **kwargs)

        self.locs = locs
        self.globs = globs
        self.varnames = varnames
        if self.varnames is None:
            self.varnames = []

    def visit_FunctionDef(self, node: ast.FunctionDef) ->
ast.FunctionDef:
        new = copy.copy(node)
        # Remove the last decorator which should be ours if the
user followed our documentation.
        new.decorator_list = new.decorator_list[:-1]
        # Inject a known name in the namespace for our library.
        new.body = [ast.parse('import omp as _omp_internal',
mode='exec').body[0]] + new.body
        return OpenMPTransformer(self.globs,
self.locs).visit(new)
```

When using our `OpenMPTransformer`, we need to pass the **context** in which the user code was defined, which we also need to recover. Unlike the source code, we can't recover the defining context from the function that is passed to the decorator. Instead, we recover the current stack *Frame* using `inspect`. A stack Frame object contains information about the current state of variables and their scopes at a given point of the execution of a **function call**. The `f_locals` and `f_globals` attributes are the dictionaries of the local and global scopes of the function. As an example, `inspect.currentframe().f_locals` should be equivalent to `locals()`

Locals and globals example:

```
>>> inspect.currentframe().f_locals == locals()
True
>>> inspect.currentframe().f_globals == globals()
True
```

Also, a stack Frame object's `f_back` attribute references the **calling** Frame object. In the case of our decorator, the calling frame would be the one that defines the user's function. We can recover the `globals` and `locals` dictionaries of that frame as the context in which the user code is defined. We can also reuse that context to compile and run the modified user function definition. Once we ran the modified definition, we can recover the function itself from the context and return it.

enable decorator:

```
def enable(*args, **kwargs):
    """
    Enable OpenMP in the given block of code.

    Note: If your function has several decorators, this one
   should be the first to run. (i.e. closest to the function
   definition)

    Use as a decorator:
    ```
    @omp.enable
    def main():
        <omp-enabled code...>
    ```

    # Call your function
    main()
    """

    def decorator(function):
        # The user code calls `enable` which itself calls
   `decorator`.
        # We need to go back two stack frames.
        caller_frame = inspect.currentframe().f_back.f_back
        globs, locs = caller_frame.f_globals,
   caller_frame.f_locals

```

```python
        # Retrieve the source code of the decorated function.
        src: str = inspect.getsource(function)

        # Convert the source code to ast.
        src_ast: ast.Module = ast.parse(src, mode='exec')

        # Patch the source ast.
        # We need to make sure that each node has a line number.
        # Since the initial function was already compiled a first time, we can recover the
        # local variables the function uses from its code object.
        patched_ast = ast.fix_missing_locations(
            EnableFunction(globs, locs, function.__code__.co_varnames).visit(src_ast)
            )

        # ALERT: Remove this debug print. (Shows the final transformed source code.)
        print(ast.unparse(patched_ast))

        # Compile the patched ast.
        patched: CodeType = compile(patched_ast, filename=inspect.getsourcefile(function), mode='exec')

        # redefine the function in the initial context.
        # TODO: Allow enabling a function with closure (nested enabled function)
        #       Note: This would involve locating the function definition in the module's source code and recompile the outer function entirely.
        exec(patched, globs, locs)

        return locs[function.__name__]

    # Simple decorator.
    if len(args) == 1 and isinstance(args[0], FunctionType):
        return decorator(args[0])

    # TODO: Parametrized decorator support.
    # TODO: If statement support.
    return decorator
```

We took a small example to showcase the on-the-fly modifications brought by our library to the Abstract Syntax Tree of the user code that we can see in Figure 3.3

Example user code:

```python
@omp.enable
def main():
    with OpenMP("parallel"):
        print('Hello, World')
```

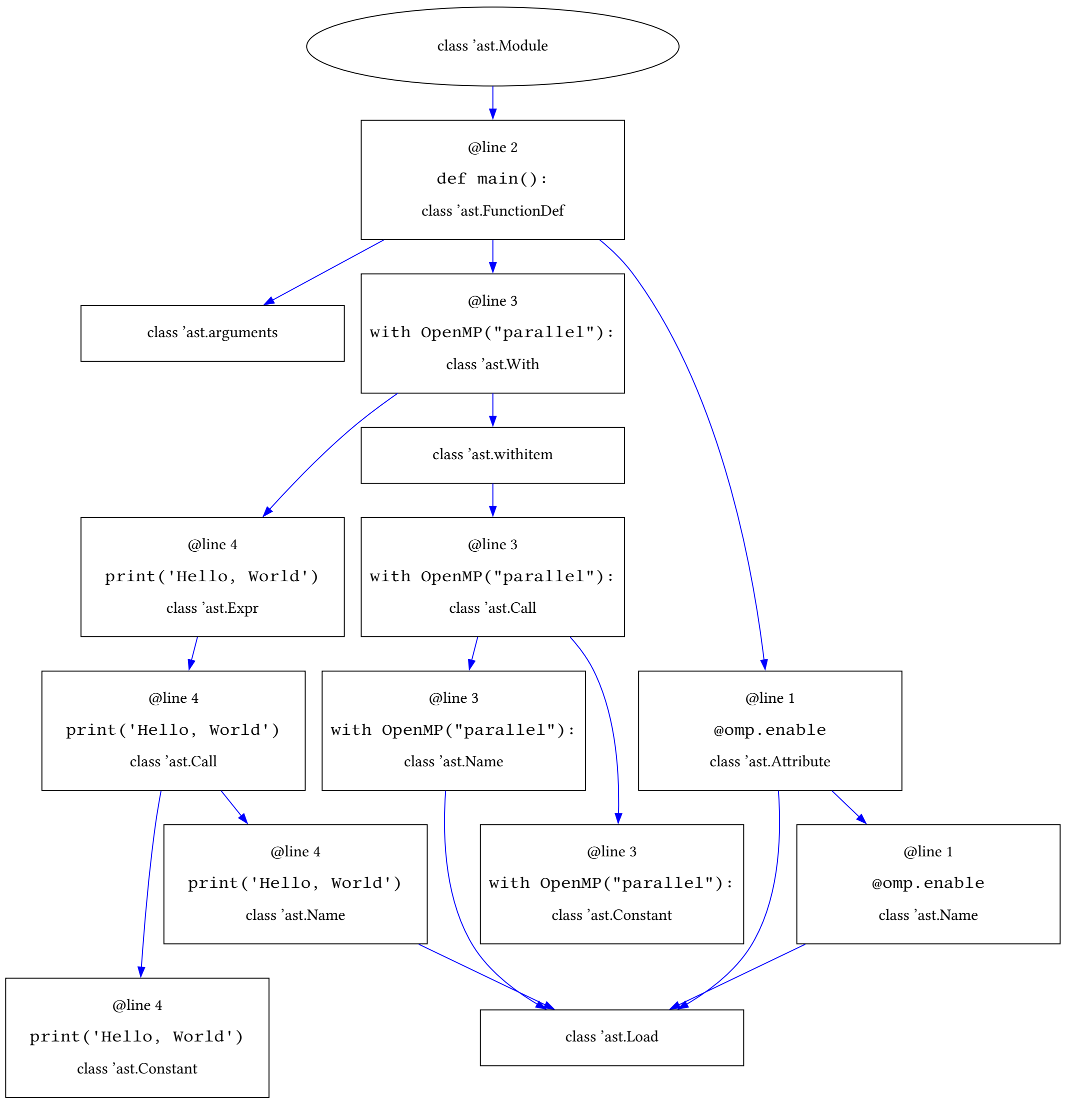
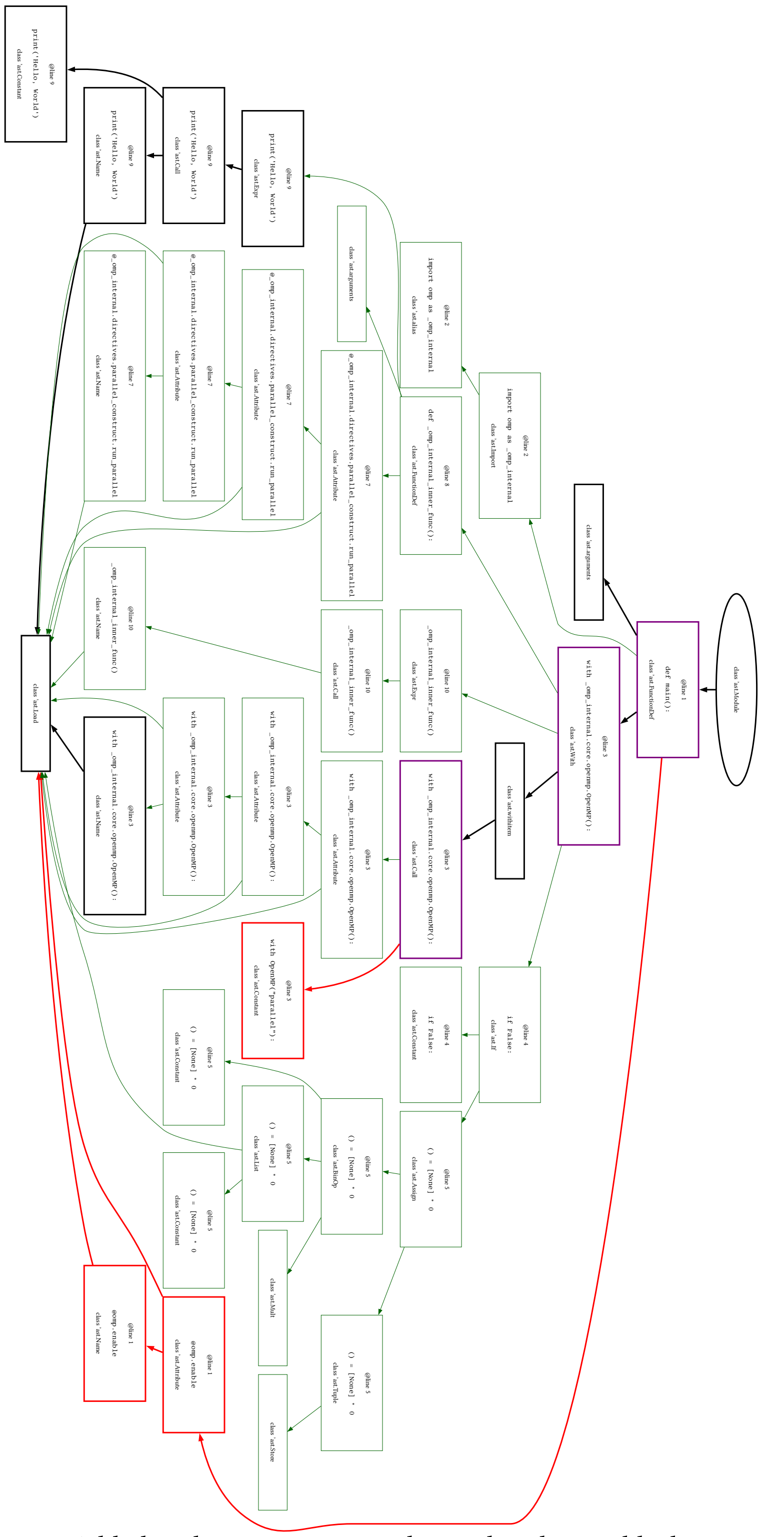


Figure 3.3: Graph of the AST representation of the user code



class 'ast.Module'
@line 1 def main(): class 'ast.FunctionDef'
class 'ast.arguments'
@line 3 with _omp_internal.core.openmp.OpenMP(): class 'ast.With'
class 'ast.withitem'
@line 2 import omp as _omp_internal class 'ast.Import'
@line 2 import omp as _omp_internal class 'ast.alias'
@line 8 def _omp_internal_inner_func(): class 'ast.FunctionDef'
@line 10 _omp_internal_inner_func() class 'ast.Expr'
@line 3 with _omp_internal.core.openmp.OpenMP(): class 'ast.Call'
@line 4 if False: class 'ast.If'
@line 4 if False: class 'ast.Constant'
@line 5 () = [None] * 0 class 'ast.Assign'
@line 5 () = [None] * 0 class 'ast.Tuple'
@line 5 () = [None] * 0 class 'ast.BinOp'
class 'ast.Store'
class 'ast.Mult'
@line 5 () = [None] * 0 class 'ast.List'
@line 5 () = [None] * 0 class 'ast.Constant'
@line 3 with OpenMP("parallel"): class 'ast.Constant'
@line 3 with _omp_internal.core.openmp.OpenMP(): class 'ast.Attribute'
@line 3 with _omp_internal.core.openmp.OpenMP(): class 'ast.Name'
@line 10 _omp_internal_inner_func() class 'ast.Call'
@line 10 _omp_internal_inner_func() class 'ast.Name'
@line 7 @_omp_internal.directives.parallel_construct.run_parallel class 'ast.Attribute'
@line 7 @_omp_internal.directives.parallel_construct.run_parallel class 'ast.Name'
class 'ast.arguments'
@line 9 print('Hello, world') class 'ast.Expr'
@line 9 print('Hello, world') class 'ast.Call'
@line 9 print('Hello, world') class 'ast.Name'
@line 9 print('Hello, world') class 'ast.Constant'
class 'ast.Load'
@line 1 @omp.enable class 'ast.Attribute'
@line 1 @omp.enable class 'ast.Name'

Added nodes are green. Unchanged nodes are black.
Modified nodes are purple. Removed nodes are red.

Figure 3.4: Graph of the library modified AST

Figures 3.4 shows the modified graphs and highlights the changes. One might notice that the nodes that remained in the graph, but were altered are only the OpenMP instructions themselves. We see for instance the removal of the `enable` decorator we mentioned earlier.

One could recognize the added nodes as the ones from the the template we saw earlier. Notice how the `Pass` node of the function body was replaced with the original user code, unchanged. See also how the `Pass` node of the `If` body was replaced with the empty, dummy, declaration.

These modifications are performed at runtime, as soon as the `main` user function is defined.

At this stage, the library has all the user-code transformation tools we needed to implement other directives, clauses and primitives.

We implemented the following directives:

- `barrier`
- `critical`
- `for`
- `parallel`
- `parallel for`
- `single`

We also implemented the following clauses:

- `nowait`
- `private`
- `reduction`
- `schedule`

Finally, we implemented the following primitives:

- `get_num_procs`
- `set_num_threads`
- `get_max_threads`
- `set_num_teams`
- `get_max_teams`

- `get_thread_num`
- `get_num_threads`
- `get_dynamic`
- `set_schedule`
- `get_schedule`

Chapter 4
Results

In this chapter we assess the performance of our solution using two user-code examples to showcase the use and limits of our Python package.

First, in the ”Tools” section, we introduce the tooling we needed to evaluate the library’s performance on these examples. We describe the specifications and versions used in our tests, and show how the tests can be reproduced.

The second section, ”Simple Sum,” discusses the first use case of sample user code. It discusses the runtimes, speedup and efficiency across different numbers of threads and different chunk sizes, and explains the results.

The third section, ”Counting Primes,” discusses our second use case, which tests a heavier loop body than the simple sum. This allows us to see the effect of the loop body on the runtimes, speedup, and efficiency. We will discuss the nature of the results and what lessons can be taken from them.

4.1 Tools

The library, as described in this documentation, does not cover the full OpenMP API. The coverage is however enough for some usages that we will present in this chapter.

The tests we will discuss in this chapter were performed on CESGA's FinisTerrae III supercomputer. More specifically, on `ilk` compute nodes, each featuring:

- 2x 32 Cores Intel Xeon Ice Lake 8352Y @2.2GHz, for a total of 64 cores.
- 256GB of RAM.
- 960GB of NVMe local storage.

Naturally, the system comes with software tooling as well:

- Linux 4.18.0-305.3.1.el8_4.x86_64 GNU/Linux
- gcc (Gentoo 10.1.0-r2 p3) 10.1.0
- ldd (Gentoo 2.31-r6 p8) 2.31 (GLIBC)
- Python 3.7.8

In order to benefit running code using the `omp` library, we need to use a **free-threaded** build of CPython. We compiled the master branch of the repository at `https://github.com/python/cpython`. The latest commit included in our copy is dated of July 28th 2024.
For reproducibility, we provide the commit hash: *bc93923a2dee00751e44da58b6967c63e3f5c392*.
To produce a free-threaded build, we first ran the configure script with the `--disable-gil` flag. We specify a prefixed installation, to prevent interfering with the system's Python installation. We can then use that prefixed python install to create a virtual environment to run our library in.

Configure CPython:

```
./configure --enable-optimizations --disable-gil
    --prefix=../prefix
make -j64
make install
../prefix/bin/python3.14t -m venv ../free-threaded
. ../free-threaded/bin/activate
```

4.2 Simple Sum

Our first example user code is a simple for loop computing the sum of integers ranging between 1 and N.

sum.py:

```python
#!/usr/bin/env python3
import omp
from omp import OpenMP

N = 40000000


@omp.enable
def main():
    acc = 0
    with OpenMP("parallel"):
        with OpenMP("for reduction(+:acc) schedule(dynamic,
   10000)"):
            for i in range(1, N):
                acc += i
    print("Actual result:  ", acc)
    print("Expected result:", N*(N-1)//2)


if __name__ == '__main__':
    main()
```

We use a reduction clause to protect the `acc` variable, but also because it has better performance than wrapping the update of `acc` in a `critical` construct. We also use dynamic scheduling because currently, our dynamic scheduling implementation is more efficient than our static scheduling implementation. One particularity of our dynamic scheduling implementation is that the access to the iterator is protected by a `threading` Lock. By increasing the chunk size, we reduce the overall number of accesses to the iterator, thus reducing the waiting time induced by the Lock. We ran the program while setting the OMP_NUM_THREADS environment variable to values ranging between 1 and 64, and measured the run time using the `time` utility. See the exact values in the appendices.

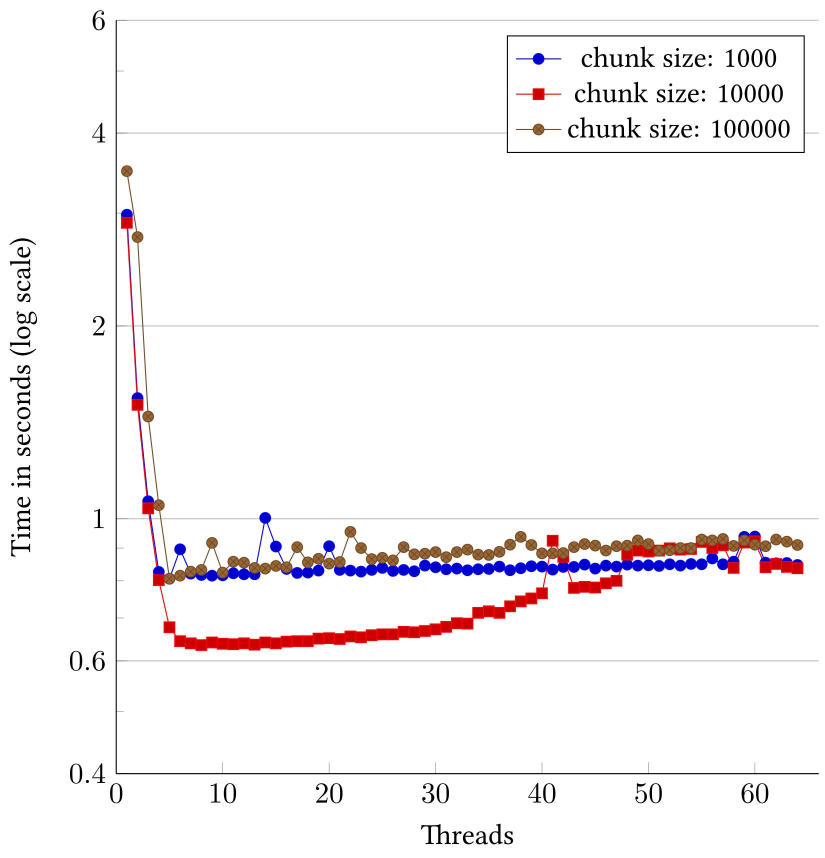


Figure 4.1: Time versus number of threads for a simple sum

As we can see in Figure 4.1, the run time does drop significantly when increasing the number of threads compared to a single-threaded run. For instance, the run time with 1 thread is of about 3 seconds, whereas it is of 0.7 seconds with 5 threads.

On the other hand, the run time does not decrease any further past 5 threads. There seems to be a **bottleneck** issue. We believe that the main reason for that bottleneck is the fact the body of the parallelized loop is so short. Realistically, the time spent updating the accumulator should be similar to the time spent iterating through the `range`. The iteration of the `range` can only be performed synchronously. The minimum run time we get is probably the time required to iterate through the `range` alone, assuming the updating of `acc` is ideally parallelized.

To verify this, we run a for loop that only iterates through the same `range`

range footprint:

```
$ time python -c 'tuple(range(40000000))'

real    0m0.986s
user    0m0.726s # This is not relevant for our usecase
sys     0m0.253s # This is not relevant for our usecase
```

The time spent to unpack that range is indeed very similar to the best run times of the parallelized simple sum.

We can also see that we get best results with a chunk size of 10000. This shows the importance of fine-tuning parameters to the usecase.

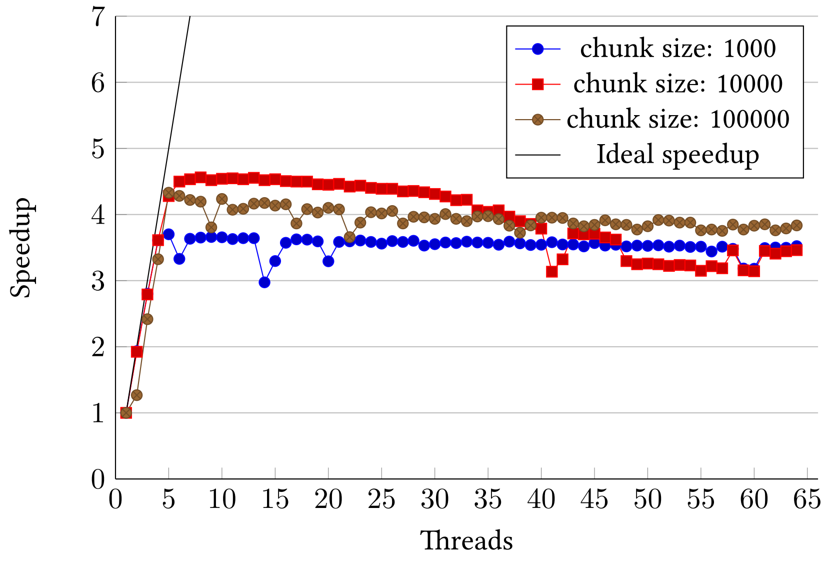


Figure 4.2: Speedup versus number of threads for a simple sum

In Figure 4.2, we see that we can reach a speedup of 4.3 with 5 threads. But adding threads past 5 does not seem to improve the speedup in this example.

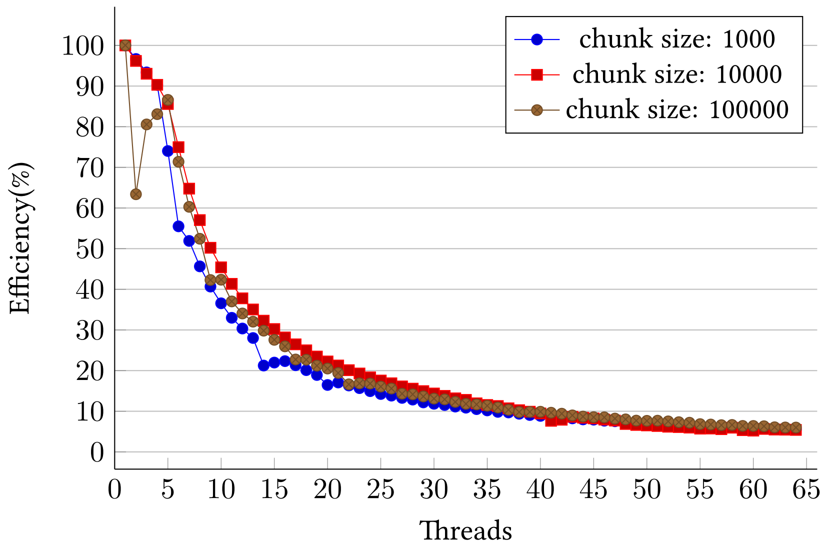


Figure 4.3: Efficiency versus number of threads for a simple sum

Finally, we can see from Figure 4.3 that past 6 threads, the efficiency drops below 60%. This shows that parallelizing a loop with a simple body can waste resources if the user does not fine tune parameters such as the number of threads.

4.3 Counting Primes

Despite getting some acceleration using our library in the simple sum example, we wanted to check that we could get better results with a heavier loop body. Here, we decide to count the number of primes within a given range. The code is similar to our simple loop, except that the check for primes number has a $O(\sqrt{n})$ complexity, which is non negligeable compared to our previous $O(1)$.

primes.py:

```python
#!/usr/bin/env python3
import omp
from omp import OpenMP

N = 10000000


def prime(n):
    for i in range(2, int(n**.5) + 1):
        if n % i == 0:
            return False
    return n > 1


@omp.enable
def main():
    acc = 0
    with OpenMP("parallel"):
        with OpenMP("for reduction(+:acc) schedule(dynamic, 100)"):
            for i in range(1, N):
                acc += prime(i)
    print("Actual result:  ", acc)


if __name__ == '__main__':
    main()
```

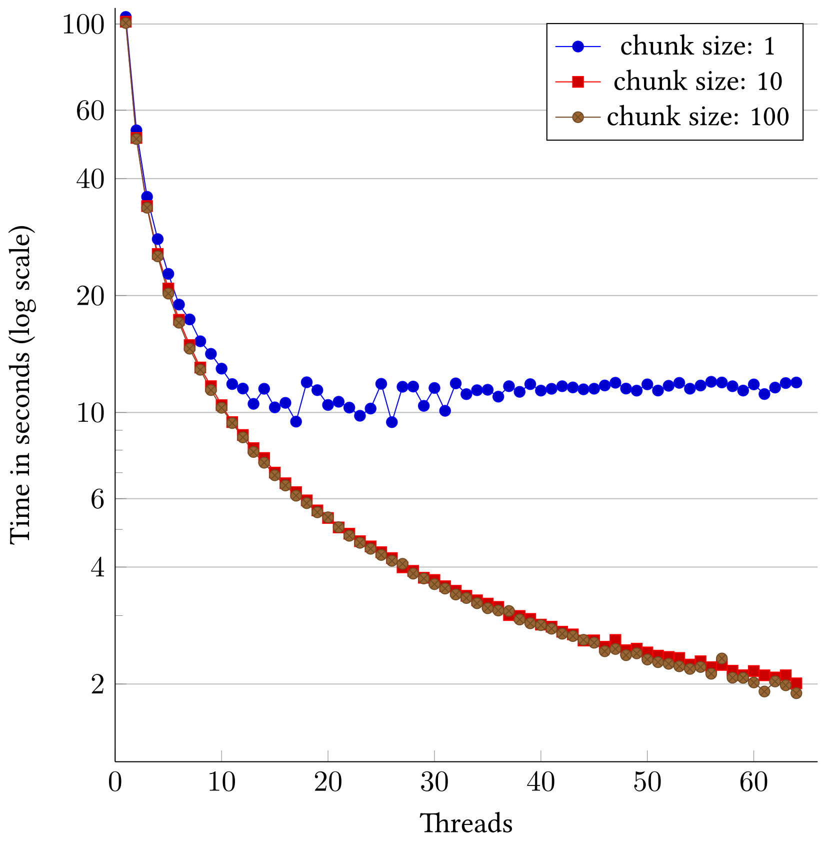


Figure 4.4: Time versus number of threads for counting primes

We can see from Figure 4.4 that as we expected, parallelizing a loop with a heavier body is easier. We shrunk the runtime from 100 seconds to just under 2 seconds. We also see that with a chunk size of 1, which is the default, we still manage to achieve a parallelized runtime of 10 seconds before hitting a bottleneck again. A heavier body loop is more forgiving and requires less fine-tuning.

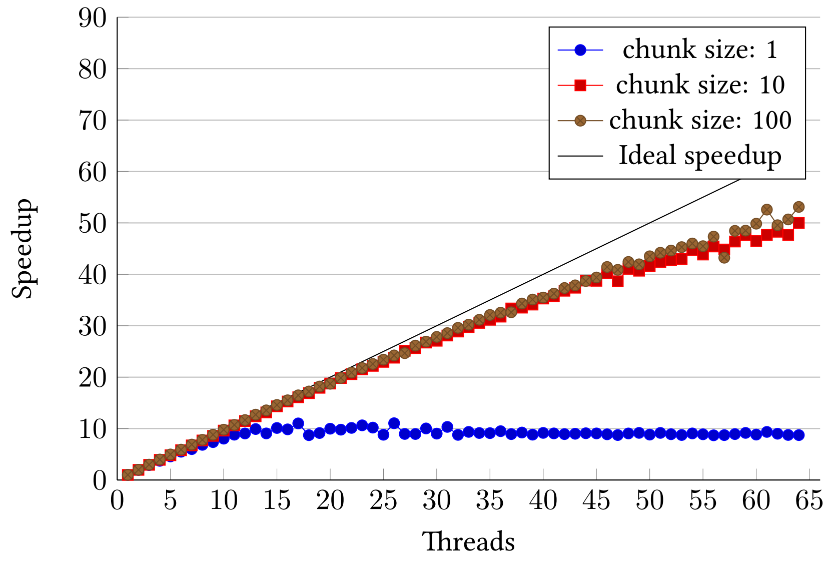


Figure 4.5: Speedup versus number of threads for counting primes

The results in Figure 4.5 are promising. Other than the bottleneck reached at 11 threads with a chunk size of 1, we could not reach the speedup limit with 64 cores with higher chunk sizes.

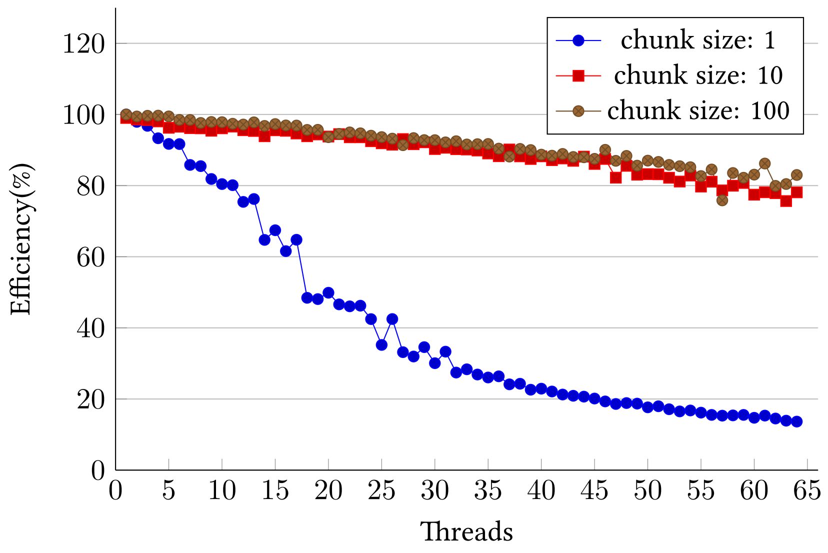


Figure 4.6: Efficiency versus number of threads for counting primes

Finally, the efficiency in Figure 4.6 remains above 80% with a chunk size of 100. This, once again, shows the importance of fine-tuning the parameters. Despite the need for fine-tuning, we believe these results are a success, and allows for incremental parallelization of user code with minimal changes compared to, as we saw earlier, using the built-in `threading` library directly.

Chapter 5

Conclusions

With the upcoming release of Python 3.13 and the introduction of PEP 703, which makes the Global Interpreter Lock optional, the potential for efficient multithreading in Python becomes significant. This new context raises the usefulness of an OpenMP-like implementation for Python. The high-level nature of Python makes it an ideal language for rapid prototyping and data analysis.

We achieved the objective of an easy-to-use syntax, similar to OpenMP in C. For this context, we limited ourselves to implementing a subset of the OpenMP API. This support is sufficient, particularly the `critical` directive as well as the `reduction` and `private` clauses which already allow the handling of many parallelization cases.

The most complex part of the code is already implemented, and future additions are reformulations of what is already present. By extending our library to include more OpenMP features, we can provide Python developers with tools to parallelize their computations effectively, similar to those available in C, C++, and Fortran.

We invite you to explore the source code at *https://github.com/douakli/omp* and the usage example in `examples/parallel_for.py`. This example demonstrates the practical application of our implementation and provides a starting point for further research and development.

Appendices

Appendix A

Simple sum, chunk size of 1000

thread	time	speedup	efficiency
1	2.98	1	100
2	1.54	1.93	96.66
3	1.07	2.8	93.36
4	0.83	3.61	90.28
5	0.81	3.7	74.02
6	0.9	3.33	55.49
7	0.82	3.63	51.91
8	0.82	3.65	45.64
9	0.82	3.66	40.67
10	0.82	3.66	36.56
11	0.82	3.63	32.99
12	0.82	3.64	30.35
13	0.82	3.64	28.02
14	1	2.97	21.24
15	0.91	3.3	21.97
16	0.84	3.57	22.33
17	0.82	3.62	21.32
18	0.82	3.62	20.11
19	0.83	3.59	18.92
20	0.91	3.29	16.46
21	0.83	3.59	17.07
22	0.83	3.59	16.34
23	0.83	3.61	15.68
24	0.83	3.59	14.94
25	0.84	3.56	14.24
26	0.83	3.6	13.84
27	0.83	3.59	13.28
28	0.83	3.6	12.87
29	0.85	3.53	12.17
30	0.84	3.55	11.84
31	0.83	3.58	11.54
32	0.84	3.57	11.15
33	0.83	3.59	10.88
34	0.83	3.58	10.52
35	0.84	3.57	10.21
36	0.84	3.54	9.84
37	0.83	3.59	9.7
38	0.84	3.56	9.38
39	0.84	3.54	9.07
40	0.84	3.54	8.86
41	0.83	3.58	8.73
42	0.84	3.55	8.45
43	0.84	3.55	8.25
44	0.85	3.52	7.99
45	0.84	3.57	7.93
46	0.85	3.53	7.67
47	0.84	3.54	7.54
48	0.85	3.52	7.33
49	0.85	3.53	7.2

50	0.85	3.53	7.05
51	0.84	3.53	6.93
52	0.85	3.51	6.76
53	0.85	3.53	6.66
54	0.85	3.51	6.5
55	0.85	3.51	6.39
56	0.87	3.44	6.14
57	0.85	3.51	6.16
58	0.86	3.48	6
59	0.94	3.18	5.4
60	0.94	3.18	5.3
61	0.85	3.49	5.73
62	0.85	3.5	5.65
63	0.85	3.5	5.55
64	0.85	3.52	5.5

Appendix B

Simple sum, chunk size of 10000

thread	time	speedup	efficiency
1	2.9	1	100
2	1.51	1.92	96.18
3	1.04	2.79	93.03
4	0.8	3.61	90.31
5	0.68	4.28	85.58
6	0.64	4.5	74.97
7	0.64	4.53	64.77
8	0.64	4.56	57.03
9	0.64	4.52	50.22
10	0.64	4.54	45.41
11	0.64	4.55	41.34
12	0.64	4.53	37.78
13	0.64	4.56	35.04
14	0.64	4.52	32.28
15	0.64	4.53	30.22
16	0.64	4.51	28.16
17	0.64	4.5	26.46
18	0.64	4.5	24.99
19	0.65	4.46	23.46
20	0.65	4.45	22.25
21	0.65	4.46	21.26
22	0.66	4.42	20.1
23	0.65	4.44	19.29
24	0.66	4.4	18.34
25	0.66	4.39	17.56
26	0.66	4.39	16.88
27	0.67	4.35	16.11
28	0.67	4.36	15.56
29	0.67	4.34	14.95
30	0.67	4.31	14.37
31	0.68	4.27	13.78
32	0.69	4.22	13.18
33	0.69	4.22	12.8
34	0.71	4.06	11.95
35	0.72	4.04	11.54
36	0.71	4.06	11.29
37	0.73	3.97	10.73
38	0.74	3.9	10.26
39	0.75	3.86	9.89
40	0.77	3.79	9.47
41	0.92	3.14	7.65
42	0.87	3.32	7.91
43	0.78	3.71	8.64
44	0.78	3.7	8.41
45	0.78	3.71	8.24
46	0.79	3.65	7.94
47	0.8	3.62	7.7
48	0.88	3.3	6.87
49	0.89	3.25	6.63

50	0.89	3.26	6.52
51	0.89	3.25	6.37
52	0.9	3.22	6.2
53	0.9	3.24	6.11
54	0.9	3.23	5.98
55	0.92	3.15	5.73
56	0.9	3.22	5.75
57	0.91	3.19	5.59
58	0.84	3.46	5.96
59	0.92	3.16	5.35
60	0.92	3.14	5.24
61	0.84	3.45	5.65
62	0.85	3.41	5.5
63	0.84	3.44	5.47
64	0.84	3.46	5.41

Appendix C

Simple sum, chunk size of 100000

thread	time	speedup	efficiency
1	3.49	1	100.03
2	2.75	1.27	63.39
3	1.44	2.42	80.56
4	1.05	3.32	83.1
5	0.81	4.33	86.6
6	0.82	4.28	71.37
7	0.83	4.22	60.29
8	0.83	4.19	52.43
9	0.92	3.81	42.29
10	0.82	4.24	42.35
11	0.86	4.07	37.02
12	0.85	4.09	34.06
13	0.84	4.16	32.04
14	0.84	4.17	29.82
15	0.84	4.14	27.57
16	0.84	4.15	25.97
17	0.9	3.86	22.73
18	0.86	4.08	22.68
19	0.87	4.03	21.21
20	0.85	4.1	20.51
21	0.86	4.08	19.41
22	0.95	3.66	16.63
23	0.9	3.88	16.86
24	0.87	4.03	16.81
25	0.87	4.02	16.06
26	0.86	4.05	15.59
27	0.9	3.86	14.31
28	0.88	3.97	14.16
29	0.88	3.96	13.64
30	0.89	3.93	13.12
31	0.87	4.01	12.93
32	0.89	3.93	12.3
33	0.9	3.9	11.82
34	0.88	3.97	11.68
35	0.88	3.97	11.36
36	0.89	3.93	10.92
37	0.91	3.83	10.35
38	0.94	3.72	9.8
39	0.91	3.84	9.83
40	0.88	3.95	9.88
41	0.88	3.95	9.64
42	0.88	3.95	9.4
43	0.9	3.86	8.99
44	0.91	3.82	8.69
45	0.91	3.84	8.54
46	0.89	3.91	8.51
47	0.91	3.85	8.2
48	0.91	3.84	8.01
49	0.93	3.77	7.7

50	0.91	3.82	7.65
51	0.89	3.92	7.68
52	0.89	3.91	7.52
53	0.9	3.88	7.32
54	0.9	3.88	7.18
55	0.93	3.76	6.84
56	0.93	3.77	6.74
57	0.93	3.75	6.58
58	0.91	3.85	6.63
59	0.93	3.77	6.39
60	0.91	3.83	6.38
61	0.91	3.85	6.31
62	0.93	3.76	6.07
63	0.92	3.79	6.01
64	0.91	3.84	5.99

Appendix D

Counting primes, chunk size of 1

thread	time	speedup	efficiency
1	104.3	1	100
2	53.26	1.96	97.92
3	35.92	2.9	96.8
4	27.94	3.73	93.31
5	22.74	4.59	91.72
6	18.96	5.5	91.67
7	17.36	6.01	85.81
8	15.25	6.84	85.47
9	14.16	7.37	81.86
10	12.97	8.04	80.43
11	11.84	8.81	80.12
12	11.52	9.05	75.42
13	10.53	9.91	76.23
14	11.51	9.06	64.73
15	10.31	10.12	67.45
16	10.59	9.85	61.57
17	9.47	11.01	64.77
18	11.96	8.72	48.44
19	11.42	9.14	48.09
20	10.46	9.98	49.88
21	10.66	9.79	46.6
22	10.29	10.13	46.05
23	9.81	10.63	46.24
24	10.24	10.19	42.46
25	11.85	8.8	35.2
26	9.44	11.05	42.48
27	11.65	8.96	33.17
28	11.66	8.94	31.94
29	10.4	10.03	34.58
30	11.56	9.02	30.07
31	10.1	10.33	33.31
32	11.88	8.78	27.43
33	11.15	9.36	28.35
34	11.41	9.14	26.88
35	11.44	9.12	26.05
36	10.99	9.49	26.37
37	11.69	8.92	24.12
38	11.3	9.23	24.29
39	11.83	8.82	22.61
40	11.39	9.16	22.9
41	11.51	9.06	22.11
42	11.69	8.93	21.25
43	11.6	8.99	20.91
44	11.47	9.09	20.66
45	11.51	9.06	20.13
46	11.74	8.89	19.32
47	11.93	8.74	18.6
48	11.52	9.05	18.86
49	11.38	9.16	18.7

50	11.82	8.83	17.66
51	11.39	9.16	17.96
52	11.72	8.9	17.12
53	11.92	8.75	16.51
54	11.51	9.06	16.77
55	11.73	8.89	16.16
56	12	8.7	15.53
57	11.95	8.73	15.32
58	11.68	8.93	15.39
59	11.39	9.16	15.52
60	11.81	8.83	14.72
61	11.15	9.35	15.33
62	11.59	9	14.51
63	11.91	8.76	13.91
64	11.94	8.73	13.64

Appendix E

Counting primes, chunk size of 10

thread	time	speedup	efficiency
1	101.37	0.99	99.01
2	50.9	1.97	98.59
3	33.99	2.95	98.44
4	25.59	3.92	98.05
5	20.86	4.81	96.26
6	17.33	5.79	96.56
7	14.91	6.73	96.16
8	13.06	7.69	96.1
9	11.69	8.59	95.43
10	10.44	9.61	96.1
11	9.45	10.63	96.6
12	8.75	11.47	95.57
13	8.1	12.4	95.36
14	7.64	13.14	93.88
15	7	14.33	95.55
16	6.58	15.27	95.41
17	6.24	16.09	94.65
18	5.94	16.9	93.88
19	5.6	17.93	94.39
20	5.35	18.76	93.79
21	5.06	19.84	94.48
22	4.88	20.59	93.59
23	4.66	21.53	93.59
24	4.52	22.2	92.48
25	4.37	22.97	91.89
26	4.22	23.79	91.5
27	3.99	25.13	93.08
28	3.91	25.66	91.63
29	3.76	26.73	92.17
30	3.71	27.08	90.28
31	3.57	28.1	90.64
32	3.48	28.88	90.24
33	3.37	29.75	90.15
34	3.29	30.55	89.87
35	3.22	31.16	89.03
36	3.16	31.77	88.26
37	3.01	33.36	90.15
38	2.99	33.52	88.22
39	2.94	34.12	87.48
40	2.84	35.29	88.23
41	2.81	35.73	87.15
42	2.73	36.81	87.64
43	2.68	37.4	86.97
44	2.59	38.78	88.14
45	2.59	38.74	86.09
46	2.49	40.25	87.49
47	2.6	38.65	82.23
48	2.44	41.07	85.56
49	2.47	40.7	83.07

50	2.41	41.61	83.23
51	2.37	42.42	83.18
52	2.35	42.75	82.21
53	2.33	43	81.14
54	2.24	44.73	82.83
55	2.29	43.85	79.73
56	2.21	45.42	81.1
57	2.24	44.83	78.65
58	2.16	46.38	79.97
59	2.11	47.64	80.74
60	2.16	46.47	77.45
61	2.11	47.66	78.13
62	2.08	48.28	77.87
63	2.11	47.66	75.65
64	2.01	49.99	78.1

Appendix F

Counting primes, chunk size of 100

thread	time	speedup	efficiency
1	100.59	1	100
2	50.58	1.99	99.43
3	33.65	2.99	99.64
4	25.23	3.99	99.68
5	20.23	4.97	99.45
6	17.02	5.91	98.48
7	14.6	6.89	98.46
8	12.88	7.81	97.62
9	11.42	8.81	97.89
10	10.28	9.78	97.83
11	9.39	10.71	97.39
12	8.63	11.66	97.17
13	7.91	12.71	97.79
14	7.42	13.55	96.78
15	6.89	14.6	97.3
16	6.49	15.51	96.92
17	6.11	16.47	96.89
18	5.84	17.22	95.66
19	5.53	18.18	95.69
20	5.37	18.72	93.61
21	5.07	19.85	94.54
22	4.82	20.89	94.96
23	4.62	21.78	94.71
24	4.46	22.57	94.04
25	4.3	23.41	93.64
26	4.15	24.23	93.21
27	4.08	24.67	91.36
28	3.85	26.14	93.36
29	3.74	26.91	92.8
30	3.61	27.83	92.78
31	3.52	28.57	92.16
32	3.4	29.59	92.48
33	3.33	30.22	91.57
34	3.23	31.16	91.65
35	3.13	32.1	91.71
36	3.09	32.54	90.4
37	3.08	32.62	88.16
38	2.93	34.32	90.32
39	2.87	35.1	90
40	2.84	35.48	88.71
41	2.78	36.25	88.41
42	2.69	37.34	88.9
43	2.66	37.86	88.05
44	2.6	38.7	87.96
45	2.55	39.39	87.53
46	2.43	41.43	90.07
47	2.46	40.87	86.97
48	2.37	42.41	88.35
49	2.4	41.95	85.61

50	2.31	43.51	87.02
51	2.28	44.2	86.66
52	2.25	44.63	85.82
53	2.22	45.29	85.46
54	2.19	46	85.18
55	2.21	45.48	82.68
56	2.13	47.34	84.53
57	2.33	43.23	75.84
58	2.08	48.46	83.54
59	2.07	48.5	82.21
60	2.02	49.85	83.08
61	1.91	52.58	86.2
62	2.03	49.55	79.92
63	1.99	50.68	80.44
64	1.89	53.11	82.99

Appendix G

Glossary of Acronyms

API	*Application Programming Interface*
CPU	*Central Processing Unit*
GIL	*Global Interpreter Lock*
GPU	*Graphics Processing Unit*
HPC	*High Performance Computing*
I/O	*Input/Output*
IPC	*Inter-Process Communication*
LLM	*Large Language Model*
OS	*Operating System*
PEP	*Python Enhancement Proposal*
RAM	*Random Access Memory*

Appendix H
Glossary of Terms

Concurrency The ability of a system to handle multiple tasks at the same time.

CPython The default and most widely used implementation of the Python programming language, written in C.

Global Interpreter Lock (GIL) A mutex in CPython that protects access to Python objects, preventing multiple native threads from executing Python bytecodes at once.

Inter-Process Communication (IPC) A set of techniques for exchanging data between multiple processes.

Multithreading A concurrent execution of two or more threads within a single process, sharing the same data space.

Mutex A mutual exclusion object that prevents multiple threads from simultaneously accessing shared resources.

OpenMP An API that supports multi-platform multithreading programming.

Parallelization The process of executing multiple tasks simultaneously, which can significantly improve performance.

PEP 703 A Python Enhancement Proposal that aims to make the Global Interpreter Lock optional in CPython, enabling true multithreading capabilities.

Shared Memory A memory accessible by multiple processes or threads to exchange information.

Bibliography

[1] "Globalinterpreterlock - python wiki," May 2024, [Online; accessed 10. May 2024]. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock

[2] "Pep 703 – making the global interpreter lock optional in cpython | peps.python.org," Mar. 2024, [Online; accessed 10. May 2024]. [Online]. Available: https://peps.python.org/pep-0703

[3] "Threading — thread-based parallelism," May 2024, [Online; accessed 10. May 2024]. [Online]. Available: https://docs.python.org/3/library/threading.html

[4] "Python initialization, finalization, and threads," May 2024, [Online; accessed 12. May 2024]. [Online]. Available: https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock

[5] T. G. Mattson, T. A. Anderson, and G. Georgakoudis, "Pyomp: Multithreaded parallel programming in python," *Computing in Science & Engineering*, vol. 23, no. 6, p. 77–80, 2021.

[6] S. GUELTON, P. BRUNET, and M. Amini, "Compiling python modules to native parallel modules using pythran and openmp annotations," 11 2013.

[7] "Multiprocessing — process-based parallelism," May 2024, [Online; accessed 10. May 2024]. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html

[8] "Concurrent.futures — launching parallel tasks," May 2024, [Online; accessed 10. May 2024]. [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html

[9] "Asyncio — asynchronous i/o," May 2024, [Online; accessed 10. May 2024]. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[10] E. Conference, “Keynote: Multithreaded python without the gil - presented by sam gross,” Nov. 2022, [Online; accessed 10. May 2024]. [Online]. Available: https://www.youtube.com/watch?v=9OOJcTp8dqE

[11] “Python glossary,” May 2024, [Online; accessed 11. May 2024]. [Online]. Available: https://docs.python.org/3/glossary.html#term-decorator

[12] “Inspect — inspect live objects,” May 2024, [Online; accessed 12. May 2024]. [Online]. Available: https://docs.python.org/3/library/inspect.html

[13] “Ast — abstract syntax trees,” May 2024, [Online; accessed 12. May 2024]. [Online]. Available: https://docs.python.org/3/library/ast.html

[14] “Python built-in functions,” May 2024, [Online; accessed 12. May 2024]. [Online]. Available: https://docs.python.org/3/library/functions.html#compile

[15] “Python data model,” May 2024, [Online; accessed 12. May 2024]. [Online]. Available: https://docs.python.org/3/reference/datamodel.html#code-objects

[16] “Python simple statements,” May 2024, [Online; accessed 13. May 2024]. [Online]. Available: https://docs.python.org/3/reference/simple_stmts.html#grammar-token-python-grammar-nonlocal_stmt

[17] O. A. R. Board, B. de Supinski, and M. Klemm, *OpenMP Application Programming Interface Specification Version 5.2.* Independently published, Nov. 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[18] O. A. R. Board, *OpenMP 5.2 Reference Guide.* Independently published, 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMPRefGuide-5.2-Web-2024.pdf