Universidad Carlos III de Madrid

Bachelor's degree in Computer Science and Engineering

Bachelor Thesis

"Evaluating performance and energy impact of programming languages"

Author

Eduardo Alarcón Navarro

Advisor

Jose Daniel García Sánchez

Leganés, Madrid, Spain

September 2025

To reach the moon, you should aim for the stars.

—

ACKNOWLEDGEMENTS

I would like to thank my family, for their continuous support and encouragement, specially this last year, where it has been the hardest for all of us. Not only they have provided me with the best education possible, financially and emotionally, but they have also taught me the importance of hard work and dedication.

I would like to thank my parents, my father for always being there when I needed him, for listening to me rant about many university projects, or crazy ideas. Furthermore, I would also like to thank my mother, for always being there for me, and for being the best mother I could have ever asked for, and for always supporting me in everything I do, even if we don't always agree on everything.

I would not have been able to finish my degree without the help of my friends, who have always been there for me, adopting me as part of their family and being there when I needed them most. During the four years of my degree, I have met many people, some have persevered in completing their degree, while others have chosen to take a different path in life. I would like to thank all of them for being there for me, and for being part of my life.

Another person I would like to thank is my thesis advisor Jose Daniel. Thank you for giving me the opportunity to work with you, after an extreme late request for a thesis. I am grateful for the trust you placed in me from the very beginning and for guiding me when I was most lost. From the admiration, it has been a great pride to close this stage with you as my advisor.

ABSTRACT

Nowadays, the importance of energy efficiency is increasing as more computationally expensive programs are being used by more and more people. The energy impact of running these programs is directly related with the programming language used to create the program as well as the design specifics.

This thesis aims to bring a specific example of the power efficiency of three programming languages: Python, Go and C++. Each one having its differences and properties, ease of use and execution speed. Each of these languages has been selected as each one has a particular characteristic that can be representative of their respective category of language.

Compiled languages with no garbage collection and no managed runtime have usually had the best execution speed as they can reach byte-code for each specific platform, but in the last years, other methods have improved significantly, such as JIT (Just in Time) compiling

To achieve realistic results, these languages were tested in multiple configurations, on different hardware, core count and operating systems to be able to eliminate any outliers.

Thus, this work will try showing the differences in energy consumption of different programming languages in a real world task, rendering a ray-traced image of multiple spheres with different materials and reflectivity.

Keywords: Compiled Language • Energy Efficiency • Interpreted Language • JIT • Ray-Tracing

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1. Motivation

Energy consumption in the software industry has been raising over the years up to a point that it is now significant at a world energy consumption.

As [1] states, in 2018 an estimated 1% of total energy consumption was attributed to datacenter alone. In 2024, it is estimated that about 1.5% of the world's energy consumption is to be blamed on data centers and server farms. These numbers may not represent much, but from the total $183,230$ TWh produced in 2023 [2] only $23,746$ TWh come from a renewable source as it can be seen from Figure 1.1, which comes to 12.96%.

Knowing which language to use for each project is decisive not only in regards to the expertise one or their team might have on that language, but also the performance and language characteristics. If you want to develop a high-performance stock trader you would never think about using a high level language such as python or Perl, but you would try sticking to compiled languages such as Java C, C++, Java or Rust.

Thus, the main motivation for this project lies in studying 3 different programming languages, with each one having peculiar characteristics, to test their respective speed and power consumption in different platforms and architectures.

This comes from the idea that the program efficiency does not come from the language itself, but the implementation of the algorithm that the programmer chooses. The language helps, but choosing the optimal algorithm is much more important.

My personal take in this project comes from my hesitation in choosing a topic to specialize in inside the Computer Science area. Having seen and used many of these languages in multiple courses along these 4 years has made me realize the importance of choosing the correct language for each problem.

2023
in terawatt-hours

Other renewables	2,428 TWh
Modern biofuels	1,318 TWh
Solar	4,264 TWh
Wind	6,040 TWh
Hydropower	11,014 TWh
Nuclear	6,824 TWh
Natural gas	40,102 TWh
Oil	54,564 TWh
Coal	45,565 TWh
Traditional biomass	11,111 TWh
Total	**183,230 TWh**

Fig. 1.1. Global electricity generation by source in 2023

1.2. Objectives

The main goal of this project is the study and analysis of three implementations of a ray-tracer program, measuring the energy consumption as well as the time each program takes to complete. It should be also noted that the platform in which the program is being run affects the energy consumption of the program.

To perform this, I have improved the code from a well-known book called Ray Tracing in One Weekend [3], translating it to go and python, updating the code so that it could handle parallel rendering.

Once the code is created, the methodology for testing the different codes need to also be created.

- x86 Intel Xeon Based
- ARM Apple Icestorm & Firestorm
- x86 Zen 2 AMD ????
- ARM Cortex-A76 CPU - Raspberry Pi 5

1.3. Document Structure

The document contains the following chapters:

- Chapter 1, *Introduction*, details the motivation of the project.

- Chapter 2, *State of the Art*, describes the main points of interest in order to fully understand the project. Theoretical and technological issues are addressed.
- Chapter 3, *Problem Statement*, general description of the project and its requirements.
- Chapter 4, *Design and Implementation*, describes the most relevant design decisions with the multi-language renderers and their multi-threaded implementation.
- Chapter 5, *Evaluation*, the analysis and benchmarks are performed and the results are exposed and discussed.
- Chapter 7, *Socioeconomic environment*, provides a comprehensive account of the project's developmental costs and its associated socio-economic implications.
- Chapter 6, *Planification*, describes the organization of the project along the development.
- Chapter 8, *Regulatory Framework*, indicates the licenses under which the project is distributed.
- Chapter 9, *Conclusions and Future Work*, briefly analyzes the results obtained and states the possible future objectives of the project.

CHAPTER 2

STATE OF THE ART

This chapter describes the paradigms and characteristics of different programming languages. Thus concepts of compiled languages, interpreters optimizations and parallelism are discussed with respect of the different programming languages.

The purpose is to provide background information necessary to understand the study and present a clear justification for the decisions made

2.1. Energy Efficient Systems

An energy efficient system is defined as a system designed and optimized to performs its functions while consuming the minimum amount of energy possible, without compromising its performance, safety and reliability.

As Muralidhar et al. [4] put it, the average power a system draws is:

$$P_{avg} = P_{dynamic} + P_{leakage}$$

The dynamic power depends on the V supply, the clock frequency, the node capacitance and the switching activity. This power can be reduced by reducing the load on the chip or by manually setting a limit on how much voltage the chip can draw (known as undervolting [5])

2.2. System Architectures

While the energy efficiency of a system is significantly affected by connected devices (e.g., a graphics card or an AI Accelerator), this study excludes any external devices and expansion cards. Therefore, the processor architecture is the primary factor determining energy consumption on the system.

2.2.1. x86 Architecture

The x86 architecture is the most widely adopted in the world of desktop and server computers, whose market share is almost entirely shared by AMD and Intel who created it in 1978. Originally called x86-16, due to the 16 bit word size, it debuted in the Intel 8086 a single core, a $3\mu m$ node processor.

Nowadays, the technology has improved, the architecture is now called x86-64, a 64 bit extension, created by AMD, and releases the full specification in August 2000. From 2006 onward, the two companies have been developing multi-core processors, adding further Single Instruction Multiple Data (SIMD) Extensions such as AVX-512 [6]. Then came the integrated graphics and finally Power Efficiency Focus.

Then, a new paradigm came, instead of having a homogeneous set of cores, cores focused on performance and efficiency were added to the same package, the hybrid architecture. This set of heterogeneous cores meant the scheduler had to be changed in the operating systems, to better allocate more demanding programs on high performing cores and lower important tasks, such as background jobs to the highly efficient cores. This technology was released by Intel on the 12th generation Intel core processors, using Intel 7 (a $7nm$ node). This approach was revolutionary for power efficiency as Padoin et al. [7] state.

2.2.2. ARM Architecture

The ARM (Advanced RISC Machine) is the newest architecture that has reached the global scale. Developed in 1986, the goal of this new 32 bit architecture was the simplicity. As Moir [8] puts it, the energy efficient came later. This allowed the ARM architecture to dominate on the mobile sector, specially on smartphones, which run on batteries.

The characteristics of this Instruction Set Architecture (ISA) are a reduced set of instructions (Reduced Instruction Set Computer (RISC)), which allows processors to have fewer transistors than Complex Instruction Set Computer (CISC) architectures such as x86, resulting in lower cost, lower temperatures and lower power consumption.

Currently, this technology is not only used in low-power light devices, but many laptops, and even desktops are using ARM chips due to their power efficiency and performance [9].

ARM also has a hybrid technology, called big.LITTLE, as described by the [10] ARM White Paper that combines high-efficiency cores and high-performance cores. This architecture dominates the mobile device market and is increasingly found in modern laptops.

<table>
<tr><th></th><th>IVB</th><th>SKX</th><th>ZEN</th><th>ZEN2</th><th>TX2</th><th>A64FX</th></tr>
<tr><td>Manufacturer</td><td>Intel</td><td>Intel</td><td>AMD</td><td>AMD</td><td>Marvell</td><td>Fujitsu</td></tr>
<tr><td>Microarchitecture</td><td>Ivy Bridge</td><td>Skylake-X</td><td>Zen</td><td>Zen2</td><td>ThunderX2</td><td>A64FX</td></tr>
<tr><td>Instr. Set Arch.</td><td>x86 with AVX2</td><td>x86 with AVX512</td><td>x86 with AVX2</td><td>x86 with AVX2</td><td>ARMv8 with Neon</td><td>ARMv8 with SVE</td></tr>
<tr><td>Model name</td><td>Xeon E5-2690v2</td><td>Xeon Gold 6148</td><td>EPYC 7451</td><td>EPYC 7452</td><td>CN9980</td><td>PRIMEHPC FX700</td></tr>
<tr><td>Base frequency</td><td>3.0 GHz</td><td>2.4 GHz</td><td>2.3 GHz</td><td>2.35 GHz</td><td>2.5 GHz</td><td>1.8 GHz</td></tr>
<tr><td>Cores</td><td>10 per socket in one NUMA domain</td><td>20 per socket 10 per SubNUMA domain</td><td>24 per socket 6 per NUMA domain</td><td>32 per socket 8 per NUMA domain</td><td>32 per socket in one NUMA domain</td><td>48 per socket 12 per NUMA domain[1]</td></tr>
<tr><td>LD/ST reciprocal throughput</td><td>(2 LD || 1 half-ST & 1 LD || 1 half-ST) with 256 bit width per cycle</td><td>(2 LD || 1 ST & 1 LD || 1 simple-ST & 2 LD || 1 ST) with 512 bit width per cycle</td><td>(2 LD || 1 ST & 1 LD || 1 ST) with 256 bit width per cycle</td><td>(2 LD || 1 ST & 2 LD || 1 ST) with 256 bit width per cycle</td><td>(2 LD || 1 ST & 1 LD || 2 ST) with 128 bit width per cycle</td><td>(2 LD || 1 ST) with 512 bit width per cycle</td></tr>
<tr><td>LD latency</td><td>4 cy per LD</td><td>4 cy per LD</td><td>4 cy per LD</td><td>4 cy per LD</td><td>4 cy per LD</td><td>5 cy per LD</td></tr>
<tr><td>L1D cache size</td><td>32 KiB per core</td><td>32 KiB per core</td><td>32 KiB per core</td><td>32 KiB per core</td><td>32 KiB per core</td><td>64 KiB per core</td></tr>
<tr><td>L1D-L2 bandwidth</td><td>32 B/cy, half-duplex per core</td><td>64 B/cy, half-duplex per core</td><td>32 B/cy, full-duplex per core</td><td>32 B/cy, full-duplex per core</td><td>64 B/cy, half-duplex per core</td><td>64 B/cy, half-duplex per core</td></tr>
<tr><td>L2 cache size</td><td>256 KiB per core</td><td>1 MiB per core</td><td>512 KiB per core</td><td>512 KiB per core</td><td>256 KiB per core</td><td>8 MiB per core</td></tr>
<tr><td>L2-L3 bandwidth</td><td>32 B/cy, half-duplex per core</td><td>16 B/cy, full-duplex per core</td><td>32 B/cy, half-duplex per core</td><td>24 B/cy, half-duplex per core</td><td>32 B/cy, half-duplex per core</td><td>no L3</td></tr>
<tr><td>L3 cache type</td><td>inclusive</td><td>victim</td><td>inclusive</td><td>victim</td><td>victim</td><td>no L3</td></tr>
<tr><td>L3 cache size</td><td>25 MiB per per socket</td><td>13.75 MiB per SubNUMA domain</td><td>8 MiB per CCX with two CCXs per NUMA domain</td><td>16 MiB per NUMA domain</td><td>32 MiB per socket</td><td>no L3</td></tr>
<tr><td>Memory bandwidth</td><td>56 GB/s with load per full socket</td><td>131 GB/s with load per full socket</td><td>149 GB/s with load per full socket</td><td>143 GB/s with load per full socket</td><td>125 GB/s with load per full socket</td><td>227 GB/s with load per full socket</td></tr>
</table>

[1] modeled in KERNCRAFT as 4 sockets with 12 cores each

Fig. 2.1. Architectures and most important parameters used for evaluation

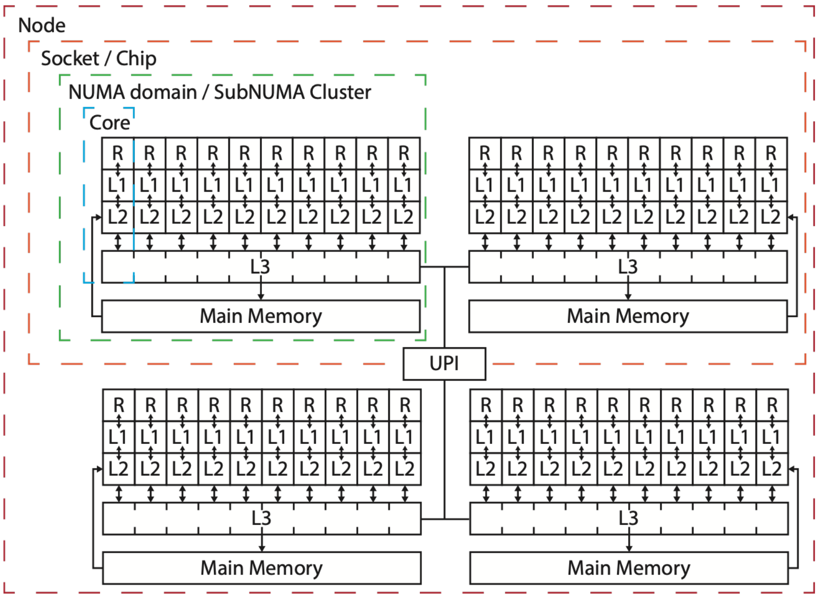


Fig. 2.2. Intel Xeon Skylake-X

2.2.3. CPU Design for Multi-core dies

Is it important to note that a computer that reports having more than 32 logical cores, usually has more than one socket, thus the performance and scaling of programs on multiple sockets can affect the energy efficiency and performance. This is due to the fact that information has to move between the multiple cache levels.

From [11]'s Figure 2.1 we can see there are multiple configurations, depending on the architecture, the amount of cache per core, how many cores there are per chip and the memory bandwidth.

The traditional layout of these Central Processing Units (CPUs) can be found from Figure 2.2, where each ten cores form a Non-Uniform Memory Access (NUMA) domain, two NUMA domains for each of the chips (sockets) and multiple chips (two in this case) form a NUMA node.

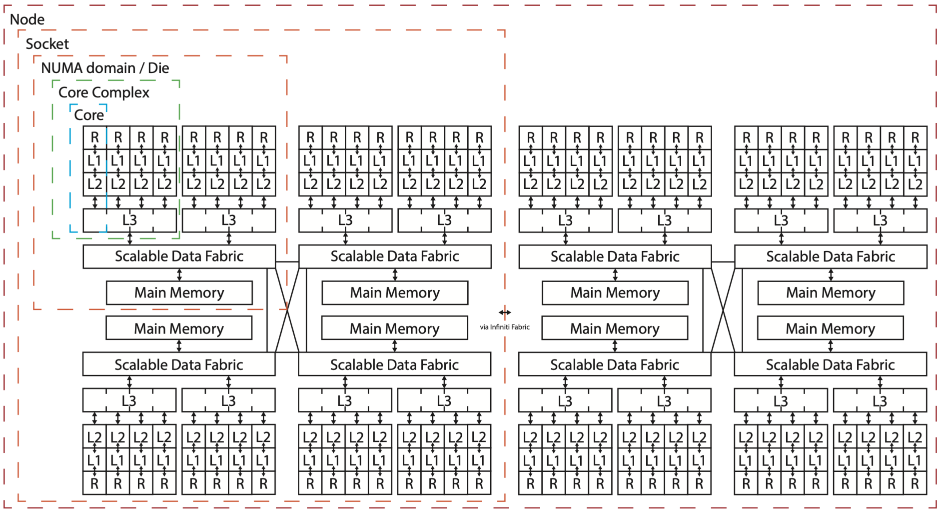


Fig. 2.3. AMD introduced an additional hierarchy level with its Zen architecture: CCX. A single core complex has up to four cores, with a sliced victim L3 cache. Two CCXs are combined onto a die, sharing a local partition of main memory. Multiple dies make up a socket

Using AMD's ZEN 2 architecture as a new CPU architecture example, we can observe there are is an additional layer, compared to Figure 2.2. As Gibbs [11] puts it in their paper, in 2.3, the victim of this design is the L3 cache, which is shared between less cores.

2.2.4. Hyperthreading

Hyperthreading is a technology that allows a single physical core to appear as two logical cores to the operating system. This means that the operating system can schedule two threads on a single physical core, allowing for better utilization of the CPU resources. This usually entails a better performance, as described by Leng et al. [12], but when CPU intensive benchmarks are run, such as integer sorting, the performance can be worse than running the same program on a system that does not have hyperthreading enabled. This is because the two threads share the same resources, such as the cache and the execution units, which can lead to contention and reduced performance.

2.3. Programming Languages

In this section, the different programming languages that have been chosen will be discussed, as well as why other similar languages were not.

TABLE 2.1
COMPARISON OF C++, PYTHON, AND GO GENERAL CHARACTERISTICS

Characteristic	C++	Python	Go
Typing System	Statically Typed: Types are checked at compile-time, catching errors early and aiding optimization.	Dynamically Typed: Type checking occurs at runtime. Optional static type hinting (PEP 484) is available.	Statically Typed: Types are checked at compile-time, ensuring type safety and early error detection.
Compilation & Execution	Compiled: Code is compiled directly to native machine code for fast execution.	Interpreted: Typically compiled to bytecode which is then executed by a VM.	Compiled: Code is compiled directly to a self-contained native machine code executable with a runtime.
Concurrency	Provides low-level primitives like threads and mutexes, requiring manual management.	Offers threading (limited by the GIL for CPU-bound tasks) and multiprocessing libraries.	Built-in support with lightweight goroutines and channels managed by the Go runtime.
Memory Management	Manual memory management, with modern C++ heavily relying on RAII and smart pointers.	Automatic via reference counting and a cyclic garbage collector.	Automatic via a concurrent, tri-color mark and sweep garbage collector.
Standard Library	Rich library with a focus on performance (e.g., STL containers, algorithms).	Extensive "batteries-included" library for a vast array of tasks, speeding up development.	Comprehensive library designed for modern needs like networking, I/O, and JSON handling.
Programming Paradigms	Multi-paradigm: Supports procedural, object-oriented (oop), and generic programming.	Multi-paradigm: Supports procedural, object-oriented, and functional styles.	Primarily procedural and concurrent. Uses composition over inheritance (no classes).

TABLE 2.2
COMPARISON OF LANGUAGE CHARACTERISTICS IMPACTING ENERGY EFFICIENCY

Characteristics Impacting Performance & Energy Efficiency			
Characteristic	**C++**	**Python**	**Go**
Typing System	Static typing and templates enable compile-time code specialization, avoiding runtime polymorphism overhead.	Dynamic typing limits static optimizations, as type checks and memory allocation occur at runtime.	Static typing allows compiler optimizations like devirtualization and function inlining.
Execution & Compilation	Mature compilers generate highly optimized machine code, leading to shorter active CPU time and lower energy use.	Code is compiled to bytecode and run on a VM. This interpreter overhead significantly impacts performance.	Compiles to efficient native machine code but needs a runtime. The compiler performs optimizations for performance.
Concurrency Model	Low-level primitives (`std::thread`) offer fine-grained control without a GIL but require manual management.	Threading is limited by the GIL for CPU-bound tasks. Multiprocessing works but has higher overhead.	Lightweight goroutines and channels allow for high concurrency with very low overhead, managed by the runtime.
Memory Management	Manual memory control (`new`/`delete`, smart pointers) and RAII provide deterministic cleanup, avoiding GC overhead.	Automatic GC on mostly heap-allocated objects increases memory footprint, GC load, and access latency.	Automatic GC with a focus on stack allocation for value types, which reduces GC pressure and improves data locality.
Standard Library	The Standard Template Library (STL) provides highly optimized, performance-focused data structures and algorithms.	Performance-critical modules are often C extensions, but the call overhead from Python remains.	Many standard library functions (e.g., networking, crypto) are highly optimized, some using assembly for critical paths.
Abstractions	Aims for "zero-cost abstractions," where high-level features are compiled away and incur no runtime overhead.	High-level abstractions and dynamic features are powerful but generally incur significant runtime overhead.	Interfaces provide abstraction with a small, well-defined runtime cost. Composition is favored over inheritance.

2.3.1. Go: Compiled Language with an Embedded Managed Runtime

Go is a language developed by Google, released in 2009, focused on concurrency. It has a runtime which manages the goroutines. As described by [13], Go has a Garbage Collector, which means while the program is running, there needs to be a thread checking for unused memory structures.

This language was designed for backend tasks, handling thousands of simultaneous connections, while having an easy syntax for any programmer. Some companies that use this are Uber, Docker, Twitch, previously Discord [14] and, although not mainly, Netflix.

From Table 2.1, it can be seen that Go is statically typed and compiled, which makes it have a good start as an efficient programming language. But from the Table 2.2, we can observe go has a managed runtime, which means the energy consumption will be higher than other languages that do not have this. This runtime is the section of the program in charge of running and scheduling goroutines. This is why go binaries have a bigger minimum size as the runtime has to be fitted in the binary, which is great for cross compilation, but not great for either energy efficiency or performance.

Go's scheduler performs a series of steps before starting to run the user's code. As described in [15], [16] and [17], the runtime can be divided into these steps:

1. **OS Loading**: The main function is not the actual entry point of a Go program. Rather, the starting point is an assembly level function within the runtime. You can find it in a file corresponding to your specific OS and architecture, for example, `rt0_linux_amd64.s`. This is the first function the Go's program code will have the OS execute after loading the binary, and its only responsibility is to get the environment setup for the Go runtime.

2. **Argument and Environment setup**: The runtime, after being loaded into memory, calls an internal function `runtime.args` that handles the arguments and environment. This function copies the arguments (`argc` and `argv`) and environment variables into a Go-managed memory space. This ensures that the rest of the Go program, including the main function, can access this information through standard library functions like `os.Args` and `os.Getenv`.

3. **Scheduler Initialization (M:P:G Model)**: The heart of the concurrency system in Go is the "M:G:P" model. Before any go code is executed, the scheduler must be initialized, which happens inside `runtime.schedinit`.

   - **`M0` and `G0` Creation**: The program starts with a single Operating System (OS) thread ($M0$). Every M thread has a special goroutine called `g0`, which is responsible for scheduling other runtime tasks.
   - **P Initializations**: A list of `Ps` or processors, which is a resource required to execute Go code, is created. The limit of `P` is determined by the `GOMAXPROCS`

environment variable or inside the Go code by using the `runtime.GOMAXPROCS()` function.

At this time, the scheduler limits are put in place, limiting the maximum number of OS threads to 10,000.

4. **Memory Allocator and GC initialization**: Go's runtime includes a complex memory allocator and Garbage Collector [18] and [19] .
   - **Memory Reservation**: The runtime reserves a large region of virtual memory, divided into 3 areas: `spans`, `bitmap` and `arena` where go objects are allocated on the heap.
   - **Allocator Structures**: Other structures such as the `mheap` (the global heap structure for Go), `mcentral` (a central cache for memory spans) and for each `P` a per-thread cache for allocating small objects without locking the main thread, called `mcache`.
   - **GC Pacer**: The pacer determines the optimal time to trigger a Garbage Collection cycle based on the `GOGC` environment variable. The goal of Go's collection system is to perform one as the heap doubles in size since the previous cycle.
5. **Package Initialization**: At this point, the runtime can start reading from the supplied files. It starts with importing the required dependencies and initializing package-level variables. Once all files are processed in lexical file name order, the `init()` function or functions are called in order.
6. **Creating the Main Goroutine**: The runtime doesn't call the `main.main` function directly. Instead, it creates a new goroutine to execute it. This is done using the internal `runtime.newproc` function. A new goroutine (`G`) is created, and its instruction pointer is set to the `main.main` function. This new goroutine is then placed into the local run queue of one of the available `P`s, making it runnable.
7. **Start the Scheduler**: Finally, the `runtime.mstart` is called on the main thread, that enters into the scheduling loop. From this point on, the Go program is running, and the scheduler is fully operational, managing the execution of all goroutines on the available threads.

2.3.2. Python: Interpreted Programming Language

Python is the most popular language according to the TIOBE Index as of May 2025 and has been since October 2021. Either because of its easy to start as a simple to start with programming language or because the actual trend of Artificial Intelligence (AI) is mostly programmed with Python, its popularity has skyrocketed.

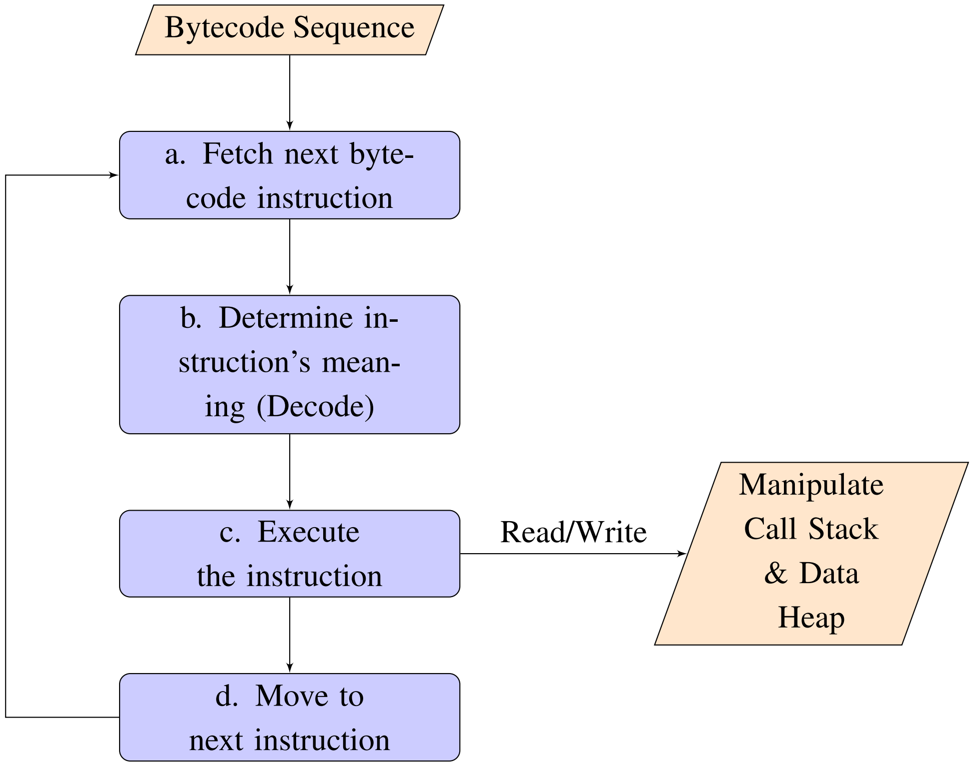


Fig. 2.4. Python's Execution loop

Python, on the contrary to most of the other popular languages is an interpreted language, which means instead of having a compiler turn the code into assembly and then into binary, it has an interpreter that runs the bytecode instructions written by the programmer one by one.

As [20] puts it: "Garbage collection also can have a significant impact on both execution time and memory usage, and can be fine-tuned to obtain better performance"

"Compiling" Python code to bytecode

There are multiple steps between the Python code that the user writes and its execution, which can be divided into two phases [21] on CPython, the default python interpreter:

1. **Phase 1: The "Frontend"**

   (a) **Reading the source code**: The python interpreter reads the *.py* files that were passed as an argument when invoked.

   (b) **Lexical Analysis (Lexing & Tokenizing)**: In this step, the interpreter breaks down the code into a sequence of tokens, the smallest meaningful unit of the language's grammar.

   (c) **Parsing**: The stream of tokens from the previous step is fed into the parser, which checks if the sequence of tokens conforms to the rules that define Python's grammar [22].

(d) **Compiling to bytecode**: The Abstract Syntax Tree (AST) is traversed generating bytecode. When the compilation is done, python caches the bytecode into a folder names `__pycache__` and stores `.pyc` files, representing the bytecode version of the same named file, so that in future executions, all the previous steps can be skipped.

2. **Phase 2: The "Backend"**

(a) **Loading the bytecode** into the Python's Virtual Machine (PVM), either from the output of the compiler or from the `.pyc` file.

(b) **Python's Execution loop**: As we can see form Figure 2.4, the loop is extremely simple. It starts by fetching the next instruction, decoding that instruction, executing it and moving the pointer to the next instruction.

(c) **Stack Frame Management**: The PVM manages function execution by pushing a stack frame, containing the function's context like its variables and return address, onto the call stack upon invocation and popping it upon completion to resume the prior state.

(d) **Memory Management**:

- Reference Counting: This is the primary mechanism. It works by having all objects keep a count of how many variables or other objects refer to themselves. If this count drops to zero, the object is removed from memory and thus that section of the memory is freed.
- Cyclic Garbage Collector: As there are some cases where the reference counting can not deal with cyclic references (e.g., when object α refers to β and β refers to α), a garbage collector process also has to run periodically. This means the efficiency of the interpreter is not very high as it need extra processes to clean up memory. This GC uses a generational approach, based on the idea that most objects are short-lived, and focuses its effort on newer objects. [1]

A Concrete Example with `dis`

Listing 2.1. Python code demonstrating the `dis` module.

```
import dis

def simple_math(a):
    x = a + 10
    return x

```

[1] In some interpreters such as CPython, you can interact with the collector using the **gc** module. [23]

```
# Use the disassembler to inspect the function's bytecode
dis.dis(simple_math)
```

Let's see this in action. The `dis` module is a "disassembler" that shows you the bytecode for a piece of Python code. The script in Listing 2.1 defines a simple function and then uses `dis` to inspect it.

Listing 2.2. Bytecode output generated by the `dis.dis` function.

```
  4           0 LOAD_FAST                0 (a)
              2 LOAD_CONST               1 (10)
              4 BINARY_ADD
              6 STORE_FAST               1 (x)

  5           8 LOAD_FAST                1 (x)
             10 RETURN_VALUE
```

The output of this script, shown in Listing 2.2, reveals the low-level bytecode instructions that the Python Virtual Machine will execute.

Other Interpreters

TABLE 2.3
ALTERNATIVE PYTHON IMPLEMENTATIONS

Implementation	Description
IronPython	Python running on .NET
MicroPython	Python running on microcontrollers and in the Web browser
Stackless Python	A branch of CPython supporting microthreads
Jython	Python running on the Java Virtual Machine

As python's interpreter is almost completely independent from it's syntax and language development, there are multiple interpreter, each one with its features. One of the most popular alternatives to CPython is PyPy, a fast implementation of Python with a Just In Time Compiler (JIT) compiler. The problem with Just in Time compilers are that there might incurr into potential warmup costs, before the functions go though the compiler. This process can optimize some hot code paths (a function or a section of the code that is run multiple times). Other examples are shown out in Table 2.3

2.3.3. C++: Directly Compiled, Unmanaged Language

C++ is one of the most famous language when it comes to high performance computing applications. Based on the programming language C, released in 1978 as a high-level language at the time, compared to assembly.

As we can see from Table 2.1, there are many characteristics on why the language is one of the most used for high performance software, for example, blender or nuke. This known examples and the multiple tests performed in multiple courses during the computer science degree.

If we take into account the energy efficiency, from Table 2.2 we can observe that being a compiled language, with multiple optimizations at the compilation level, zero-cost abstractions, no runtime and direct memory management makes it one of the best low energy consumption language in theory. In this section, the different programming languages that have been chosen will be discussed, as well as why other similar languages were not.

Compilers

There are two main compilers for C++ widely used in the industry Clang++ and G++:

G++ is the C++ compiler for the GNU Compiler Collection (GCC). It is widely considered as a seasoned, reliable veteran; it's the default on most Linux distributions and has a long history of producing highly optimized code for final release builds. While its error messages have improved significantly over the years, they can sometimes be verbose and a bit cryptic, leaving you to decipher a long template expansion error.

Clang++ is the C++ compiler front-end for the LLVM project. It often feels like it was designed specifically to make a developer's life easier, excelling in two key areas: speed and diagnostics. Clang++ is famous for its remarkably fast compilation times and for error messages that are not only clear and color-coded but often suggest the exact fix, creating a much tighter and less frustrating coding loop. This focus on tooling is why it's also the engine behind many modern IDE features and static analysis tools.

2.3.4. Other languages not used

There are many more languages, but to reduce the scope of the project and have a good analysis on each of the languages to be analyzed, a reduced group had to be selected.

As a contender for a fast, high energy efficient language we could have used Rust, a recently new programming language, focused in performance and type-safety. As Rust is a compiled language and uses the same LLVM backend for compilation, a similar result is to be expected from this benchmark compared to the C++ implementation.

Other programming languages that could have been good contenders to be tested, not because of their efficiency but because of their widespread use could have been:

TABLE 2.4
LANGUAGES EXCLUDED FROM THE STUDY AND JUSTIFICATION

Language	Reason for Exclusion
Java / C#	These languages primarily execute on managed runtimes (the JVM and .NET CLR, respectively). Their common Just-In-Time (JIT) compilation model is fundamentally different from the Ahead-Of-Time (AOT) native compilation of C++ and Go. Including them would be similar to go's implementation with a specific runtime .
JavaScript / TypeScript	These language was designed for more web-centric environments, these languages run on JavaScript engines and typically use a single-threaded event loop for concurrency. This distinct execution paradigm and primary application domain fall outside the scope of this study, which focuses on general-purpose compiled languages. Some runtimes that Javascipt use are Node, Deno or bun, but all of them have to use a core, either V8 (for node and Deno) or JavaScriptCode (for bun).
Ada	While a statically typed and being Ahead-of-time compiled language, Ada is highly specialized for high-integrity, real-time, and safety-critical systems (e.g., avionics, defense). Its lower mainstream adoption and niche focus make it less representative for this study.
Zig	As a modern systems language, Zig aligns well with the technical characteristics of C++ and Go. However, it is a relatively new language that has not yet achieved the same level of industrial adoption, ecosystem maturity, or long-term stability as the selected languages. The study's focus is on established, widely-used technologies to ensure the relevance of the findings to the current software development landscape.

2.4. Previous Benchmarks

Research in this area has intensified recently, driven by the growing global imperative to improve energy efficiency. This topic is no longer a niche concern but has garnered significant interest across industrial, economic, and policy-making sectors worldwide.

One of the first papers published that sparked the flame in the energy efficiency aspect of software development is Pereira et al. [24], that analyzed the performance of twenty seven languages in ten different programming problems. The problem I found out with these kinds of tests is that the algorithms usually are quite simple and do not represent real-world applications.

As Kempen et al. [25] put it, "Despite the fact that these studies are statistical and only establish associations, they have nonetheless been broadly interpreted as establishing a causal relationship, that the choice of programming language has a direct effect on a system's energy consumption. This misinterpretation stems in part from the work's presentation, not only in ranking of languages by efficiency, but also from the specific claim that "it is almost always possible to choose the best language" when considering execution time and energy consumption [26]". Continuing with that idea, they also state that short benchmarks are not realistic of a real-life program being executed on a machine.

The most important aspect of comparing two languages when doing a benchmark, is making sure that the algorithm used to solve the task is the same between the multiple languages. Thus, the benchmarks that test the languages using libraries sometimes poison the results by adding another language without their knowledge. For instance, if the python program uses `numpy`, most of that library is implemented in C, or if we were to use `PyTorch`, that library is mostly implemented in C++-

Other studies such as [26], specifically on their Table 3, on the top languages, for `binary-trees`, `fannkuch-redux`, and `fasta`, are C, C++, Rust and fortran, exchanging positions depending on the test. As we see from this list, these are all compiled languages, which is not a surprise, as Abdulsalam et al. [27] comment on their paper comparing multiple compiler flags and other languages.

As Lion et al. [28] state in their publication "studies have found that V8 and CPython can be 8.01x and 29.50x slower on average that their C++ counterparts respectively" but they also state that, "choosing a language for your application simple because it is 'fast' is the ultimate form of premature optimization" [29]. They have also found out that with respect to a language with a runtime, it can provide a better performance and scalability. Although this might be counter-intuitive, Go's scheduler automatically maps user threads to kernel threads thus reducing the number of context switches.

Another challenge this area faces is measuring the energy consumption. Some CPUs have internal registers that can provide these measurements, like the intel RAPL, but other machines might have those registers not accessible to the user or, in case of most ARM processors, these registers do not exist. Another technique is using the measurement from an external power plug, like the TP-link HS110, used by Søndergaard et al. [30] in their measurements. The problem with this kinds of measurement devices it that the entire system is measured instead of only the CPU or the RAM and the precision it provides is much lower, being able to read only at a 1 Hz frequency.

CHAPTER 3

PROBLEM STATEMENT

3.1. Project Description

Raytracer in multiple languages, comparing the energy efficiency of each one.

3.2. Requirements

UR -> User Requirement CA -> Capacity RE -> Restriction

3.2.1. User Requirements

Capacity

TABLE 3.1
REQUIREMENT UR-CA-XX

UR-CA-XX	
Description	*Requirement's Description*
Necessity	*Low / Medium / High*
Priority	*Low / Medium / High*
Stability	*Stable / Unstable*: How easy it is for the requirement to change along the development of the project
Verifiability	*Verifiable / Non-Verifiable*

TABLE 3.2
REQUIREMENT UR-CA-01

UR-CA-01	
Description	The user must be able to run the language benchmarks on MacOS and Linux.
Necessity	High
Priority	Medium
Stability	Stable
Verifiability	Verifiable

TABLE 3.3
REQUIREMENT UR-CA-02

UR-CA-02	
Description	The user must be able to inspect every aspect of the code.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.4
REQUIREMENT UR-CA-03

UR-CA-03	
Description	The user must be able to add their own implementation of the program to be tested.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.5
REQUIREMENT UR-CA-04

UR-CA-04	
Description	The user must be able to check the outputs of the programs.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.6
REQUIREMENT UR-CA-05

UR-CA-05	
Description	The user must be able to check the enery consumption of the benchmarks per-core
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.7
REQUIREMENT UR-CA-06

UR-CA-06	
Description	The user must be able to decide the amount of cores needed via the CLI
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.8
REQUIREMENT UR-CA-07

UR-CA-07	
Description	The user must be able to specify if the banchmark should use `taskset` to fix the number of avaiable cores.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.9
REQUIREMENT UR-CA-08

UR-CA-08	
Description	The user should be able to modify the execution command to add specific performance metric and energy consumption parameters before the execution of the benchmark.
Necessity	Low
Priority	Low
Stability	Unstable
Verifiability	Verifiable

TABLE 3.10
REQUIREMENT UR-CA-09

UR-CA-09	
Description	The user must be able to check the time taken to run the different programs / benchmarks
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.11
REQUIREMENT UR-CA-10

UR-CA-10	
Description	The user must be able to change the parameters of the image generated
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.12
REQUIREMENT UR-CA-11

UR-CA-11	
Description	The user must be able to visualize the resulting images of the program execution
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.13
REQUIREMENT UR-CA-12

UR-CA-12	
Description	The user must be able to compare the resulting images of the program execution
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.14
REQUIREMENT UR-CA-13

UR-CA-13	
Description	The system must inform the user how many cores it is using though the terminal
Necessity	Medium
Priority	Low
Stability	Stable
Verifiability	Verifiable

TABLE 3.15
REQUIREMENT UR-CA-14

UR-CA-14	
Description	The user must be able to compare different executions on the same platform.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable

TABLE 3.16
REQUIREMENT UR-CA-15

UR-CA-15	
Description	The user must be able to view the remaining lines of the image to be renderes.
Necessity	Low
Priority	Low
Stability	Stable
Verifiability	Verifiable

Restriction

TABLE 3.17
REQUIREMENT UR-RE-01

UR-RE-01	
Description	The same architecture should be used for every program
Necessity	Low
Priority	Low
Stability	Unstable
Verifiability	Verifiable

TABLE 3.18
REQUIREMENT UR-RE-02

UR-RE-02	
Description	The system must have a "one-command" execution for any given language
Necessity	Medium
Priority	Low
Stability	Stable
Verifiability	Verifiable

3.3. Functional & Non Functional Requirements

3.3.1. Non Functional Requirements

TABLE 3.19
REQUIREMENT SR-NF-01

SR-NF-01	
Description	The different programs must have the same architecture
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-RE-01*, *Requirement UR-CA-14*

TABLE 3.20
REQUIREMENT SR-NF-02

SR-NF-02	
Description	The system should be open-source
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-RE-02*, *Requirement UR-CA-03*, *Requirement UR-CA-05*, *Requirement UR-CA-06*, *Requirement UR-CA-07*, *Requirement UR-CA-08*, *Requirement UR-CA-10*, *Requirement UR-CA-12*, *Requirement UR-CA-13*

TABLE 3.21
REQUIREMENT SR-NF-03

SR-NF-03	
Description	The ray-tracer should be implemented in C++
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-RE-01*

TABLE 3.22
REQUIREMENT SR-NF-04

SR-NF-04	
Description	The ray-tracer should be implemented in Python
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-RE-01*

TABLE 3.23
REQUIREMENT SR-NF-05

SR-NF-05	
Description	The ray-tracer should be implemented in Go
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-RE-01*

TABLE 3.24
REQUIREMENT SR-NF-06

SR-NF-06	
Description	The ray-tracer's output should be stored in a file
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-04*, *Requirement UR-CA-14*

TABLE 3.25
REQUIREMENT SR-NF-07

SR-NF-07	
Description	The ray-tracer's randomness should not impact the structure of the scene
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-10*, *Requirement UR-CA-14*

TABLE 3.26
REQUIREMENT SR-NF-08

SR-NF-08	
Description	The system must implement different time-measuring systems for the different platforms
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-09*, *Requirement UR-CA-14*

3.3.2. Functional Requirements

TABLE 3.27
REQUIREMENT SR-FN-01

SR-FN-01	
Description	The system must measure the energy consumption of each of the benchmarks regardless of the implementation language.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-05*, *Requirement UR-CA-09*

TABLE 3.28
REQUIREMENT SR-FN-02

SR-FN-02	
Description	The system must output the file where the energy consumption result is kept at the end of the execution.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-09*, *Requirement UR-CA-14*

TABLE 3.29
REQUIREMENT SR-FN-03

SR-FN-03	
Description	The system must output the file where the execution time is kept at the end of the execution.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-09*, *Requirement UR-CA-14*

TABLE 3.30
REQUIREMENT SR-FN-04

SR-FN-04	
Description	The system must reduce outlier results by executing the benchmarks multiple times.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-12*

TABLE 3.31
REQUIREMENT SR-FN-05

SR-FN-05	
Description	The system must show the lines that are remaining to be processed.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-15*

TABLE 3.32
REQUIREMENT SR-FN-06

SR-FN-06	
Description	The system must have a script to process MacOs Powermetrics results.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-05*

TABLE 3.33
REQUIREMENT SR-FN-07

SR-FN-07	
Description	The system must have a file from which all benchmarks can be launched.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-05*, *Requirement UR-CA-14*

TABLE 3.34
REQUIREMENT SR-FN-08

SR-FN-08	
Description	The system must have a file from which parameters for the simulation can be changed.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-05*, *Requirement UR-CA-06*, *Requirement UR-CA-14*

TABLE 3.35
REQUIREMENT SR-FN-09

SR-FN-09	
Description	The system must have a utility that checks the diference between two output images.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-04*

TABLE 3.36
REQUIREMENT SR-FN-10

SR-FN-10	
Description	The running of the benchamrks on any device must not interfere with the reading of the energy consumed by the program.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-09*

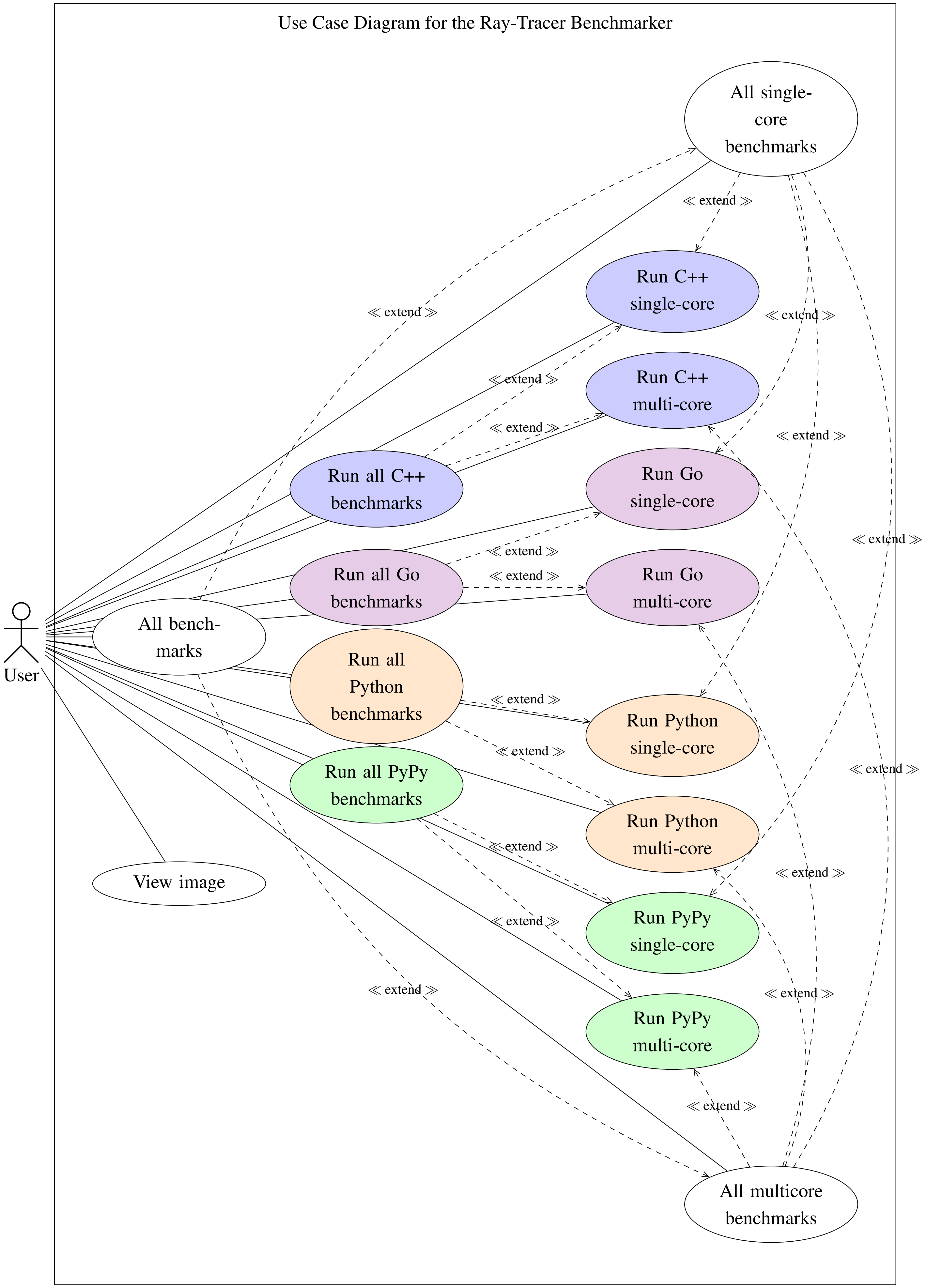
TABLE 3.37
REQUIREMENT SR-FN-11

SR-FN-11	
Description	The running of the benchamrks on any device must not interfere with the reading of the time taken to execute the program.
Necessity	High
Priority	High
Stability	Stable
Verifiability	Verifiable
Origin	*Requirement UR-CA-09*

3.4. Use Case

✓

Ray-Tracer Benchmarker



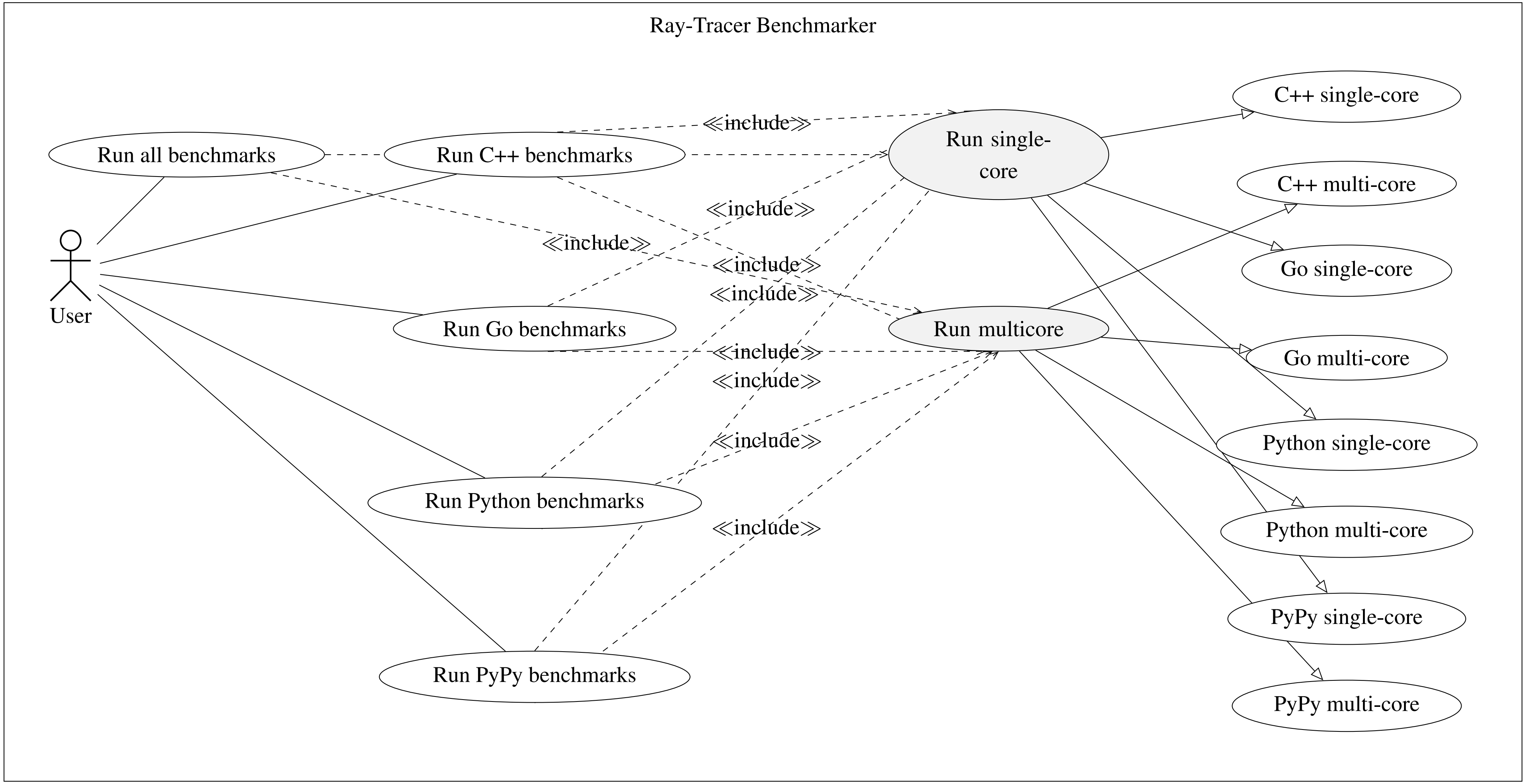


Fig. 3.2. Cleaned-up use-case diagram for the Ray-Tracer Benchmarker

3.5. Traceability

CHAPTER 4

DESIGN AND IMPLEMENTATION

Thing cosa cosa cosa TODO

4.1. General Program Design

The "30,000 foot view" of the programs is a ray-tracer, that has multiple spheres, with different materials, indexes of refraction and sizes. Each of the pixels is calculated individually, and then, when all pixels have been processed, they are outputted to a `ppm` file.

Figure 4.1 TODO: hablar algo de aqui

4.2. Scene

This first section of the program reads the input file and stores the spheres into their appropriate data-structures.

4.2.1. Sphere_data design

To ensure the consistency between programs and runs, I decided to create a file that would specify the layout of all spheres, and include the parameters for the camera setting, position and render settings:

- **ratio <width: double> <height: double>** ⇒ Aspect ratio of the output image (width / height).

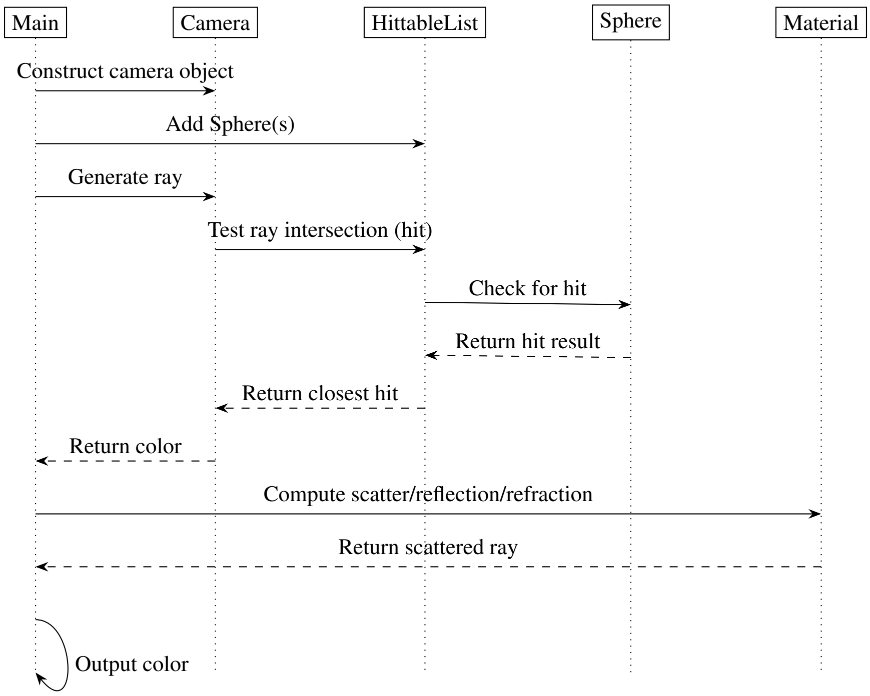


Fig. 4.1. General program data flow

- **width <int>** ⇒ The number of pixels for the width in the output image.
- **samplesPerPixel <int>** ⇒ How many times each of the pixels is processed. The higher this number is, the slower the render, but the less noise that the output image has. See Figure 4.3 and Figure 4.2 for examples of the same image with different `samplesPerPixel` values.
- **maxDepth <int>** ⇒ Specifies how many bounces a ray has to perform before getting the resulting color.
- **vfov <int>** ⇒ State the Field of Vision (FOV) of the camera.
- **lookFrom <x: double> <y: double> <z: double>** ⇒ Position of the camera in 3D space, where x is width, y is the height and z is the depth.
- **lookAt <x: double> <y: double> <z: double>** ⇒ States the relative "up" orientation of the camera.
- **vup <x: double> <y: double> <z: double>** ⇒ Vector that describes what is "up" in the scene
- **defocusAngle <double>** ⇒ This parameter represents the "aperture". A higher number will mean more objects will be in focus, and a smaller number results in a shallower dept of field.
- **focusDist <double>** ⇒ Specified the distance from camera lookfrom point to a plane where the elements are in perfect focus

4.2.2. Language Specific

To try eliminating any possible influence of libraries created in other programming languages, all programs have been created only using their own standard library. The only exception for this is OpenMP used for C++ parallelization.

C++

When using C++, the intent was to use some of of C++ modern features that would make development easier and adapted to new standards such as the use of smart pointers, `constexpr` and range-based loops. It was designed as an object-oriented program, with polymorphism though virtual functions (inside `material`, `hittable`).

The idea of making it header only was the benefits of inlining, easy to distribute and easily separable concepts and, as the compilation time is not crucial, the re-compilation of the headers every time there is a modification is not a drawback.

To perform multi-threaded operations, I chose the OpenMP library, because of its convenience of parallelizing and great performance. It has been the standard for decades, originating from the Fortran world (1960s)

Go

When choosing Go as to build the ray-tracer, as Go does not support inheritance like other oop languages, I had to use `interfaces`, which are the tools go provide for polymorphism. Interfaces are a type that defines a set of method signatures. Thus, for any struct that has the signature methods described in an interface, it can be called as an object from that type.

Listing 4.1. Go interface example.

```go
type Material interface {
	Scatter(rIn Ray, rec HitRecord) (bool, Color, Ray)
}

type Hittable interface {
	Hit(r Ray, rayT Interval, rec *HitRecord) bool
}
```

In my specific implementation, two instances of these keywords were used to denote all types of materials and all Hittable objects, that have to implement a scatter function and a hit function as described in Listing 4.1.

Python

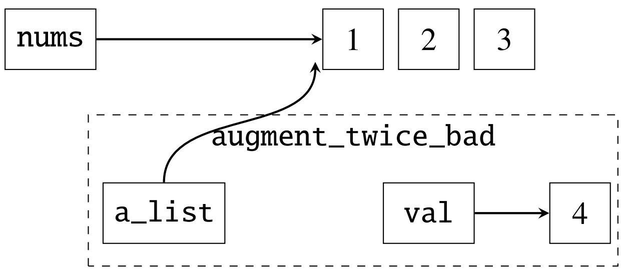
python is one of the most open languages, where there are many ways of developing the same program. There have been some problems with python's parameters in functions, wether they are passed as parameters or as references. Unlike in C++ where you can add & to symbolize the passing the parameter by reference in the function signature, or using the * in Go, in Python, at first, it seems you can not specify this behavior. But if you research into the inner-workings of pythons functions and how they work, it seems that "Python passes function arguments by assigning to them" as [31] states at PyCon 2015:

```
def augment_twice_bad(a_list, val):
    """Put val on the end of 'a_list' twice."""
    a_list = a_list + [val, val]

nums = [1, 2, 3]
augment_twice_bad(nums, 4)
print(nums)              # [1, 2, 3]
```

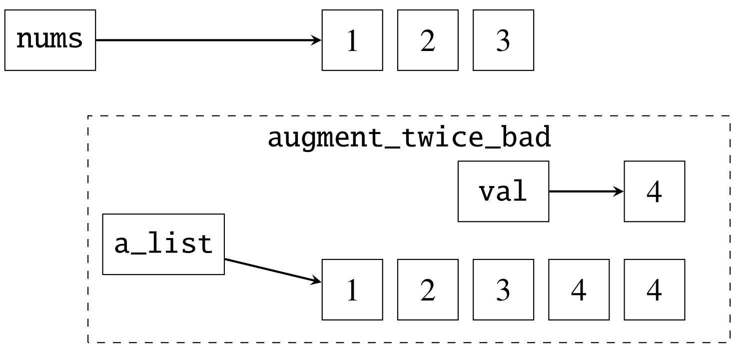
When calling the function `augment_twice_bad`, the parameters are assigned the values `nums` and 4 respectively.

`augment_twice_bad(nums, 4)`



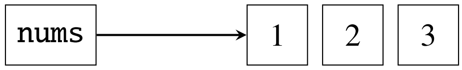
The next statement is an assignment. The expression on the right-hand side makes a new list, which is then assigned to `a_list` thus any further modification is made to the local variable only:

`a_list = a_list + [val, val]`



When the function ends, its local names are destroyed, and any values no longer referenced are reclaimed, leaving us just where we started:

print(nums)



4.3. Object

All objects in this program are spheres, even the "ground" is a sphere with a big enough radius that it seems a plane.

C++

Each of the spheres in C++ is a class called `sphere`, as all objects in this scene ar spheres. `sphere` is a derived class from `hittable`, an abstract class, such that in the case that, in the future the program is modified to have more objects, it is easily implemented.

Listing 4.2. Sphere Class for C++

```
class sphere : public hittable {
  public:
    sphere(point3 const & center, double radius,
           shared_ptr<material> mat)
      : center(center), radius(std::fmax(0, radius)), mat(mat) { }

    bool hit(ray const & r, interval ray_t,
             hit_record & rec) const override { ... }

  private:
    point3 center;
    double radius;
    shared_ptr<material> mat;
};
```

Go

Each of the spheres in Go is a struct, one for each of the materials implemented (Lambertian, Metal, Dielectric). Each of these types of materials implement the `Scatter` method, described in Listing 4.1.

Listing 4.3. Go materials structs.

```
// Solid color
type Lambertian struct {
        Albedo Color
}

// Fuzz: 0 for perfect mirror, higher for fuzzier reflection
```

```
type Metal struct {
        Albedo Color
        Fuzz   float64
}

// Transparent material such as water or ice
type Dielectric struct {
        RefractionIndex float64
}
```

Python

All spheres in Python are different classes that inherit from the same Material class:

Listing 4.4. Python abscract class.

```
class Material(ABC):
    \@abstractmethod
    def scatter(self, r_in: Ray, rec: 'HitRecord') -> tuple[bool,
        Color, Ray]:
        """Returns (scatter_happened, attenuation, scattered_ray)"""
```

4.4. Renderer

The main loop of the program is processing the object read from the file, added to the scene. This loop has two version in each of the programs designed:

- **Single-threaded loop**: The program only runs using one core. It has a double loop where it processes all the pixels in the image, one by one. This is an extremely CPU intensive process, as there are many pixels and iterations to go though each of those pixels. After all pixels are processed, they are outputted into the `output_file`
- **Multi-threaded loop**: There are many ways a multi-threaded renderer can work, even on different programming languages, different implementations have been chosen for specific reasons regarding their parallelization implementations. But in general, each of the pixels is processed and then they are all joined into an array / list that is outputted to a file.

4.4.1. Multiprocessing

As previously stated, each of the programming languages, not only uses a different approach into how they have been parallelized, but even the algorithm had to be changed, as the implementation of python's interpreters makes the obvious parallelization perform surprisingly bad (this will be discussed in its section)

C++

To implement multiprocessing in C++, the openMP library has been used, as it allows to implement parallelism with a low-effort compared to the great results it provides.

Listing 4.5. OpenMP Pragma instruction.

```
#pragma omp parallel for schedule(dynamic, 1) default(none)      \
    shared(image, world, lines_remaining, cout_mutex, std::cout) \
    firstprivate(samples_per_pixel, max_depth, image_width,
       image_height)
      for (int pixel = 0;
           pixel < image_width * image_height;
           pixel++) {
      ...
      }
```

Dividing this `#pragma` directive into its components to better understand why each of the sections exist and its effects on parallelizing:

- **#pragma omp parallel for**: This construct merges a parallel region with a for-loop, enabling work-sharing. Specifically, a group of threads is created, and all the iterations of the for-loop are distributed among these threads.
- **schedule(dynamic, 1)**: Uses dynamic scheduling, meaning each thread grabs one job at a time. Using 1 creates some more scheduling overhead, but it ensures fine-grained balancing.
- **default(none)**: Disables all implicit data-sharing forcing the programmer to scope each variable used inside the parallel region. This helps at checking race conditions at compile time.
- **shared(image, world, lines_remaining, cout_mutex, std::cout)**: The named variables refer to a single instance in shared memory, visible to all threads:
  - image: the pixel buffer, where al threads dump the processed pixel. Threads must coordinate writes so they don't stomp on each other.
  - world: the scene description, with all the spheres.
  - lines_remaining: and atomic counter for progress reporting
  - cout_mutex + std::cout: Locking the cout_mutex before writing to std::cout to serialize console output.
- **firstprivate**$(samples\_per_pixel, max\_depth, image\_width, image\_height)$: Each thread has its own copy of these variables, with the values copied form the master thread, they are constants, read-only, although you can modify them localy, but they do not copy to other threads.

Go

To parallelize in go, its standard library provides a system called goroutines. These are "Green threadss" that are created by Go's runtime every time the keyword `go` comes before a function.

Listing 4.6. Goroutines.

```go
var wg sync.WaitGroup
waitChan := make(chan struct{}, numThreads)
lines_remaining := c.imageHeight

for pixel_idx := range c.imageHeight * c.ImageWidth {
    waitChan <- struct{}{}
    wg.Add(1)
    go func(pixel_idx int) {
        defer wg.Done()
        ...
        <-waitChan
    }(pixel_idx)
}
wg.Wait()
```

To be able to limit the number of threads that can be created, a channel with a size of `numThreads` is created and each time a new goroutine is going to be created, it tries to add a struct to the channel which, if there is an empty place, it adds the struct and allows the program to continue. But, it the channel is full, the program stops and does not allow any continuation of the program until the channel has a free spot, which is generated after the pixel is added to the resulting image.

To prevent the program from continue running before all the goroutines are finished, a Wait Group (wg) is used to prevent the main thread from continuing the main execution until all threads have finished (which is the same as the wg being empty.

Python

To parallelize in python, I had to use the `concurrent.futures` library to properly parallelize the python execution.

This library can create two types of "executors" which are:

- **ThreadPoolExecutor**: for I/O-bound tasks (uses threads)
- **ProcessPoolExecutor**: for CPU-bound tasks (uses processes)

To submit a task, you can use the following semantics, using python's list comprehension to create all the jobs:

Listing 4.7. Python submiting jobs ProcessPoolExecutor.

```
futures = [
    executor.submit(self.process_row, j, world)
    for j in range(self.image_height)
]
```

This creates a job for every pixel, running the function `self.process_row` inside the camera object, with parameters `j` and `world`.

To retrieve the results, you can simply iterate the futures object, as if it was a list of objects:

Listing 4.8. Python retieving data from Process execution pool.

```
for future in futures:
    j, row_pixels = future.result()
    processed_rows += 1
    ...
    for i in range(self.image_width):
        img.set_pixel(i, j, row_pixels[i])
```

There is an interesting aspect on why the difference between the loops in Python and the rest of the programming languages: Compiled languages iterate over every pixel, while python iterates over every line, why is this? Because a copy of the entire python environment has to be created and the overhead of copying so much information slows down the program extremely. We will see the impact of these change in the evaluation section of the report.

4.5. Output

The output of the program was initially thought to be though the standard output of the terminal, and redirecting the output to a `PPM` file, but for measuring performance concerns and stability between different programs and their implementation of interacting with the operating system, it was decided that the program would create a file and output all the data into that file.

The design of the output is a PPM file, in which the first three lines define the content, aspect and maximum values of a file.

TABLE 4.1
PPM FILE HEADER FIELDS

Field	Description
P3	Magic number (P3 = ASCII, P6 = binary)
400 225	Width and height (in pixels)
255	Maximum color value

In my specific case, I used P3, as I am outputting the Red Green Blue (RGB) values individually and a 255 maximum color value, to simplify the outputting of the resulting image.

This is an example image, of an image that this program would generate at maximum reasonable quality settings, 1920 width, 300 samples per pixel and 200 max depth.

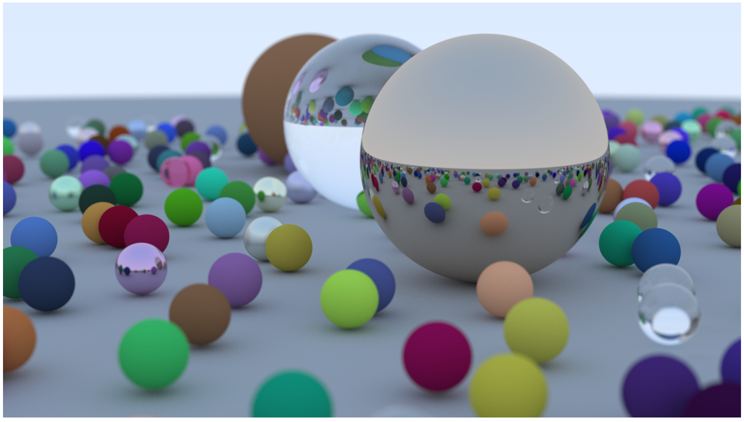


Fig. 4.2. Example of program output, 1920 width, 300 samples per pixel and 200 max depth

And this is an example of the same picture, but with the settings used to test the programs on the personal laptop and server, which takes 168x more time than Figure 4.2

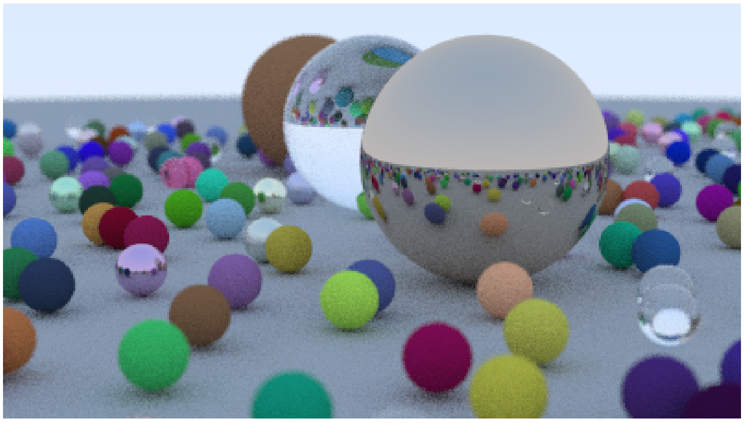


Fig. 4.3. Example of program output, 400 width, 50 samples per pixel and 50 max depth

CHAPTER 5

EVALUATION

This section reflects the results of all the testing performed with the different programming languages and architectures / types of computer.

TABLE 5.1

COMPARISON OF LANGUAGE PERFORMANCE ON DIFFERENT PLATFORMS

	C+	Go	Python	PyPy
Intel Xeon Gold 6326 x2	GCC 14.2.0	go1.24.2	Python 3.12.3	PyPy 7.3.19
MacbookPro M4 Pro	Apple Clang 17.0.0	go1.24.2	Python 3.13.3	PyPy 7.3.19
RPi 5				
Ryzen 3800x				

It should be noted that python should not be used for performance critical applications, as it is an interpreted language, and it is not designed for high performance computing. However, it is a great language for rapid prototyping and development, and it is widely used in the industry.

Go's intent is to be a fast, efficient, and easy to use language, and it is designed for multithreading and concurrency, which makes it a great choice for high performance computing, specifically being designed for backend development for web applications, and it is widely used in the industry.

5.1. Measurement Platforms

5.1.1. Many Core Platform

This platform is the most powerful combinations as well as power-hungry combinations of all of my suite of devices. This is a rack server, with two processors Intel Xeon Gold 6326, that have each 16 cores, 32 threads, contributing to a total of 32 cores, 64 threads. It also has the largest amount of RAM from this testing, with 256GB of DDR4 memory.

As it has two sockets, one per CPU chip, there has to be an intercommunication between these processors if a process spreads out to more than 32 threads, or is set by the user using the command `taskset`, fixing the cores the process can run on.

Evaluated parameters

This system was the most versatile in terms of how many tests could be done, as it has many processors, and uses Linux on x86, a great advantage to force processes to run on specific cores

The tests were done on a variety of core configuration, always setting, for core numbers less than 16, cores in the same processor.

- **1 Core**: Testing with one core, producing the baseline for the programs energy consumption and time.
- **2 Cores**: Testing with 2 cores provides the first glimpse of parallelization.
- **4 Cores**: Testing with 4 cores because many computer from some time ago had for cores.
- **8 Cores**: Testing with 8 cores gives us a great insight into how many processors in the market work, and it is half of the amount of cores inside one chip.
- **14 Cores**: Testing with 14 cores, because it is the number of cores available on the Laptop and wanted to have an execution time comparison.
- **16 Cores**: Testing with 16 cores as it is the amount or real cores on a single chip. This must be one of the most energy efficient and fast tests, if there were only one CPU.
- **28 Cores (in different CPUs, but all real cores, no logical cores)**: Testing with 28 cores, distributed with two sockets is interesting because there has to be some information sharing over some bun inside the motherboard to synchronize both CPUs. This wont be as energy efficient, but may be faster.
- **28 Cores (in the same CPU, 16 cores, 32 virtual cores)**: Testing with 28 cores, inside the same CPU, the performance should be slower as there are less real cores to tackle the work, but it has the advantage of not needing to share data to another socket.

- **32 Cores (same socket)**: Testing with 32 cores in the same socket is using all available logical threads of a system, the 16 real cores and the other 16 threads the CPU has thanks to Hyper-Threading.
- **32 Cores (only real cores)**: Testing with 32 real cores, across two sockets should be the most powerful combination for CPU intensive tasks, as all the operations should be able to be carried out without many interruptions.
- **48 Cores**: Testing with 48 cores forces us to use all real cores and some logical cores.
- **60 Cores**: Testing with 60 cores is also interesting and not 64, as this would force the machine to interrupt the program we are benchmarking to perform routine operations, such as checking for incoming connections, or logging.

Results

The results for the server are shown in the following figures and tables. The energy consumption is measured in joules, and the execution time is measured in seconds.

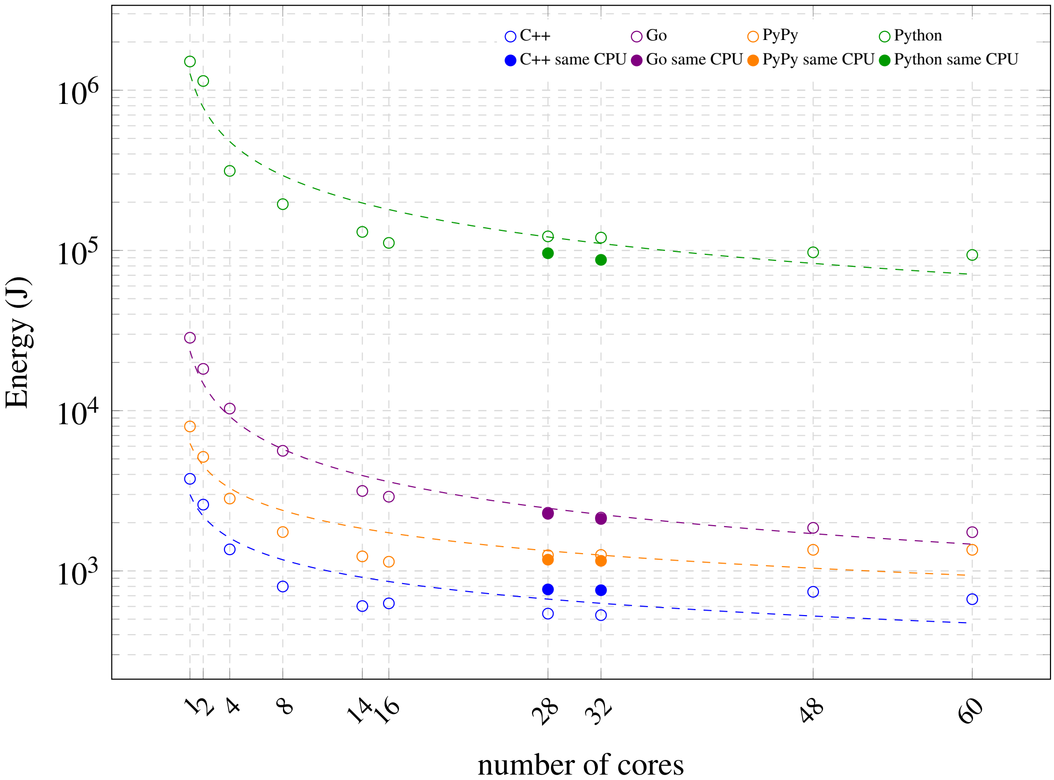


Fig. 5.1. Energy consumption of the pkg (package, chips) server in Joules for different core configurations

From Figure 5.1 we can see that the energy consumption of the server is not linear with the number of cores. It can be observed that the energy consumption decreases as the number of cores increases, but there is a point in the graph and Table 5.2 where the

Table 5.2

Energy usage (pkg) by implementation and core count

energy-pkg	C++	Go	PyPy	Python
1	3,756.26	28,522.69	7,972.65	**1,510,534.76**
2	2,591.91	18,231.97	5,147.60	1,141,490.15
4	1,362.61	10,304.27	2,828.21	313,537.85
8	799.23	5,617.27	1,747.96	194,458.98
14	603.37	3,155.30	1,232.03	130,506.94
16	627.16	2,904.52	1,140.80	111,438.01
28	**541.37**	2,306.35	1,252.93	122,384.40
28 same CPU	**766.51**	2,271.71	1,175.52	**96,049.86**
32	**529.61**	2,151.74	1,257.81	120,259.78
32 same CPU	**757.79**	2,109.37	1,155.74	**87,343.61**
48	740.57	1,856.93	1,354.03	97,331.00
60	666.04	1,744.76	1,354.03	93,718.97

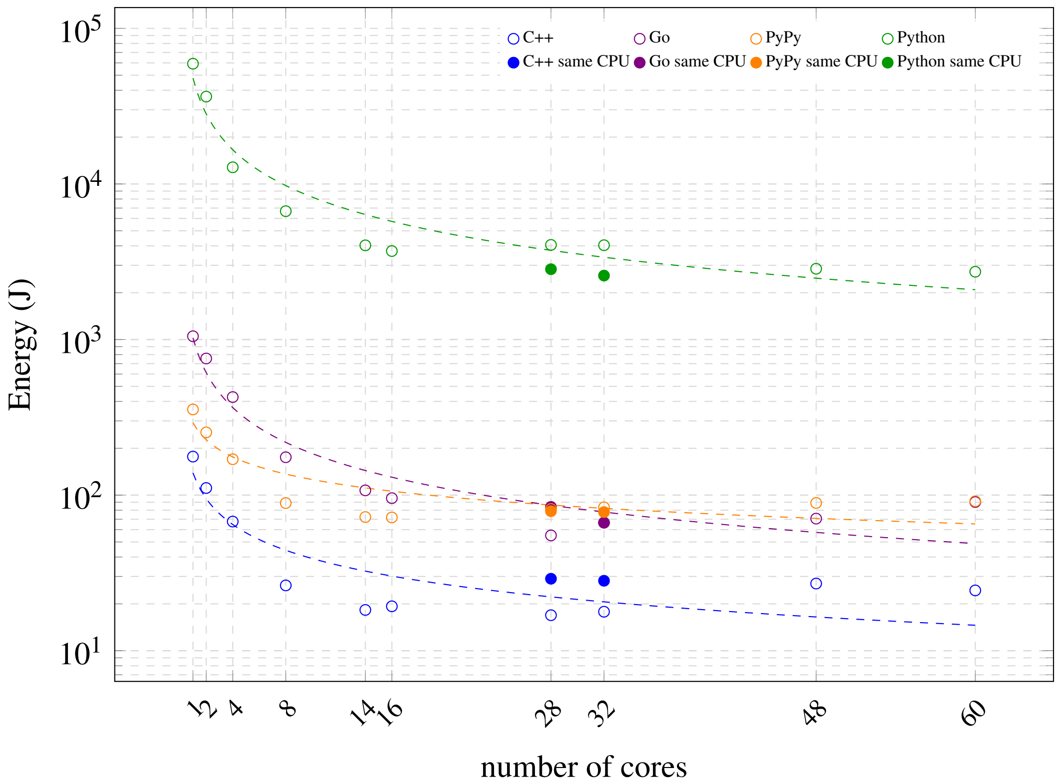


Fig. 5.2. Energy consumption of the server's RAM in Joules for different core configurations

Table 5.3

Energy usage (RAM) by implementation and core count

energy-ram	C++	Go	PyPy	Python
1	176.83	1,049.92	355.42	59,269.48
2	111.08	755.43	253.06	36,442.43
4	67.60	426.69	170.08	12,799.44
8	26.28	175.15	88.94	6,683.08
14	18.28	107.26	72.28	4,031.07
16	19.29	95.60	71.84	3,705.32
28	16.91	55.07	81.44	4,057.46
28 same CPU	29.01	83.98	78.81	2,831.52
32	17.80	66.51	83.50	4,037.98
32 same CPU	28.15	66.54	77.79	2.576.33
48	27.02	70.64	89.06	2,850.62
60	24.40	90.23	91.31	2,727.18

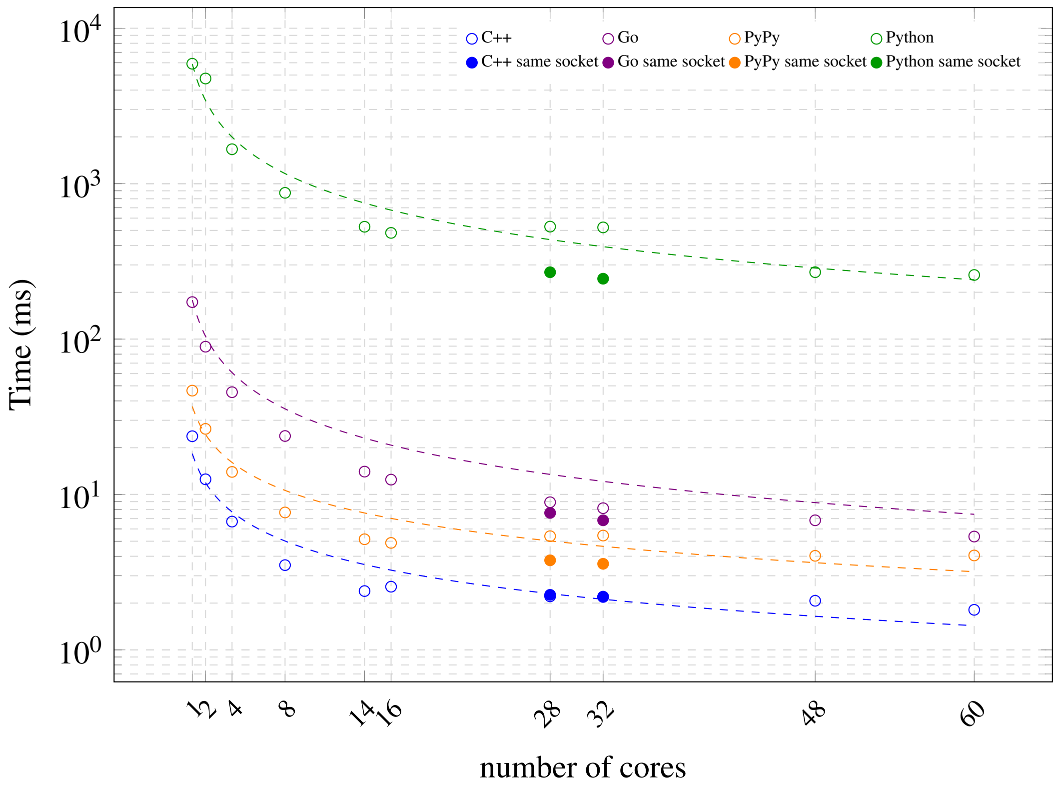


Fig. 5.3. Execution time of the server in Joules for different core configurations

TABLE 5.4

EXECUTION TIME BY IMPLEMENTATION AND CORE COUNT

time	C++	Go	PyPy	Python
1	23.70	172.95	46.58	5,913.41
2	12.52	89.32	26.43	4,749.55
4	6.69	45.51	13.97	1,665.41
8	3.51	23.78	7.66	872.89
14	2.39	14.03	5.15	528.12
16	2.55	12.46	4.88	482.30
28	2.21	8.91	5.39	529.22
28 same CPU	2.26	7.61	3.77	269.13
32	2.20	8.16	5.44	523.04
32 same CPU	2.19	6.82	3.58	244.57
48	2.07	6.82	4.03	269.41
60	1.81	5.36	4.05	258.44

energy consumption starts to increase slightly again, as well as the execution times in Figure 5.3, but not as much as the energy consumption. This is due to hyperthreading [2] in the CPUs, which allows the CPUs to run two threads per core, but this is not as efficient as running a single thread per core, as the CPUs have to share resources between the two threads.

It is obvious from the multiple graphs and tables that the C+ implementation is the most energy efficient and fastest by a significant margin, followed suprsingly by the PyPy execution of the Python code, which is faster than the Go implementation, and the Python implementation is the slowest and most energy consuming by an extremely large amount.

But, when loking close at the 28 core and 32 core tests, focusing in C++, we can see the energy consumption is lower when using cores from different CPUs rather than consuming more, as there is some energy efficiency loss when synchronizing the data between the two CPUs. But what happens in this case is that the C++ parallelization algorithm makes each of the cores have a very hard on CPU workload, resulting in a more efficient result. This results in agreement with subsection 2.2.4, where it is explained that hyperthreading is not as efficient for specific tasks.

I want to specifically talk about the 60 cores test, as it is the most interesting one. In this test, the energy consumption is lower than in the 48 cores test, as well as the execution time on the C++ implementation, but on the Go implementation, both energy consumption and execution time are higher than in the 48 cores test. This is because the Go implementation is not as efficient as the C+ implementation, and the Go runtime has

[2]Hyperthreading is enabled in this system as it it not mine and I can not dissable it to perform testing. To set the process to a fixed CPU, I used `taskset -c [cores]` ie `taskset -c 0-15,32-47` for running across multiple CPUs and `taskset -c 0-31` to force the prorgam to only run in a single CPU

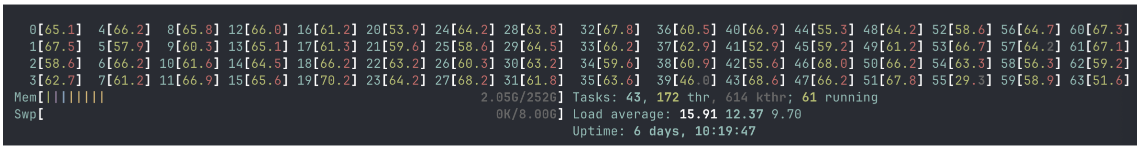


Fig. 5.4. htop showing the cores not being used at 100% when using many cores for processing in a per-pixel multi threading renderer

to manage more goroutines, which adds overhead.

Considering the 32 and 48 cores tests with the python program, the energy consumption reduces significanly when the program start using virtual cores, as the program is able to run on more cores, and the python runtime is not very demanding, being able to use these cores efficiently, as shown in Figure 5.1 and Figure 5.3 is an advantage to python with respect to itself.

It also must be noted that the cores during the 48 core benchmark were being used at 100% of their capacity, while in the 60 cores test, the cores were mostly being usead at a lower percentage, as shown in Figure 5.4. This is because the Go runtime is not able to efficiently use all the cores when there are more than 48 cores available, and it is not able to schedule the goroutines efficiently as these routines finish so fast that the Go runtime is not able to keep all the cores busy.

If we changed the implementation to a per-row renderer, on the go-side, the Go runtime would be able to use all the cores more efficiently, as it would be able to schedule the goroutines more efficiently, and the execution time would be lower, but the energy consumption would be higher, as the cores would be used at 100% of their capacity. Thus, in this case, as we will see in other sections, having a faster execution time is not always the best option in terms of energy consumption.

5.1.2. Personal Desktop

Evaluated parameters

Results

5.1.3. Personal SOTA Laptop

This lapot is said to have one of the fastest single-core performance in the market. It has a 14 core ARM processor, using the big.LITTLE architecture, with 10 high performance cores and 4 high efficiency cores. It has 48GB of RAM, which is enough to run any of the tests.

Evaluated parameters

This platform is a personal laptop, with a 14 core processor, the Apple M4 Pro, which has 10 high performance cores and 4 high efficiency cores, which is a big.LITTLE archi-

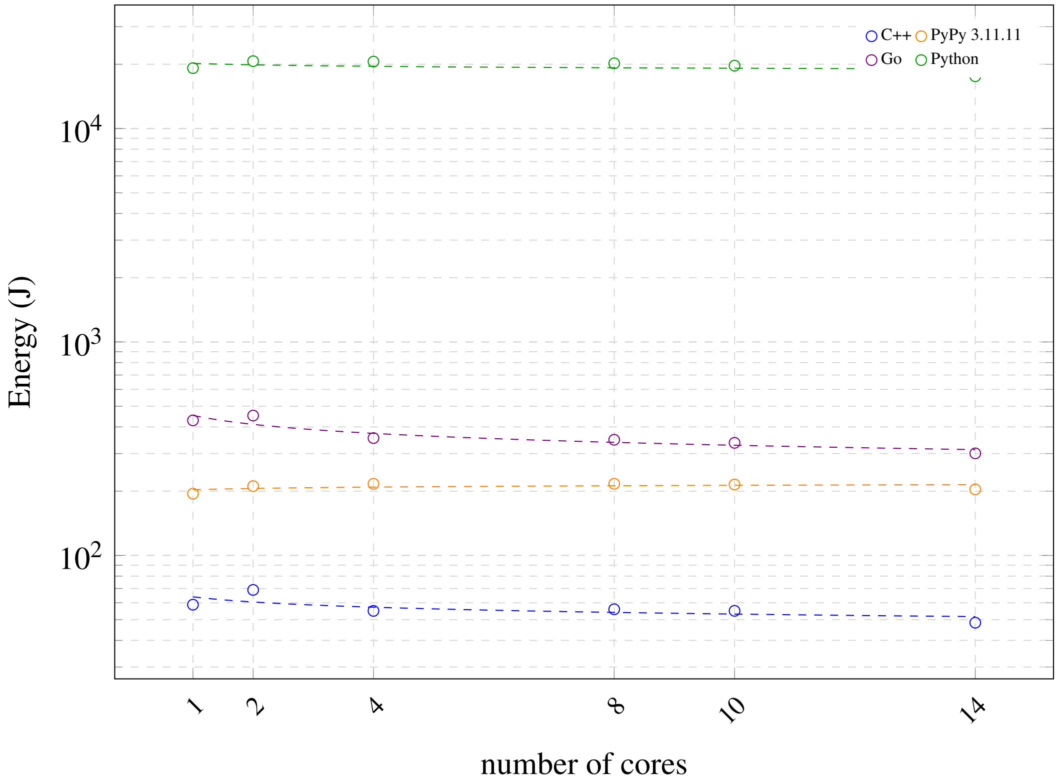


Fig. 5.5. Energy consumption of the MBP algorithm in different programming languages.

tecture. This means that the high performance cores are used for CPU intensive tasks, while the high efficiency cores are used for less demanding tasks, such as web browsing or watching videos. But as Apple does not allow the user seting the cores to be used by a specific process, like it happens in Linux, we can not test the high efficiency cores isolated from the high performance cores, as the operating system will decide for us which cores to use for each process.

Results

TABLE 5.5

POWER SUM BY IMPLEMENTATION AND CORE COUNT

Cores	C++	Go	PyPy 3.11.11	Python
1	58.79	429.23	194.69	19,183.95
2	68.93	452.12	211.21	20,680.94
4	54.97	354.64	216.53	20,580.73
8	55.94	348.14	216.60	20,198.35
10	55.00	336.88	214.96	19,701.50
14	48.39	300.66	203.92	17,567.03

5.1.4. Raspberry Pi 5

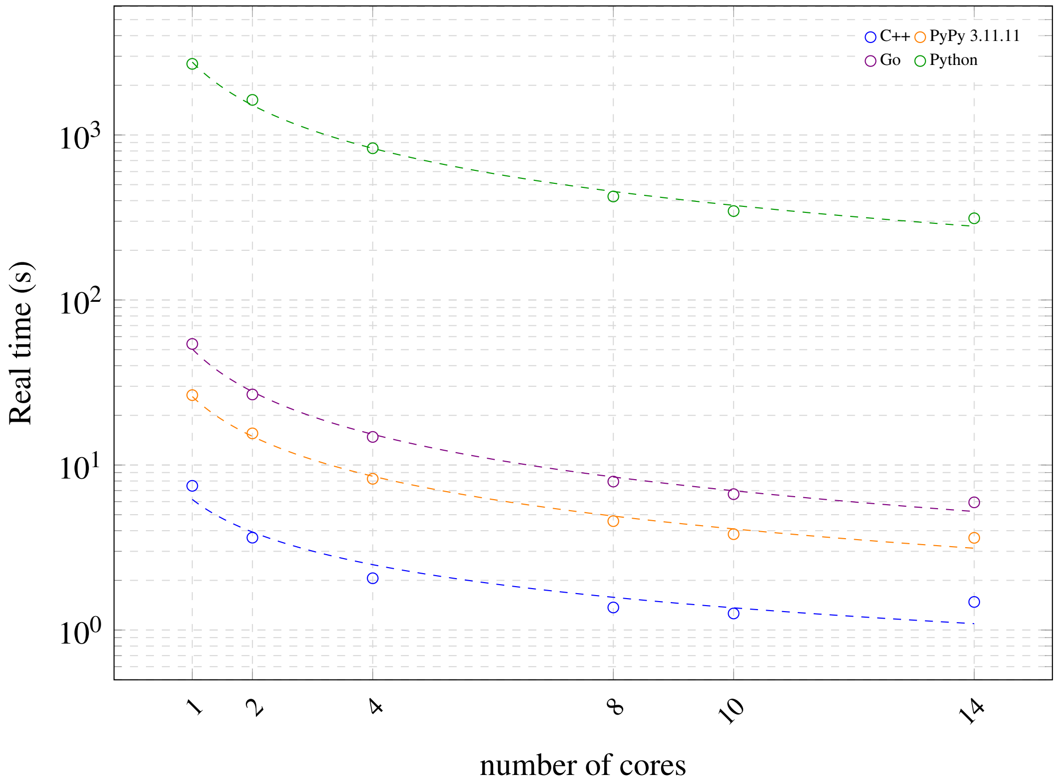


Fig. 5.6. Execution time of the MPB example in different programming languages.

TABLE 5.6

REAL EXECUTION TIME BY IMPLEMENTATION AND CORE COUNT

Cores	C++	Go	PyPy 3.11.11	Python
1	7.48	54.16	26.52	2,697.08
2	3.63	26.76	15.55	1,631.14
4	2.06	14.80	8.25	830.61
8	1.37	7.94	4.57	423.92
10	1.26	6.66	3.81	345.83
14	1.48	5.94	3.62	312.64

Evaluated parameters

Results

5.2. Comment on paralellizing different languages

In this section, I would like to make some comment on the paralelization on different languages, and why some might experience a different behaviour.

5.2.1. Go

When choosing how many "cores" the tests are using, for the Go implementation, I used the size of the `waitChan` channel. This number can be changed to be more than the total number of threads in the system, which sometimes increases the performance.

As it can be seen from Table 5.7, the Go implementation is able to use more than the total number of threads in the system, and it is able to use them efficiently, as the Go runtime is able to schedule the goroutines efficiently.

5.2.2. Python

When iterating though every pixel in Python, as the environment has to be copied for every single pixel, the cores are not being used at 100% of their capacity, specifically, while testing I saw that the cores were being used at around 5% of their capacity. Meaning the creation of too many threads is not beneficial, as the overhead of creating the threads is larger than the actual work being done by each thread. Another factor that Python, each time a task is submitted to a process, Python needs to serialize (pickle) the entire world object and other parameters, then deserialize them in the worker process, which means that, if this has to happen for every pixel, the serializing and deselializing tasks run for much longer than the actual pixel processing.

TABLE 5.7

GO GOROUTINES AND THREADS USED IN THE TESTS

Cores	Goroutines	Energy consumed (pkg)	Execution time
1	1	28,522.69	172.95
1	2	28,919.31	175.38
2	2	18,231.97	89.319
2	4	18,224.50	89.275
4	4	10,304.27	45.508
4	8	10,299.06	45.482
8	8	5,617.27	23.781
8	16	5,580.52	23.594
14	14	3,155.30	14.034
14	28	3,151.93	14.001
16	16	2,904.52	12.456
16	32	3,018.54	12.435
28	28	2,306.35	8.9061
28 Same CPU	28	2,271.71	7.6137
28	56	2,314.29	7.7914
28 Same CPU	56	2,290.85	8.8153
32	32	2,151.74	8.1632
32 Same CPU	32	2,109.37	6.8224
32	64	2,121.88	8.1017
32 Same CPU	64	2,142.21	6.8960
48	48	1,856.93	5.7187
48	96	1,848.47	5.6737
60	60	1,744.76	5.3571
60	120	1,737.80	5.3208
60	200	1,724.73	5.2556
60	250	1,719.49	5.2182

CHAPTER 6

PLANIFICATION

CHAPTER 7

SOCIOECONOMIC ENVIRONMENT

7.1. Budget

7.1.1. Human Resources

7.1.2. Material Resources

Hardware

7.1.3. Software

7.1.4. Indirect Costs

7.1.5. Total Cost

7.2. Socio-Economic Impact

CHAPTER 8

REGULATORY FRAMEWORK

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

Example of figure:

Fig. 9.1. Figure name here

Example of table:

TABLE 9.1

LOREM IPSUM

I		II		III			IV
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Source: BOE

BIBLIOGRAPHY

[1] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, "Recalibrating global data center energy-use estimates," *Science*, vol. 367, no. 6481, pp. 984–986, 2020. DOI: 10.1126/science.aba3758. eprint: https://www.science.org/doi/pdf/10.1126/science.aba3758. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aba3758.

[2] H. Ritchie, P. Rosado, and M. Roser. "Energy production and consumption. "[Online]. Available: https://ourworldindata.org/energy-production-consumption.

[3] P. Shirley, T. D. Black, and S. Hollasch. "Ray tracing in one weekend. "[Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[4] R. Muralidhar, R. Borovica-Gajic, and R. Buyya, "Energy efficient computing systems: Architectures, abstractions and modeling to techniques and standards," *CoRR*, vol. abs/2007.09976, 2020. arXiv: 2007.09976. [Online]. Available: https://arxiv.org/abs/2007.09976.

[5] A. Muc, T. Muchowski, M. Kluczyk2, and A. Szeleziński, "Analysis of the use of undervolting to reduce electricity consumption and environmental impact of computers," *Rocznik Ochrona Środowiska*, 2020.

[6] Intel. "Maximize roi and performance for demanding workloads with intel avx-512. "[Online]. Available: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-avx-512.html.

[7] E. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. Navaux, and J.-F. Méhaut, "Performance/energy trade-off in scientific computing: The case of arm big.little and intel sandy bridge," *IET Computers & Digital Techniques*, vol. 9, pp. 27–35, Dec. 2014. DOI: 10.1049/iet-cdt.2014.0074.

[8] C. Moir. "The remarkable story of arm. "[Online]. Available: https://medium.com/swlh/the-remarkable-story-of-arm-85760399c38d.

[9] W. Wolff and B. Porter, "Performance optimization on big.little architectures: A memory-latency aware approach," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20, London, United Kingdom: Association for Computing Machinery, 2020, pp. 51–61. DOI: 10.1145/3372799.3394370. [Online]. Available: https://doi.org/10.1145/3372799.3394370.

[10] A. Holdings, "Big.little technology: The future of mobile," ARM Limited, White Paper, Sep. 2013. Accessed: Jun. 7, 2025. [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf.

[11] M. Gibbs, "Design and implementation of pay for performance," *SSRN Electronic Journal*, pp. 35–36, Feb. 2012. DOI: 10.2139/ssrn.2003655.

[12] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," pp. 6–11, Jan. 2002.

[13] M. Rosecrance. "The garbage collector," Accessed: Jun. 5, 2025. [Online]. Available: https://www.youtube.com/watch?v=gPxFOMuhnUU.

[14] J. Howarth. "Why discord is switching from go to rust," Accessed: Mar. 15, 2025. [Online]. Available: https://discord.com/blog/why-discord-is-switching-from-go-to-rust.

[15] J. Espino, *Understanding the go runtime*, 2023. Accessed: Jun. 7, 2025. [Online]. Available: https://golab.io/talks/understanding-the-go-runtime.

[16] W. Kennedy. "Scheduling in go : Part ii - go scheduler," Accessed: Jun. 7, 2025. [Online]. Available: https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html.

[17] T. Liu. "Memory allocations," Accessed: Jun. 7, 2025. [Online]. Available: https://go101.org/optimizations/0.3-memory-allocations.html.

[18] SoByte. "Principle of memory allocator implementation in go language," Accessed: Jun. 8, 2025. [Online]. Available: https://www.sobyte.net/post/2021-12/golang-memory-allocator/.

[19] S. Matsiukevich. "Golang internals, part 6: Bootstrapping and memory allocator initialization," Accessed: Jun. 8, 2025. [Online]. Available: https://www.altoros.com/blog/golang-internals-part-6-bootstrapping-and-memory-allocator-initialization/.

[20] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," *SIGPLAN Not.*, vol. 40, no. 10, pp. 313–326, Oct. 2005. DOI: 10.1145/1103845.1094836. [Online]. Available: https://doi.org/10.1145/1103845.1094836.

[21] P. S. Foundation. "Cpython internal documentation," Accessed: Jun. 7, 2025. [Online]. Available: https://github.com/python/cpython/tree/main/InternalDocs.

[22] P. S. Foundation. "Full grammar specification." Part of the Python Language Reference for Python 3, Accessed: Jun. 7, 2025. [Online]. Available: https://docs.python.org/3/reference/grammar.html.

[23] P. S. Foundation. "Gc - garbage collector interface," Accessed: Jun. 7, 2025. [Online]. Available: https://docs.python.org/3/library/gc.html.

[24] R. Pereira et al., "Energy efficiency across programming languages: How do energy, time, and memory relate?" In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017, Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. DOI: 10.1145/3136014.3136031. [Online]. Available: https://doi.org/10.1145/3136014.3136031.

[25] N. van Kempen, H.-J. Kwon, D. T. Nguyen, and E. D. Berger, *It's not easy being green: On the energy efficiency of programming languages*, 2025. DOI: 10.48550/arXiv.2410.05460. arXiv: 2410.05460 [cs.PL]. [Online]. Available: https://arxiv.org/abs/2410.05460.

[26] R. Pereira et al., "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102 609, 2021. DOI: https://doi.org/10.1016/j.scico.2021.102609. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[27] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong, "Program energy efficiency: The impact of language, compiler and implementation choices," in *International Green Computing Conference*, 2014, pp. 1–6. DOI: 10.1109/IGCC.2014.7039169.

[28] D. Lion, A. Chiu, M. Stumm, and D. Yuan, "Investigating managed language runtime performance: Why JavaScript and python are 8x and 29x slower than c++, yet java and go can be faster?" In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 835–852. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/lion.

[29] N. Humrich. "Yes, python is slow, and i don't care," Accessed: Mar. 15, 2025. [Online]. Available: https://medium.com/pyslackers/yes-python-is-slow-and-i-dont-care-13763980b5a1.

[30] J. Z. Søndergaard, M. C. B. Nielsen, S. A. B. Jensen, and V. F. J. and, "Measuring energy efficiency of jit compiled programming languages with micro-benchmarks," Aalborg University, Tech. Rep., 2025.

[31] N. Batchelder, *Facts and myths about names and values*, 2015. Accessed: Jun. 16, 2025. [Online]. Available: https://nedbatchelder.com/text/names1.html.