UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

µPyCSP

Two approaches to implementing CSP-like concurrency modeling in microcontrollers.

Jon Helge Langaas Johansen

INF-3981, June 2020

*For **the boys**.*

“If a man does not have sauce, then he is lost. But the same man can get lost in the sauce.”
–Gucci Mane

“The most disastrous thing that you can ever learn is your first programming language.”
–Alan Kay

Abstract

In this thesis, we have explored the possibility of bringing CSP-like concurrency modeling to an embedded environment. With the growth of IoT and embedded software, many of the hardships that come with concurrent coding in an inherently event-driven environment become more apparent. Convoluted and complex code is often used to handle the many fallacies of concurrency. We are solving this problem by abstracting away the concurrency and providing an easy to use interface, in a growing language; Micropython. Through two separate library implementations using different underlying asynchronous architecture, harvesting the potentials and of multi-threading and coroutine based CSP-processes.

We have shown through measurements and example implementations that both implementations provide viable performance, and uncovered the different advantages and disadvantages to each approach.

Acknowledgements

First and foremost, I would like to thank my family for all the love, encouragement, and faith. You guys bring out the best in me.

I want to send a special thanks to my older brother, who is always available to help me, guide me, and inspire me to try harder and do better.

I have to express my gratitude to my supervisor John Markus Bjørndalen, for sharing this project with and for continuously pushing me in the right direction for my thesis. Thanks for all discussions on everything related and not, and all the help and encouragement.

To my classmates, thanks for five incredible years, for all the late nights, early mornings and long lunches, without you, this would not have been possible.

I have to thank my fellow students and members of
Tromsøstudentenes Dataforening, for always providing useful distractions, necessary procrastination, and an unhealthy amount of coffee.

and for everyone else, you're pretty cool too.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The recent years have brought significant growth in the development of Internet of Things (IOT) [1]. The increased interest and possibilities for what IoT can provide within a variety of fields such as medicinal, research, and business development, to younger audiences learning to develop through hobby projects. Producers have manufactured newer, better, and smaller microcontrollers to keep up with popular demand, provide more functionality, and create products with intended use varying from lightbulbs and simple thermometers to more advanced products used to monitor the environment. Input / Output (I/O) Input, such as sensor or network-data, is asynchronous in nature. This often leads the developer to create convoluted code patterns, either hogging the computing capability of the IOT device by performing the I/O call in a blocking manner or creating interrupt-based callback routines, which quickly increases the application's complexity. While this can provide feasible for some application solutions, these asynchronous code blocks and separate routines are suitable for a concurrency model.

One of these concurrency models is Communicating Sequential Processes (CSP)[2]. The CSP model has been adapted in multiple languages as a way of dividing and orchestrating concurrent application code in a structured and simple way. Concurrency is the nature of coexisting routines. Frequently, a routine is dependent on data input from another routine, such as reading from a sensor and sending the readings over a network connection. CSP provides a simple, yet a rigid model for data dependency between concurrent processes, which suits the application code for IOT projects naturally.

A widespread and growing platform for IOT is Micropython[1]. Micropython is a language-platform based upon Python[2] specifically designed for the IOT platform. Micropython, being a hardware-focused version of Python, has implemented multiple ways of treating concurrency, intended for use with I/O. With multiprocessing, multithreading, and asynchronous I/O supported, the tools to create programs relying on concurrent and asynchronous code are available. Orchestrating these mechanisms into a CSP-model-library, which hides the internal CSP logic behind abstractions, provides a more comfortable interface for developers to develop leaner applications more efficiently.

PyCSP[3] is a CSP library available for the Python programming language, which provides inter-process communication concepts known from CSP[2]. PyCSP has the intent of aiding in constructing and managing concurrent programs. PyCSP has multiple versions with various approaches to CSP-like orchestration, using different underlying mechanisms. PyCSP is, however, not available for Micropython, and while the languages are similar, Micropython has some shortcomings, creating the need for specialized implementations of PyCSP to work properly.

1.1 Thesis Statement

The multiple PyCSP approaches to implementing the CSP-model yields different potential candidates for an ideal approach to a concurrency library for a Micropython-based microcontroller. Investigating two of these approaches, one using multi-threading, and another using coroutine-based mechanisms [4] will provide useful context, to further determine the advantages and disadvantages of each of the developed modules. This context will be gained through experiments focusing on performance and maintaining the ease-of-use given by PyCSP. Summarizing this leads to the thesis statement:

This thesis aims to create an efficient concurrency library, with a familiar and easy to use interface provided by a version of the PyCSP library, aimed for use in a microcontroller using the Micropython environment. Determining which PyCSP version yielding the best result is done through the implementation of two different libraries and approaches, showing and measuring of the advantages and disadvantages of both.

1. https://micropython.org/
2. https://www.python.org/

1.2 Thesis Contribution

The thesis outcome is two Micropython libraries providing the PyCSP interface, with different architectures underneath. These are considered ports of their corresponding Python PyCSP modules. The two modules which will enlighten each approach's distinct advantages and disadvantages and provide useful insight as to which has better grounds for further development. The modules will do so by implementing benchmarks to document the performance.

1.3 Organization

Chapter 2 - Related work contains a summary of researched work related to the topic of the thesis.

Chapter 3 - Technical background summarizes the technical and theoretical knowledge required to make decisions regarding design and implementation.

Chapter 4 - Design outlines the design choices made for the implementation.

Chapter 5 - Implementation describes the implementations specific details of the contributions.

Chapter 6 - Experiments details the design and intent of experiments used to evaluate performance of the contributions.

Chapter 7 - Results present the configurations and results of the experiments conducted.

Chapter 8 - Evalutation explains the context, implications and reflections of the results of experiments conducted.

Chapter 9 - Discussion brings forth further implications and aspects not provided by evalutation of results alone.

Chapter 10 - Future work presents possible future directions to go in for further development of the contributions.

Chapter 11 - Conclusions summarizes the thesis, its findings and presents concluding remarks.

2

Related works

In this section, a summary and comparison of related work and the thesis are made. With focus reasonings, potential problems, and differences in the process of solving similar problems.

2.1 Portable CSP Based Design for Embedded Multi-Core Systems

Sputh et al.[5] presents an effort to provide CSP capabilities to an embedded system, with similar intentions as and reasoning as this thesis, focusing on reducing the complexity of writing concurrent code, as it can quickly become convoluted and complicated to control all the side-effect of concurrency. The system described attempts doing so through both software and hardware implementations of the CSP-capabilities, in distinction from the software-only approach intended for this project.

An experience presented is the ease of porting onto a different underlying architecture, once the levels of abstractions provided by CSP-like modeling were implemented. They do this by implementing the software approach first, and then hardware. Similar to this project, they use the same interface, with different underlying architectures underneath the CSP-abstractions, to provide multiple versions of CSP-capabilities, and compare them.

A challenge presented was finding a suitable environment with all the necessary dependencies available for the target system. The embedded OS, which was said to be POSIX compliant, only provided partial support for the POSIX pthread Application programming interface (API). As a part of the project, this API had to be extended to suit the design requirements.

2.2 Medusa: Managing Concurrency and Communication in Embedded Systems

Barr et al. [6] describe challenges with the use of concurrency in embedded systems being very suitable, due to the event-driven nature of the embedded system, but often ending up as "hand-coded interrupt routines". The Medusa project is presented as an extension to the Owl, embedded Python system [7]. The Medusa language provides a concurrency environment for embedded systems. Medusa is based on Python due to its ease-of-use, similar to this thesis. However, a different approach when it comes to concurrency modeling and orchestration. Medusa takes from Erlang[1] and uses the Actor Model[8] for message passing instead of CSP.

Medusa implements threads, allocated on the heap instead of the more conventional stack allocation, leaving the threads with no stack. As a result of this, the threads are light-weight, and it gives the ability to use large amounts of threads in a minimal environment as embedded devices often are. A side effect of the light-weight threads is a small overhead for spawning them. Context switches also see high performance, increasing efficiency even further.

1. https://www.erlang.org/

3

Technical background

In this section, technology relevant to the project is described to enhance the further reading experiment. The information presented should not be interpreted as a full description or documentation; it merely serves the purpose of explaining concepts and some of the researched elements used in the project.

3.1 Concurrency

The term concurrency in computer science refers to procedures able to be executed in an unspecified order with a deterministic result. Concurrent procedures can be executed simultaneously, not affecting each other's execution. Concurrency is considered the way of structuring procedures in a manner that satisfies the criteria as mentioned above.

Concurrency does not inherently mean that the execution of the procedure is done at the same point of time as another procedure. This is more specifically parallelism, a way of executing concurrent procedures.

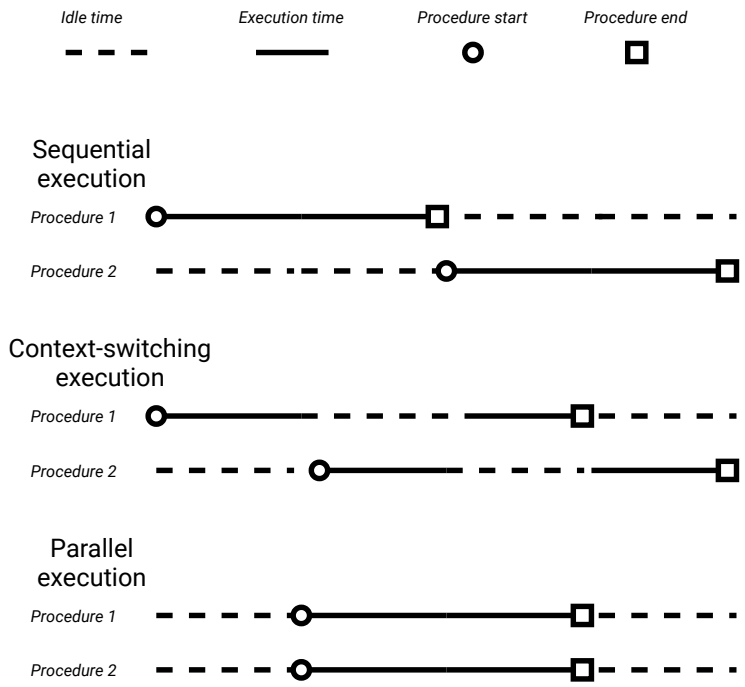

Figure 3.1: Procedures in sequential, concurrent and parallel execution

Concurrent procedures can also be executed in a context-switching manner, where two concurrent procedures share a resource, such as execution time. Figure 3.1 displays an example of the distribution of execution time can be when executing procedures in sequential, parallel, and context-switching fashion.

3.2 Python & Asynchronous code

Python is a high-level, object-oriented, dynamically, and strongly typed language. With a focus on simple, easy to learn syntax and code readability, developers of many skill levels have taken a liking to the language.

Python is an interpreted language, meaning the code is evaluated in execution. The Python interpreter uses a global lock, named the Global Interpreter Lock (GIL) to prevent multiple Python threads from modifying the same object, potentially causing race conditions.

There are multiple versions of Python, and the most commonly used is *CPython*, named after the language implemented in. If not specified, one can assume

mentions of *Python* is referring to this version.

Python has a broad standard library[1], with modules for a wide range of application types. External modules can be installed from the Python Package Index[2], providing even further modularity and support for extensive functionality.

3.2.1 Functions in Python

Functions in Python are blocks of code defined using the `def` keyword. By default, functions are considered *void* unless a `return` keyword specifies other return values. Void functions return `None` objects.

Listing 3.1: Function, definition and usage

```
# Definiton
def hello_world():
    print("Hello, world!")

# Usage
hello_world()

# Output is "Hello, world!"
```

Functions in Python are *first-class objects*, allowing for extended usage, such as storing function references in variables and lists, passing them as arguments, and returning values.

Objects in Python can also be used similarly to functions. By implementing an object's `__call__` method, the object is callable, like any other regular function.

Listing 3.2: Object as a function, definition and usage

```
# Definition
class object_as_function:
    def __init__(self):
        pass
    def __call__(self):
        print("Good morning, world!")

# Usage
```

1. https://docs.python.org/3/library/
2. https://pypi.org/

```
hello_world = object_as_function()
hello_world()

# Output is "Good morning, world!"
```

Generator Functions

Generators in Python are re-entrant functions. The generator function is defined in the same manner as a regular function, but at a point of execution, it *yields* a value instead of returning.

Calling a generator directly, as a function, will not start the generator, but it will return the *generator object*. Generators are *iterable*, to receive values from the generator object, the `__next__` method is called on the generator. This is done through the builtin `__next__` method. The generator will then start execution until it reaches a `yield` or `return` keyword. If a generator reaches a `return` keyword, it will throw a `StopIteration` exception, signaling the exhaustion of a generator.

The most common usage of generators is iterative tasks. This is because the usage of the generator is considered to be *on-demand*. It does not allocate memory space for all the values potentially yielded and used, only that which is yielded.

Listing 3.3: Generator, definition and usage

```
# Definition
def generator():
    // Simple generation of values from 0 to 5.
    i = 0
    while(i <= 5):
        yield i
        i++
    return // Return as the generator is exhausted

# Usage
counter = Generator()
for i in counter:
    print(i, end=", ")

# Output is 0, 1, 2, 3, 4, 5
```

Coroutine Functions

Coroutines in Python are functions defined with `async def` syntax, opposed to the standard `def` syntax. When called with standard syntax, these return the coroutine object. Coroutines introduce an essential functionality that regular functions do not provide. The ability to use asynchronous syntax such as: `await` `yield`, `async with` and `async for` within the code block.

There are a few ways of executing coroutines. The simplest method is awaiting the coroutine, which will halt execution until the code in the coroutine has finished executing. Awaiting a coroutine within a implicitly turns the calling parent-function into a coroutine itself. By definition, the top-level function has to execute these coroutines differently than using the `await` keyword to avoid becoming a coroutine object itself. Executing coroutines from a regular function without turning it into a coroutine is done using asyncio's *(described in the next section)* provided functionality.

Coroutine functions form the basis required for writing asynchronous code in Python. They leverage the concept of shared execution through suspension to allow other code blocks to execute, effectively creating asynchronous code through context switching.

Listing 3.4: Coroutine, definition and usage

```
# Definiton
async def hello_world():
    print("Hello, world!")

# Usage
import asyncio
asyncio.run(hello_world())

# Output is "Hello, world!"
```

3.2.2 Asynchronous Code in Python

Asyncio [9] is a module as part of the Python standard library, which provides an API to aid in the usage of coroutine-based asynchronous code. It was introduced in Python version 3.4 and has seen multiple changes in newer versions of Python.

Asyncio the concept of an *event loop*, which orchestrates the scheduling and rescheduling of events. The event loop uses internal queues for active and inactive events. The active queue is First In First Out (FIFO)-based, while the

inactive queue contains events scheduled with a delay, such as a given amount of time. For each iteration, the event loops handle internal queue management, such as checking for tasks on a timed sleep or timed out tasks on the inactive queue, then recover and run the next task in the active queue. The event loop provides interfaces for synchronous top-level functions to wait until all the scheduled tasks reach completion, preserving synchronicity.

The events used for scheduling in asyncio are separated into three main types of objects. These are *coroutines (described in the previous section)*, *tasks*, and *futures*. The commonality between these is that all fall under the category of awaitable objects.

Tasks are wrapper objects used to schedule coroutines in the event loop. Tasks are created through the asyncio interface, and immediately gives the event loop a notion of the task and its scheduling.

Futures are objects that represent an eventual result. A future does not have an executable code block attached to it as tasks or coroutines do. Futures simply hold the promise that they will be resolved by another operation, enabling the owner of the future to await this result. An awaiting parent routine can continue once the future has resolved with a result. Notably, futures are a part of the low-level asyncio API, and direct usage is usually avoided [10].

3.3 Multi-threading In Python

In Python, multi-threading is available through the two built-in libraries `threading`[3] and `_thread` [4]. The `threading` library provides a convenient and straightforward API to create and manage thread lifetimes. The *_thread* library supplies low-level API for functions used internally in the *threading* library's Thread objects to create and identify threads, as well as common mutexes such as barriers and locks.

3.3.1 Threads

Threads require a target *task* upon creation, which is done either through customizing a class inheriting from the `Thread` class with a new `run` method, or passing a function reference through the `target` argument in the constructor

3. https://docs.python.org/3/library/threading.html
4. https://docs.python.org/3/library/_thread.html

of a `Thread object`.

An initialized thread is considered *not alive*. The thread will first start the execution of the job when invoking the Thread objects' `start` method, which will run the thread concurrently. The `thread` object also has a method `join`, which halts the callers execution until a thread has finished its execution.

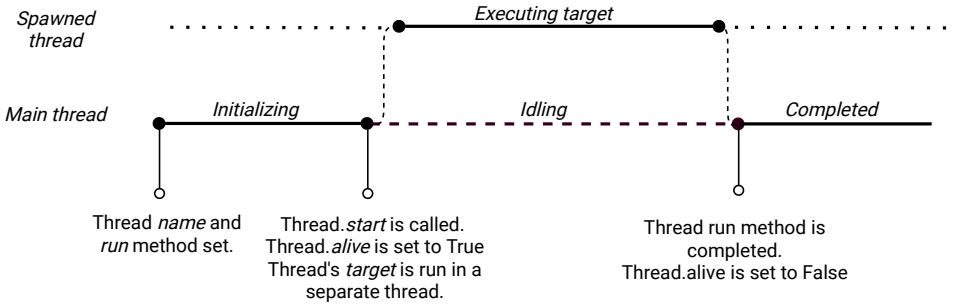

Figure 3.2: A timeline of a Python thread's lifetime states

Since threads operate within the same memory space [11], both starting and stopping threads are quick compared to, for example, subprocesses. This is due to the memory context remaining intact and not being switched. Shared memory, however, can easily lead to both deadlocks as well as race-conditions if not appropriately handled.

3.3.2 Threads And The Global Interpreter Lock

As mentioned in section 3.2, the GIL prevents Python threads from accessing and modifying the same objects. Doing so can disturb an object's reference to value and lead to unintended race conditions. The GIL, while being helpful, can become a severe bottleneck. By removing the possibility of interpreting and evaluating Python code at the same instance of time, this effectively disables the possibility of parallelizing the execution of a thread's target. A possibility of avoiding this issue is releasing the global interpreter lock by utilizing code not written in Python. A common use case for this is within performance-intensive code. Libraries such as, for example, Numpy [5], which focus on mathematic operations, which often can be parallelized, use implementations in C to increase the performance of their interfaced calculations. Another use case for this is I/O operations, which are not handled by Python code, thus avoiding the GIL.

5. https://numpy.org/

3.3.3 Reentrant Locks

Re-enterant Lock (RLock), are lock-like objects available through the threading library. The RLock object enables a thread to acquire a lock multiple times. After a thread initially acquires the RLock, the same thread will be available to acquire the lock again. For the lock to be fully released, it has to be released the same number of times as it is acquired.

The RLock is useful when working with threads and functionality with multiple critical sections protected by the lock. A regular lock would already be acquired by the thread, and cause a deadlock if the thread attempted to acquire it. Using Rlock, The thread will be able to access numerous of these sections with the same lock, with the RLock internal logic keeping track of which *depth* the thread is *acquiring* the lock.

3.4 Micropython

Micropython[6] is a version of Python targeting micro-controllers. Micropython aims to preserve the easy-to-use familiar Python language, with adjustments to better suit micro-controllers. The Micropython project is open-sourced and has many contributors. Micropython implements the syntax of Python 3.4, with few exceptions, and can run on devices with as little as 16K memory. An out-of-the-box read–eval–print loop (REPL), like the one found in Python, is available and provides the familiar functionality of either direct usage or file input.

One of the most noticeable things that separate it from Python is the high focus on modularity. Python features a broad standard library with all-you-need included, where Micropythons philosophy is more oriented towards configure-what-you-need. This "opt-in" approach reduces the size of the Micropython environment since the sheer size of the Python environment can be impracticable to use on a micro-controller platform. The result of this modularity in practice is that the Micropython environment has very few standard-libraries, and many of the Python standard libraries are instead moved to the micropython equivalent of the Python Package Index, called micropython-lib. These are available through the Micropython version of pip, upip. Some of the standard libraries also have limited support, as some functionality is only partially implemented or not available. The Micropython documentation attempts to keep track of known incompatibilities.

6. https://micropython.org/

Micropython does provide interfaces for some new extra libraries, such as "machine" for hardware interfacing, "network" for network configuration, and several other more "micro-controllers-related" libraries that have no place within Python, and in unique to Micropython. The interface concept provides a description of how each port of micropython should be implemented to provide the corresponding functionality to the intended target hardware for the port.

Micropython is updated regularly with new functionality and bug fixes.

3.4.1 Asyncio For Micropython

Asyncio for Micropython is not a part of the standard library [7]. There are, however, several packages in micropython-lib referencing asyncio. Throughout the Micropython community, it seems that the packages named uasyncio are the "official" micropython asyncio package. The six modules namespaced under uasyncio are kept separate to preserve the configurable modularity of micropython, each containing portions of the asyncio functionality.

The uasyncio module features implementations for a subset of the essential asyncio features such as an event-loop, scheduling of coroutines, and execution mechanisms.

One of the substantial differences between asyncio and uasyncio is the lack of future objects. The awaitable objects available are coroutines and a stubbed version of tasks.

3.4.2 Threading in Micropython

Micropython has support for multi-threading. While the high-level threading library threading is not available, the low-level _thread module is included in the standard library. The _thread module is described as highly experimental [12] and relies entirely on the Python documentation.

One of the differences between the Micropython and Python version of _thread is that spawning new threads in Micropython has a configurable option of not using a GIL.

7. Micropython version 1.12 has received a completely rewritten version of Asyncio, now included in the standard library. This version of Asyncio is called v3.0.0. This topic will be revisited in chapter 10, Future Work

3.5 Communicating sequential processes

Communicating sequential processes [2] is a concurrent programming language that leverages specific commands to communicate between processes [8] safely. While being a simple language, it describes a model where a process uses communication through input and output commands, which helps to solve issues regarding shared state between processes. A process has a direct reference to another process in its input or output argument, which defines a correspondence between the processes.

Directly referencing an input and output from a process creates a limitation for the input command. An input could be expected to come from multiple sources. The solution to this is using a separate port, a communication channel that acts as an intermediary to aid with indirect communication. Channels have been adapted by many languages which have taken inspiration by CSP when approaching concurrency [13][14]. The notion of multiple inputs channels is avoided in the original CSP language paper for simplicity but has surfaced is considered a primary feature of CSP modeled concurrency.

An addition of a buffered channel enables the ability to communicate asynchronously, as the processes which interact with a channel can utilize it as a mailbox for output. Similarly to a mailbox, one process can deliver state onto the channel and move on, while another process can pick up communication when it is ready.

3.6 PyCSP

PyCSP[3] is a library that adapts the CSP model in the Python language. There are multiple implementations of CSP, which vary in which underlying asynchronous architecture is used to provide a simple interface for running concurrent programs. The library uses concepts described in the original CSP[3] model, such as channels and processes, but has seasoned and uses new adaptions of the concepts. The goal is eased usage and increased performance when running asynchronous code, through concurrent modeling.

Channels The adaption of channels stems directly from the original CSP model. Channels' primary operations are read and write, which correspond to input and output from the CSP model.

8. Processes in relation to CSP describe a block of code or procedure that does not refer to the typical UNIX process

Processes The processes in PyCSP are the separate code blocks that leverage communication over channels. The implementation of these processes is varied, using different underlying asynchronous paradigms to execute. A decorator is available to turn a function into a PyCSP-process efficiently.

An important distinguishment is that the processes used internally in PyCSP do not correspond to OS processes, similarly to the CSP process.

Channel Poison Mechanism A mechanism implemented in PyCSP called poison is used to shut down the network of channels and processes. Poisoning a channel propagates poison efficiently to all the processes that use the channel, which poisons all the channels the poisoned process knows before it exits the process. Poisoning a process causes it to exit with an exception.

Being able to propagate poison throughout a CSP network is an effective way of shutting down processes. An example is a network of processes, where one of the processes is responsible for storing results. Once the processes have received enough results and decide to terminate, it can poison the network and subsequently shut down all the other processes. The other processes do not have to consider how many results they are producing before terminating and can contain pure producer code logic.

This poison mechanism is one of the primary reasons why the process requires a distinguished interface from the underlying architecture of a PyCSP version.

Alternatives, Guards & Selecting Alternatives is a concept which provides flexibility for a process performing operations. An alternative is used to *select* between multiple channels which one to operate. In combination with a Guard, which intent is to set some protective limitation as to the readiness of the channel, this can be used to exclusively select a channel that is ready to be selected.

3.7 aPyCSP, Asyncio-based CSP

aPyCSP[4] is an implementation of PyCSP utilizing the asyncio module's scheduling and asynchronous model as its basis. Implementing CSP-processes as coroutines has a few advantages. Due to the shared memory and minimal cost of context switching, there is a low cost of spawning more coroutines. A more substantial amount of CSP-processes can be spawned as a result of this.

An initial implementation of aPyCSP used locks and conditions as a result of being intended to be mimicking a threaded version, but instead using asyncio [3]. A second implementation took advantage of the cooperative multitasking that comes naturally from coroutines. The cooperative multitasking ensures that only one "process" is active at the same time, and the yielding of a coroutine is done at safe stages of execution, eliminating the need for primitives such as locks to provide thread-safety.

The experiments related to these versions show that the lock-free implementation has better performance and shorter and less convoluted code.

3.8 PyCSP - Lockversion

PyCSP - Lockversion is a version of PyCSP which took inspiration from the aPyCSP version. Instead of coroutines and asyncio, the implementation of processes is done by using threads. While this is not a new approach in PyCSP, the observations made during the development of aPyCSP resulted in a new version more alike the aPyCSP.

Using threads requires thoughts on how the state is protected, and a mechanism to help synchronize the threads is used. With inspiration from aPyCSP, a future object, which can be a wait-and-notify mechanism, is implemented using a global shared reentering lock.

This multi-threaded approach performs better than the original PyCSP version using threads.

4 Design

During the course of the development of the project, certain design choices regarding the have been made. This section is intended to describe these choices and the reasoning behind them.

4.1 Inherited Module Design

The two modules developed in this project, inherit their design from the "original" PyCSP modules. The released publications regarding PyCSP describe interface design and the internal logic of the modules.

A decision made to fork the existing PyCSP projects rather than start the library code anew. The intent behind the project is not to reinvent the wheel, but rather port the original code. By performing as little changes possible, a strong link between the original and the micropython variant preserved. This link not only ensures a better understanding of both original and ported versions but opens the possibility of maintaining both source codes with more or less the same code-changes.

The modules the design is inherited from are some of the leanest PyCSP versions in terms of code size. aPyCSP started its development from the 0.3.0 version of PyCSP, because of the smallest and less complex code [4]. The PyCSP lock-version bases its design on the aPyCSP version.

Less intrusive changes, such as moving files into separate submodules to maintain tidiness of the library is not considered altering the design of the project, but rather a part of the project architecture, described in the following section.

4.2 Platforms

The project's target platform is the Micropython environment run on microcontrollers. As indicated in section 3.4, there are different ports of Micropython adapted to different microcontrollers. The variety of versions of Micropython means that there are in all likeliness that there are some underlying differences between the ports. It is essential to keep in mind that the modules should operate on a microcontroller with specific ports, and not the UNIX version of Micropython. The intent is to create libraries suitable for all micropython-based controllers, and developing the modules as general for Micropython as possible.

To approach the project from the right angle, a micropython-based controller is used both for testing and experiments. The device chosen for development and testing is the Pycom FiPy[1]. Pycom, the manufacturer of the FiPY, use Micropython for most of their product series, which means adapting to this device makes the module available on their portfolio of microcontrollers. They have an open-source fork of Micropython [2], which is avidly maintained. The microcontroller itself is based on ESP32[3], a commonly used microcontroller that features a dual-core processor with clock-frequency up to 240MHz, Wi-Fi, and Bluetooth connectivity.

4.3 Project Architecture

With respect to the above-mentioned platform design choice, there are a few matters regarding the structure of the project that needs to be considered. Figure 4.1 gives an overview of the flow of dependencies and imports.

1. https://pycom.io/product/fipy/
2. https://github.com/pycom/pycom-micropython-sigfox
3. https://www.espressif.com/en/products/socs/esp32/overview

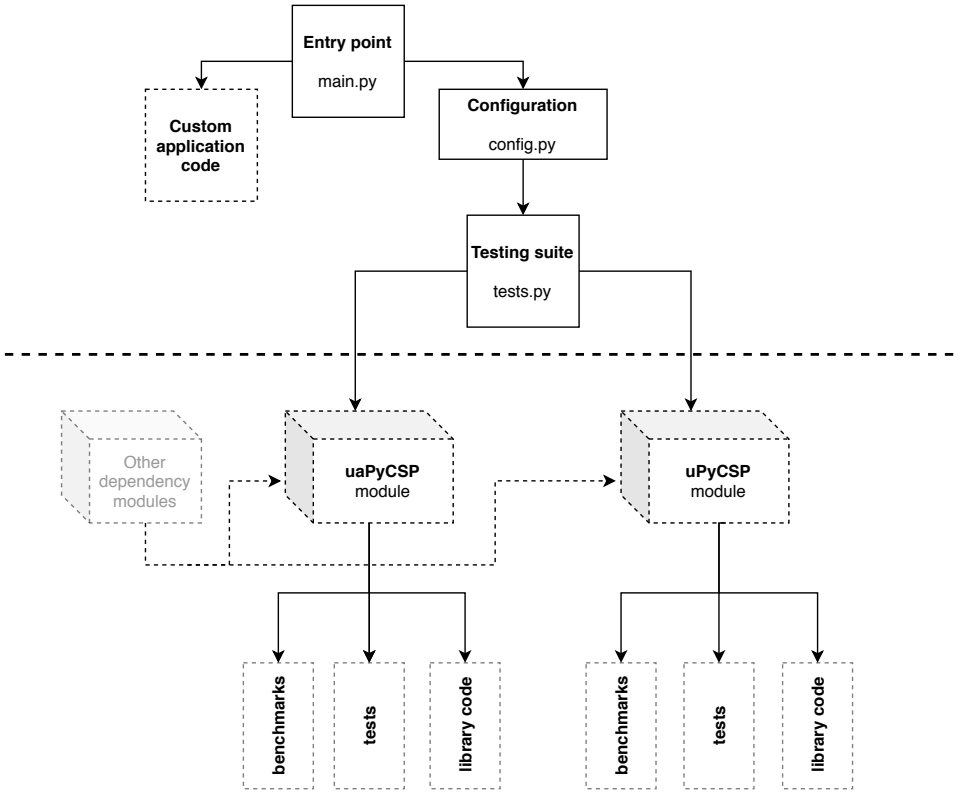

Figure 4.1: Project architecture

Entry point

While the UNIX distributions of Micropython vary between OS platforms, the project's structure has almost no requirements in terms of structure. The limitations are apparent when focusing on a microcontroller target. Devices without an underlying OS, such as the FiPY, often require an entry point file `main.py` to bootstrap the rest of the application code. This entry point works adequately for use on UNIX distributions as well.

Configuration

Concerning the ability to execute different tests, benchmarks, and different versions of the PyCSP code, a configuration step that enables executing specific benchmarks or tests. This configuration has to be hard-coded within config.py and has to be configured prior to execution as the FiPy platform does not accept arguments for execution.

While the configuration step and test suite are convenient for development, and

running time-based performance experiments, all of the tests and benchmarks should be possible to run in an isolated setting, to leave the possibility open for experiments such as measuring memory or power usage.

PyCSP Library Code

The code written for the modules should be operating in isolation. As the libraries' intention, is just that, libraries, the code retaining to the configuration of the test and development environment are separated. This leaves the possibility of extracting the modules, without the entry point or configurations. The libraries dependencies, however, are not necessarily included within the library.

Module Dependencies

While there are whole module dependencies outside of the module code, these are placed as stand-alone modules alongside[4] the module for the most part. The intention to keep a clear separation of the modules. This leaves the possibility of replacing the dependencies with newer versions or better implementations without affecting the PyCSP module code.

4. Some of these dependencies require reimplementation as a part of this project, while others are available through the upip module and Micropython standard library, described in section 3.4

5 Implementation

In this chapter, the implementation specifics of the project will be described. Smaller implementation changes, generic to both versions of PyCSP, will be discussed in a common section, while the particulars to each approach are described in their respective sections.

For this implementation the following versions have been used:

Micropython - Unix Version 1.11

Pycom Micropython Version 1.20.1.r1 (Equivalent to v1.11 of Micropython)

uasyncio Version 2.0.0

5.1 uaPyCSP

The asyncio-based PyCSP version aPyCSP is ported to Micropython and thus named uaPyCSP to distinguish between the two versions. This section contains the specific changes made to the source code of the library, done to create the uaPyCSP library.

5.1.1 Futures

Futures, as mentioned in section 3.4.1 is a feature from asyncio, not provided in uasyncio. The Futures waiting mechanism is used in the channels to await operations queued on a channel. Upon scheduling a command on a channel, it internally uses a future to wait for results. It is critical to provide the same functionality through reimplementation to maintain the same structure as the original implementation of aPyCSP. Two different implementations of future objects are implemented.

The usage of sparse usage futures in aPyCSP, allows for only a subset of methods to be reimplemented for usage in uaPyCSP. With a focus on the core functionality of the future object, rather than a feature-packed version, it fits into the application without any intervention into the uasyncio library. The Future class feature a constructor method, an await method, and a set-result method.

First Version of Futures

As seen in listing 5.1, the future object is relatively simple. After initialization, the options are either awaiting the future or setting a result.

The `__await__` method itself is a coroutine that yields until the done flag has is set. Mainly this causes the coroutine to "take a break", so another coroutine has the possibility of completing the future by setting the result. Upon reentrance, the state is reevaluated and returning a result if completed.

Listing 5.1: Initial Future implementation

```
class uFuture1:
    def __init__(self, loop=None):
        self.done = False
        self.result = None

    async def __await__(self):
        while not self.done:
            yield
        return self.result

    # CPython compability
    __iter__ = __await__

    def set_result(self, result):
        self.done = True
```

```
        self.result = result
```

Second Version of Futures

The second version of futures, which has a bit more complex inner logic (seen in listing 5.2), handles awaiting differently. With the intention of suspending and rescheduling, instead of reevaluating the completed state multiple times, this `__await__` method stores a reference of the waiting coroutine in an internal list. It then yields a `False` value.

Listing 5.2: Second implementation of Future, with a suspending mechanism

```
class uFuture:
    def __init__(self, loop=None):
        self.done = False
        if loop:
            self.loop = loop
        else:
            self.loop = uasyncio.get_event_loop()
        self.result = None
        self.waiters = []

    async def __await__(self):
        while not self.done:
            self.waiters.append(self.loop.cur_task)
            yield False
        return self.result

    # CPython compability
    __iter__ = __await__

    def set_result(self, result):
        self.done = True
        self.result = result
        for task in self.waiters:
            self.loop.call_soon(task)
```

While undocumented in uasyncio, yielding a `False` value prevents it from being rescheduled on the Event loop. This creates the effect of the coroutine being taken out of the scheduling. Initially, in the development, the future object yielded a reference to itself, and by creating a type-check in the event loop, it provided the same non-rescheduling functionality. This provides more readability for the code, rather than yielding a False value. It was decided

against, as doing so requires changes to asyncio code.

Upon completion of the future object, in `set_result`, the list of *removed* coroutines is looped through and rescheduled on the event loop associated with the future.

Creating Futures

A method create_future, known from asyncio, has also been reimplemented. While originally being a method on the event loop class, this is now a standalone function, with an optional loop argument. This function also creates an effective way of alternating between using the different future object, when in an experimental phase such as this project.

5.1.2 Gather

Gather is a mechanism from asyncio. The gather function takes multiple awaitable objects as arguments and returns a list of the resulting (returned) values. The gathering mechanism is not featured in uasyncio.

Internally gather sets up the scheduling of all the entered awaitables, before yielding to the execution to perform these. Upon completion of all the awaitables, the gather function compiles a list containing results from the each of the awaitables. The result list is ordered corresponding to the order of the inputted awaitables, returned to the caller of gather.

The gather function is implemented as a class, leveraging the `__iter__` method, which creates the illusion of the gather functionality being a coroutine function, similar to how the `__call__` method can be used to make a class appear as a function, described in section 3.2.1. This is done to adapt correctly to the asyncio interface where gather is an awaitable [15].

Upon initialization, a barrier object is created, alongside a result list. The barrier object is used to aid in synchronization of the awaitables scheduled. The gather functionality continues to loop through the task and schedule them using the uasyncio provided create_task function. Notable in listing 5.3 the wrapping of the objects before scheduling them. Wrapping the objects not only allows for a mechanism to store the result properly in the result list but also creates the possibility of adequately catching and raising a ChannelPoisonException to propagate poison. Since the gather function is a coroutine, it will be scheduled in the CSP-network similarly as a CSP-process, but lacks the proper poison propagation mechanisms Process objects has.

Listing 5.3: Gather implementation

```
class gather():
    def __init__(self, *procs):
        ncoros = len(procs)
        self.barrier = Barrier(ncoros)
        self.results = [None] * ncoros
        self.poisoned = False
        loop = asyncio.get_event_loop()
        for n, proc in enumerate(procs):
            loop.create_task(self.wrap(proc, n)())

    def __iter__(self):
        try:
            while not self.barrier.complete():
                yield
            return self.results
        finally:
            if self.poisoned == True:
                raise ChannelPoisonException

    def wrap(self, coro, idx):
        async def wrapped():
            try:
                self.results[idx] = await coro
            except ChannelPoisonException:
                self.poisoned = True
            self.barrier.trigger()
        return wrapped
```

Awaiting the gather functionality uses the `__iter__` method, which similarly to how the future objects in section 5.1.1 work, yield until the barrier signals its completion. It also checks and raises the ChannelPoisonException for the executing channel to propagate poison further, if the gather object is poisoned.

5.1.3 uasyncio - Run Until Complete

A minor fix for the uasyncio event loop's `run_until_complete` method has been implemented. The `run_until_complete` method is used for the `run_csp` function in PyCSP. The difference observed between uasyncio and asyncio, is that the version from uasyncio of the function does not return the results after execution of the scheduled coroutines. A minor change to this method creates

the possibility of successfully returning the result from the coroutine scheduled using this function. The change can be seen in listing 5.4.

This has been done directly within the uasyncio library. While intrusive, the decision to do so is due to the fact that this clearly is a bug within uasyncio, rather than an intentional design.

Listing 5.4: Fix for run_until_complete method

```
# Original
def run_until_complete(self, coro):
    def _run_and_stop():
        yield from coro
        yield StopLoop(0)
    self.call_soon(_run_and_stop())
    self.run_forever()

# Fixed version
def run_until_complete(self, coro):
    def _run_and_stop():
        res = yield from coro
        yield StopLoop(res)
    self.call_soon(_run_and_stop())
    return self.run_forever()
```

5.2 uPyCSP

The original PyCSP and PyCSP-Lockversion are distinguishable in name, but since there currently is no other version of PyCSP supported in Micropython, the name of the PyCSP-Lockversion ported will be uPyCSP. This section contains descriptions of the specifics done to implement the module.

5.2.1 uThreading - Threads

Threads, described in section 3.3.1, has a feature rich implementation in in Python. For the reimplementation of the threads, a more narrow approach is done. The result are thread objects, which only supports core functionality.

Initialization Initialization of a Thread object mostly configures initial values for some of the variables keeping track of the state of the thread, as well as the name of the thread. If a target method is passed (along

with arguments and keyword arguments), these are stored here. The allocation of a lock object from the _thread library is done. Description of its use in the following section.

State-lock & Join A state-lock object, allocated in the initialization, is held by the thread while it executes its target method. Once the target method has finished, the lock is released. The implications further down the line become clear when other threads attempt to perform a join on this thread. While they are able to acquire the lock when the thread no longer holds on to it, they are required to wait for it while it does. Once they hold on to the lock, they instantly release it to allow for other threads to join. Using a lock in this manner creates the effect of "waiting" for the thread to finish.

An essential factor in this design is resource consumption when waiting for a thread to complete execution. Leveraging the lock primitive, instead of setting up while-statement-conditioning the lifetime status of the thread, which would result in a spinning thread, using all the CPU power possible. The lock from _thread is implemented in such a manner that the thread is suspended, and not active until it can acquire the lock.

Start & Bootstrapping The thread's start method initializes the thread, in which the target function is executed on. The start method also sets up manages some of the internal mechanisms of the thread object. These internal mechanisms, such as acquiring a state lock and setting lifetime status, are done right before spawning the new thread.

A separate bootstrapping method sets identity from within the new thread and executes the target method. As seen in Listing 5.5The bootstrapping method makes sure that the thread releases its state lock, and re-sets its lifetime status after the execution of the run method by leveraging a try-catch-finally block.

The concept of using a bootstrap method, instead of placing this functionality within the run method, derives from the inheritance usage of the Thread class. Any important internal mechanisms in the run method would be overridden when inheriting from the Thread class and implementing the run method.

Listing 5.5: The implementation of a Thread's `start`, `bootstrap` and `run` methods

```
class Thread:
    def __init__(self, ...):
        # Initialization logic

    def start(self):
        if self.alive:
            # Thread is already alive
            raise RuntimeError
        # Get identifier for the thread
        self.alive = True
        self.started = True
        self.state_lock.acquire()
        _thread.start_new_thread(self.__bootstrap, ())

    def __bootstrap(self):
        try:
            self.indent = _thread.get_ident()
            self.run()
        finally:
            self.state_lock.release()
            self.alive = False

    def run(self):
        self.target(*self.args, **self.kwargs)
```

5.2.2 uThreading - RLock, Reentering Locking Mechanism

Another critical piece provided by the threading module is the RLock object. As mentioned in section 3.8, a global RLock, is used in the implementation of future objects.

Acquiring the Lock Attempting to acquire a reentering lock establishes if a thread already holds the lock and if so, that thread is the same thread that attempts to acquire it. [1] If so, a counter is increased, and the method returns the value 1, signaling that the lock was acquired. If the lock is not acquired or acquired by another thread, a call to the underlying _thread method to acquire the actual lock is done. In the case of the lock being acquired, this results in a thread waiting for the lock to become available.

1. Finding a threads identity is done through _threads get_ident function, wrapped in a get_ident function in uthreading, alike the original threading module.

Once the lock is acquired by a thread, the owner and count state is set.

Releasing the Lock Releasing the lock subtracts one from the counter property. If the property reaches 0, the owner is removed, and the lock is released through the underlying _thread library's mechanism. Attempting to release a lock, which is not held by the thread invoking the call results in a RuntimeError.

5.2.3 Itertools

itertools is a module containing building blocks for iterators. itertools is not available in Micropython.

In the Thread class, a counter is used for default naming of threads [2]. A small substitute for the counter generator replacement for counter is implemented to maintain the naming functionality within the thread class.

5.3 Compability Changes

There are multiple places throughout the libraries where a newer syntax then what is available in Micropython. These are minor changes and are not considered drastic enough to be mentioned in detail. Examples of these are print statements, and constructors of inheriting classes which have a slightly different syntax.

Import statements using aliases are also used throughout the implementation to ensure that changes are kept to a minimum. Conditional imports create support for running the module code with Python support, in addition to the Micropython target support.

5.4 Collections' Deque

The Double-ended queue (deque) object, available from the collections module, is used to as read-and-write queues for Channels. The collections module is not available in Micropython, and the module ucollections is used as a replacement.

2. Thread names are by default a string "Thread-N", where N is an int starting at 1 increasing by 1 for every thread spawned. Names are not unique and can be overridden manually through the name property

This replacement does not provide all the features of the original module, and the Deque available from it has limitations.

The Deque from ucollections has two methods implemented; append, which adds an item on the right side of the queue, and popleft, which removes the first item from the left end of the queue. These two methods make it possible to use it as a FIFO queue.

Max Length

The ucollections' deque object also requires a max-length in its constructor, unlike the deque from collections. The max-length has an optional silent/exception toggle available if the queue has reached maximum elements. On appending, the silent approach removes the first element from the queue, while the exception toggle throws an error if it is full.

Filtering a Deque Object

In both implementations of PyCSP, uses the builtin filter-method to operate the write and read queue. However, the __iter__ method is not implemented for the ucollections deque object, excluding the possibility of using the filter method. An implemented workaround to the missing filtering is done by iterating the full length of the queue using popleft, and the filtering conditional for appending the element back on the queue, as seen in listing 5.6.

Listing 5.6: Manual filtering of deque object.

```
def _remove_alt_from_pqueue(self, queue, alt):
""" Common method to remove an alt from the
    read or write queue.
"""
    for _ in range(len(queue)):
        op = queue.popleft()
        if (not op.cmd == 'ALT' and op.alt == alt):
            queue.append(op)
    return queue
```

5.4.1 List Implementation

As an alternative to using double-ended queues within the Channels, a Python List can be used instead. Lists have dynamic length and require no input configuration. Filtering lists are also possible, which results in a minimal

implementation for removing alternatives from the queue.

Alternate implementations of both uaPyCSP and uPyCSP have Channels using lists as queues instead of Deque.

5.5 Timer - Chrono

Many of the performance benchmarks, an exact measurement of time, are critical to the results. In Python and Micropython, the time module provides high-resolution accurate readings. However, when using testing this module on a FiPy, which uses a Pycom port of Micropython, the time module's resolution is is inadequate, as it reports time in seconds.

In the Pycom port of Micropython, measuring time is handled by another module. A chronometer class Chrono[3] from a module Timer provides readings with microsecond precision. The Chrono class is not available in the official micropython version or Python.

For compatibility between platforms, a Chrono class is implemented, using the time module as its basis. This allows the same benchmarking code to be similar for all platforms, by using a platform-based import statement alternating the import of the Chrono class.

3. https://docs.pycom.io/firmwareapi/pycom/machine/timer/

6 Experiments

The chapter will outline the different experiments used in this project, with the intention of describing the mechanics behind the experiments, what the experiments achieve, and the value of the results.

6.1 CommsTime

The CommsTime benchmark is used to measure communication overhead between CSP-processes using channels. The result reflects not only the communication, but due to the simple design (seen in figure 6.1) indirectly measures the overhead associated with switching executions between the processes.

The benchmark is initiated by a message sent from a Prefix process through a channel to the Delta2 process. The Prefix process initiates with a message, then just relays the messages received upon a channel, onto the next. The Delta2 process (which has two implementations, described in the following section) sends the message onto two channels. One of the channels is used for the Consumer process, which keeps track of the number the messages passed through the benchmark and eventually will poison the network. The other channel leads to the Successor process. This process increments the value received before sending along to the next channel, leading back to the Prefix process.

By sending a large number of messages through the CommsTime network, one can, with accuracy calculate the communication time of a message per channel, and measuring the round-trip time of a message by dividing the result by the number of messages and finding per-channel time by dividing it by the number of channels.

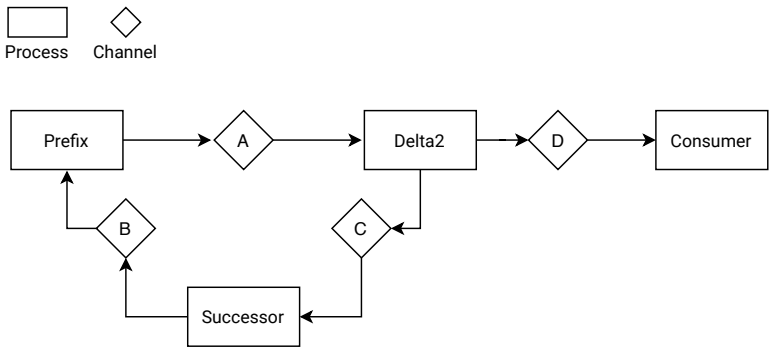

Figure 6.1: A visual representation of the CommsTime benchmark

6.1.1 Delta2

The Delta2 process [3], which is used to write an inputted value onto two output channels, has two implementations. A parallel version uses two writer-processes, spawned in a Parallel section. The sequential version executes these write operations one after the other. The alternate implementations of Delta2 allow for measuring a performance difference between parallel and sequential execution of processes, where the parallel version additionally spawns processes.

6.2 Stressed Alternatives

The Stressed Alternatives experiment [16] uses multiple writer processes to write to multiple channels as rapidly as possible, while a single stressed reading process is performing a given number of selects, using alternatives to read from the same pool of channels. This benchmark effectively measures the cost of using the select mechanism for alternatives.

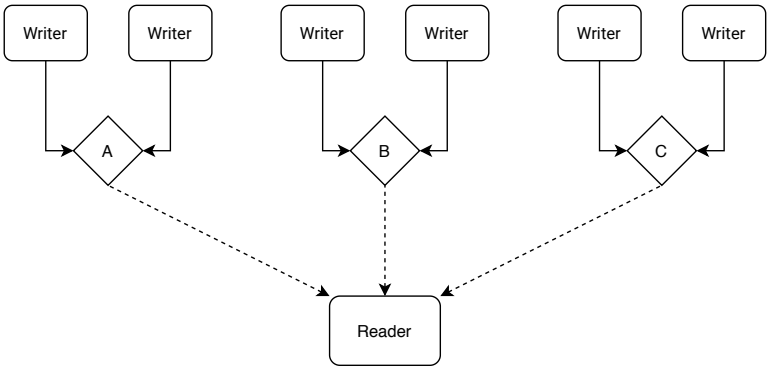

Figure 6.2: A visual representation of the Stressed Alternatives Experiment experiment with three channels, and two writers per channel

A side effect of the experiment, which in its uses a vast amount of processes, is the ability to get a good indicator of the number of processes the library supports[17].

In addition, the experiment utilizes the functionality related to the alternatives and selects, such as channel operation queueing, and changes to the internals of the queuing mechanism should be visual in the results.

6.3 List & Deque Benchmark

The List & Deque Benchmark is a microbenchmark measuring the performance of common operations in the builtin List object, and the Deque object provided by the ucollections module. The goal of the benchmark is to measure specifically the cost of performing corresponding functionality between the objects.

The benchmarks comprise of multiple small benchmarks testing the performance of the different operations used in the application. These operations are repeated a number of times to improve accuracy. These different operations are:

Append Appends an object to the end of the List or Deque object.

Popleft Removes an object from the beginning (left side) of the List or Deque object.

Filter Filters objects from the List or Deque of based on a conditional statement.

The benchmark provides useful context to the choice of using one or the other for different mechanisms within the library.

6.4 The Waiting Queue

The Waiting Queue experiment spawns a number of coroutines, which all wait for futures to complete. In the mean-time, a pair of processes tasked with sending a number of messages back and forth through channels, simulating regular usage of the library. Once the goal amount of messages is sent and received through channels, the processes terminate, and all the waiting futures are completed. The result is benchmarking the time it takes to pass a given amount of messages, with a specified amount of futures active in the background.

With the implementation of multiple futures, an observation was that the initial implementation performed better than the rescheduling futures in the CommsTime experiment described in section 6.1. The initially spawned futures increase the size of the active queue with each coroutine waiting for a future, which resulted in a question as to if the number of coroutines waiting in the queue might result in differences in performance in use cases where the number of processes utilizing futures increases.

The goal of the benchmark is to see if there are changes in the performance depending on how many processes are idling in the run queue. The potential here is to discover if there are any notable performance differences in the two future implementations and uncover potential aspects with either of them.

7

Results

The results discussed in this section aim to document the performance of the implementation through the experiments brought forth in chapter 6. All experiments are reproducible with the source code, environment, and the given configurations.

7.1 Platforms & Environments

UNIX The UNIX environment is run on a Lenovo ThinkCentre M920Q[] equipped with an Intel Core i5-8500T Processor(6 cores, 2.1Ghz 3.5Ghz) and has 16GB memory at 2666MHz. The system is using Ubuntu 18.04, with Micropython version 1.11 (Linux version) and Python 3.8.0.

FiPY The FiPy environment is run on an ESP32 microcontroller[1] (2 cores, 240mhz, 4MB memory). The system is using Pycom Micropython 1.20.1.r1[2]

1. https://www.espressif.com/en/products/socs/esp32/overview
2. Equivalent to v1.11 of Micropython

7.2 Communication Time - CommsTime

CommsTime, as described in the previous chapter, measures communication performance. The communication time is measured in messages round-trips of the CommsTime network. A number of messages' round-trip time is recorded, before division by the number messages, and then again by the four channels. The presented results are are the communication time for a single message through for a single channel

7.2.1 UNIX Platform

The Unix platform, which has support for both Python and Micropython environments, can compare the execution of four PyCSP libraries on the same hardware. The modules aPyCSP and PyCSP are executed using Python, and uaPyCSP and uPyCSP are executed using Micropython.

Configurations & Results

The CommsTime benchmark data presented in Table 7.1 and Figure 7.1 is a result of 5000 messages passed through the network. Using 100 runs, and presenting the minimum, maximum, and average communication times for a message on a channel. Additionally, both the Sequential and Parallel version of the Delta2 process is used.

Delta2	Environment	Version	Min(µs)	Avg(µs)	Max(µs)
Parallel	**Python**	PyCSP	36.687	37.563	39.618
		aPyCSP	12.773	13.053	13.403
	Micropython	uPyCSP	28.742	29.580	30.595
		uaPyCSP	22.077	22.255	22.975
Sequential	**Python**	PyCSP	12.000	13.293	17.941
		aPyCSP	5.905	6.186	6.784
	Micropython	uPyCSP	17.622	17.999	18.396
		uaPyCSP	13.483	13.606	13.847

Table 7.1: Results from from 100 executions of the CommsTime experiment from multiple PyCSP-versions executed on UNIX platform, configured with 5000 messages passed through the CommsTime processes.

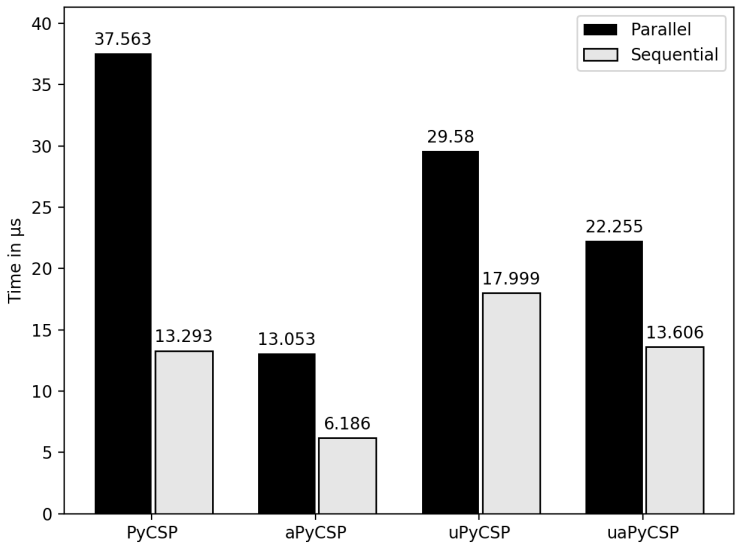

Figure 7.1: Average results from the CommsTime experiment from multiple PyCSP-versions executed on UNIX platform based on Table 7.1.

7.2.2 FiPY Platform

The experiment executed using the FiPY Platform does not support Python, hence why only Micropython libraries are used. Additionally, the computational speed is significantly lower, and a smaller number of messages are required to measure communication time with precision.

Configurations & Results

The experiment is executed with 200 messages, and the results is presented in Figure 7.2 and Table 7.2. Results are presented with minimum, maximum and average communication times from the 100 runs, given in µs.

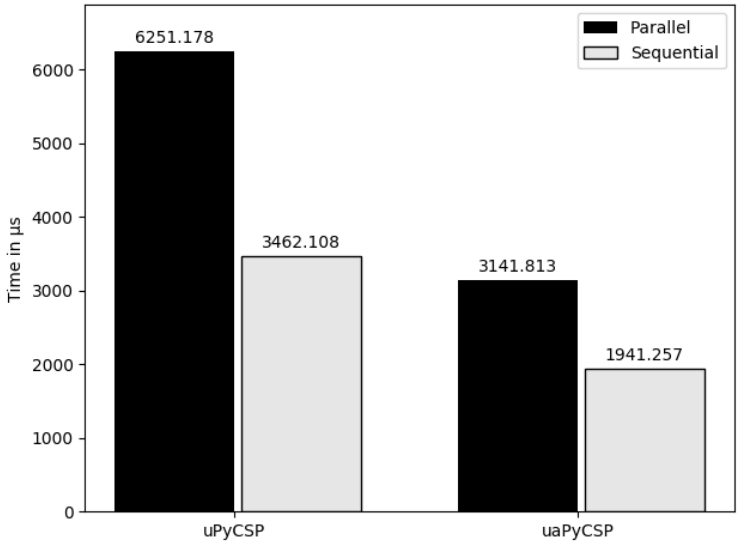

Figure 7.2: Average results from the CommsTime experiment from multiple PyCSP-versions executed on FiPy platform based on Table 7.2

Delta2	Version	Min(µs)	Average(µs)	Max(µs)
Parallel	uPyCSP	5934.724	6251.178	6703.854
	uaPyCSP	2941.495	3141.813	3473.825
Sequential	uPyCSP	3100.868	3462.108	3827.499
	uaPyCSP	1802.616	1941.257	2191.131

Table 7.2: Results from the CommsTime experiment on FiPY Platform. The experiment used 200 messages, with 100 runs.

7.3 Stressed Alternatives

The Stressed Alternative benchmark used to measure the performance of Alternatives and Select. The experiment uses a reader CSP-process, performing a number of selects onto a given list of channels. The use of multiple selections, reduces the any potential overhead caused by measuring disturbances is the time measuring mechanism. Each channel has a set sized pool of writer CSP-processes. Writers perform write operations endlessly onto channels. After the conclusion, upon finishing the number of selections, the reader will issue a Channel poison, shutting down the network. This execution is repeated a set number of times to ensure precise results. The total time spent on the number of selections is then divided by the number of selections, which yields an estimate of how much time a single selection takes.

Additionally, there are two options for performing Alternative selects, either by using the `with` keyword on an Alternative, or by calling the Alternative's `select` method. Both are options are tested.

7.3.1 UNIX Platform

On the UNIX platform, performance using two configurations are measured. This is done to see how the amount of processes in total has an effect on the performance of the primary task, performing select operations.

Large CSP-pool configuration and results

The first configuration uses a large pool of CSP-processes; 100 channels, with 5 writer processes per channel. This leads to the total amount of CSP-processes being 501. The reader process performs 5000 selections. The results from Table 7.3 are minimum, maximum, and average results observed from 100 executions.

Select	Environment	Version	Min(µs)	Average(µs)	Max(µs)
with	Python	aPyCSP	78.716	79.358	80.656
		PyCSP	47.757	59.170	97.495
	Micropython	uaPyCSP [1]	249.102	251.054	253.422
		uaPyCSP [2]	246.971	249.032	251.841
		uPyCSP	24.615	26.283	34.524
alt.select	Python	aPyCSP	78.349	79.104	84.470
		PyCSP	49.583	58.457	103.198
	Micropython	uaPyCSP [1]	250.459	252.375	254.045
		uaPyCSP [2]	247.531	250.031	252.369
		uPyCSP	24.862	26.632	29.528

1. Using rescheduling futures 2. Using yielding futures

Table 7.3: Results from Stressed Alternatives benchmark run on the UNIX platform. The results are from performing 5000 selects on 100 channels. Each channel has 5 writers.

Small CSP-pool configuration and results

A second configuration uses a reduced pool of CSP-processes. Using only 5 channels, with 2 writer processes each, for a total of 11 CSP-processes in the network. The readers perform 5000 selections. The results from Table 7.4 are minimum, maximum, and average results observed from 100 executions.

Select	Environment	Version	Min(µs)	Average(µs)	Max(µs)
with	Python	aPyCSP	10.801	10.930	12.359
		PyCSP	17.738	19.247	26.880
	Micropython	uaPyCSP [1]	27.009	27.185	28.878
		uaPyCSP [2]	25.648	25.783	27.305
		uPyCSP	18.433	19.087	22.838
alt.select	Python	aPyCSP	10.294	10.371	10.858
		PyCSP	17.436	18.817	21.336
	Micropython	uaPyCSP [1]	26.474	26.679	27.326
		uaPyCSP [2]	25.063	25.250	25.727
		uPyCSP	18.673	19.359	20.671

1. Using rescheduling futures 2. Using yielding futures

Table 7.4: Results from Stressed Alternatives benchmark run on the Unix platform. The results are from performing 5000 selects on 5 channels. Each channel has 2 writers.

7.3.2 FiPY Platform

On the FiPY platform, using a large pool of CSP-processes is not possible. A limitation introduced by the uPyCSP version, as attempting to use more than 20 CSP-processes, causes a MemoryError. As previously stated in 6.2, the Stressed Alternatives benchmark can be used to find the maximum amount of CSP-processes possible to spawn with a library and is an important result in itself.

Configurations and results

Since spawning a large pool of threads was not possible, a smaller pool of 5 channels, with 2 writers each, totaling at 11 CSP-processes is used. The number of selects is reduced to 200, as the computational hardware on FiPY is a lot slower, and does not need the same sized pool to reduce the overhead caused by time measurement. The results from execution on uaPYCSP and uPyCSP are found in Table 7.5

Alternative select	Version	Min(µs)	Average(µs)	Max(µs)
with	uaPyCSP [1]	4773.066	4986.087	5655.597
	uaPyCSP [2]	4480.089	4712.095	5454.641
	uPyCSP	3299.178	3523.560	4053.391
alt.select	uaPyCSP [1]	4611.241	4844.526	5403.270
	uaPyCSP [2]	4331.133	4568.494	5224.5830
	uPyCSP	3309.888	3522.165	4185.317

1. Using rescheduling futures 2. Using yielding futures

Table 7.5: Results from Stressed Alternatives benchmark run on the Unix platform. The results are from performing 200 selects on 5 channels. Each channel has 2 writers.

7.4 Lists & Deque Benchmark

The microbenchmark described in section 6.3, measures the speed of Deque and List objects. It reports the time it takes to perform operations with a number of elements using the different objects. The experiment is performed with a varying number of elements within the object, to see how the data structure handles the different sizes.

The resulting values are the time it takes to add and remove a given number of elements using append and popleft, while the results for filter is the reported time it takes to perform a single filter operation on the given number elements using the underlying data structure.

7.4.1 UNIX Platform

The benchmark is run with a number of elements in the interval of 500 ~16000, starting at 500, doubling for each iteration value. The measurements are presented in Figure 7.3, and Table 7.6 and display the average execution times of 500 runs per iteration.

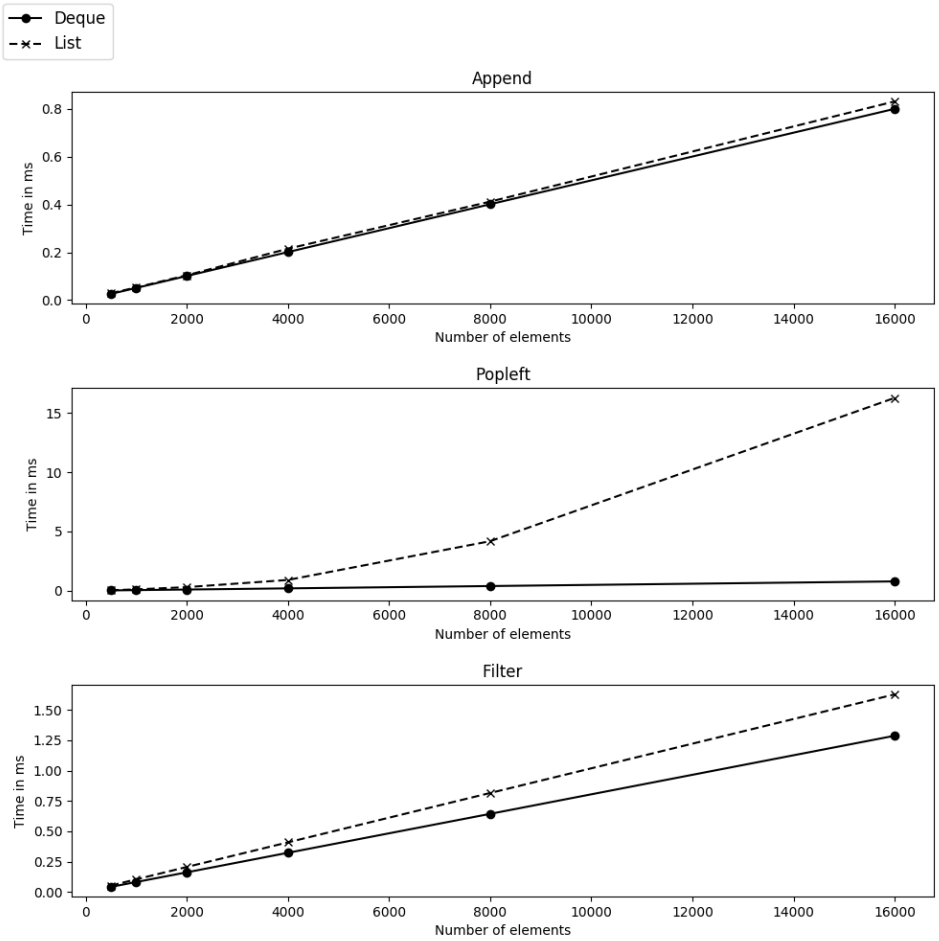

Figure 7.3: Average benchmark measurements of Deque and List objects performance on Append, Popleft and Filter operations with set sizes in the interval of 500 to 16000 elements, based on numbers from Table 7.6. Executed on UNIX platform

N	Append	Popleft	Filter
500	26	25	42
1000	51	51	82
2000	101	100	162
4000	201	198	323
8000	401	394	644
16000	800	786	1288

(a) Deque results in µs

N	Append	Popleft	Filter
500	29	46	53
1000	53	113	104
2000	104	302	206
4000	215	908	409
8000	412	4179	816
16000	831	1629	1629

(b) List results in µs

Table 7.6: Benchmark measurements of Deque and List objects performance on Append, Popleft and Filter operations with set sizes in the interval of 500 to 16000 elements. Executed on UNIX platform

7.4.2 FiPY Platform

The benchmark is run with a number of elements in the interval of 50 ~1600, starting at 50, doubling for each iteration value. A small set size is used as computational speeds are slower, and accurate time measurement is achieved with fewer elements used in the set. The measurements are presented in Figure 7.4, and Table 7.7 and display the average execution times of 500 runs per iteration.

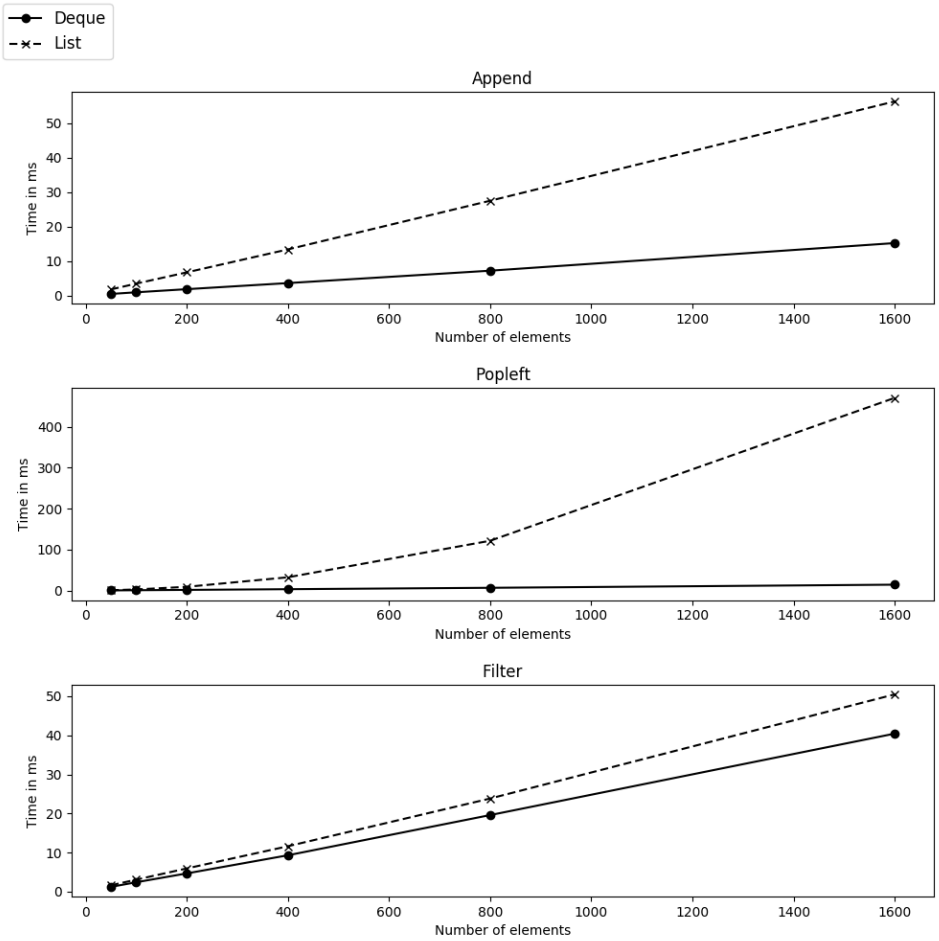

Figure 7.4: Average benchmark measurements of Deque and List objects performance on Append, Popleft and Filter operations with set sizes in the interval of 50 to 1600 elements, based on numbers from Table 7.7. Executed on FiPY platform

N	Append	Popleft	Filter
500	485	463	1218
1000	964	909	2401
2000	1868	1769	4666
4000	3633	3460	9305
8000	7345	6869	19592
16000	15243	14506	40435

(a) Deque results in µs

N	Append	Popleft	Filter
500	1794	1016	1626
1000	3476	2879	3066
2000	6742	9256	5936
4000	13406	32486	11618
8000	27530	121650	23828
16000	56357	470440	50480

(b) List results in µs

Table 7.7: Benchmark measurements of Deque and List objects performance on Append, Popleft and Filter operations with set sizes in the interval of 50 to 1600 elements. Executed on FiPY platform

7.5 The Waiting Queue Experiment

The Waiting Queue experiment design described in section 6.4 is implemented using a Successor CSP-Process, found in the CommstTime benchmark, as well as a specialized Successor CSP-process. These two successors pass messages back and forth using two Channels. Once a goal number of messages is sent back and forth, the specialized successor reads the time, resolves futures, and poisoning the CSP network. Meanwhile, a number of Waiting CSP-processes are run alongside these Successors. The waiters perform an await on a future, creating the effect of a number of extra processes in the schedulers' queue waiting for future objects to be resolved. The amount of waiter processes is increased in an interval ranging from 1~30, to see the performance changes with an increasing amount of waiter processes used, and thus futures await mechanism.

The result after the benchmark is the total time it takes to send several messages, with a given amount of processes waiting for futures to be resolved at the same time.

7.5.1 Unix Platform

For the Unix Platform, 1000 messages are sent back and forth between the two Successors. The execution is run 100 times, and the average measurement from those executions is found in Figure 7.5.

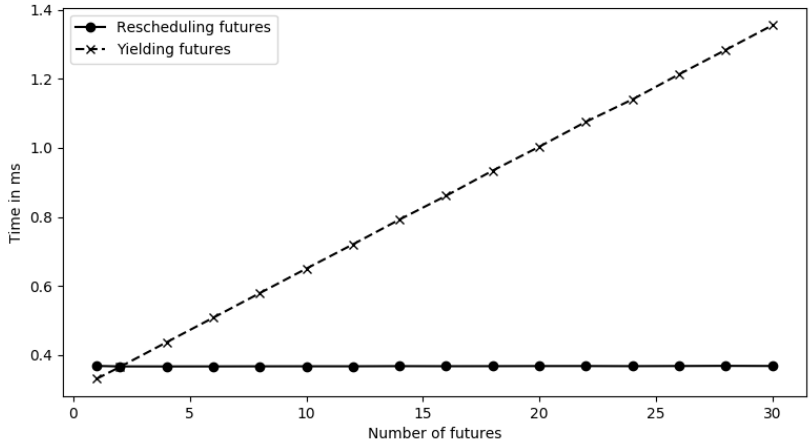

Figure 7.5: Results from The Waiting Queue experiment on Unix platform, using 100 messages and a range of 1~30 futures

7.5.2 FiPy Platform

For the FiPy Platform, 100 messages are sent back and forth between the two Successors. The execution is run 100 times, and the average measurement from those executions is found in Figure 7.6.

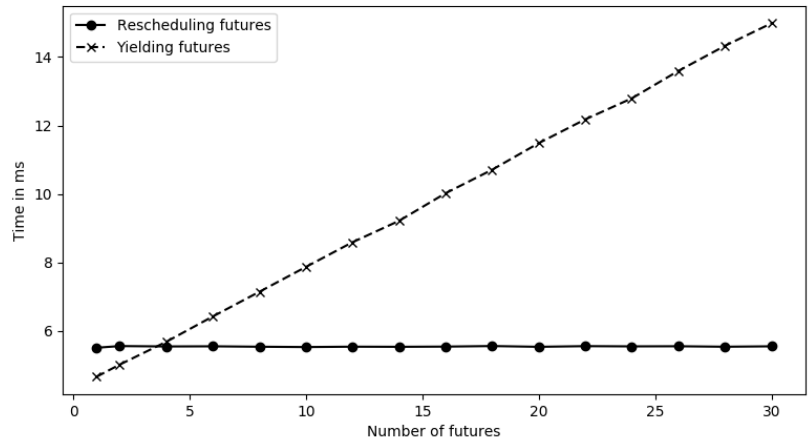

Figure 7.6: Results from The Waiting Queue experiment on FiPY platform, using 100 messages and a range of 1~30 futures

8 Evaluation

In this chapter, an evaluation of the thesis implementations will be done using the measurements brought forth in the previous chapter, focusing on providing context and enlightening the implications of the results, presented in the form of performance advantages and disadvantages of the built libraries.

8.1 CommsTime

Readable from 7.1, is that the overall minimum and maximum recorded executions are relatively close to the average values reported. The most significant variances are found in the results for PyCSP. There seem to be no disturbances influencing the results from the experiments performed, and the results presented appear legitimate.

Upon inspection of Figure 7.1, it becomes evident that the sequential Delta2 process, regardless of the used version, Python environment and platform, performs better than the corresponding parallel variant. The sequential Delta2 executes writes operations to the two output channels in sequence rather than in parallel. As described in Bjorndalen et al. [4], using the Parallel construct is done by spawning to two new CSP processes. In uPyCSP and PyCSP-Lockversion, this means spawning two new threads, which has a high cost. In aPyCSP and uaPyCSP spawning these two CSP-processes comes at a reduced cost as the processes are implemented as coroutines, and the difference

between the two Delta2 variants is smaller, but still considerable. The results from the FiPY environment, displayed in Table 7.2 and Figure 7.2, confirms the trends of increased performance with a sequential Delta2 Process.

The performance from both the original versions of aPyCSP and PyCSP Lock-sharing is run with the Python environment. These are performed using a different environment, and should not be used used for direct comparison between the Micropython and Python libraries. However, they bring context as to how the original libraries perform compared to each other while using the same hardware-environment as the Micropython libraries. In Figure 7.1 the performance of uaPyCSP is better than uPyCSP, similarly to the results when comparing aPyCSP and PyCSP-Lockversion.

A difference observed in Figure 7.1 is that there is that the gap between the communication time in uPyCSP and uaPyCSP both in parallel and sequential Delta2 is considerable smaller, than it is between aPyCSP and PyCSP-Lockversion. This points to uPyCSP performing very well, and a possible explanation for this is that threads are less time-intensive to spawn in Micropython than in Python, due to a smaller implementation. This theory strengthened when making a comparison of uPyCSP and PyCSP-Lockversion, both using the parallel Delta2.

Looking at Figure 7.2, a unique trait is that both the sequential and parallel execution on uaPyCSP perform better than uPyCSP's sequential execution. As mentioned previously, CSP-processes as threads come at a higher cost then CSP-processes as coroutines. A possibility is that the cost-ratio of spawning threads-to-coroutine may be even higher on for the FiPY platform. The support for spawning threads is a platform and Operating System (OS)-specific implementation, and could return different results on different platforms and OS.

8.2 Stressed Alternatives

From Tables 7.3, 7.4 and 7.5, it is clear that the difference in usage of the "with" statement and alt.select has little or no impact on the performance. Throughout both platforms and environments, there is no evidence that there is a preferred way of using the selection mechanism. This means that both ways of selecting alternatives are equally good, and can both be used without the concern of performance loss.

Tables 7.3 and 7.4 show that the performance ratio between uPyCSP and uaPyCSP is inconsistent with two different configurations. The change in

performance of the uaPyCSP with an increased number of CSP-processes run is dramatic, compared to the corresponding change in performance uPyCSP displays. The change in performance is a result of the FIFO scheduling used by the uasyncio Event loop. Once there are more processes in the run queue, the reader process is scheduled more seldom. However, the threads from uPyCSP rely on scheduling performed by the underlying platform OS, which can be far more complex. A suspicion, yet to be confirmed is that the scheduler prioritizes the reading process, as all the writing processes end up in the _thread.acquire_lock() call within the reentering lock.

As already mentioned in section 7.3.2, a large pool of CSP-processes is not run on the FiPY platform. Due to a limitation in the number of threads possible to have running on the FiPY platform.

8.3 The Waiting Queue

From Figures 7.5 and 7.6, it appears that results for both FiPy and UNIX environments are alike. The results of the experiment show that message passing with yielding futures scheduled in the background has a linear performance, while an increasing number of rescheduling futures do not affect the performance of the on-going message passing.

With only a few yielding futures it affects the message passing less than the rescheduling futures. The cost of rescheduling futures is why the yielding futures perform better when there are only a few of them. Even if it is a single CSP-process, the rescheduling of coroutines is more time-consuming than only yielding to reschedule.

Rescheduling quickly becomes more effective, once more CSP-processes are waiting for futures to complete. The point of intersection in both figures show that with as little as two and four futures scheduled, on the Unix and FiPy platforms respectively, it is enough for the rescheduling futures to perform better. Once there are more futures, the constant cost of context-switching into these futures to check for completion becomes higher than the cost of rescheduling them, thus leading to a linear cost of context switching. The rescheduling futures, however, do not require a context-switch, as the process waiting for the future is removed from the run-queue and not scheduled back in until the future is completed.

8.4 List & Deque Benchmark

It is clear from the results of the micro-benchmark shown in Figures 7.3 and 7.3 that the performance in Unix and FiPy platforms are moderately inconsistent.

With focus on the readings from the two platforms executing append, it is apparent that the List and Deque object perform almost equally, with Deques coming slightly ahead in the Unix platform benchmark. Upon further inspection of the results from Table 7.7, it is clear that the Deque object performs over 3.5 times better than the List object on the FiPY platform across all set sizes. FiPy runs on a port of Micropython, and these ports have different implementations and optimizations of common built-in objects, and this may very well be a result of such an optimization. Another explanation for the result is the underlying hardware. While the Deque object has a fixed-size on-initialization-allocated memory space, the List is dynamically size and allocated memory on-demand. It is possible that FiPy platform is slower than the UNIX platform when it comes to memory allocation.

The performance when performing Popleft on both platforms is similar. Popleft is significantly faster on Deque objects. Deque is by design created to have a complexity of $O(1)$ for removing elements at the end or the start of the set, whereas Lists, require all the elements to be shifted by one element in memory when removing the first element of the set, thus resulting in a complexity of $O(N)$, as seen in results from both platforms.

The figures 7.3 and 7.4 show that filter operations work similarly on both platforms. Filter, implemented in Deque as a combination of performing popleft on all elements and appending the elements matching a condition back in the set, results in $O(N)$ complexity and linear performance. Lists, which also displays a linear performance, use an iterator to condition the elements for removal, also at $O(N)$ complexity yields slightly worse performance.

9 Discussion

This chapter intends to discuss non-performance related advantages and disadvantages of the two built libraries based upon experiences gained throughout the development of the thesis contribution.

9.1 Library Dependencies

A large portion of this thesis has been dedicated to reimplementing functionality outside of the PyCSP code-base to create two functioning PyCSP libraries in the form of dependencies. It is necessary to reflect upon the consequences and implications of these dependencies bring.

By reimplementing parts of existing libraries, and building these reimplementations on top of the underlying dependencies as extensions of the dependencies, these extensions become dependent themselves. These extensions, rather than libraries, are prone to errors and bugs. The underlying libraries can introduce changes, altering the behavior intended from the substitute libraries. It is almost inevitable that some underlying mechanisms or parts will change with newer versions, and revisiting the extensions to maintain the functionality is expected if it is supposed to be compatible with future versions of the underlying libraries.

In uPyCSP, the main dependency is a partially implemented version of the

threading library built upon the _thread library. The threading substitute is working as intended in its current state. However, the _thread library is highly experimental, and documentation as to what is implemented and not is inadequate. Any insights into the Micropython specifics of the library is missing. The result of this is uncertainty about the current state, for example, the performance of the threads, and the future state of the library, and thus the substitute built on top of it. We have already experienced, as described in section 8.2, unexpected undocumented results.

Another interesting topic is to what degree substitute libraries should be implemented. As a result of being dependent on functionality from the threading module, only the simplified CFuture version from PyCSP-Lockversion is reimplemented. While it is entirely possible to implement Conditions from the threading library, and thus have both versions of CFutures operational, it is another part of the functionality provided by an extension that has to be maintained.

The uasyncio library used for uaPyCSP was at the start of the thesis, a minimal and undocumented module. This has seen a large update during the course of the thesis. The uasyncio module has been completely rewritten. The consequence of this is that with the intent to use uaPyCSP with the new and improved version of uasyncio, the implementation of replacement functionality requires another look to be compatible with new versions.

While development on Micropython and its library may introduce several defects and bugs in the libraries developed, it is not all bad. Using dependencies is, in most cases, a positive trait and a natural way of separating concerns and create abstractions. Updates frequently increase performance and reduce bugs. Micropython is continuously updated, and more functionality from Python is introduced. Maintaining the implemented PyCSP-libraries and updating its dependencies may remove the need for substituting libraries, resulting in fewer bugs and better performance at little development cost, and it speaks to the increased relevance of the PyCSP-libraries.

9.2 uaPyCSP & Futures

From the evaluation section 8.3 it is pointed out which version of the futures the purely performance-related advantages to each of the future implementations. It is clear that the rescheduling future scales better in CSP-networks, where many CSP-processes are waiting for results using futures. An advantage with the rescheduling future is reducing the CPU-load. Constantly entering yielding futures will result in heavy usage of the CPU, which is an embedded

environment that can be a quite sparse resource, both in terms of computational power and with consideration to power consumption as a side effect of CPU usage.

Another perspective of interest is the compatibility of the future in newer versions of uasyncio. While the rescheduling future makes use of the undocumented "yielding false" feature of uasyncio, this may not be supported in newer versions of uasyncio, as this feature is most likely an implementation-specific artifact rather than intended design. The yielding future which uses a simple yield, will most likely still be compatible, as the yield keyword itself is fundamental to generators, and thus the implementation of uasyncio.

9.3 uPyCSP & Threads

The evaluation of uPyCSP has revealed that using threads in Micropython has a viable performance. Measurements show that the uPyCSP perform better than its Python equivalent in, for example, the Stressed Alternatives benchmarks, and should be considered a viable underlying architecture to achieve concurrency in embedded systems due to sheer performance.

However, as mentioned in 7.3.2, it was discovered in the Stressed Alternatives benchmark, that any attempt to exceed a thread count of over 20 resulted in system failure. This uncovers a severe limitation for the uPyCSP library. Without making any assumption as to the amount of CSP-processes, an embedded system application code usually requires, it is clear that the number of threads possible to use at the same time is rather low.

It may be possible to increase the thread-count. The _thread library documentation for Python states that threads have the possibility of reducing the stack size allocated upon initialization. By reducing this size, the thread consumes a smaller portion of the memory, which could prove to be a solution to the memory-related error raised when attempting to spawn a higher amount of threads. There have been no investigation and experiments to see if the viability of this theory, or if this is even possible using the Micropython threads.

9.4 Relevance of performance

While better performance will always be relevant, it is also worthwhile reflecting on the libraries' use-cases. With the intent of using the implemented modules in embedded hardware, several factors outside of the concurrency manage-

ment are contributing to the performance of the entire application system. The relevancy of the difference in communication overhead may decline, for example, if the application is sending network data with low bandwidth. Both the implementations show promising results performance-wise, and taking into consideration the common use-cases for IoT, the bottlenecks of application code are likely not within the PyCSP libraries.

9.5 Usability

As mentioned in Bjorndalen et al.[4], adapting to aPyCSP and subsequently uaPyCSP, thought as to where to yield execution, is required, due to being cooperative multitasking. In effect, this means that the programmer using the uaPyCSP module will have to consider which points of execution to pause execution. To which degree this potentially convolutes the application code is not determined, but it is noteworthy. uPyCSP, on the other hand, relies on underlying scheduling mechanisms, and after adapting to CSP-like modeling for the process input and output, all signs of concurrency should be gone in abstractions.

One crucial difference between the two libraries are the support for the underlying libraries. uPyCSP requires the intended port of Micropython to implement the _thread module adequately, which it is indirectly dependent of. aPyCSP, on the other hand, depends on uasyncio, which is written entirely in Micropython and can be used independently of Micropython port and is installed through the upip module.

9.6 Channels Queue implementations

As mentioned in section 5.4, one of the limitations of uaPyCSP and uPyCSP is the usage of Deque objects in the Channel queue implementation, as Deque objects require a max length. The implication of using a Deque object is; when using uPyCSP and uaPyCSP, the programmer has to manage the max queue sizes upon channel creation. To determine the size of the queue needed, the programmer will have to put thought to how many CSP-processes are going to communicate using the Channel. A possible solution is to use a dynamic-length object, such as a list object to contain the set of queued processes.

Deque objects in Micropython are missing parts of their original Python implementations. As outlined in section 5.4, the builtin filter method is not possible to use for Deque objects, due to a missing implementation of the `__iter__`

method. The result of this is a more convoluted workaround to be able to achieve the same result. The workaround strays from the original libraries code, whereas an implementation using a List object, would be able to use the filter method, and pose fewer code-changes compared.

Through experiments and evaluation of the results, it is determined that the Deque object is, in all use cases, superior in terms of performance. However, the experiments performed are based on micro-benchmark, which is designed to exaggerate and find the performance differences between the two objects. In regular use of the uPyCSP and aPyCSP library, it is seldom, if even possible, to enqueue the number of processes used in the micro-benchmark, due to process limitations.

The general trade-off by imposing the implementations using List as its mechanism for Channels read and write queues seem to boil down to usability versus performance. By opting for the Deque object, a user-defined size is as mentioned required. PyCSP has, in its design, considered ease of use as an essential factor in decision-making [3]. Posing the requirement that the user has to define the number of processes using a channel introduces complexity to the usage of one of the most critical aspects of the libraries. The performance difference experienced using List as a substitute with an actual CSP application would be minimal and, in most if not all use cases, insignificant.

10 Future work

This section will describe future work that could prove interesting in the context of this thesis. The future work will be described in as much detail as convenient and should not be considered the detailed specification for solutions. The work presented here is intended to build the foundations laid by this thesis's contributions.

10.1 A new version of uasyncio

Parallel to the work in this thesis a uasyncio has had a complete reimplementation. A new version 3.0.0 of uasyncio is now a part of the Micropython standard library from version 1.12 and upwards. This has multiple implications for the uaPyCSP library implemented, which is here presented as further work in terms of research and development.

This new version of uasyncio has new features and bug fixes, providing some of the same functionality as implemented in this thesis. One of the things fixed is the run_until_complete method for the event loop. This thesis made an intrusive change made into the uasyncio library (described in section 5.1.3) to fix a bug experienced. Upon using the new version of uasyncio, this change will no longer have to be applied, and the same functionality is provided out-of-the-box from uasyncio upon installation.

Another significant change in the new version of uasyncio is the Gather method. Gather, as mentioned in section 5.1.2, was not featured in the previous version. A local gather function is implemented as an internal part of uaPyCSP. This can hopefully be replaced with the uasyncio-supplied Gather instead. A concern here is the Channel Poison mechanism not functioning correctly, though this is only a concern and has not been verified.

Future objects are not introduced in the new version of uasyncio, and the implemented future object will have to be re-implemented to fit the new version of uasyncio. How comprehensive this work is, is yet to be determined. With all likeliness, the rescheduling futures will require a change as to how they remove the waiting coroutines from the event loops run queue. However, yielding futures will most likely adapt better, as they use a simple `yield` mechanism, fundamental to uasyncio, as described in section 9.2.

It would prove valuable to see how these two uasyncio versions alter the performance of uaPyCSP and to determine if either sets a better foundation as an underlying architecture for CSP-based concurrency.

10.2 uPyCSP Without a Global Interpreter Lock

Multithreading in Micropython can be done without using the GIL. The GIL is, as mentioned in 3.3.2 a possible bottleneck for threaded applications. An interesting approach would be to attempt a version of uPyCSP where the GIL is released.

The implications of releasing the GIL, is possibly increased performance for threads. If not prohibited by the GIL, threads have the possibility of using the Micropython interpreter in multiple threads at the same time, effectively parallelizing the thread's target code. With the aid of CSP-based concurrency, many of the challenges related to parallelism are already handled by the inherited CSP design, thus making it a suitable setting to attempt to parallelize the CSP processes.

However, the _thread library is experimental, and finding documentation of how to disable the GIL, has proven to be a challenge. Further research into this, could provide the uPyCSP library a whole new modus operandi, working as parallel threads.

10.3 Experimenting with Threads Stacksize

One of the larger limitations in uPyCSP is not being able to spawn a lot of CSP-processes, due to underlying memory limitations when running on a FiPy unit. In section 9.3, it is presented that there may be a possible solution to this problem. By reducing the stack size of the spawned threads, it is less likely that memory consumption reaches its limits, and thus potentially opening up for larger-scale CSP-networks, even on embedded controllers.

Further research into the potential of this could result in a much more performant, and versatile version of uPyCSP.

11

Conclusion

To conclude and summarize the thesis, it can be helpful to reiterate the original thesis statement:

This thesis aims to create an efficient concurrency library, with a familiar and easy to use interface provided by a version of the PyCSP library, aimed for use in a microcontroller using the Micropython environment. Determining which PyCSP version yielding the best result is done through the implementation of two different libraries and approaches, showing and measuring of the advantages and disadvantages of both.

Two versions of the PyCSP has been researched, and successfully implemented in Micropython, and are compatible to be used with microcontrollers. Both the libraries have implemented benchmarks, which both show promising results performance-wise. Evaluating the benchmark has resulted in highlighting specific characteristics for both implementations. Through this evaluation and a discussion reflecting different experiences gained, the thesis has uncovered multiple advantages and disadvantages to each implementation outside of computational efficiency. Both versions have pieces of further work, which could reveal more about the potential of the libraries.

The two libraries showed unique traits. Whereas the uaPyCSP gives the possibility of large networks of CSP-processes, it had the drawback of slower communication speeds while doing so. uPyCSP had issues spawning enough threads but showed impressive speed on a platform where it was able to do so.

With all this in mind, it is hard to determine which one of the implementations is more fitting the description of the thesis statement, as they both have advantages and disadvantages in various essential aspects, and both may very well fit.

Bibliography

[1] S. Lucero *et al.*, "Iot platforms: enabling the internet of things," *White paper*, 2016.

[2] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, p. 666–677, Aug. 1978.

[3] J. M. Bjørndalen, B. Vinter, and O. J. Anshus, "PyCSP-Communicating Sequential Processes for Python.," in *Cpa*, pp. 229–248, 2007.

[4] J. M. Bjørndalen, B. Vinter, and O. J. Anshus, "aPyCSP-Asynchronous PyCSP Using Python Coroutines and asyncio," 2018.

[5] P. Welch, J. Kerridge, and F. Barnes, "Portable csp based design for embedded multi-core systems," in *Communicating Process Architectures 2006: WoTUG-29: Proceedings of the 29th WoTUG Technical Meeting, 17-20 September 2006, Napier University, Edinburgh, Scotland*, vol. 64, p. 123, IOS Press, 2006.

[6] T. W. Barr and S. Rixner, "Medusa: Managing concurrency and communication in embedded systems," in *2014 USENIX Annual Technical Conference*, pp. 439–450, 2014.

[7] T. W. Barr, R. Smith, and S. Rixner, "Design and implementation of an embedded python run-time system," in *Presented as part of the 2012 USENIX Annual Technical Conference*, pp. 297–308, 2012.

[8] C. Hewitt, "Actor model of computation: scalable robust information systems," *arXiv preprint arXiv:1008.1459*, 2010.

[9] "Pep 3156 – asynchronous io support rebooted: the 'asyncio' module." `https://www.python.org/dev/peps/pep-3156/`. Accessed on 2020-25-06.

[10] "Python docs – coroutines and tasks - awaitables." `https://docs.python.org/3/library/asyncio-task.html#awaitables`. Accessed on 2020-25-

06.

[11] A. D. Birrell, *An introduction to programming with threads*. 1989.

[12] "Micropython v1.11 docs – _thread - multithreading support." `http://docs.micropython.org/en/v1.11/library/_thread.html`. Accessed on 2020-20-06.

[13] D. May, "Occam," *ACM Sigplan Notices*, vol. 18, no. 4, pp. 69–79, 1983.

[14] A. A. Donovan and B. W. Kernighan, *The Go programming language*. Addison-Wesley Professional, 2015.

[15] "Python docs – coroutines and tasks - running tasks concurrently." `https://docs.python.org/3/library/asyncio-task.html#running-tasks-concurrently`. Accessed on 2020-29-06.

[16] K. Chalmers, "Development and evaluation of a modern c++csp library," 08 2016.

[17] K. Chalmers and S. Clayton, "Csp for .net based on jcsp.," pp. 59–76, 01 2006.