Communicating Process Architectures 2009
P.H. Welch et al. (Eds.)
IOS Press, 2009

Three Unique Implementations of Processes for PyCSP

Rune Møllegaard FRIBORG [a,1], John Markus BJØRNDALEN [b] and Brian VINTER [a]
[a] *Department of Computer Science, University of Copenhagen*
[b] *Department of Computer Science, University of Tromsø*

Abstract. In this work we motivate and describe three unique implementations of processes for PyCSP: process, thread and greenlet based. The overall purpose is to demonstrate the feasibility of Communicating Sequential Processes as a framework for different application types and target platforms. The result is a set of three implementations of PyCSP with identical interfaces to the point where a PyCSP developer need only change which implementation is imported to switch to any of the other implementations. The three implementations have different strengths; processes favors parallel processing, threading portability and greenlets favor many processes with frequent communication. The paper includes examples of applications in all three categories.

Keywords. Python, CSP, PyCSP, Concurrency, Threads, Processes, Co-routines

Introduction

The original PyCSP [1] implemented processes as threads, motivated by an application domain with scientific users and the assumption that these applications would spend most of their time in external C calls. While the original PyCSP was well received, users often aired two common complaints. First and foremost programmers were disappointed that pure Python applications would not show actual parallelism on shared memory machines, most frequently multi-core machines, because of Python's Global Interpreter Lock. The second common disappointment was the limited number of threads supported, typically an operating system limitation in the number of threads per process, and the overhead of switching between the threads.

In this paper we present a new version of PyCSP that addresses these issues using three different implementations of its concurrency primitives.

PyCSP

The PyCSP library presented in this paper is based on the version of PyCSP presented in [2] which we believe reduces the complexity for the programmer significantly. It is a new implementation of CSP constructs in Python, that replaces the original PyCSP implementation from [1]. This new PyCSP uses threads like the original PyCSP, but introduces four major changes and uses a better and simpler approach to handle the internal synchronization of channel communications. The four major changes are: simplification to one channel type, input and output guards, automatic poisoning of CSP networks and making the produced Python code look more like occam where possible.

[1]Corresponding Author: *Rune Møllegaard Friborg, Department of Computer Science, University of Copenhagen, DK-2100 Copenhagen, Denmark.* Tel.: +45 3532 1421; Fax: +45 3521 1401; E-mail: `runef@diku.dk`.

When we refer to the threads implementation of PyCSP, we are referring to the new PyCSP presented in [2] and referenced in this paper as `pycsp.threads`. This is used as our base to implement the alternatives to threading presented in this paper.

1. Motivation

We have looked at three underlying mechanisms for managing tasks and concurrency: co-routines, threads and processes. Each provide different levels of parallelism that come with increasing overhead. All of them are available in different forms, and in this paper we define them as follows:

Co-routines provide concurrency similar to user-level threads and are scheduled and executed by a user-level runtime system. One of the main advantages is very low overhead.
Threads are kernel-level threads scheduled by the operating system, has a separate execution stack, but share a global address space.
Processes are operating system processes and data can only be shared through explicit system calls.

When programming a concurrent application, it is necessary to choose one or several of the above. If the choice turns out to be wrong, then the application needs to be rewritten. A rewrite is not a simple task, since the mechanisms are very different by design.

Using Python and PyCSP, we want to simplify moving between the three implementations. The intended users are scientists that are able to program in Python and who want to create concurrent applications that can utilize several cores. Python is a popular programming language among scientists because of a simple and readable syntax and the many scientific modules available. It is also easy to extend with code written in C or Fortran and does not require explicit compilation.

1.1. Release of GIL to Utilize Multi-Core Systems

Normally PyCSP is limited to execution on a single core. This is a limitation within the CPython[1] interpreter and is caused by the Global Interpreter Lock (GIL) that ensures exclusive access to Python objects. It is very difficult to achieve any speedup in Python from running multiple threads unless the actual computation is performed in external modules that release the GIL. Instead of releasing and acquiring the GIL in external modules it is possible to use multiple processes that run separate CPython interpreters with separate GILs. In Python 2.6 we can use the new multiprocessing module [3] to handle processes, enabling us to compare threads to processes. The comparison in Table 1 shows the result of computing Monte Carlo pi in parallel using threads and processes.

Table 1. Comparison of threads and multiprocessing on a dual core system with Python 2.6.2

Workers	1	2	3	4	10
Threads	0.98s	1.52s	1.56s	1.55s	1.57s
Processes	1.01s	0.57s	0.54s	0.54s	0.56s

The GIL is to blame for the poor performance for threads illustrated in Table 1. It is possible to obtain good performance for threads, but to do this you must compute in an external module and manually release the GIL. The unladen-swallow project [4] aims to remove the Global Interpreter Lock entirely from CPython.

[1]CPython is the official Python interpreter.

1.2. Maximum Threads Available

On a standard configured operating system, the maximum number of threads in a single application is limited to around 1000. In PyCSP, every CSP process is implemented as a thread. Thus, there can be no more CSP processes than the maximum number of threads. We want to overcome this and give PyCSP the ability to handle CSP networks consisting of more than 100000 CSP processes, by using co-routines.

We thus decided to address these issues by providing two additional implementations, one that provides real parallelism for multi-core machines and one that does not expose the processes to the operating system. All versions should implement the exact same interface, and a programmer should need only to change the code that imports PyCSP to change between the three different versions. Having a common interface for three implementations of PyCSP has another purpose besides being a fast and effective method for changing the concurrent execution platform. It is also an easy method for students to learn what consequences it has to run a specific PyCSP application with co-routines, threads or processes. PyCSP is often chosen by students in the Extreme Multiprogramming Class, which is a popular course at the University of Copenhagen teaching Communicating Sequential Processes [5].

2. Three Implementations of PyCSP

The three implementations of concurrency in PyCSP – `pycsp.threads`, `pycsp.processes` and `pycsp.greenlets` – are packaged together in the `pycsp` module. Although packaged together these are completely separate implementations sharing a common API. It is possible to combine the implementations to produce a heterogeneous application with threads, processes and greenlets, but the support is limited since the choice (Alternation) construct does not work with channels from separate implementations and when communicating between implementations only channels from the processes implementation are supported. The primary purpose of packaging the three implementations in one module is to motivate the developer to switch between them as needed. A common API is used for all implementations making it trivial to switch between them, as shown in Listing 1. A summary of advantages and limitations for each of the implementations are listed at the end of this section.

Listing 1: Switching between implementations of the PyCSP API

```
# Use threads                        # Use processes
from pycsp.threads import *          from pycsp.processes import *
```

When switching to another implementation, the PyCSP application may execute very differently as processes may be scheduled in another order and less fair. Hidden latencies may also become more apparent when all other processes are waiting to be scheduled. In the following sections we present an overview of the implementations in order to understand how they affect the execution of a PyCSP application.

2.1. *pycsp.threads*

This implementation uses the standard threading module in Python, which implements kernel-level threads. All threads access the same memory space, thus when communicating data only the reference to the data is copied. If the data is a mutable Python type it can be updated from multiple threads in parallel, though it is not recommended to do so since it might cause unexpected data corruption and does not fit with the CSP programming model.

Details of `pycsp.threads` are presented in [2] and is a remake of the original PyCSP [1].

2.2. *pycsp.greenlets*

Greenlets are lightweight (user-level) threads, and all execute in the same thread. A simple scheduler has been created to handle new greenlets, dying greenlets and greenlets that are rescheduled after blocking on communication. The scheduler has a simple FIFO policy and will always try to choose the first greenlet among the greenlets ready to run.

The PyCSP API has been extended with an `@io` decorator that can wrap blocking IO operations and run the operations in a separate thread. In `pycsp.threads` and `pycsp.processes`, this decorator has no function while in `pycsp.greenlets` an `Io` object is created. It is necessary to introduce this construct because the greenlets are all running in one thread, and if one greenlet blocks without yielding control to the scheduler, all greenlets in this thread are blocked. For threads and processes, this is not a problem because the operating system can yield on IO and use time slices to interrupt execution, thus rescheduling new threads or processes. Greenlets are never forced to yield to another greenlet. Instead, they must yield execution control by themselves.

Invoking the `__call__` method on the `Io` object will create a separate thread running the wrapped function. After the separate thread has been started, the greenlet yields control to the scheduler in order to schedule a new greenlet. Listing 2 provides an example of how to use `@io`. Without `@io`, the greenlet would not yield, thus blocking all other greenlets ready to be scheduled. This would serialize the processes, and the total runtime of Listing 2 would be around 50 seconds instead of the expected 10 seconds.

Listing 2: Yielding on blocking IO operations

```
@io
def wait(seconds):
    time.sleep(seconds)

@process
def delay_output(msg, seconds):
    wait(seconds)
    print msg

Parallel(
    [delay_output('%d second delay' % i, i) for i in range(1, 11)]
    )
```

Communicating on channels from outside a PyCSP greenlet process is not supported, since the scheduler needs to work on a greenlet process to coordinate channel communication. This means that you can not communicate with the main greenlet at the top-level environment. Calls to `pycsp.greenlets` functions from a `@io` thread will fail for the same reason. Calls to the `pycsp.threads` or `pycsp.processes` implementations are recommended to be wrapped with the `@io` decorator, otherwise they could block the scheduler and cause a deadlock.

2.3. *pycsp.processes*

This implementation uses the multiprocessing module available in Python 2.6+. Processes started with the multiprocessing module are executed in separate instances of the Python interpreter. On systems supporting the UNIX system call `fork`, starting separate Python interpreters with a copy of all objects is trivial. On Microsoft Windows, this is much more challenging for the multiprocessing module, since no equivalent of `fork` is available. The multiprocessing module simulates the `fork` system call by starting a new Python interpreter, loading all necessary modules, serializing / unserializing objects and initiating the requested function. This is very slow compared to `fork`, but it still works in lack of a better alternative for Windows.

When an application is written in pure Python and PyCSP, it is now possible with `pycsp.processes` to utilize multi-core CPUs. For most cases all PyCSP applications will be able to run without any changes, but if the data communicated does not support serialization, the application will fail. An example of such data is an object containing pointers initialized by external modules, fortunately this type of data is not very common in Python applications.

`pycsp.processes` uses shared memory pointers internally and must allocate everything before any processes are forked. For this reason, it might in extreme cases be necessary to tweak a set of constants for `pycsp.processes`. To do this, a singleton Configuration class is instantiated as shown in the example (Listing 3). New constants must be set before any other use of `pycsp.processes`, since everything is allocated on first use.

Listing 3: Example of setting and getting a constant.

```
from pycsp.processes import *
Configuration().set(PROCESSES_SHARED_CONDITIONS, 50)
Configuration().get(PROCESSES_SHARED_CONDITIONS) # returns 50
```

Using this configuration class it is possible to change the size of shared memory and the amount of shared locks and conditions allocated on initialization. The allocated shared memory is used as buffers for channel communication, which means that the size of data communicated on channels at any given time can never exceed the size of the buffer. The default size of the shared memory buffer is set to 100MB, but can easily be increased by setting the constant `PROCESSES_ALLOC_MSG_BUFFER`.

2.4. Summary of Advantages and Limitations

The following is a summary of the advantages (+) and limitations (-) of the individual implementations before moving on to the *Implementation* and *Experiments* section:

Threads:

+ Only references to data are passed by channel communication.
+ Other Python modules usually only expect threads.
+ Compatible with all platforms supporting Python 2.6+.
- Limited by the Global Interpreter Lock (GIL), resulting in very poor performance for code not releasing the GIL.
- Limited in the maximum number of CSP processes.

Greenlets:

+ More optimal switching between CSP processes, since we can limit the context switches to the point where they are blocking. Performance does not decrease with more CSP processes competing for execution.
+ Very small footprint per CSP process, making it possible to run a large number of processes, only limited by the amount of memory available.
+ Fast channel communications ($\approx 20\mu s$).
- No utilization of more than one CPU core.
- Unfair execution, since execution control is only yielded when a CSP process blocks on a channel.
- Requires that the developer wraps blocking IO operations in an `@io` decorator to yield execution to another CSP process.

Processes:

+ Can utilize more cores, without requiring the developer to release the GIL.
- Fewer processes possible than `pycsp.threads` and `pycsp.greenlets`.
- Windows support is limited, because of lack of the `fork` system call.
- All data communicated are serialized, which requires the data type be supported by the pickle module.
+ A positive side-effect of serializing data is that data is copied when communicated, rendering it impossible to edit the received data from the sending process.

3. Implementation

When processes communicate through external choice at both the reading and writing end a number of challenges must be addressed to avoid live-lock and dead-lock problems, this is well researched in [6,7,8]. The PyCSP solution introduces what we believe to be a new algorithm for this problem. The algorithm is very simple and quite fast in the common case.

Every channel has two queues associated with it, one for pending read-operations and one for pending write-operations. Every active choice (Alternation) is represented with a request structure, this request has a lock, to ensure mutual exclusion on changes to the request, an unique id, a status field, and the actual operation, i.e. read or write with associated data. When an Alternation is run a reference to the request structure is added to the queue it belongs to, i.e. input-requests (IR) and output-requests (OR), on every channel in the choice. Then all requests are tested against all potentially matching requests on all involved channels. When a match is found the state of the request structure is changed to `Done` to ensure that the request is matched only once. When the arbitration function comes across an inactive request structure it is evicted from the queue.

Listing 4: The double_lock operation in pseudo code

```
def double_lock(req_1, req_2):
    if req_1.id < req_2.id:
        lock(req_1.lock)
        lock(req_2.lock)
    else:
        lock(req_2.lock)
        lock(req_1.lock)
```

Live-lock is avoided by using blocking locks only, so if a legal match exists it will always be found the first time it is available. Deadlock is avoided by using the unique id of a request to sort the order in which locks are acquired, thus we have an operation, double_lock (Listing 4), that acquires two individual locks in order and returns once both locks are obtained. If two threads attempt to lock the same requests they will always do so in the same order and thus never deadlock.

Listing 5: The arbitration algorithm

```
for w in write_queue:
    for r in read_queue:
        double_lock(w, r)
        match(w, r)
        unlock(w, r)
```

The arbitration algorithm in Listing 5 then performs the protected matching by acquiring locks with the double_lock operation. For every Alternation, read or write action there is exactly one request and this request is always enqueued on the destination channel queues before the arbitration algorithm is run. It may seem unnecessarily expensive at first glance, but it is important to remember that if we do not enqueue the request before matching against potential matches, then there exists a scenario where a read-operation and a matching write-operation may be arbitrated in step-lock without detecting each other. An example of a correctly committed Alternation is shown in figure 1.

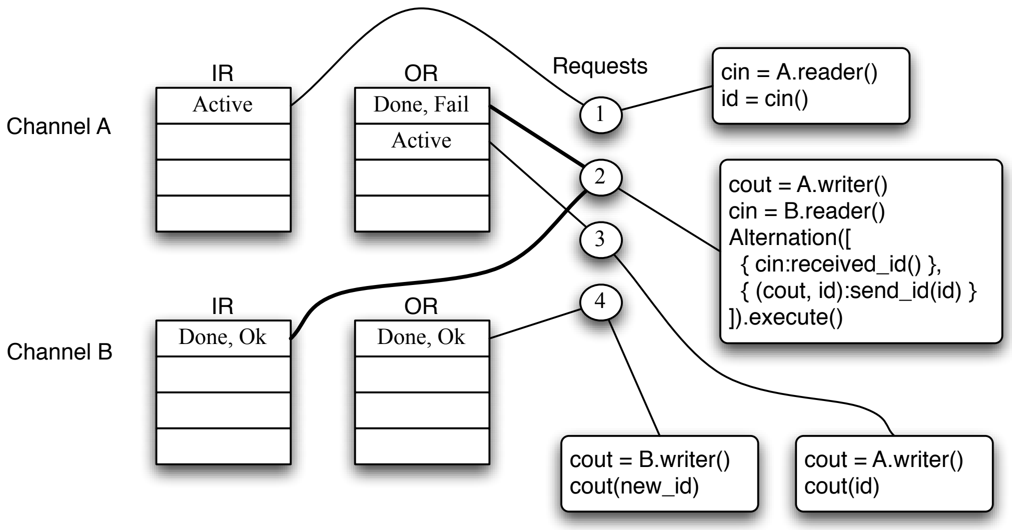


Figure 1. Snapshot of synchronization with two channels and four communicating processes. Channel B has found a match between two request structures; one in the input request queue (IR) and one in the output request queue (OR). Next, channel A will match the two active requests on channel A's request queues.

The presented algorithm for handling synchronization in PyCSP is relevant for `pycsp.threads` and `pycsp.processes`, while the `pycsp.greenlets` does not need this to ensure correctness. The algorithm is a main feature of the new PyCSP, if interested in other features of `pycsp.threads` then the description of these can be found in [2]. Next we will focus on the implementation details for `pycsp.processes` and `pycsp.greenlets`.

3.1. pycsp.greenlets

For co-routines, the greenlet module [9] was chosen because it is a very small module, easy to install, provides full control (no internal scheduler) and allows yielding from nested functions. Python's own generators which make it possible to create a co-routine-like API, do not allow yielding from nested functions, which would not allow us to yield when blocked on a channel communication. Another option was to use Stackless Python [10] for our implementation. Stackless Python was originally based on the greenlet design and has since then matured. It is slightly faster than the greenlet module and allows a larger number of allocated co-routines. However, having to install an extra Python interpreter to make the co-routine implementation run was found unacceptable, leaving the greenlet module as the only valid choice left.

A limitation with co-routines is that everything runs in a single thread, which means that a blocking call will block all other co-routines as well. This is especially a problem with IO operations, since the blocking action might happen in a system call, which we are not able to detect in the Python environment. The `@io` decorator attempts to solve this by wrapping a function into a *run* method on an `Io` thread object. This `Io` thread object is created on-the-fly and yields execution to the scheduler after starting the thread. When the thread's run method finishes, the return value is saved and the calling co-routine is moved onto the scheduler's next queue. Wrapping a function in an `@io` decorator introduces an overhead of starting and stopping a thread. We carried out a test, to see whether this overhead could be minimized by using a thread worker pool. The overhead was found to be similar to the time needed to start and stop a thread, thus the idea of a thread worker pool was abandoned. The idea of delegating a blocking system call to a separate thread was presented by Barnes [11] for the Kent Retargetable occam-π Compiler. occam-π implements a set of channels `keyboard` and `screen` that can be used to communicate to processes reserved for these IO operations. This could also be an option for PyCSP, but it was decided that the `@io` decorator would provide more flexibility for the programmer.

The channel communication overhead is much lower for greenlets than the other two implementations because we can avoid the conditions and locks when synchronizing.

To optimize for fast switching on channel communications, a central queue of blocked greenlets is not used when handling synchronizations. Whenever a greenlet blocks on channel communication, it saves a self-reference together with the channel communication request. Since channel communication requests are located in queues on channels these can be viewed as wait queues, from where a request is matched with an offer for communication. It is now the responsibility of another greenlet that matches this channel communication request to place the blocking greenlet on the scheduler's *next queue*. The scheduler uses a simple FIFO policy, thus choosing the first element of the *next queue* for execution. The *next queue* is usually short as most greenlets will be blocked on channel communication.

Listing 6: Blocking and scheduling a new greenlet CSP process

```
# Reschedule, without putting this process on either
# the next[] or a blocking[] list.
def wait(self):
    while self.state == ACTIVE:
        self.s.getNext().greenlet.switch()
```

When switching, we switch directly from CSP process to CSP process without spending any time on having to switch to a scheduler process. The code in Listing 6 is the wait method, which is executed when a CSP process blocks on channel communication. The method is responsible for scheduling the next CSP process. The `self.s` attribute is a reference to the scheduler, which is implemented as a singleton class. If the next and new queues are empty, then `getNext()` will return a reference to the scheduler greenlet which will then be activated. The scheduler greenlet will then investigate whether there are any current Timeout guards or `@io` threads active. In case all queues are empty it will terminate since everything must have been executed.

3.2. pycsp.processes

Using processes instead of threads requires that we run separate Python interpreters. For fast communication we can choose among several existing inter-process communication techniques, which includes message passing, synchronization and shared memory. Which techniques are available and how they are implemented differs between platforms. In order to have cross-platform support we construct the `pycsp.processes` implementation on top of the multiprocessing module available in Python 2.6. The multiprocessing module presents a uniform method of creating processes, shared values and shared locks. When Python objects are communicated through shared values, they are serialized using the pickle module [12]. Some Python objects cannot be serialized, shared values and locks are two examples of these. This requires us to initialize everything at startup, so that references can be passed to new processes as arguments. A singleton `ShmManager` class maintains all references to shared values and locks. This instance is automatically located in the memory address space of newly created processes.

Every channel instance requires a lock to protect critical regions, and every channel communication requires a condition linked to the channel request offered to processes to ensure that this request is updated in a critical region and can be signaled when updated. This usage of locks and conditions can be a problem when having many processes and channels. The total number of available locks and conditions in shared memory is much lower for the multiprocessing module than for the threading module. The solution was to let the ShmManager class maintain a small pool of shared conditions and locks. The size of the lock pool needs to be large enough to prevent a delay when entering a critical region. Likewise the size of the condition pool should be large enough to avoid waking up to many false processes, causing an overhead in context switches. 20 locks and 20 conditions seem to be enough for most situations possible with `pycsp.processes`, though a small performance increase is possible for the micro benchmark experiments by using more conditions.

Sending data around in a CSP network requires a method to actually transfer data from one process to another. Since all references to shared memory have to be initialized and allocated at startup a message buffer is allocated in shared memory. Unfortunately Python only supports allocating shared memory through the multiprocessing module, thus we will have to handle the memory management in PyCSP by calling get and set methods on objects allocated using the multiprocessing module. A large shared string buffer is allocated and partitioned into blocks of a static size. To handle the allocation of the required number of blocks for a channel communication and freeing them again afterwards, a dynamic memory allocator is implemented. The memory allocator uses a simple strategy that resembles the next-fit strategy:

init A list of free blocks is initialized with one entry that equals the entire message buffer.

alloc Any size is allocated by searching the list of free blocks for an entry that has enough space. The needed blocks are then cut from this entry and an index to the first block is returned.

free Allocated blocks are freed by appending an entry containing the index and size of the free blocks list.

Every new allocation will fragment the message buffer into smaller sections. If at some point we cannot find a partitioned area large enough, a step of combining free blocks is executed. This solution makes it possible to send both large messages and very small messages. If necessary, the buffer and block size can be tweaked using the `Configuration().set()` functionality.

We do not expect this dynamic memory allocator to affect the performance of parallelism in general, even though the allocation of a buffer is protected by a shared lock. The amortized cost of allocating buffers is low, since most allocations will be able to allocate blocks from the first entry in the list of free blocks and while the more rare and expensive action of reassembling blocks introduces a delay, it is a delay that will not affect the overall execution much. In the micro benchmarks (Section 5.1) and in the Mandelbrot experiment (Section 5.3) we successfully communicate small and larger data sizes.

4. Related Work

Communicating Sequential Processes (CSP) was defined by Hoare [5] in 1978, but it is during the last decade that we have seen numerous new libraries and compilers for CSP. Several implementations are optimized for multi-core CPUs that are becoming the de-facto standard when buying even small desktop computers. occam-π [13] and C++CSP2 [14] are two CSP implementations which stand out by being able to utilize multiple cores and use user-level threads for fast context switching. User-level threads are more efficient and provides greater flexibility than kernel-threads. They exists only to the user and can be made to use very little memory. It is possible to optimize the scheduling of threads to fit with the internal priority in the application, because the scheduler is in user code and the operating system is not involved. occam-π implements processes as user-level threads and uses a very robust and optimized scheduler that can handle millions of processes. The utilization of multiple cores is handled automatically by the scheduler and is described in detail in [15]. This is different from C++CSP2, where it is necessary to specify whether processes should be run as user-level threads or kernel-level threads.

Several libraries exist for Python that enable the Python programmer to manage tasks or threads, but they do not enable the programmer to easily change from threads to co-routines. Some of these libraries are Stackless Python [10], Fibra [16] and the multiprocessing module [3] and they provide an abstraction that uses the concept of processes and channels resembling a subset of the constructs available in the CSP algebra. Stackless Python is a branch of the standard CPython interpreter and provides very small and efficient co-routines (greenlets), bidirectional channels and a round-robin scheduler. Fibra is based on Python generators that are similar to co-routines, but it is impossible to hide the fact that a co-routine is a Python generator since the keyword `yield` is the only method to switch between generators. In Fibra, co-routines communicate through tubes by yielding values to a scheduler. The multiprocessing module in Python 2.6 provides a method of using operating system processes, shared memory and pipes for buffered communication. Operating system processes are heavy processes requiring a large amount of memory, but contrary to threads they are not affected by the Global Interpreter Lock (GIL).

However, no libraries exist for Python that provide the functionality of the choice construct that makes it possible to program with non-deterministic behaviour in the communication between processes.

5. Experiments

We have run three different experiments, to show the strengths and weaknesses of the PyCSP implementations. The first experiment consists of two micro benchmarks where one is showing how the implementations handle an increasing amount of processes until reaching the maximum possible amount. The other micro benchmark is showing how well an implementation copes with performing an increasing amount of concurrent communications in a network of static size. After the micro benchmarks, we generate primes using a simple PyCSP application as a case for when it is convenient to be able to switch from threads or processes to co-routines. Finally, a benchmark computing the Mandelbrot set is used to compare speedup on an 8-core system. The Mandelbrot set is computed twice using two different strategies and producing two very different speedup plots. One has the Global Interpreter Lock (GIL) released during computation by computing in an external module and one was computed using the numpy module [17]. All benchmarks are executed on a computer with 8 cores: two Intel Xeon E5310 Quad Core processors and 8 GB RAM running Ubuntu 9.04.

5.1. Micro Benchmarks

The results of these micro benchmarks provides a detailed view of how the implementations behave when they are stressed. The benchmarks are designed with the purpose of measuring the channel communication time including the necessary time required to context switch. Extra unnecessary context switches may be added by the operating system and is related to the PyCSP implementation used.

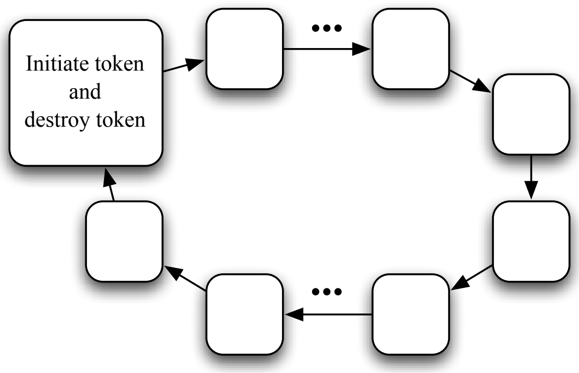


Figure 2. Ring of variable size

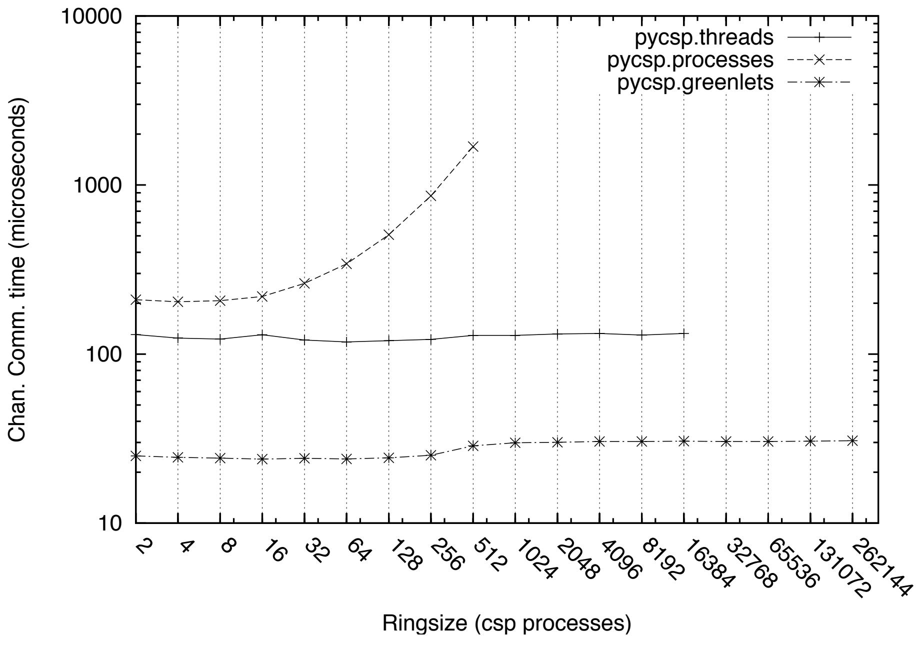


Figure 3. Micro benchmark measuring the channel communication time including the overhead of context switching for an increasing number of CSP processes.

Using the ring design in Figure 2, we run a benchmark that sends a token around a ring of increasing size. The ring benchmark was inspired from a similar micro benchmark in [15]. N elements are connected in a ring and every element passes a token from the previous element to the next. This challenges the PyCSP implementations ability to handle an increasing amount of processes and channels. The time measurements does not include startup and shutdown time and each measured run is divided by the size of the ring to compute an average channel communication time.

The test system has been tweaked to allow a larger number of threads and processes than the default. The results for our test system (in Figure 3) show that we can reach 512, 16384 and 262144 CSP processes depending on the PyCSP implementation used. It is obvious that `pycsp.processes` should only be used for applications with few CSP processes because of the exponential increase in latency, though it is possible to configure `pycsp.processes` using `Configuration().set(PROCESSES_SHARED_CONDITIONS, 50)` and achieve marginally better performance. As expected, `pycsp.greenlets` is able to handle a large number of CSP processes with only a small decrease in performance.

Investigating the performance in a different perspective, we use four rings of static size N and then send 1 to N-1 tokens to circle concurrently. In the previous benchmark there was only one communication at a time, which is a rare situation for an actual application. With this benchmark we see `pycsp.processes` performs much better, since it can now utilize more cores. Based on the results in Figure 4 we can conclude that `pycsp.processes` has a higher throughput of channel communications than `pycsp.threads` when enough concurrent communications can utilize several cores.

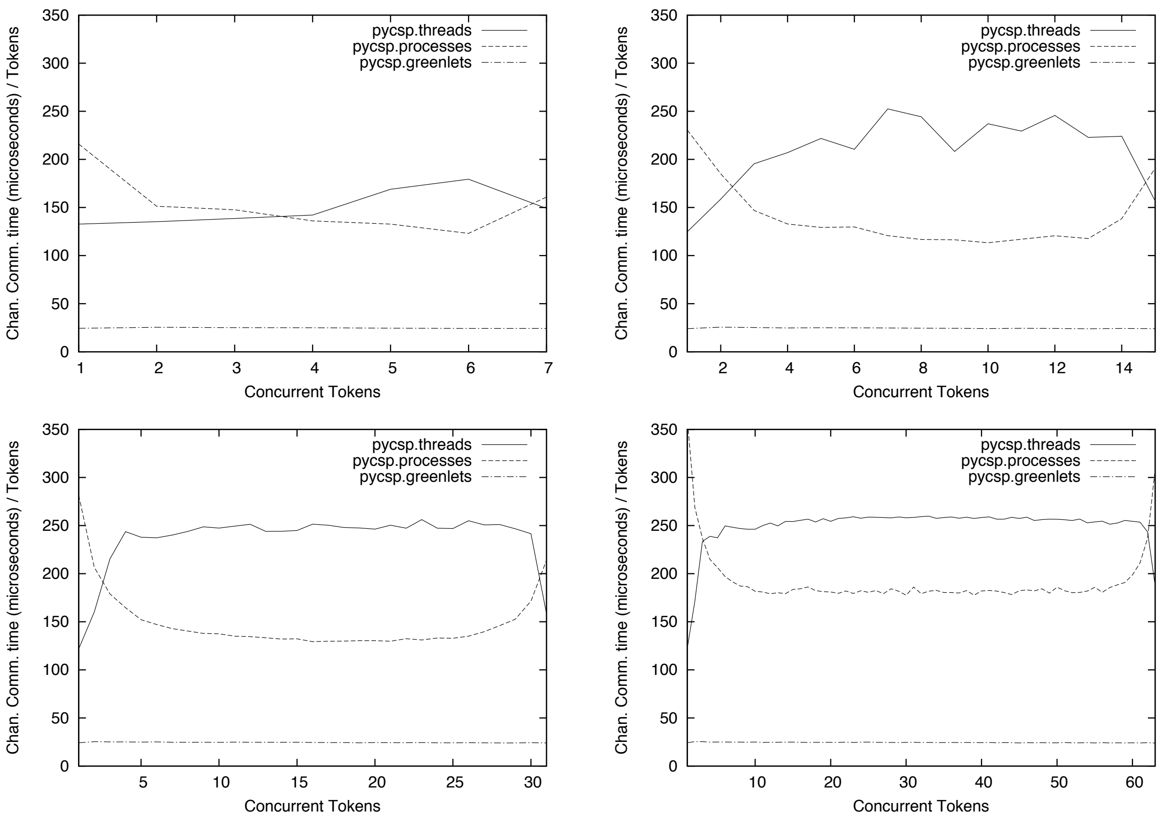


Figure 4. Micro benchmarks measuring the average channel communication time including the overhead of context switching for an increasing number of concurrent tokens in four rings of size 8, 16, 32 and 64.

Looking at the results for the four rings of size N in Figure 4, an interesting pattern is observed whenever the number of concurrent tokens comes close to N. For N-1 concurrent tokens the performance of `pycsp.threads` are almost equal to the performance of one concurrent token. The reason for this behaviour is explained by the blocking nature of CSP, because when all processes but one has a token, then only this one is able to receive. This behaviour mimics the behaviour of the test with one token and explains why the results in Figure 4 are mirrored around the center.

From these micro benchmarks we can see that, `pycsp.threads` performs consistently in both benchmarks. `pycsp.processes` does poorly in Figure 3 where the cost of adding more processes is high, but perform better in Figure 4 where a number of concurrent tokens are added. Finally `pycsp.greenlets` has proved able to do fast switching and many processes, regardless of the amount of concurrent tokens.

5.2. Primes

This is a simple and inefficient implementation of prime number generation found in [18]. The CSP design of the implementation is shown in Figure 5. It adds one CSP process for every computed prime, which sets a limit on how many primes can be calculated using this design. The maximum number of primes equals the maximum amount of CSP processes or channels possible. The latency involved in spawning new CSP processes and performing context switches varies when swapping between threads, processes and greenlets.

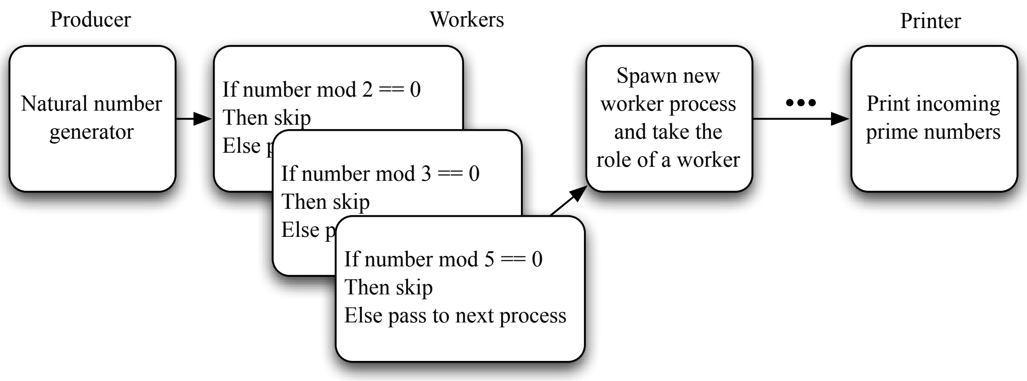

Producer
Workers
Printer
Natural number generator
If number mod 2 == 0 Then skip Else p
If number mod 3 == 0 Then skip Else p
If number mod 5 == 0 Then skip Else pass to next process
Spawn new worker process and take the role of a worker
•••
Print incoming prime numbers

Figure 5. Primes CSP design

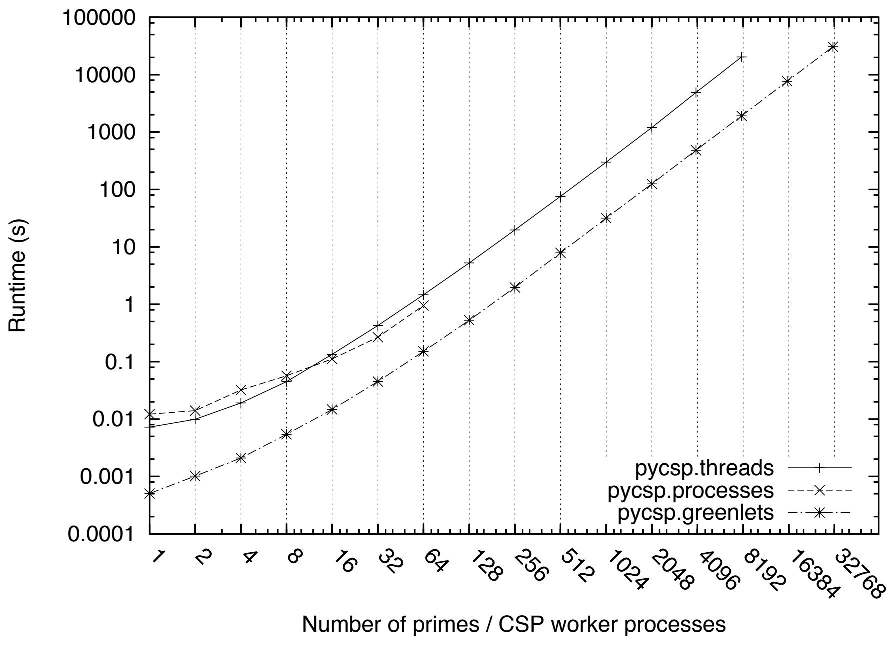


Figure 6. Results of primes experiment.

We run a benchmark computing primes, plotting the runtime results in Figure 6. The processes implementation failed with the message "maximum recursion depth exceeded" after creating 90 processes. This is a limitation in the Python multiprocessing module which is only apparent when spawning new processes from child processes.

This primes benchmark does not compare to a simple implementation in pure Python, which would be orders of magnitude faster than the implementation using PyCSP. This benchmark is meant as a method to compare one aspect of the PyCSP implementations and it proves why greenlets is an important player compared to threads and processes. Running for an entire day (86400s) would produce ≈ 16000 primes using the threads implementation and ≈ 60000 primes using the greenlets implementation. Also 16384 threads is close to an upper limit for threads, while greenlets has no real upper limit on the amount of greenlets.

5.3. Computing the Mandelbrot Set

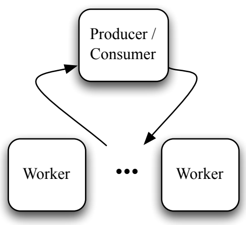
This experiment is a producer-consumer-worker example that tests PyCSP's ability to utilize multiple cores. It produces the image in Figure 7 at a requested resolution. The image requires up to 5000 iterations for some pixels and is located in the Mandelbrot set at the coordinates:

xmin	=	-1.6744096758873175
xmax	=	-1.6744096714940624
ymin	=	0.00004716419197284976
ymax	=	0.000047167062611931696

The simple CSP design in Figure 7 communicates jobs from the producer-consumer to the workers using the Alternation in Listing 7. Workers can request and submit jobs in any order they like.

Listing 7: Producer-Consumer: Delegating and receiving jobs

```
while len(jobs) > 0 or len(results) < jobcount:
    if len(jobs) > 0:
        Alternation([{
                workerIn: received_job,
                (workerOut, jobs[-1]): send_job
            }]).execute()
    else:
        received_job(workerIn())
```



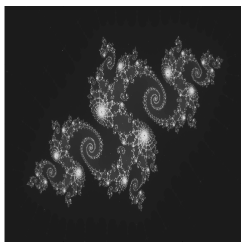


Figure 7. The Mandelbrot CSP design and the computed Mandelbrot set.

The experiment is divided into two different runs. They differ by using two different implementations of the worker process. One releases the GIL during computation by using the ctypes module [19] to call compiled code contained in an operating system specific dynamic library. Executing external code using ctypes is advanced, but does also provide a performance improvement over the other method which is using the numpy module [17] to manipulate and compute on matrices. The numpy module is a package used for scientific computing and provides a N-dimensional array object including tools to manipulate this array object. The numpy module also releases the GIL on every call, but this is much more fine-grained than the course-grained release and acquire used in the ctypes module, thus a larger overhead is expected for the numpy module.

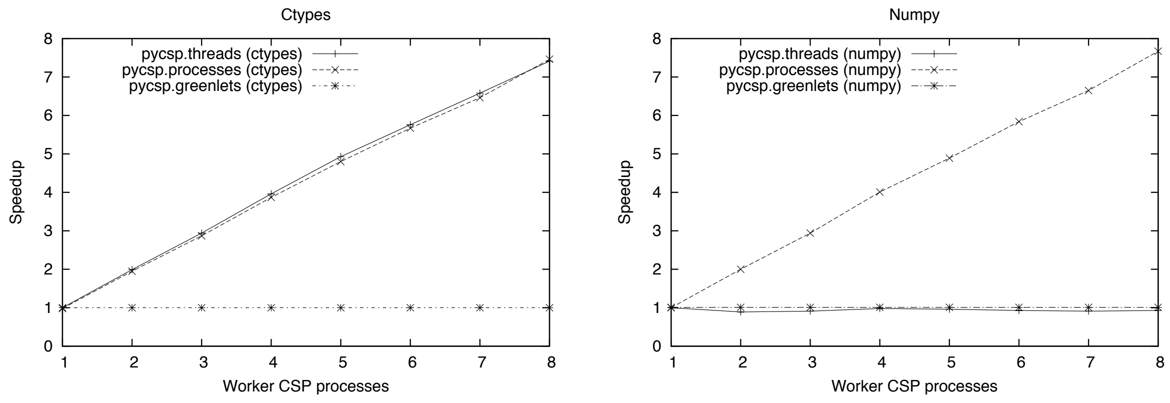


Figure 8. Speedup plots of computing the Mandelbrot set displayed in Figure 7. The resolution is $1000 * 1000$ and the work is divided in 100 jobs. The run time for the case with a single worker is used as the base for the speedup calculation and was 592.5 seconds for the numpy benchmark and 10.6 seconds for the ctypes benchmark.

The results in Figure 8 clearly shows that `pycsp.processes` is superior in this application by attaining a good speedup in both runs. It is interesting that `pycsp.processes` is able to compete with `pycsp.threads` when using the ctypes worker, since `pycsp.processes` for every communication includes an extra overhead of serializing data to a string format, allocating a message buffer, copying the string data to the message buffer, retrieving the string data from the message buffer, freeing the message buffer and finally unserializing the string data into a copy of the original data. As expected we have no multi-core speedup at all from using `pycsp.greenlets`. We could have wrapped the computation in the `@io` decorator and gained a speedup for the ctypes benchmark, but this is not the purpose of the `@io` decorator and would encourage wrong usage of the new PyCSP library.

Based on the experiments performed, the three implementations have different strengths; processes favors parallel processing, threading favors portability and applications that release the GIL and greenlets favor many processes and frequent communication.

6. Conclusions

With the PyCSP version presented in this paper, any application written in Python and using PyCSP can change the concurrent execution model from threads to co-routines or processes just by changing which module is imported. Depending on a user's domain and application a user can choose to circumvent the Global Interpreter Lock by using processes, provided that the application does not create more than the maximum allowed processes for the operating system. Alternatively, a user may want to speed up the communication time by a factor of ten by using greenlets. Then again if the application is changed further and the user suddenly wants to return to using threads, this is a simple task that does not require the user to transfer code changes to an older revision.

Using `pycsp.processes` it is now possible to utilize all cores of an 8-core system without requiring the computation to take place in an external module. This is important for programmers who want to utilize more cores when the performance of `pycsp.threads` is limited by the Global Interpreter Lock. Additionally, running more than 262144 processes in a single PyCSP application is made possible using `pycsp.greenlets`. This amount is smaller than what is possible with occam-π [13] or C++CSP2 [14], but it does open up to the possibility of developing more fine-grained CSP-designs using PyCSP.

PyCSP is available at Google-code using the project name pycsp [20].

6.1. Future Work

The obvious next step would be to create `pycsp.net`, a distributed version of PyCSP that connects processes by networked channels. `pycsp.net` would be required to be fully compatible with the current API, so that any PyCSP application can be transformed into a distributed application, just by changing the imported module. Channels could be given names so that they could be registered on a nameserver and identified from different hosts.

Using `pycsp.net` and running the Mandelbrot benchmark application from the *Experiments* section would allow us to utilize multiple machines. The producer-consumer would be started on one host, and starting additional worker processes on other hosts would be trivial, since they would request the correct channel reference from the nameserver by a known name and automatically start requesting jobs.

References

[1] John Markus Bjørndalen, Brian Vinter, and Otto J. Anshus. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

[2] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP Revisited. In Peter H. Welch, editor, *Communicating Process Architectures 2009*, Amsterdam, The Netherlands. WoTUG, IOS Press.

[3] Python multiprocessing module. `http://docs.python.org/library/multiprocessing.html`.

[4] unladen-swallow distribution. `http://code.google.com/p/unladen-swallow/`.

[5] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[6] Peter H. Welch, Neil C.C. Brown, Jim Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In Steve Schneider, Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 349–370, Amsterdam, The Netherlands, July 2007. WoTUG, IOS.

[7] Peter H. Welch. Tuna: Multiway synchronisation outputs. `http://www.cs.york.ac.uk/nature/tuna/outputs/mm-sync/`, 2006.

[8] Alastair R. Allen, Oliver Faust, and Bernhard Sputh. Transfer Request Broker: Resolving Input-Output Choice. In Frederick R.M. Barnes, Jan F. Broenink, Alistair A. McEwan, Adam Sampson, G. S. Stiles, and Peter H. Welch, editors, *Communicating Process Architectures 2008*, September 2008.

[9] greenlet distribution. `http://pypi.python.org/pypi/greenlet`.

[10] Christian Tismer. Continuations and stackless python. *Proceedings of the 8th International Python Conference*, Jan 2000.

[11] Frederick R.M. Barnes. Blocking system calls in KRoC/Linux. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering Series*, pages 155–178. Computing Laboratory, University of Kent, IOS Press, September 2000.

[12] Python pickle module. `http://docs.python.org/library/pickle.html`.

[13] occam-pi distribution. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.

[14] Neil C.C. Brown. C++CSP2: A many-to-many threading model for multicore architectures. *Communicating Process Architectures 2007: WoTUG-30*, page 23, Jan 2007.

[15] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009.

[16] Fibra distribution. `http://code.google.com/p/fibra/`.

[17] numpy distribution. `http://numpy.scipy.org/`.

[18] Donald E. Knuth. *The Art of Computer Programming - Volume 2 - Seminumerical Algorithms*. Addison-Wesley, third edition, 1998.

[19] Python ctypes module. `http://docs.python.org/library/ctypes.html`.

[20] PyCSP distribution. `http://code.google.com/p/pycsp`.