PyCSP - Communicating Sequential Processes for Python

John Markus Bjørndalen[1], Brian Vinter[2], and Otto Anshus[1]

[1] Department of Computer Science, University of Tromsø
[2] Department of Computer Science, University of Copenhagen

Abstract. The Python programming language is effective for rapidly writing programs and experimenting with them. It is increasingly being used in computational sciences, and in teaching computer science.
CSP is effective for describing concurrency. It has become especially relevant with the emergence of commodity multi-core architectures. We are interested in exploring how a combination of Python and CSP can benefit both the computational sciences and the hands-on teaching of distributed and parallel computing in computer science.
To make this possible, we have developed PyCSP, a CSP library for Python. PyCSP presently supports the core CSP abstractions.
We introduce the PyCSP library, its implementation, a few performance benchmarks, and show example code using PyCSP.
PyCSP has been used in the Extreme Multiprogramming Class at the CS department, university of Copenhagen with promising results.

1 Introduction

Python [1] has become a popular programming language in many fields. One of these fields is scientific programming, where efforts such as SciPy (Scientific Tools for Python) [2] has provided programmers with tools for developing new simulation models, as well as tools for scripting, managing and using existing codes and applications written in C, C++ and Fortran in new applications.

For many scientific applications, the time-consuming operations can be executed in libraries written in lower-level languages that provide faster execution, while automation, analysis and control of information flow and communication may be more easily expressed in Python. To see some examples of current uses and projects, we refer to the 2007 May/June issue of IEEE Computer in Science & Engineering, which is devoted to Python[3].

There are several libraries for Python supporting many communication paradigms, allowing programmers to take advantage of clusters and distributed computing. However, to the best of our knowledge, there is no implementation of the basic abstractions of CSP (Communicating Sequential Processes) [3, 4] for Python. This is the situation that we are trying to remedy with our implementation of CSP for Python: PyCSP[5, 6].

[3] Available on line at `http://www.computer.org/cise`.

PyCSP is under development at the University of Tromsø, Norway, and University of Copenhagen, Denmark. It is intended both as a research tool and as a compact library used to introduce CSP to Computer Science and eScience students. Students may already be familiar with the Python programming language from other courses and projects, and with the support for CSP they get better abstractions for expressing concurrency.

One of the goals of PyCSP is to keep the implementation short and readable so that we can walk students through the main parts of the implementation, not only explaining how the concepts work, but also how the concepts can be implemented. This may help students to understand the concepts more fully and possibly help them to implement and use CSP in other non-CSP languages.

In this article, we use the term *CSP Process* to distinguish a CSP Process implemented using PyCSP from an operating system Process (OS Process).

2 A quick introduction to PyCSP

We refer to [5] for a more full description of PyCSP and some example applications. The syntax of PyCSP has changed slightly since the 2007 paper, however, so we will use the new syntax here. We have also added network channels since the 2007 paper (described below).

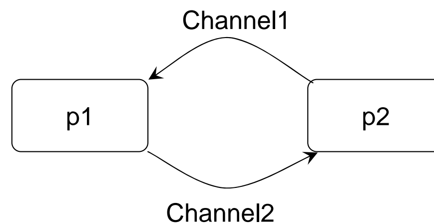

Fig. 1: Basic PyCSP process network, with two processes: P1 and P2. The processes are connected and communicate over two channels: Channel1 and Channel2.

Two central abstractions of CSP and PyCSP are the process and the channel. A running PyCSP program typically comprise several CSP processes communicating by sending messages over channels. Figure 1 shows an example, where two processes, P1 and P2, communicate over two channels: Channel1 and Channel2.

Listing 1.1 shows a complete PyCSP program implementing the process network in Figure 1. A PyCSP process is defined by writing a function prefixed with the `@process` decorator. This ensures that a PyCSP Process is created with the provided parameters when the function is called.

The PyCSP process starts execution within a `Sequence` or `Parallel` construct, as shown in the listing. The `Parallel` construct initiates execution of

Listing 1.1: Complete PyCSP program with two processes

```
import time
from pycsp import *

@process
def P1(cin, cout):
    while True:
       v = cin()
       print "P1, read from input channel:", v
       time.sleep(1)
       cout(v)

@process
def P2(cin, cout):
    i = 0
    while True:
        cout(i)
        v = cin()
        print "P2, read from input channel:", v
        i += 1

chan1 = One2OneChannel()
chan2 = One2OneChannel()

Parallel(P1(chan1.read, chan2.write),
         P2(chan2.read, chan1.write))
```

the provided processes, executing them in parallel, while the `Sequence` objects starts them one by one, waiting for one process to finish before starting the next. Both constructs ensure that all processes have finished before the construct finishes.

Network support: PyCSP uses *named channels* for network communication. A named channel is *hosted* by an OS process by calling `registerNamedChannel(chan,name)`. This registers the object with the networking library and creates an entry in a name server. A remote process can get a proxy to the newly registered channel by calling `getNamedChannel(name)`, which returns a proxy object. The returned Proxy object can be used similarly to local channel objects, and supports passing most Python objects.

3 Concurrency with PyCSP

Python supports kernel threads using the Python *threading* module. The main limitation with this is that the Python runtime is protected with a single lock, the Global Interpreter Lock (GIL). Only one kernel thread can execute Python code at any time, having to release the GIL to allow other threads to execute.

PyCSP uses kernel threads to implement CSP Processes. This places some restrictions on us when it comes to utilizing multicore CPU's and multiprocessor systems, but the restriction may not be as strict as it sounds. Equation solvers and other computationally heavy libraries tend to be implemented in C, C++ and Fortran, while coordination and scripting is done using Python. Libraries that release and claim the lock when leaving and reentering Python respectively

allows for concurrency while doing the heavy-lifting of computations in library calls called from CSP Processes.

For users that write their own computational libraries, the *ctypes* module (included in Python 2.5) automatically releases and claims the GIL for the user and provides a simple interface for using external libraries from Python.

For more fine-grained concurrency, we are currently investigating how to combine PyCSP with occam and C++CSP[7].

4 Conclusions

Early experiences with PyCSP are promising: PyCSP was offered as an option along with occam, C++CSP [7] and JCSP [8–10] in last year's Extreme Multiprogramming Class at the CS department, university of Copenhagen. Several students opted for PyCSP even with the warning that it was early prototype software. No students experienced problems related with the stability of the code however. An informal look-over seems to indicate that the solutions that uses PyCSP were shorter and easier to understand than solutions using statically typed languages.

A simple benchmark in [5] shows that PyCSP performs within a factor 6.5 to 10.5 of the Java JCSP implementation on a simple process switching and channel communication benchmark (commstime).

PyCSP can be downloaded from [6].

References

1. Python programming language home page. `http://www.python.org/`.
2. Scientific tools for Python (SciPy) homepage. `http://www.scipy.org/`.
3. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM, 21(8):666-677*, pages 666–677, August 1978.
4. C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1985.
5. John Markus Bjørndalen, Otto Anshus, and Brian Vinter. Pycsp - communicating sequential processes for python. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, jul 2007.
6. PyCSP distribution. `http://www.cs.uit.no/~johnm/code/PyCSP/`.
7. Neil Brown and Peter Welch. An introduction to the Kent C++CSP Library. *CPA, Communicating Process Architectures*, September 2003.
8. JCSP - Communicating Sequential Processes for Java. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.
9. J.Moores. Native JCSP: the CSP-for-java library with a Low-Overhead CPS Kernel. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering*, pages 263–273. WoTUG, IOS Press (Amsterdam), September 2000.
10. P.H.Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.