PyCSP - controlled concurrency

Vinter, Brian; Friborg, Rune Møllegaard; Bjørndalen, John Markus

Published in:
International Journal of Information Processing and Management

Publication date:
2010

Document version
Peer reviewed version

PyCSP – controlled concurrency

Brian Vinter
Department of Computer Science
University of Copenhagen
DK-2100 Copenhagen, Denmark
vinter@diku.dk

Rune Møllegaard Friborg
Department of Computer Science
University of Copenhagen
DK-2100 Copenhagen, Denmark
runef@diku.dk

John Markus Bjørndalen
Department of Computer Science
University of Tromsø
N-9037 Tromsø, Norway
jmb@cs.uit.no

Abstract

Producing readable and correct programs while at the same time taking advantage of multi-core architectures is a challenge. PyCSP is an implementation of Communicating Sequential Processes algebra (CSP) for the Python programming language, taking advantage of CSP's formal and verifiable approach to controlling concurrency and the readability of Python source code. We describe PyCSP, demonstrate it through examples and demonstrate how PyCSP compares to Pthreads using a benchmark.

1 Introduction

Maintaining scientific codes is a well-known challenge; many applications are written by scientists without any formal computer science or software engineering qualifications, and usually grown "organically" from a small kernel to hundreds of thousands of code-lines. These applications have mostly targeted simple single core systems and have still grown to a complexity where the cost of maintaining the codes is prohibiting, and where the continued correctness of the code is often questionable. This problem is being addressed today by training scientists in some kind of structured program development, however emerging architectures, which are massively parallel and often heterogeneous, may again raise the complexity of software development to a level where non computer scientists will not be able to produce reliable scientific software.

1.1 Motivation

PyCSP [3] is intended to help scientists develop correct, maintainable and portable code for emerging architectures. Python is highly suited for scientific applications. While it is interpreted and thus very slow, scientific libraries efficiently utilize the underlying hardware. CSP provides a formal and verifiable approach to controlling concurrency, fits directly into scientific workflows, and maps directly onto many graphical tools that present scientific workflows such as Taverna[11], Knime[2] and LabView[7].

CPUs are produced with multiple cores today and every announced future CPU generation[12] seems to feature an ever increasing number of cores. As single core performance increase very slowly, researchers are required to exploit this parallel hardware for increased performance. To this end a number of parallel libraries like BLAS and programming tools like Intel Parallel Studio[8] are appearing. Unfortunately parallel libraries are often not enough to achieve acceptable speed and even with advanced tools parallel programming remains a source of added complexity and new bugs in software development.

The intended users for PyCSP are not computer scientists, but scientists in general. It can not be expected that general scientists will learn CSP as formulated by Hoare [6], thus the approach in this paper to controlling concurrency is based on CSP, but does not require any knowledge of CSP. The key elements of controlling concurrency using PyCSP is presented in the *PyCSP* section.

1.2 Related Work

During the last decade we have seen numerous new libraries and compilers for CSP. Several implementations are optimized for multi-core CPUs that are becoming the defacto standard when buying even small desktop computers. Occam-π [9], C++CSP [4] and JCSP [14] are three robust CSP implementations of CSP. C++CSP and JCSP are libraries for C++ and Java, while Occam-π uses CSP inherently in the programming language.

2 CSP

The Communicating Sequential Processes algebra, CSP [6], was invented more than 25 years ago and while it was highly popular and thoroughly investigated in its first years, interest dropped off in the late 1980 because the algebra appeared to be a solution in search of a problem, namely modelling massively concurrent processes and providing tools to solve many of the common problems associated with writing parallel and concurrent applications.

CSP provides many attractive features with respect to the next generation processors; it is a formal algebra with automated tools to help prove correctness, it works with entirely isolated process spaces, thus the inherent coherence problem is eliminated by design, and it lends itself to being modelled through both programming languages and graphical design tools.

Another attractive feature of CSP, which has so far not been investigated, is the fact that it should lend itself towards modelling heterogeneous systems. This is important for the next generation processors since heterogeneity has already been introduced: the CELL-BE processor features two architectures on the core, while the Tesla processors require a classic processor for managing the overall system and the scalar portions of an application.

3 PyCSP

PyCSP provides an API that can be used to write concurrent applications using CSP. PyCSP was introduced in 2007 [3] and revisited in 2009 [13]. The API is implemented in four versions: Threads, processes, greenlets and net. Since all implementations share the same API it is trivial to swap from one implementation to another. Having several implementations sharing one API was presented in [5].

- pycsp.threads - A CSP process is implemented as an OS thread. The internal synchronization is handled by thread-locking mechanisms. This is the default implementation. Because of the Python Global Interpeter Lock, this is best suited for applications that spend most of their time in external routines.
- pycsp.processes - A CSP process is implemented as an OS process. The internal synchronization is more complex than pycsp.threads and is built on top of the multiprocessing module available in Python 2.6. This implementation is not affected by the Global Interpreter Lock[1], but has some limitations on a Windows OS and generally has a larger communication overhead than the threaded version.
- pycsp.greenlets - This uses co-routines instead of threads. Greenlets is a simple co-routine implementation that is bundled together with pylib. It provides the possibility to create 100.000 CSP processes in a single CSP network. This version is optimal for single-core architectures since it provides the fastest communication.
- pycsp.net - A proof-of-concept net implementation of pycsp.threads. All synchronization is handled in a single process. This provides the same functionality as pycsp.threads, but adding a larger cost and a bottleneck by introducing the ChannelServerProcess. It uses Pyro [1] for communication.

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

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

Table 1. Switching between implementations of the PyCSP API

3.1 Processes

A process in PyCSP is an isolated unit of execution, not physically isolated as an operating system process, but by design should not share objects with other processes. A process is specified by using the `@process` decorator as depicted in listing 1.

This creates a increment class that can produce increment process instances. Executing the process is covered in the Concurrency section.

3.2 Networks of Processes

The only allowed methods to communicate between processes are to either pass arguments when creating processes or by sending messages across channels. All communications are blocking operations and are guaranteed to be sent

[1] Python uses a Global Interpreter Lock, the GIL, to protect the interpreter when multiple threads execute Python code. The GIL limits concurrency when executing Python code, but libraries commonly mitigate the problem by releasing the GIL when executing external code.

exactly once and received by exactly one process. A channel can have any number of writing processes and any number of reading processes. It is created using the Channel class.

```
a = Channel('A')
```

Processes are usually passed their input and output channels as parameters which can then be used to communicate with other processes. An example of this usage is shown in listing 1.

```
@process
def increment(cin, cout, inc_value=1):
    while True:
        cout(cin() + inc_value)
```

Listing 1. Example process with IO

To communicate on a channel an application is required to create an input or output end. This is created using the IN and OUT functions, which will return a ChannelEnd object. Requesting a channel end object will also join the actual channel. This adds information to the channel, letting it know how many readers and writers that are connected to it. The number of readers and writers is used to automate poisoning explained in Section 3.5.

To create an increment instance P and provide it with channel ends we do the following:

```
P = increment(IN(a), OUT(b))
```

Reading on an input channel end `cin = IN(a)` is done by invoking `cin()`. Writing msg X on an output channel end `cout = OUT(a)` is done by invoking `cout(X)`.

Calling `IN(a)` or `OUT(a)` on a channel a to create a channel end is usually combined with creating processes, by providing channel ends as arguments to new processes.

3.3 Concurrency

Creating a process will simply instantiate a copy of the process but not execute or start it in any way. A set of processes may be executed using one of three ways, Sequence, Parallel or Spawn. Sequence and Parallel are synchronous and will only terminate once all processes in their parameter list are terminated. Sequence executes the processes one at a time while Parallel executes all the processes concurrently. Spawn starts the set of processes and then returns, one may view it as an asynchronous version of Parallel.

When processes are passed to `Parallel(...)`, they are queued for execution and the Parallel construct will block until all have finished.

```
a,b = Channel('A'), Channel('B')
Parallel(
    counter(OUT(a)), 10),
    increment(IN(a), OUT(b)),
    printer(IN(b))
)
```

Listing 2. Initiating processes for execution

The code in listings 2 completes when the counter, increment and the printer processes has completed. In this case it will never complete. Section 3.5 explains how to end the `Parallel(processes)` execution.

3.4 Nondeterminism

When an input or output channel end is invoked, as explained in Section 3.2, you are committed to this channel until this communication has completed. Using the Alternation class, it is possible to commit to a guard set until exactly one of these is selected.

A guard set is represented as a list of dictionaries where the keys are either input channel ends from which to read or two-tuples where the first entry is an output channel end and the second the value that should be written to the corresponding channel. The value of each dictionary entry is a function of type choice. This function may be executed if the guard becomes true. If the guard is an input guard then the choice function will always have the parameter ChannelInput available which is the value that was read from the channel.

Alternation has two methods

- Execute – which waits for a guard to complete and then executes the associated choice function
- Select - which returns a two-tuple, the guard that was chosen by Alternation and if the guard was an input-guard, the message that was read.

Note that alternation always performs the guard that was chosen, i.e. channel input or output is executed within the alternation so even the empty choice with an alternation execution or a choice where the results are simply ignored, still performs the guarded input or output. An example of alternation usage is shown in listings 3.

```
@choice
def read_action(ChannelInput=None):
    print ChannelInput

@choice
def write_action():
    print 'W'
```

```
@process
def par_in_out(cin1, cin2, cout3, cnt):
    for i in range(cnt):
        Alternation([
            { cin1:read_action },
            { cin2:read_action },
            { (cout3,i):write_action }
        ]).execute()
```

Listing 3. Alternation

The guard types included in the distribution are:

- Channel end input
- Channel end output
- Timeout(seconds) - When expired, it will commit.
- Skip() - At first change it will commit.

The order of guards in a guard set is important. A guard set having a Skip() guard as the first item will always commit to this Skip() guard, thus Skip() is usually used as the last item in a guard set. A usage of Timeout() might be like this:

```
(guard_selected, msg) = Alternation([
        { cin:None },
        { Timeout(seconds=1):None }
    ]).select()

if isinstance(guard_selected, Timeout):
    print 'timeout!'
```

3.5 Termination

A controlled shutdown of a CSP network can be performed by using poisoning [10]. Poisoning of a network may happen in one of two ways, either as an explicit poison which will propagate the entire network instantly and cause a fast termination, or as an incremental retirement which allows all processes to finish their current work before termination. An explicit poison is performed using the `poison(channel/channelend)` call. The less intrusive poison can be performed by using the `retire(channelend)` call. Calling retire will cause a decrement of an internal counter inside a channel. When a `retire()` call causes a channel to have 0 readers or 0 writers left, the channel is permanently retired and any access will cause an exception as described below.

Upon the permanent retirement or poisoning of a channel, all processes that access the channel will raise a `ChannelRetireException()` or `ChannelPoisonException()` respectively. Any following reads or writes on same channel will also raise an exception. Whether this exception is caught inside the process or passed on is left to the programmer. The default behaviour is that the Process class will catch the exception and then, depending on whether it is poisoned or retired, the following occurs: all channels and channel ends in the argument list of the poisoned process are poisoned, thus propagating a poison signal through all known channels. All channel ends in the argument list of the retired process is retired, thus propagating a retire signal through to all known channel ends.

In listings 4 a programmer chooses to catch `ChannelRetireException()` inside a process.

```
@process
def printer(cin):
    try:
        i = 0
        while True:
            print cin()
            i += 1
    except ChannelRetireException:
        print 'Printed', i, 'values'
```

Listing 4. Controlling termination

```
@process
def counter(cout, N):
    for i in range(N):
        cout(i)
    retire(cout)
```

Listing 5. Initiating termination

To initiate termination, we poison the network nicely by calling retire in listings 5. This propagates a retire when all output ends of a channel are retired. Instead of `retire()`, `poison()` could be used which poisons the channel instantly. Poisoning channels or retiring channel ends are a nice way to shut down processes.

4 Examples

Our first example is an application that computes the Mandelbrot set. This example is also used in section 5 to compare PyCSP performance to Pthreads. It consists of a manager process and a number of worker processes. The design is modelled in figure 1. The manager divides the computation into jobs and loops on the Alternation in listing 6, until all jobs have been computed.

```
while jobs or len(results) < jobcount:
    if jobs:
        Alternation([{

            # If selected, a job is read
            # from workerIn
            workerIn:received_job,
```

```
            # If selected, the job at the
            # end of the jobs list is
            # written to workerOut
            (workerOut, jobs[-1]):send_job

        }]).execute()
    else:
        received_job(workerIn())
```

Listing 6. Manager process: Deliver and receive loop

This Alternation handles all synchronization between the manager and the workers. For every loop, one of two things will occur. Either a new job is sent to a worker asking for a job, or a new job result is received from a worker finished computing. When all jobs are computed and received, they are glued together and the computation has finished.

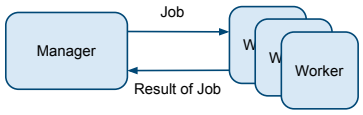

Figure 1. Mandelbrot PyCSP design

The second example is a simple webserver that runs a set of services. In figure 2 we demonstrate how easily this implementation can be mapped into a model consisting of connected processes. A service can register itself by sending an output channel end to the dispatcher. This is then entered into a service dictionary inside the dispatcher. When an incoming request is received it is sent to a matching service if one exists. A popular service might register several processes to handle a greater load. The dispatcher is able to listen for both new services registering and incoming requests by using an Alternation on input guards.

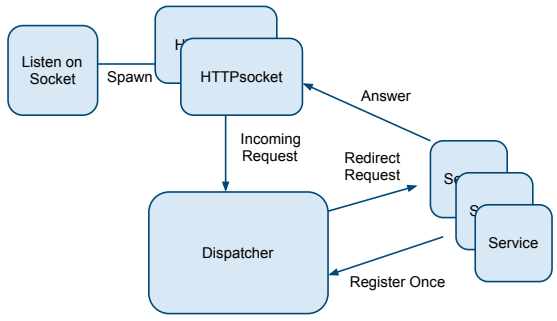

Figure 2. Webserver PyCSP design

An example of a hello world service is shown in listing 7. It performs a self-registering step and then goes into a loop. In this loop requests are received and bundled together with an output channel end. When returning the result for a request, the result is sent on `cout`, which communicates directly to the HTTPSocket process handling the client connection.

```
@process
def HelloWorld(register):
    req_chan = Channel()
    cin = IN(req_chan)
    register(('/hello.html', OUT(req_chan)))
    while True:
        (req_str, cout) = cin()
        cout("Hello World")
```

Listing 7. HelloWorld process

The simple webserver has one bottleneck, the dispatcher. All other processes can be multiplied in numbers, to do loadbalancing. When a request is dispatched to a service, the dispatcher is free to handle other requests and is not required to wait for any services to finish.

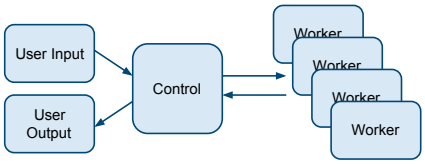

Figure 3. Stochastic Minimum Search PyCSP design

The final example combines processes allocated for user input and output with performing a stochastic search for a local minimum in parallel. The PyCSP design is shown in figure 3 and maps directly to the actual code. A version of the code necessary to perform this search, with a simplified **`gmin`** process, is shown in listing 8. A channel used to update the workers with the function f, going from the master to the workers has been removed in the listed code. This simple network shows how a concurrent interactive application can be made using PyCSP. The **`userin`** process can easily parse inputs from the keyboard and perform actions depending on the input. In this example when the user enters an input, the network is terminated with a poison call.

```
@process
def gmin(f ,chout):
  try:
    while True:
      # Find local minimum of function f
      # from random point
      chout(min)
```

```
@process
def master(kbd, scr, workers_i):
  log = []
  while True:
    Alternation([{
                  kbd:None,
                  workers_i:"""
log.append(ChannelInput)
log.sort()
scr(log[0])
"""
                  }]).execute()

@process
def userin(kbd):
  raw_input("Terminate")
  poison(kbd)

@process
def userout(scr):
  while True:
    print scr()

kbd = Channel()
scr   = Channel()
workers = Channel()
N=10
f= 'x**2+y**2-pylab.cos(18*x)
                      -pylab.cos(18*y)+2'

Parallel(userin(OUT(kbd)),
  userout(IN(scr)),
  master(IN(kbd), OUT(scr), IN(workers)),
  [gmin(f, OUT(workers)) for i in range(N)]
)
```

Listing 8. Stochastic Minimum Search

As long as the application is running the workers will search for a new local minimum, constantly updating the `master` process with new candidates which are printed to the screen by `userout`. Upon termination the poison signal is propagated to all processes and the application exists.

5 Performance

To compare the overhead of using PyCSP with a C program using Pthreads, we run a Mandelbrot benchmark. The benchmark uses the same C function for computing the values of a group of pixels. The PyCSP version calls the C function using the standard Python ctypes library.

Since the computation time for each region of the computed picture is irregular, we use a bag-of-tasks scheme to provide automatic load-balancing. Each worker thread retrieves a task description from a task queue protected with a lock (C version) or a manger process using channels (PyCSP version), computes the pixels requested in that task description, and stores the results before fetching a new task.

The bechmarks are executed on a computer with 8 cores: two Intel Xeon E5310 Quad Core processors and 8GB RAM running Ubuntu 9.04. We use PyCSP version 0.6.0 with Python 2.6.2.

The measured time for each run includes the startup and completion time for the worker threads or PyCSP processes, but not the startup time of the main program.

5.1 Results

Figure 4 shows the speedups of the PyCSP and Pthreads implementations of the benchmark when run using various problem sizes. The number of tasks is kept constant at 100, while the size of the total problem is varied from 10x10 pixels to 2560x2560 pixels.

As expected, the Pthreads version approaches linear speedup earlier than the PyCSP version: the 640x640, 1280x1280 and 2560x2560 problems are close to linear, while in the PyCSP version only the two largest problems are close to linear. The execution time for the largest problem with a single worker is similar in both versions: 49.48 seconds for PyCSP and 49.36 seconds for Phtreads. For 8 workers, the numbers are 6.45 seconds for PyCSP and 6.28 seconds for Pthreads.

The main difference between the versions is the overhead of the interpreted Python language, and also that the PyCSP workers need to wake up and interact with a manager process, while in the Pthreads version, there workers only need to grab and release a lock.

The results shows that for this benchmark, PyCSP and the choices made in the PyCSP solution add a small overhead compared to the Pthreads version, but that the overhead is not overly large: the smallest problem size that approaches linear speedup is 4 times larger for the PyCSP version than the Pthreads version.

6 Conclusions

We have described PyCSP, an implementation of Communicating Sequential Process algebra (CSP) for the Python programming language. PyCSP takes advantage of CSP's formal and verifiable approach to controlling concurrency. The close mapping between the graphical representation of CSP programs and the PyCSP source code makes it easy to compare design documents and implementations, helping programmers manage the complexity that is often introduced when introducing parallel architectures.

In this paper we show that using PyCSP, we can get fairly close to the efficiency of a Pthreads implementation of a Mandelbrot benchmark. The smallest problem size that provides a near linear speedup with PyCSP using 8 CPU

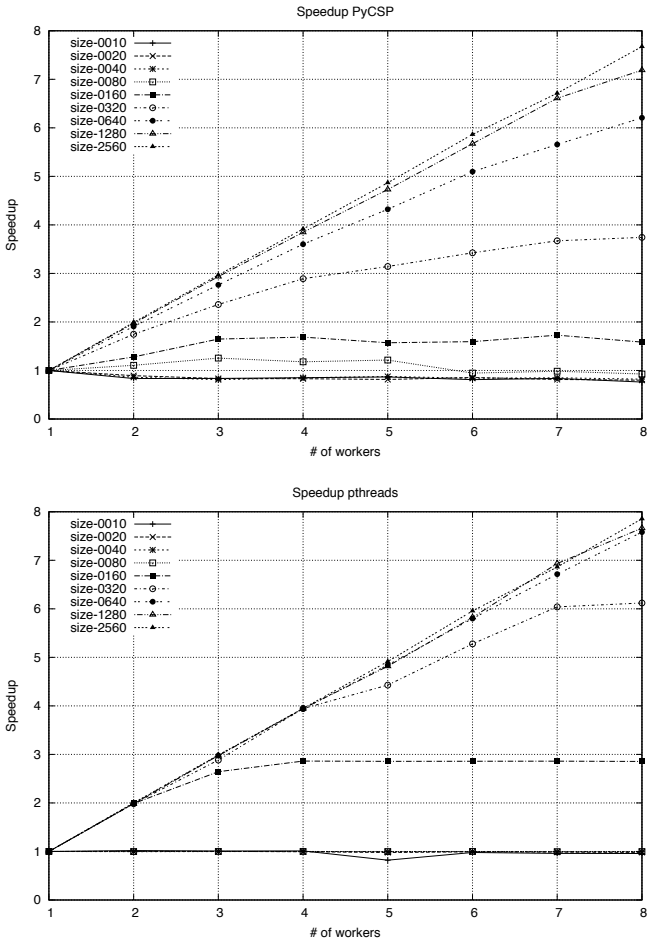

Figure 4. Speedup of PyCSP and Pthreads Mandelbrot computations.

cores is four times the smallest problem size that provide a near linear speedup with the Pthreads version. This is close enough that we believe PyCSP to be usable for scientific computations.

References

[1] Pyro: Python remote objects. http://pyro.sourceforge.net/.

[2] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel. Knime: The konstanz information miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer, 2007.

[3] J. M. Bjørndalen, B. Vinter, and O. Anshus. PyCSP - Communicating Sequential Processes for Python. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

[4] N. C. Brown. C++CSP2: A Many-to-Many Threading. In A. A. McEwan, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–206, jul 2007.

[5] R. M. Friborg, J. M. Bjørndalen, and B. Vinter. Three Unique Implementations of Processes for PyCSP. In *Communicating Process Architectures 2009*, nov 2009.

[6] C. Hoare. Communicating sequential processes. *Communications of the ACM, 21(8):666-677*, pages 666–677, August 1978.

[7] LabView. http://www.ni.com/labview/.

[8] Intel parallel studio. http://software.intel.com/en-us/intel-parallel-studio-home/.

[9] P.H.Welch and F. Barnes. Communicating mobile processes - introducing occam-pi. In *Communicating Sequential Processes*, Lecture Notes in Computer Science, pages 175–210. Springer-Verlag, 2005.

[10] B. H. Sputh and A. R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. *CPA, Communicating Process Architectures*, September 2005.

[11] Taverna Project. http://taverna.sourceforge.net.

[12] B. Vinter. Next generation processes. In B. Topping and P. Ivanyi, editors, *Parallel, Distributed and Grid Computing for Engineering*, Computational Science, Engineering and Technology, pages 21–33. Saxo-Coburg Publications, 2009.

[13] B. Vinter, J. M. Bjørndalen, and R. M. Friborg. PyCSP Revisited. In *Communicating Process Architectures 2009*, nov 2009.

[14] P. H. Welch, N. C. Brown, J. Moores, K. Chalmers, and B. Sputh. Integrating and Extending JCSP. In A. A. McEwan, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369, jul 2007.