Communicating Process Architectures 2007
Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch
IOS Press, 2007

PyCSP - Communicating Sequential Processes for Python

John Markus Bjørndalen [a,1], Brian Vinter [b] and Otto Anshus [a]
[a] *Department of Computer Science, University of Tromsø*
[b] *Department of Computer Science, University of Copenhagen*

Abstract. The Python programming language is effective for rapidly specifying 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. An early prototype of PyCSP has been used in this year's Extreme Multiprogramming Class at the CS department, university of Copenhagen with promising results.

Keywords. CSP, Python, eScience, Computational Science, Teaching, Parallel, Concurrency, Clusters

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[1]. A study of the performance when using Python for scientific computing tasks is available in [3]. Langtangen's book [4] provides an introduction and many examples of how Python can be used for Scientific Computing. More information can also be found at the SciPy homepage [2].

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) [5,6] for Python. This is the situation that we are trying to remedy with our implementation of CSP for Python: PyCSP.

[1]Corresponding Author: *John Markus Bjørndalen, Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway*. Tel.: +47 7764 5252; Fax: +47 7764 4580; E-mail: `jmb@cs.uit.no`.
[1]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.

Early experiences with PyCSP are promising: PyCSP was offered as an option along with occam, C++CSP [7] and JCSP [8,9,10,11] in this 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.

PyCSP can be downloaded from [12].

This paper assumes familiarity with the basic abstractions of CSP, as well as some familiarity of other recent implementations of CSP, such as CSP for Java, JCSP.

eScience

eScience refers to the application of computational methods in natural science. While no formal definition exists, common components in eScience are mathematical modeling, data acquisition and handling, scientific visualization and high performance computing. eScience thus expects no formal computer science training but rather a strong scientific background in general. A consequence of this is that applications for eScience often lack the most basic computer science techniques for ensuring correctness and performance.

Computational methods are becoming pervasive in science, and a growing number of students will need increasing knowledge of sequential, concurrent, parallel, and distributed computing to be efficient in their studies and research. We observe that the knowledge in many areas are lacking, including choice of language, methods for reuse and techniques for parallelization and distribution.

In our experience, the choice by students of which languages to use is typically made based on which languages they already know. The less they know, the worse match there will be between tools and problem. Scientific communities can end up using a programming language because of dependency of older programs they are using and enhancing. The tendency to stay with previously used languages limits how practical it is to use a language better suited to solve the problems at hand.

Java is used in many sciences. The availability of text-books for Java may have contributed to this, but we do not think it is because of its (relatively low) performance or (large) memory footprint [13]. We see extensive use of Perl in biology (see [14,15,16] for some pointers) and C++ in physics, though both languages require an in-depth knowledge of their implementation for efficient, let alone correct, use.

We prefer the use of Python for scientific computing because it is easy to adapt to the problem at hand: it requires little knowledge of the language to use it correctly, the source code is usually relatively short and readable, and through efforts such as SciPy, it seamlessly supports integration of high performance implementations of common scientific libraries.

Finally, multi-core architectures are now becoming the standard and as eScience is insatiable for performance, using multiple cores will soon be the norm in scientific computing. Multi-core architectures, shared memory multi-processors, cluster-computers and even meta-computing systems may easily be utilized by PyCSP simply by changing the run-time environment to match the architectures, without requiring the programmer to rewrite applications or consider the underlying architecture.

Computer Science

Concurrency, distribution and parallelism is often considered to be a hard subject. Educating future computer scientists and programmers, now that increased parallelism appears to become the norm, is a challenge that needs to be met even if we do not yet know which programming models will prevail. Clearly, much research remains to be done [17].

One approach is to educate students by guiding them through hands-on use of the models, and experimentally comparing them. This should provide students with a set of tools and models, aiding them in handling legacy systems on existing architectures, porting legacy systems to new architectures, and creating new systems. Python is a promising language to help us do this.

For more advanced students, Python can be used for the introduction and comparison of concepts, while more specialized languages that focus on given programming models can be used to study the respective models in greater detail. This would give us an opportunity to discuss trade-offs between using specialized languages that may have less library support vs. general purpose languages that need to support the models through libraries.

Systems that are candidates to use include: MPI (Message Passing Interface) [18], which can be taught using systems such as Pypar[19] or pyMPI[20]. To cover Tuple Spaces[21], we have SimpleTS[22]. RMI (Remote Method Invocation) and similar approaches can be taught using Fnorb [23] or Pyro (Python Remote Objects). PATHS [24,25] uses Python internally, and has a Python interface. Multi-threading and shared memory with various approaches to synchronization can be taught using libraries and modules that come with Python.

Most of these approaches and systems have implementations that we can use from Python, but we lack a Python CSP implementation. This is the situation that we are trying to remedy with PyCSP.

Terminology and conventions

We will refer to CSP processes as *processes*, while we refer to user level processes scheduled by the operating system as *OS processes*. Many CSP processes will run inside a (Python) user level OS process.

To reduce the size of code listings in the paper, we have chosen to remove documentation strings and some of the comments in the code listings. The code is instead explained in the text of the paper.

Organization

The paper is organized as follows: Section 1 provides a short introduction to and mini-tutorial of PyCSP. Section 2 describes the implementation of PyCSP and some of the design choices we have made. Section 3 describes some of the eScience applications we are working on. Section 4 present the ever-present commstime benchmark, while future work and conclusions are presented in Sections 5 and 6 respectively.

1. A short introduction to PyCSP

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 shows a complete PyCSP program implementing the process network in Figure 1. A PyCSP process is created by instantiating the Python *Process* class, passing as the

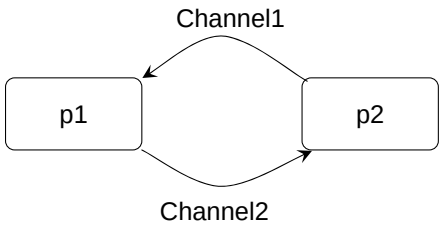

Figure 1 Basic PyCSP process network, with two processes: P1 and P2. The processes are connected and communicate over two channels: Channel1 and Channel2.

Listing 1: Complete PyCSP program with two processes

```
1  import time
2  from pycsp import *
3
4  def P1(cin, cout):
5      while True:
6          v = cin()
7          print "P1, read from input channel:", v
8          time.sleep(1)
9          cout(v)
10
11 def P2(cin, cout):
12     i = 0
13     while True:
14         cout(i)
15         v = cin()
16         print "P2, read from input channel:", v
17         i += 1
18
19 chan1 = One2OneChannel()
20 chan2 = One2OneChannel()
21
22 Parallel(Process(P1, chan1.read, chan2.write),
23          Process(P2, chan2.read, chan1.write))
```

first parameter a function that implements the functionality of the CSP process. The rest of the parameters to the Process constructor are passed directly to the function when the function starts.

A PyCSP channel is created by instantiating one of the channel classes. In the example, we create two One2OneChannels, which are PyCSP channels that can only have one reader and one writer attached[2].

Processes are usually connected in a network by passing channel ends to each of the processes when they are created. In the example, the reading end of channel 1 is passed to process P1, while the writing end of channel 1 is passed to process P2. To the functions implementing the processes, the channel ends appears as ordinary functions passed as parameters (cin and cout in the example).

Accidentally accessing the wrong end of a channel can cause deadlocks, or at the very least lead to incorrect behaviour. To ensure that processes only use the correct ends of channels, PyCSP uses a simple approach: by passing the read and write methods of channels to the corresponding processes, we are certain that processes do not accidentally use the wrong methods. Python makes this trick simple since an object's methods are bound to the object.

[2]This design is inherited from JCSP. Other channel variants exist, and will be described later.

If a PyCSP process only has a reference to the read() method of a channel (and not to the channel object), the read() method is still able to access the internals of the channel.

This is similar to the way channel ends are used in JCSP Network Edition (from Quickstone) and the recently integrated JCSP core and network edition [26]. The main difference is that we don't need to define and implement separate interfaces and functions for returning the channel ends (like JCSP's in() and out()) since we use functionality built into Python.

Our simple program also contains another central CSP abstraction: the Parallel construct. To allow processes to execute in parallel, we have to specify which processes should be executed, and initialize execution within the Parallel construct. In PyCSP, this is done by writing Parallel, and listing all the processes that should be executed in parallel.

The Parallel construct initiates execution of the provided processes, and then waits for the completion of all of them before it finishes.

The output of the program in Listing 1 is as follows[3]:

```
~/pycsp/pycsp-0-1/test> python2.5 simple.py
P1, read from input channel: 0
P2, read from input channel: 0
P1, read from input channel: 1
P2, read from input channel: 1
P1, read from input channel: 2
P2, read from input channel: 2
P1, read from input channel: 3
P2, read from input channel: 3
....
```

1.1. Alternative

Another central abstraction in PyCSP is the Alternative command. One of the basic examples of the use of an Alternative construct is a process that needs to select from a number of input channels depending on which one has data already available for reading. This is done in CSP using a set of *guards*, which have two states: ready and unready. If a guard is ready, the expression associated with that guard can be executed. If several guards are ready in an Alternative statement, one of them is (non-deterministically) selected and the statements being guarded by the selected guard are executed.

PyCSP uses a similar approach to JCSP: a channel's *read()* method can act as a guard. When a number of channel *read()* operations are registered with an Alternative command, the Alternative's *select()* method can be used to detect which of the channels have available input. The following listing is an example:

Listing 2: Alternative example

```
1 ch1 = One2OneChannel()
2 ch2 = One2OneChannel()
3 ...
4 alt = Alternative(ch1.read, ch2.read)
5 ...
6 ret = alt.select()
7 print "Reading from the selected channel:", ret()
```

Two channels are created, and the input methods of the channels are passed to the Alternative construct. An Alternative object is created (`alt`), which can be used to select from the input guards.

[3]Note that PyCSP does not provide any guarantees about output interleaving when using the standard Python 'print' statement.

In PyCSP, the selected guard is returned from the select() call. Thus, since we select from two input guards, it is possible to read directly from the returned object.

1.2. Library Contents

The PyCSP library currently contains the following constructs:

- Channels: One2One, One2Any, Any2One, Any2Any, BlackHole
- Channel Poison
- Alternative
- Guards: Guard, Skip, and input channels
- Parallel and Sequence constructs
- Processes
- Some components based on the JCSP.plugNplay library

2. PyCSP Implementation

PyCSP is implemented as pure Python code, aiming for a portable implementation. This enables us to run PyCSP on devices ranging from mobile phones and embedded devices up to most desktop and high performance computing architectures.

Another goal is to aim for compact and readable code that can be taught to students. We intend to walk students through the implementation without having to spend time on many of the problems created by, for instance, statically typed languages which sometimes tend to be rather verbose and hide some of the abstractions that we try to teach.

We are currently using Python version 2.5, which provides us with some new language features that help us write more compact understandable code. An example is the new *with* statement, which, among other things, simplifies some of the code using locks. Listing 3 shows an example where scaffolding with *try/finally* is replaced with a single *with* statement.

Listing 3: Simplifying lock usage by using the *with* statement

```
1 # old-style lock usage, ensuring that locks are always
2 # released when returning from the code
3 somelock.acquire()
4 try:
5     dosomething
6 finally:
7     somelock.release()
8
9 # alternative Python code using the 'with' statement
10 with somelock:
11     dosomething
```

The *with* statement takes care of acquiring the lock before executing the following code block ("dosomething"). It also takes care of releasing the lock when leaving the code block, even if exceptions are thrown in the code block.

The PyCSP implementation mainly borrows ideas from the JCSP [8] implementation, but also uses ides from C++CSP [7], and two independent CSP implementations for Microsoft's .NET platform (Chalmers [27] and Lehmberg [28]).

2.1. Threads, CSP processes and OS processes

The current implementation uses the Python *threading.Thread* class to implement CSP Processes. Python uses kernel threads to implement multi-threading, which should allow us to draw advantage of multi-core and multi-processor architectures.

The main drawback with this is that Python has a single interpreter lock, restricting the execution of Python byte-code to a single thread at a time. This may not be a major problem for our intended use, since we expect most of the execution time of compute-intensive applications to be spent in C or Fortran libraries, which can release the interpreter lock when executing the library code, allowing multiple threads to execute concurrently.

Another limitation is that we use up a kernel thread for every CSP Process, limiting the number of CSP Processes we can run concurrently since operating systems usually have an upper limit on the number of threads and processes a user can run concurrently.

Both problems can be solved by introducing network channels, which allow us to use PyCSP to communicate between multiple OS Processes on the same host or on multiple hosts in clusters. Network channels is on our agenda (see Section 5).

Kernel threads introduce extra scheduling and synchronization overhead compared to CSP implementations that use fibers or user threads to implement CSP Processes. For the intended applications that are expected to use C or Fortran libraries for compute-intensive and time-consuming tasks, we do not expect the difference to cause any major performance problems. User threads are also likely to introduce extra complexity compared to the current implementation when we try to avoid stalling the rest of the process network when one CSP Process calls a blocking system call or a time-consuming library call.

2.1.1. Synchronization and Python decorators

Python has no *synchronized* keyword, but with recent versions of Python, *decorators* allow us to implement similar functionality. Code listing 4 shows an implementation of a synchronized decorator and its usage in a class. The Python decorators are essentially wrappers around a given method, allowing us to wrap code around existing functions.

We started the project using Python 2.4, where the `@synchronized` decorator took care of the necessary framework of lock acquiring and releasing as well as exception handling with try/finally (as shown earlier in the first part of listing 3). When we decided to use Python 2.5, with its *with* statement, the `@synchronized` decorator was simplified to the code shown in listing 4. The `self._cond` attribute is a standard condition variable from the Python *threading* module.

Listing 4: Python decorator for monitor/synchronized

```
1 def synchronized(func):
2     "Decorator to help create monitors"
3     def _call(self, *args, **kwargs):
4         with self._cond:
5             return func(self, *args, **kwargs)
6     return _call
```

A decorator is applied to a function by prefixing the function with `@decoratorname`, as in the following example where the decorator wraps a method in a class:

Listing 5: Example decorator use

```
1 class Foo:
2    def __init__(self):
3       ....
4    @synchronized
5    def somemethod(self, arg1, arg2):
6       dosomething...
```

Calling *somemethod* in this example, will result in the call being redirected through the *synchronized* decorator function, which handles the lock and then forwards the call to the original function.

Compared to Java, which has *synchronized* built-in, this adds extra code, although not by much. Decorators, however, allow us to use similar techniques to simplify other tasks in the CSP library, such as managing channel poison propagation (see Section 2.4.1).

We are currently evaluating whether the *@synchronized* decorator should be removed in future version of PyCSP. The advantage of keeping it there is that it clearly labels the intention of the programmer, but the drawback is that decorators can only be applied to functions, while the *with* statement can be applied to any block of code.

Another reason for keeping the decorator is that we can insert *@synchronized* before other method decorators, ensuring that the lock is acquired before executing other decorators. This is currently used for channel poisoning.

2.2. Processes

PyCSP processes are encapsulated using the `Process` class, which is a subclass of the Python `threading.Thread` class. Listing 6 shows an implementation of the `Process` class (the full implementation uses the new `run()` function necessary for handling channel poisoning, shown in Section 2.4.1, Listing 11).

Listing 6: PyCSP process implementation.

```
1 class Process(threading.Thread):
2     def __init__(self, fn, *args, **kwargs):
3         threading.Thread.__init__(self)
4         self.fn = fn
5         self.args = args
6         self.kwargs = kwargs
7     def run(self):
8         self.fn(*self.args, **self.kwargs)
```

Rather than creating a new class for each type of process, we have chosen to use the Process class directly as the Process construct in PyCSP. Programmers create a PyCSP process by creating an instance of the Process object, passing as the first argument a Python function that implements the process. The rest of the arguments to the Process object are passed to the function as arguments and keyword arguments. This is similar to one of the methods of creating threads in Python: passing a function to the constructor of the `threading.Thread` class.

The advantage of this is that source code tends to be shorter and clearer than source code where classes have to be made for every type of process. Listing 7 shows an example, where we first define the *Successor* process used in the *commstime* benchmark, and then create a successor process, passing two channel ends to the process.

Listing 7: Process example

```
1 def Successor(cin, cout):
2     while True:
3         cout(cin()+1)
4
5 p = Process(Successor, chan1.read, chan2.write)
```

Although we believe this method to be easier than creating classes for most uses, users may still want to create a new class for some process types. This can be supported either by sub-classing *Process*, or by taking advantage of the fact that Python objects can act as functions: any Python object can behave as a function if it has a `__call__()` method. Any object with a call method can be passed to *Process* in the same way as the Successor function was in listing 7.

A process object does not start automatically after creation. Instead, it exists as a container for a potential execution. To start the execution, a Parallel or Sequence construct is needed[4].

2.3. Parallel and Sequence

Parallel and Sequence have the following straight-forward implementations (see listing 8). Parallel is implemented as a class where the constructor takes a list of Process objects, calls start() on each of the processes (initiating execution of the processes), and then calls join() on each of the processes to synchronize with the termination of all of the processes. The constructor of the Parallel container object returns when the processes in the Parallel construct have terminated.

Sequence is similar, but instead of starting the threads and joining with them, the Sequence constructor calls the run() method directly on each of the processes in the sequence specified by the programmer.

Listing 8: Implementation of Parallel and Sequence

```
1  class Parallel:
2      def __init__(self, *processes):
3          self.procs = processes
4          # run, then sync with them.
5          for p in self.procs:
6              p.start()
7          for p in self.procs:
8              p.join()
9
10 class Sequence:
11     def __init__(self, *processes):
12         self.procs = processes
13         for p in self.procs:
14             p.run()
```

2.4. Channels

Similar to the Chalmers et. als CSP for .NET implementation [27], we protect the user from accidentally using the wrong end of a channel. We do this by passing write and read methods of the channel objects directly to processes. The necessary bits for doing this already exist in the language, so in PyCSP, channel ends are passed to the processes as in listing 9.

Listing 9: Passing channel ends to processes. The read end of a channel is passed to a new process (`bar`).

```
1 def bar(cin):
2     val = cin()
3     print 'Got a value from the channel:', val
4
5 ch = Channel('foo')
6 Parallel(Process(bar, ch.read),
7          ...
```

In Python, the methods of an instantiated object are already bound to the object the methods belongs to. Thus, a function that only has a reference to one of the channel methods

[4]In practice, a user can abuse the fact that the process object is a Python thread, and start it manually with p.start() or p.run(), but this is not the intended use in PyCSP.

can still call the method by treating the reference as an ordinary function. In listing 9, `bar` uses the passed channel input (cin = read) directly by calling the cin function.

The PyCSP channels allow any object to be passed over the channel, including channel ends and CSP processes. This may not be the case for the future network channels as some objects, such as CSP Processes, will be more difficult to pass across a network connection (see Section 5).

PyCSP channels also take a name as an optional argument to the constructor, as in listing 9. Channel names are currently only used for debugging purposes.

The current PyCSP version implements the following channels from JCSP: One2One, Any2One, One2Any, Any2Any, and BlackHole. The One2One and Any2One channels can be used as input guards (see Section 2.5).

2.4.1. Channel Poisoning

PyCSP channels supports *Channel Poison* [29] to aid in terminating a process network. Any process that tries to read or write to a poisoned channel is terminated, and the channels passed to that process upon creation are also poisoned. There is currently no support for automatic poisoning of channels created inside the process, or mobile channels passed between processes.

Poisoning and poison propagation is implemented by adding a poisoncheck decorator around the channel methods (Listing 10). The poisoncheck decorator checks whether a channel is poisoned before and after calls to the channel and throws a ChannelPoisonException if poison is detected. The exception is caught in the `Process` class (specifically in the run() method). The process object then examines the parameters to the process and poisons any channels passed to the process.

Listing 10: Channel poison check decorator

```
def chan_poisoncheck(func):
    "Decorator for making sure that poisoned channels raise exceptions"
    def _call(self, *args, **kwargs):
        if self.poisoned:
            raise ChannelPoisonException(self)
        try:
            return func(self, *args, **kwargs)
        finally:
            if self.poisoned:
                raise ChannelPoisonException(self)
    return _call
```

Listing 11: PyCSP process - adding support for poison propagation.

```
class Process(threading.Thread):
    ....
    def run(self):
        try:
            self.fn(*self.args, **self.kwargs)
        except ChannelPoisonException, e:
            # look for any chan. end methods (only inspect method objects)
            InstMethType = type(self.run)
            for chfn in [x for x in self.args if type(x) == InstMethType]:
                ch = chfn.im_self # get channel object from channel end
                if isinstance(ch, Channel):
                    # Only call poison() on proper channels
                    ch.poison()
```

2.5. Alternative

Alternative in PyCSP follows the implementation in JCSP in principle, but with a few alterations to allow for a more Python-style implementation.

Alternative is a Python class, where the constructor takes a list of guards (see example in listing 12). When the priSelect() operation is called, each of the guards are enabled in turn, as in the JCSP implementation.

Listing 12: Alternative example

```
# assuming that we already have two channel inputs: in1, and in2
sg = Skip()
alt = Alternative(in1, in2, sg)
ret = alt.priSelect()
if ret != sg:
    # Alt did not return the skip guard
    print "Reading from the selected channel:", ret()
```

Contrary to JCSP, the PyCSP Alternative returns a reference to the selected guard, which allows the program to use the guard directly. In the above example, we check the returned object. If it is the skip guard, we ignore the results. Otherwise, we attempt to read from the returned channel.

Listing 13: JCSP Alt example, modified from Regulate demo

```
...
final Skip sg = new Skip();
final Guard[] guards = {in1, in2, sg};        // prioritised order
final int IN1 = 0, IN2 = 1, SG = 2;           // index into guards

final Alternative alt = new Alternative (guards);

switch (alt.priSelect()) {
  case IN1:
    x1 = in1.read();
    break;
  case IN2:
    x2 = in2.read();
    break;
  case SG:
    break;
}
```

The advantage of returning the guard directly, compared to the JCSP example in listing 13, is that the programmer can not mix up the indexes into the provided guard array, and we do not need a switch when the returned guard can be called directly as a function. The latter should be common when selecting from multiple inputs.

When we need to check the identity of the returned guard, the PyCSP code needs to use a series of if- and elif-statements comparing the identity of the returned guard with the guards provided to the Alternative construct. We do not consider this a drawback compared to the JCSP method: Python does not have a switch statement, and a pattern similar to the provided JCSP example would normally be implemented using a series of if- and elif-statements.

PyCSP currently only supports priSelect() which mimics the behaviour of priSelect() in JCSP. Select() is using priSelect() as an implementation of select(). As soon as fairSelect() is implemented, select() will be set to use fairSelect() to mimic the behaviour in JCSP.

2.6. Guards

The guards in PyCSP follows the implementation of JCSP guards. Since our current examples and test-applications have not demanded many of the JCSP guards yet, the current im-

plementation only has two guards apart from the channel input guards: Skip and the Guard base class. Other guards will be added in the future to handle cases such as timeouts.

The One2One and Any2One channels can be used as input guards.

3. Applications

3.1. Radiation Planning

The first eScience application targets planning of stereotactic radiation of brain-tumours. The challenge in the problem is to set up a number of radiation sources in such a manner that it minimizes the amount of energy that is absorbed by healthy brain-tissue while still depositing enough energy within the tumour. The modeling of the radiation is a simple Monte-Carlo simulation of a radiation source where the probability of a ray depositing its energy in any point is proportional to the density of the tissue, or inversely proportional to the light on the CT scan of the brain. The images in Figure 2 are before and after the simulation.

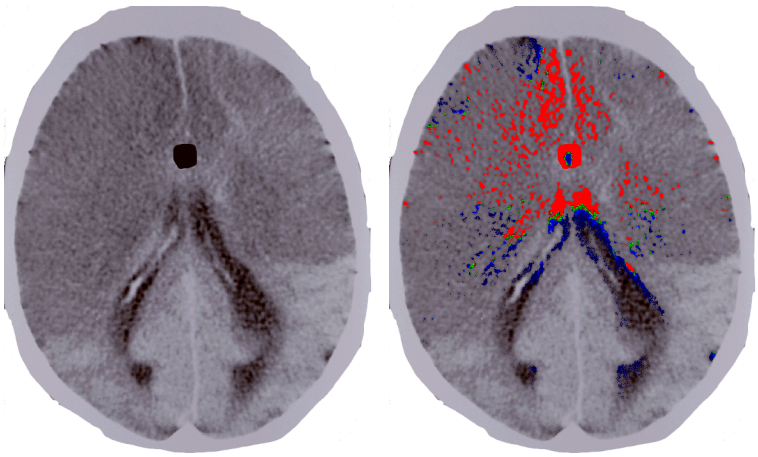

Figure 2 Radiation Planning. CT brain scan Before (left) and after (right) radiation simulation.

Since we have a number of radiation sources, parallelization of the application through CSP has a trivial solution by allowing each radiation source to be simulated in parallel. Unfortunately that approach limits the potential parallelism in the application to the number of radiation sources, thus a more scalable solution was chosen where the radiation sources produces vectors of particles and a number of processes receive these vectors through an any2any channel and traces the particles in the vector through the brain tissue. Applying this approach allows us to reach very high degrees of parallelism, in principle hundreds of millions of ray-tracing processes, since the number of rays that are simulated in real-world scenarios are in the billions.

Figure 3 shows the CSP network used in the application. The code for this setup, including termination process, is shown in listing 14. In the listing, the names of the processes have been shortened to fit in the listing: `source` is the Radiation Source and `trace` is the Ray Tracer.

When a radiation source has finished creating all its particles, it sends a "finished" message on its termination channel (the "c" channel in listing 14). The terminator process waits for all radiation sources to finish, then poisons the channel used to transmit particle vectors to the ray tracers (the "ec" channel). This terminates all ray tracer processes when they attempt to read a new particle vector.

Terminating the network this way is safe, since: a) a radiation source will not terminate until it has safely transmitted all its particles to a ray tracer, and thus, the Terminator will

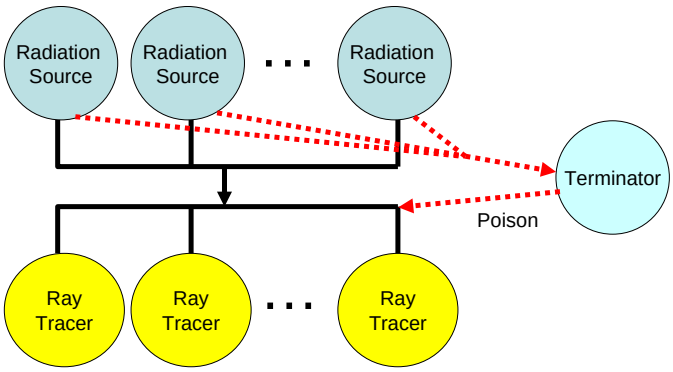

Figure 3 CSP network for parallelizing the brain-tumour radiation simulation. Note that there are usually more Ray-tracers than there are radiation sources.

not poison the channel before all radiation has been transmitted, and b) a Ray tracer process will not be poisoned and terminated until it goes back to read from its input channel after processing the final radiation.

Note that the source code contains unnecessary replication of code. The main reason for this is to provide a simple example. Larger networks of processes could use standard Python list comprehensions to create similar networks with fewer lines of code than this listing.

Listing 14: PyCSP raytrace network

```
def terminator(n, cin, cout):
    for i in range(n):
        cin()
    poisonChannel(cout)

c = Any2AnyChannel()
ec = Any2OneChannel()

Parallel(Process(source,  (85.0,75.0),  (1.0,0.8), 50000, c.write, ec.write),
         Process(source, (10.0,230.0),  (1.0,0.0), 50000, c.write, ec.write),
         Process(source,(550.0,230.0), (-1.0,0.0), 50000, c.write, ec.write),
         Process(source, (475.0,90.0), (-1.0,.75), 50000, c.write, ec.write),
         Process(source,  (280.0,0.0),  (0.0,1.0), 50000, c.write, ec.write),
         Process(terminator, 5, ec.read, c.write),
         Process(trace, c.read),
         Process(trace, c.read),
         Process(trace, c.read),
         Process(trace, c.read),
         Process(trace, c.read),
         Process(trace, c.read))
```

3.2. *Circuit Design*

As an exercise in designing simulated experiments we have another example where digital circuits are build as a networks of CSP processes each functioning as a trivial small Boolean logic gate These gates may by grouped to form more complex components, adders and multiplexers, etc. Even simple circuits includes tens of processes and easily hundreds or even thousands[5].

The circuit design code is straightforward except for wire-junctions, which electrically are trivial, but in a CSP model needs to be handled explicitly by a junction. Thus a full adder need to be set up as in listing 15.

[5]Welch [30] provides examples and a more detailed discussion about emulating digital logic using CSP and occam.

Listing 15: PyCSP circuit design code

```
def Adder(A, B, Cin, S, Cout):
    Aa = One2OneChannel()
    Ab = One2OneChannel()
    Ba = One2OneChannel()
    Bb = One2OneChannel()
    Ca = One2OneChannel()
    Cb = One2OneChannel()
    i1 = One2OneChannel()
    i1a = One2OneChannel()
    i1b = One2OneChannel()
    i2 = One2OneChannel()
    i3 = One2OneChannel()

    Parallel(Process(delta, A.read, Aa.write, Ab.write),
             Process(delta, B.read, Ba.write, Bb.write),
             Process(delta, Cin.read, Ca.write, Cb.write),
             Process(delta, i1.read, i1a.write, i1b.write),
             Process(XOR, Aa.read, Ba.read, i1.write),
             Process(XOR, i1a.read, Ca.read, S.write),
             Process(AND, Ab.read, Bb.read, i2.write),
             Process(AND, i1b.read, Cb.read, i3.write),
             Process(OR, i2.read, i3.read, Cout.write))
```

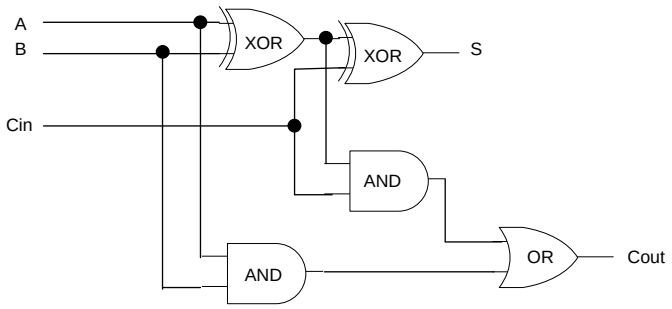

Figure 4 Full-adder diagram, the CSP implementation has 9 processes in it.

3.3. Protein Folding

Protein folding is an extremely hot topic in medical research these days, unfortunately protein folding is extremely computationally demanding and requires a huge supercomputer to fold even the simplest proteins. Luckily the task of calculating protein-foldings is quite well suited for parallel processing.

Proteins are made up of amino-acids, of which there are 20 types. Thus a protein can be viewed as a sequence of amino-acids and folding such a sequence means that the sequence "curls up" until there is a minimum of unbound energy present in the protein. For teaching purpose we need not concern ourselves with the chemistry behind the protein-foldings. Instead we can play with a simplified version of proteins called prototeins – proto-type proteins.

Our simplified prototeins are folded in only two dimensions and only in 90 degree angles. This is much simpler than real three dimensional foldings with angles depending on the amino-acids that are present at the fold, but as a model it is quite sufficient. Our amino-acids are also reduced to two types; Hydrophobic (H) and Hydrophilic (P). When our prototein is folded it will seek the minimal unbound energy, modeled by the highest number of H-H neighborships.

Each folding results in an residual energy level and the solution to a protein-folding problem is to find the folding that has the minimum residual energy level. The actual folding

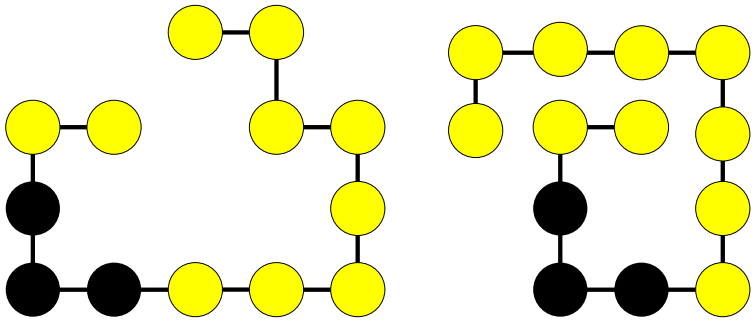

Figure 5 A non-optimal (left) and an improved (right) prototein folding of 13 amino-acids.

is performed as a search-tree of the potential solutions, much like the Travelling-Salesman-Problem, but without the option for branch-and-bound. Thus the CSP solution is well known and implemented as a producer-consumer algorithm.

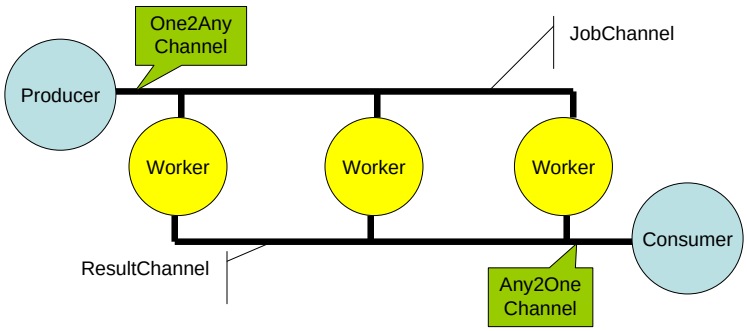

Figure 6 CSP network for handling the prototein folding example.

The code for this model, including a termination process that is not shown in the figure looks as:

Listing 16: PyCSP prototein network

```
1 feeder = One2AnyChannel()
2 collector = Any2OneChannel()
3 done = Any2OneChannel()
4
5 Parallel( Process(producer, protein, map, place, feeder.write),
6           Process(worker, feeder.read, collector.write, done.write),
7           Process(worker, feeder.read, collector.write, done.write),
8           Process(worker, feeder.read, collector.write, done.write),
9           Process(worker, feeder.read, collector.write, done.write),
10          Process(worker, feeder.read, collector.write, done.write),
11          Process(barrier, 5, done.read, collector.write),
12          Process(consumer, collector.read) )
```

3.4. Commstime

The classic *commstime* benchmark [31] is used in many of the recent CSP papers. The source code for the consumer process, written as a PyCSP process, is shown in listing 17. Listing 18 shows the source code for setting up and running the network of processes in the benchmark. The network uses a Delta2 process that is similar to the JCSP SeqDelta2Int process: the process forwards its input to the two output channels in sequence.

The output from the Consumer process is the execution time per communication, computed as time per loop divided by 4, which is reported as "microseconds/communication" in JCSP.

The final line in listing 17 shows an example usage of channel poison, terminating the commstime process network when the consumer process finishes.

Performance results of commstime are shown in Section 4.2

Listing 17: Consumer process

```
def Consumer(cin):
    "Commstime consumer process"
    N = 5000
    ts = time.time     # get a reference to the time function
    t1 = ts()
    cin()              # warmup, only one loop
    t1 = ts()
    for i in range(N):
        cin()
    t2 = ts()
    dt = t2-t1
    tchan = dt / (4 * N)
    print "DT = %f.\nTime per comm : %f/(4*%d) = %f s = %f us" % \
          (dt, dt, N, tchan, tchan * 1000000)
    print "consumer done, poisoning channel"
    poisonChannel(cin)
```

Listing 18: Commstime benchmark

```
def CommsTimeBM():
    # Create channels
    a = One2OneChannel()
    b = One2OneChannel()
    c = One2OneChannel()
    d = One2OneChannel()

    print "Running commstime"
    Parallel(Process(Prefix, c.read, a.write, prefixItem=0),
             Process(Delta2, a.read, b.write, d.write),
             Process(Successor, b.read, c.write),
             Process(Consumer, d.read))
```

4. Experiments

Since we do not yet have network support, and since the execution of Python byte-code is limited to one thread at the time, the potential for parallelism is not very large for our application examples. Thus, we currently only have performance numbers for the commstime benchmark.

The benchmarks were executed on the following hosts, all using Python 2.5:

AMD AMD Athlon 64 X2 Dual-Core 4400+, 2.2GHz, 2GB RAM, Running Ubuntu Linux 6.10 in 32-bit mode. Both cores were enabled.

R360 Dell Precision Workstation 360, Intel P4 Prescott, 3.2GHz, 2GB RAM, with Hypterthreading enabled. Running Rocks cluster distribution of Linux.

R370 Dell Precision Workstation 370, Intel P4 Prescott, 3.2GHz, 2GB RAM, with Hypterthreading enabled. Running Rocks cluster distribution of Linux, in 64-bit mode.

Qtek Qtek 9100 mobile phone, 195MHz TI OMAP 850 processor, 64MB RAM, Windows Mobile 5 operating system.

4.1. Optimization

Python compiles the source code to byte-code and runs the byte-code in an interpreter. Further optimizations of the byte-code can be made with the Psyco [32] Python module, which works similarly to a just-in-time compiler.

Enabling Psyco optimization is as easy as importing the Psyco module and calling one of the optimizer functions in the module:

Listing 19: Using Psyco byte-code optimization

```
1 import psyco
2 psyco.full()
```

According to the Psyco documentation, users often experience speed-ups of 2 to 100 times on Python programs, with a typical result being a 4x speed-up of the execution time.

The benchmarks below are presented with and without Psyco optimizations for the two machines running in 32-bit mode. There is no Psyco-support for 64-bit Linux or for the Qtek mobile phone, so Psyco-optimization experiments were not tried on these machines.

4.2. Commstime

The commstime benchmark was executed with N set to 5000 on all hosts (see Listing 17), with the exception of the Qtek mobile phone, where it was set to 500 due to the slower CPU in the phone. The reported numbers in Table 1 are the minimum, maximum and average of 10 runs of the commstime benchmark.

In addition, we ran the JCSP benchmark on the AMD machine, using JCSP 1.0rc7 with Sun JDK 1.5-06. For the JCSP experiments, we specified that we wanted sequential output from the Delta process (using SeqDelta2Int) rather than parallel output. The reported results are the minimum, maximum and average of the "microseconds / communication" output from 20 runs of the benchmark. No errors or spurious wakeups were reported by commstime.

Implementation	Optimization	min	max	avg
AMD, PyCSP		74.78μs	88.40μs	84.81μs
AMD, PyCSP	Psyco	48.15μs	54.91μs	52.67μs
R360, PyCSP		141.67μs	142.51μs	142.09μs
R360, PyCSP	Psyco	89.50μs	91.57μs	90.37μs
R370, PyCSP		128.14μs	129.12μs	128.61μs
Qtek mobile phone, PyCSP		6500μs	6500μs	6500μs
AMD, JCSP, w/SeqDelta		6μs	9μs	8.1μs

Table 1 Commstime results

There is clearly an advantage in running Psyco to optimize the Python byte code: a factor 1.6 improvement in the AMD case, and a factor 1.57 in the R360 case. This is lower than the improvement that the Psyco developers claim is common, but some of the explanation for this may be that a large fraction of the commstime execution time is outside the reach of Psyco: C library code for locks, system calls and operating system code.

There is also a significant difference in execution time favoring the AMD multi-core processor compared to the Intel Hyperthreading processors. We do not know the reason for this yet, as there are several factors changing between the processors: Hyperthreading vs. multi-core, AMD vs. Intel P4 Prescott implementation of instruction sets, memory buses, and Linux distributions with different kernel versions.

Comparing the "average" column for PyCSP and JCSP, we see that PyCSP without Psyco is about an order of magnitude slower than JCSP, and PyCSP with Psyco is about 6.5 times slower than JCSP. This is within the range that we expected. PyCSP is not intended for fine granularity CSP networks where a significant part of the time is spent communicating. It is intended for reasonable CSP performance in applications where most of the computation time is spent in C or Fortran library code. In that sense, commstime is the worst-case benchmark: it stresses the code that we expect to spend the least amount of time to check that PyCSP does not introduce unreasonable communication overhead.

The experiments show that PyCSPs channel communication overhead is not prohibitive for scientific applications.

5. Future work

Network support is an important addition to PyCSP since this will allow us to make use of clusters. It may also improve utilization of multi-core architectures, remedying some of the problems with Pythons Global Interpreter Lock (GIL)[6] since we can run multiple OS processes on the same host, each hosting a set of PyCSP processes.

Initial prototyping of network support is likely to use Pyro (Python Remote Objects) to speed up development efforts, keep the code small, and to allow us to identify issues and potential implementation techniques. We do not expect to follow the JCSP or C++CSP implementations too closely, since we are hoping that Python will allow us to express some of the ideas in a more compact way.

One of the problems we are likely to encounter is how to handle mobile processes and mobile channels, or whether we should allow them in the first place. Passing processes and channels over channels within the same Python process is not a problem, since channels can essentially pass any object, and passing a PyCSP process across a channel would not influence the execution of the PyCSP process.

Moving PyCSP processes across network channels is not as simple though. Migration could be handled by suspending the PyCSP Process, passing the state across a network channel, and restarting the state in another OS Process. There are two complicating factors however: the first is that we expect users to make use of C and Fortran libraries, and we have no control of pointers and references used by those libraries. The second factor is that we are using kernel threads to implement PyCSP processes. Suspending a PyCSP Process by suspending the Python kernel thread executing it and handling potential problems with locks held by the thread, open files and other objects may prove difficult, save in the most trivial cases. Thus, it may in fact be impossible to migrate PyCSP processes to another address space in a safe way.

The same might end up being a problem with channels and channel ends: waiting queues for locks are difficult to migrate in a safe way if they are maintained by the operating system. Migrating a channel reference across to another address space, however, should be safe if we ensure that any access to the referenced object is forwarded back to the home node of the channel.

An alternative approach is to introduce remote evaluators and code execution. With Python and Pyro, we can pass expressions (as text strings), functions (as objects), classes and even entire Python modules across the network to remote Python processes, and have the remote Python process evaluate and execute the provided code. We have used this in other projects, and it may be a viable alternative to moving processes across address spaces.

[6]We have seen several questions and discussions about removing the GIL over the years, but it appears that the GIL is here to stay for the foreseeable future. For more information, please see the Python Frequently Asked Questions on the Python homepage.

Further development to support more constructs from the core JCSP and plugNplay libraries are also underway.

6. Conclusions

In this paper we have presented the preliminary results from working on integrating CSP in the standard Python model. PyCSP does not seek to be a high-performance CSP implementation but, like Python itself, seeks to provide an easy and transparent environment for scientists and students to work on models and algorithms. If high performance is needed, it may be achieved through the use of native-code libraries, and we do not envision CSP used at that level.

We believe that we have shown how scientists may easily model experiments similar to physical setups by using CSP enabled processes as black-boxes for more complex experiments. The advantage of CSP in this context becomes the isolation of private data and thus elimination of race-conditions and legacy dependencies that may come from using an object oriented model for scientific computing. While performance is not key to this work, we have shown that with commstime round-trip as low as $50\mu s$, the overhead of using PyCSP will not become prohibiting to using the model for real scientific computations.

The presented version of PyCSP is still work in progress and significant changes may still be applied. However, future developments will be directed towards portability, scalability, network support, and usability rather than performance and "feature-explosion".

Early experiences with PyCSP are promising: PyCSP was offered as an option along with occam, C++CSP [7] and JCSP [8,9,10] in this 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.

PyCSP can be downloaded from [12].

References

[1] Python programming language home page. `http://www.python.org/`.

[2] Scientific tools for Python (SciPy) homepage. `http://www.scipy.org/`.

[3] Xing Cai, Hans Petter Langtangen, and Halvard Moe. On the performance of the python programming language for serial and parallel scientific computations. *Scientific Programming, Vol 13, Issue 1, IOS Press*, pages 31–56, 2005.

[4] Hans Petter Langtangen. *Python Scripting for Computational Science, 2nd Ed.* Springer-Verlag Berlin and Heidelberg Gmbh & Co., 2005.

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

[6] C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1985.

[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.

[11] P.H.Welch, J.R.Aldous, and J.Foster. CSP networking for java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002.
[12] PyCSP distribution. `http://www.cs.uit.no/~johnm/code/PyCSP/`.
[13] Ronald F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and numerical computing. *Computing in Science and Engineering, Volume 3, Issue 2*, pages 18–24, 2001.
[14] BioPerl. http://www.bioperl.org/.
[15] Perl for Bioinformatics and Internet. http://biptest.weizmann.ac.il/course/prog/.
[16] James Tisdall. *Beginning Perl for Bioinformatics*. O'Reilly, 2001. ISBN 0-596-00080-4. Also see http://www.perl.com/pub/a/2002/01/02/bioinf.html.
[17] Krste Asanovic et. al. The Landscape of Parallel Computing Research: A View from Berkeley, 2006. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences University of Californai at Berkeley.
[18] MPI: A Message-Passing Interface Standard. *Message Passing Interface Forum*, March 1994.
[19] Pypar software package. `http://datamining.anu.edu.au/~ole/pypar/`.
[20] pyMPI software package. `http://pympi.sourceforge.net/`.
[21] N. Carriero and D. Gelernter. Linda in Context. *Commun. ACM*, 32(4):pp. 444–458, April 1989.
[22] SimpleTS - Tuple Spaces implementation in Python, John Markus Bjørndalen, unpublished. Source code available at `http://www.cs.uit.no/~johnm/code/`.
[23] Fnorb software package. `http://fnorb.sourceforge.net/`.
[24] John Markus Bjørndalen, Otto Anshus, Tore Larsen, and Brian Vinter. PATHS - Integrating the Principles of Method-Combination and Remote Procedure Calls for Run-Time Configuration and Tuning of High-Performance Distributed Application. In *Norsk Informatikk Konferanse*, pages 164–175, November 2001.
[25] John Markus Bjørndalen, Otto Anshus, Tore Larsen, Lars Ailo Bongo, and Brian Vinter. Scalable Processing and Communication Performance in a Multi-Media Related Context. *Euromicro 2002, Dortmund, Germany*, September 2002.
[26] Peter H. Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, jul 2007.
[27] Kevin Chalmers and Sarah Clayton. CSP for .NET Based on JCSP. *CPA, Communicating Process Architectures*, September 2006.
[28] Alex A. Lehmberg and Martin N. Olsen. An Introduction to CSP.NET. *CPA, Communicating Process Architectures*, September 2006.
[29] Bernhard H.C. Sputh and Alastair R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. *CPA, Communicating Process Architectures*, September 2005.
[30] P.H. Welch. Emulating Digital Logic using Transputer Networks (Very High Level Parallelism = Simplicity = Performance). In *Proceedings of the Parallel Architectures and Languages Europe International Conference*, volume 258 of *Springer-Verlag Lecture Notes in Computer Science*, pages 357–373, Eindhoven, Netherlands, June 1987. Springer-Verlag. sponsored by the CEC ESPRIT Programme.
[31] Fred Barnes and Peter H. Welch. Prioritised Dynamic Communicating Processes - Part I. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 321–352, sep 2002.
[32] Psyco optimizer for Python. `http://psyco.sourceforge.net/`.