# Hydra: a Python Framework for Parallel Computing

Waide Tristram

Karen Bradshaw

3rd November 2009

RHODES UNIVERSITY

*Where leaders learn*

sponsored by

COMVERSE

STORTECH For Your Information

amatole telecommunication services

Telkom

THRIP TECHNOLOGY AND HUMAN RESOURCES FOR INDUSTRY PROGRAMME

MARS TECHNOLOGIES

openVOICE

tellabs

Bright Ideas® Projects 39

# Hydra in ½ hour

- An Opportunity
- Why Python and CSP?
- Aim
- Approach
- Framework
- Results
- Conclusions

# An Opportunity

- Desktop and Server CPUs have changed quite considerably over the last few years
- No longer a race for GHz
- Shift to multi-core CPUs
- Main drawback is the difficulty involved in writing concurrent software able to make use of these parallel CPUs
- Performance gains aren’t automatic when adding more cores
  - Developers need to explicitly code concurrency into their software to benefit from multiple processors
  - Tools and frameworks are required to ease the process

# Python ?

- Python is a good candidate for such a framework
  - Powerful built-in data types
  - Extensive and powerful libraries
  - Supports multiple programming paradigms
  - Increased use in scientific computing

    SciPy, NumPy, BioPython

- Suffers from some concurrency limitations
  - Global Interpreter Lock – single thread at a time
  - Affects modules based on Python’s threading module
  - Multiple Python interpreter processes can bypass this
  - Co-ordinating multiple Python interpreters is tricky

# CSP ?

- Message-passing model good start
- CSP provides key constructs for developing programs based on the message-passing
- Several CSP implementations exist for modern languages such as Java and C/C++
- CSP implementation for Python, PyCSP, is limited by the GIL (newer versions address this)
- Current CSP implementations require the programmer to convert CSP algorithm into the appropriate form

# So ....

- Investigate the feasibility of a concurrent framework for Python that overcomes the GIL based on the original CSP notation

- Develop prototype framework that:
  - provides concurrent programming functionality for Python based on CSP constructs
  - properly harnesses power of multi-processor systems
  - provides a high level approach instead of requiring that CSP algorithms be manually converted

# Approach

- Identify or develop suitable grammar
- Select a suitable compiler generator
- Identify suitable existing libraries to form the base of the framework
- Develop the parser and code generator for the grammar
- Basic testing

# Approach - Grammar

- Grammar was developed as a modified version of the original CSP notation
- Novel syntax chosen over an existing machine readable syntax such as that used by FDR
  - Can keep the language small – prototype
  - Allows for the incorporation of Python expressions
  - Reduce parser complexity

# Approach - Grammar

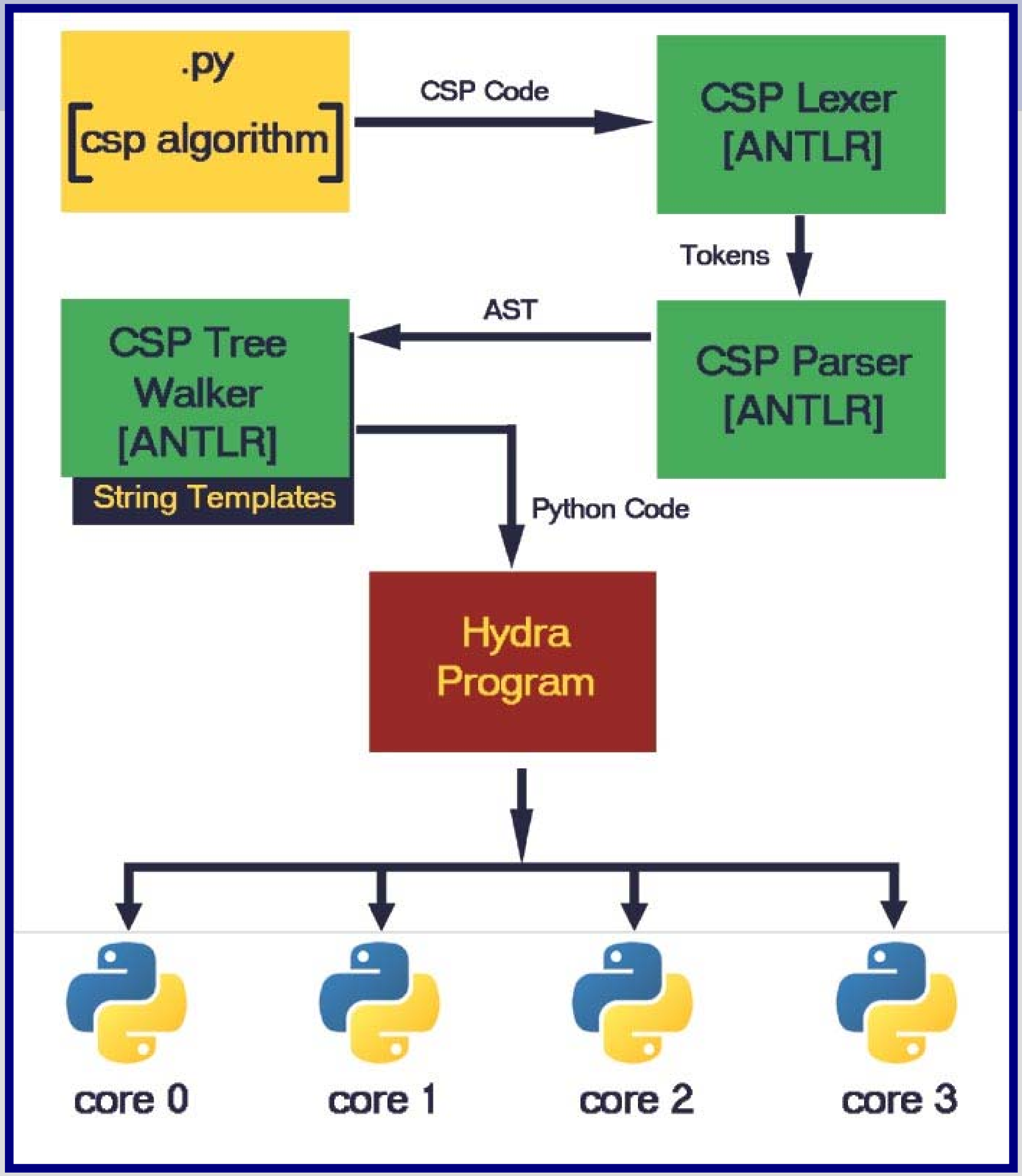
- Number of modifications required
  - Process construct uses [[ instead of [ to avoid ambiguity with the Alternative construct.
  - Inclusion of Python import statements at the start of the program: _include{import time}
  - Expression handling removed in favour of having Python interpret the expressions as Python code; anything within { }

# Approach - Libraries

- PYRO – Python Remote Objects
  - Powerful library for distributed Python objects with easy access
  - Handles the network communication between objects
  - Used as CSP style *channels* for inter-process communication
- PyCSP
  - Python module that provides a number of CSP constructs
  - Channels can be created as PYRO objects
  - *Process* and *Parallel* implemented using Python threads

  - However, newer versions (v0.6) create Processes as OS processes and network processes

# Approach – Compiler Design



# Framework – Using Hydra

- Include the ***csp*** module from the Hydra package in Python program
- Write Hydra CSP code in a triple-quoted Python string or read it into a string from a file
- Call the ***cspexec*** method with the string as an argument

```
from Hydra.csp import cspexec
code = """[[
    prod ::
        data : integer;
        data := 4;
]];    """
cspexec(code, progname='simple')
```

# Framework - Implementation

- Parallel construct
  - Defines the concurrent architecture of the program
  - Takes a list of processes to be executed in parallel
  - During execution, these processes are spawned asynchronously and may execute in parallel
- Drawbacks
  - Spawning a Python interpreter for every parallel process is not viable
  - Only the top-level parallel processes run in separate VMs and nested parallel processes use Python’s threading library

# Framework - Communication

- I / O commands define the **channels** of communication (and synchronisation)

- Channels are implemented as remote PyCSP channel objects using PYRO
  - Named according to source and destination processes
  - Carefully tracked and recorded
  - Registered with PYRO nameserver before execution

- I / O commands generate simple ***read*** / ***write*** method calls on appropriate Channel objects

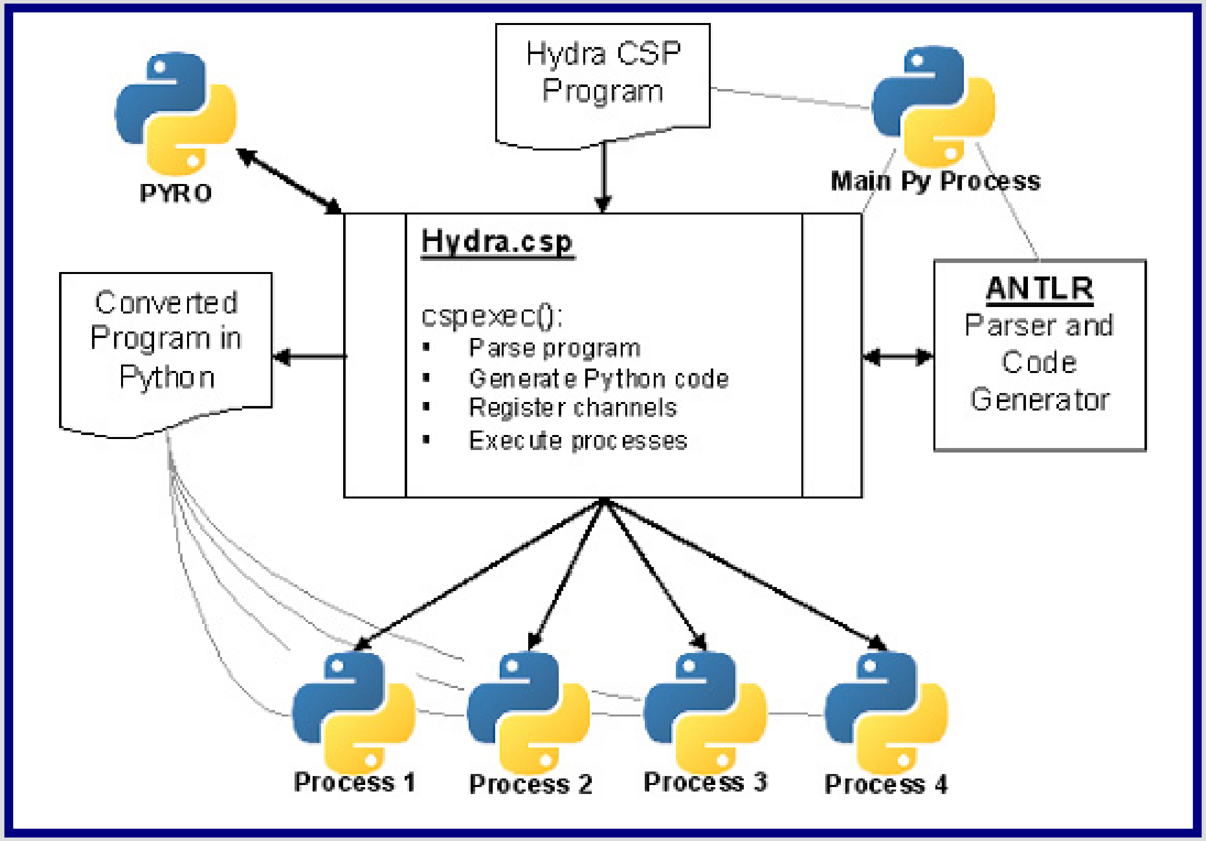
# Framework – Hydra CSP

- Process construct
  - Represented as a PyCSP Process for simplicity
  - Care taken to retrieve relevant Channel objects from PYRO
  - Need to handle definition of anonymous CSP processes
- Flow control
  - Repetitive, alternative and guarded statements implemented using appropriately constructed Python **while** and **if-else** statements
  - Input guards are implemented using PyCSP's Alternative class and the priSelect() method and can be mixed with boolean guards

# Framework - Bootstrapping

- Hydra CSP-based program defined as a Python file
- PyCSP's network channel functionality requires channels to be registered with PYRO
- Processes asynchronously executed by spawning a new Python interpreter using a loop and Python threads (process started by passing its name as a cmdline argument).
- The **cspexec** method then waits for the Processes to finish executing and allows the user to view the results before ending the program.

# The Framework



# Results

- Prototype for investigating use of CSP within Python
  - Performance was not considered
  - Use of Python expressions and statements embedded in CSP
  - By no means rigorous testing (correctness and communication)
  - Focus on multiprocessor execution in Python
  - Execution observed using operating system's process and CPU load monitoring tools
  - Simple producer-consumer program running in an infinite loop performing numerous mathematical operations

- Processes
  - Four Python processes were spawned for this example
  - Average CPU loads over program execution.

    CPU Core 1: 83%

    CPU Core 2: 79%

# Results - Sample Hydra program

```
from Hydra.csp import cspexec
prodcons = """
 _include{from time import time}
[[
   producer ::
              x : integer;     x := 1;
              *[
                        {x <= 10000} -> {print "prod: x = " + str(x)};
                                         consumer ! x; x := {time()};
              ];
||
    consumer ::
               -- code omitted
 ]]; """
cspexec(prodcons, progname='prodcons')
```

# Results – Python conversion

```
import sys
from pycsp import *
from pycsp.plugNplay import *
from pycsp.net import *
from time import time
def __program(_proc_):
	@process
	def producer():
		__procname = 'producer'
		__chan_consumer_out = getNamedChannel("producer->consumer")
		x = None
		x = 1
		__lctrl_1 = True
		while(__lctrl_1):
			 if False:
				pass
			elif x <= 10000:
				print "prod: " + str(x)
				__chan_consumer_out.write(x)
				x = time()
			else:
				__lctrl_1 = False
	@process
	def consumer():
		# code omitted
```

# Conclusions

Is possible to convert a CSP algorithm into suitably concurrent Python code using the chosen approach and tools

- Conversion process is automatic – easier for non-programmers
- More flexible than standard CSP as Python expressions and functionality can be used
- Parallel execution is possible

# Questions?