Hydra: A Python Framework for Parallel Computing

Waide B. TRISTRAM, Karen L. BRADSHAW
Department of Computer Science, Rhodes University, Grahamstown, South Africa
g05t1067@campus.ru.ac.za, k.bradshaw@ru.ac.za

Abstract. This paper investigates the feasibility of developing a CSP to Python translator using a concurrent framework for Python. The objective of this translation framework, developed under the name of Hydra, is to produce a tool that helps programmers implement concurrent software easily using CSP algorithms. This objective was achieved using the ANTLR compiler generator tool, Python Remote Objects and PyCSP. The resulting Hydra prototype takes an algorithm defined in CSP, parses and converts it to Python and then executes the program using multiple instances of the Python interpreter. Testing has revealed that the Hydra prototype appears to function correctly, allowing simultaneous process execution. Therefore, it can be concluded that converting CSP to Python using a concurrent framework such as Hydra is both possible and adds flexibility to CSP with embedded Python statements.

Keywords. concurrency, CSP, language translation, Python

Introduction

Parallel architectures started making an appearance from as early as the mid-1960s and continue to be the primary design for high performance computing systems. This is particularly evident in modern supercomputers, such as IBM's Roadrunner and Blue Gene/L, which make use of thousands of processors to achieve their astonishing computational power. However, these systems are only available to a select few scientists and researchers and it was not until recently that multi-processor computers started becoming readily available to consumers.

The availability of dual and quad core CPUs targeted at the consumer, workstation and server markets has created a problem in the field of software development. Multi-core computers have the power and potential to greatly outperform their single-core counterparts, but this potential can only be realised if the software is able to make use of multiple processors [1]. Both consumers and researchers stand to gain from the performance increases afforded by multi-core CPUs and parallel software. Researchers are now able to construct small high performance computing systems for their data processing needs by combining a number of relatively cheap multi-core CPU systems.

The Hydra project aims to provide a Python framework for parallel execution based on Communicating Sequential Processes (CSP). The focus of this paper is our investigation into the feasibility of translating CSP to Python code for the Hydra project.

1. Background and Related Work

This paper investigates the feasibility of a concurrency framework for Python based around a CSP to Python translator. There is a significant amount of research in the related fields of language translation, concurrency and parallel computing, and Communicating Sequential Processes. However, a detailed review of the work is beyond the scope of this paper.

Therefore, only the relevant aspects of the related fields are briefly presented and discussed in this section. These discussions focus primarily around work that has facilitated the development of our concurrent framework for Python based on CSP.

1.1. Language Parsing and Translation

The typical translation process involves a number of stages, ranging from identifying the syntactic constructs to constraint analysis and finally, producing the output code [2,3]. While constructing parsers can be done by hand, a tool known as a compiler generator is typically used to produce the translator. Compiler generators accept the target grammar and generate the various components of the compiler [3].

1.2. Parallel Computing and CSP

Communicating Sequential Processes was first introduced in 1978 by Hoare. A number of operations and constructs were identified as the primary methods for structuring computer programs [4]. Hoare identified *input* and *output* operations as being important but noted that these were not well understood. He also noted that the *repetitive*, *alternative* and *sequential* constructs were well understood, whereas there was less agreement on other constructs such as *subroutines*, *monitors*, *procedures*, *processes* and *classes* [4].

Processor development at the time was such that multiprocessor systems and increased parallelism were required to improve computation speed. However, Hoare noted that this parallelism was being hidden from the programmer as a deterministic, sequential machine. He saw that a more effective approach would be to introduce this parallelism at the programming level by defining *communication* and *synchronization* methods [4]. It is this approach that we are attempting to incorporate into Python using Hydra.

1.2.1. The CSP Programming Notation

The programming language or notation specified by Hoare is based on a number of fundamental proposals. The first of these is the use of the *alternative* command in conjunction with *guarded commands*, and the related *guards*, as a sequential control structure and a means to control non-determinism. Associated with the guarded and alternative commands is the *repetitive* command, which loops until all its guards terminate. Secondly, the *parallel* command specifies a means to start parallel execution of a number of processes or commands by starting them simultaneously, and synchronizing on termination of each of the parallel processes. Parallel processes may not communicate directly, except through the use of *message passing* [4].

To support the message passing concept, *input* and *output* commands are specified. These commands enable communication between processes. Essentially, a *channel* is created and used for synchronous communication when a source process names a destination process for output and the destination process names the source process for input. This effectively introduces the *rendezvous* as the primary method of synchronization [4].

1.2.2. The CSP Meta-Language

Hoare continued to refine CSP and it evolved substantially compared to the notation described in his earlier paper. CSP had become a *process algebra* that allows for the formal description and verification of interactions in a concurrent system [4].

The new notation consists of two primitives, namely the *process* and the *event*, and a number of algebraic operators. Concurrent and sequential systems can then be defined through a combination of these operators and primitives. An important addition to CSP is the introduction of *traces*, which allow for the description of each possible behaviour in a system as a sequence of actions [4].

1.3. The Python Programming Language

Python is a powerful, very high level programming language, supporting multiple programming paradigms [5,6]. Python has a strong, dynamic typing system and robust automatic memory management. It is very well suited for use both as a scripting language, much like *Perl*, and as a general purpose programming language. Python places a great deal of emphasis on programmer productivity and supports this via its expansive standard library and support for third-party extensions.

1.3.1. Features and Benefits

There are numerous benefits and features that make Python a very attractive language both to beginner programmers and for advanced scientific programming [6,7]. It has high-level built-in data types, such as the dictionary, list and tuple. Python has strong introspection capabilities and provides easy to use object orientation features. There is also plenty of support and readily available documentation.

Python supports full modularity and hierarchical packages for extending functionality and can also be embedded within applications as a scripting interface. This makes it very useful for linking together previously unrelated modules [7]. Python can therefore, be used for the rapid prototyping of algorithms, with any performance critical modules being rewritten in C and added as extensions. All of the above factors along with the availability of science orientated packages, such as SciPy and NumPy, have aided in the acceptance of Python in the computational science community.

1.3.2. Limitations

As an interpreted language, Python's performance is not as good as compiled languages such as C++, but the performance is sufficient for most applications. Python's greatest limitation is its global interpreter lock. The Python VM makes use of a global interpreter lock (GIL) to ensure that only one thread runs in the VM at any time [8].

So, while Python supports multi-threading, these threads are time-sliced instead of executing in a truly parallel fashion. Attempts to remove the GIL, such as Greg Stein's *"free threading"* patches, resulted in an overall drop in performance [8]. This performance decrease for non-threaded programs was unacceptable and the patches were abandoned and no further attempts to remove the GIL were made [8].

However, there are ways to circumvent the GIL limitation to achieve multiple processor usage. The first of the suggested methods is to make use of C extensions. The C extension can release the GIL and maintain the executing thread within the C code [8]. The second method is to divide the tasks between multiple Python interpreter processes, which must be spawned with appropriate communication and synchronization mechanisms [8]. There are frameworks and tools that provide functionality for communication between distinct Python processes, such as *River* [9], Trickle [10] and PYRO [11].

2. Methodology

As stated above, the aim of the Hydra project is to provide a Python framework for parallel execution based on Communicating Sequential Processes. However, the scope of this initial research was restricted to investigating the feasibility of translating CSP algorithms to concurrent Python code. The approach taken was that of research through design and development, requiring a working prototype for use in further research. However, this prototyping approach lead to a number of compromises in the development of the framework.

2.1. Approach

A number of issues needed to be addressed before being able to convert a CSP algorithm into a working, concurrent Python program. These issues are highlighted and briefly described in this section, while the decisions regarding these issues are fully discussed in Section 3.

First, an appropriate grammar for CSP was defined. There are a few variations of the original CSP grammar introduced by Hoare and of the later CSP process algebra or CSP meta-language [4]. While the CSP meta-language allows one to verify a concurrent algorithm through the use of process algebra [4], the original CSP notation provides a more suitable syntax for programming. Therefore a grammar has been devised based on the original CSP notation.

The decision to develop our own CSP syntax instead of using an existing dialect, such as that used by FDR, was influenced by the prototype nature of the project. In order to keep the language to be implemented small, it was decided to use a novel syntax so that much of the type support, expression and pattern matching elements of CSP could be ignored initially and we could concentrate on the concurrent aspects thereof. We also wanted to allow the programmer to embed Python expressions in their CSP, thus making the language much more powerful. At a later stage, a more acceptable dialect could be used with some modifications to the front-end of the parser.

Suitable compiler generators were then identified. There are a number of requirements that the compiler generator needed to meet before being considered for use. Firstly, its parsing technique needs to be powerful enough to deal with the CSP notation. It must be able to cope with ambiguity, with backtracking and suitable lookahead sufficient for handling any ambiguous cases. Secondly, it must provide functionality for generating target code as opposed to merely returning the identified tokens. Thirdly, the compiler generator must provide a clear and easy to use method for defining the grammar. This is to ensure that the grammar is maintainable and extensible. Finally, it would be beneficial for the generated parser to be coded in Python for easier integration into the Hydra framework. Good error handling mechanisms would also be beneficial.

The final choice of compiler generator was then made, based on its capabilities and shortcomings according to the above criteria. The availability of documentation and development activity was also investigated to ensure that issues or bugs are easily resolved. Once the compiler generator had been identified, the CSP parser was implemented using the selected tool. Due to the prototyping approach, the implementation of semantic checks and error reporting was kept rather basic and incomplete.

The code generation aspect required the identification of suitable libraries and frameworks on which to build the parallel constructs. Once the libraries had been chosen, the appropriate code segments were then designed to represent the CSP constructs as closely as possible using the features of the selected libraries. This put down the groundwork for the actual code generation process.

Once the underlying concurrent framework was complete, the actual code generator was developed. This step tied in closely with the parser generation and made use of the features provided by the compiler generator. The code generator was designed to take the abstract syntax tree (AST) returned by the parser and generate an equivalent Python program.

Finally, sample programs were developed and converted using Hydra. The output code was analysed by hand to identify any glaring errors and the program was run and checked for correct execution. Success is indicated by the correct execution and functioning of the Hydra-based program and its communication channels, and whether or not multiple processors are used as shown by the CPU load and process metrics. However, more rigorous testing is required to fully validate the correctness of the conversion process and execution.

2.2. *Translation to an Intermediate CSP Implementation*

While there are many projects that add CSP features to existing programming languages, there are very few attempts to convert directly from CSP to executable code [12]. JCSP and CTJ provide CSP features to Java [13,14]. CCSP and C++CSP provide similar CSP features for C and C++, respectively [15,16]. PyCSP introduces CSP features to Python and is discussed further in Section 3.2.2 [17]. From the list of modern language CSP implementations mentioned above, it would appear that no further work is required to expose CSP to programmers. However, these implementations require the programmer to convert their CSP code into the appropriate form for the implementation they desire to use. For small programs, this task is relatively easy. But once the programs start to get larger and more complex, the process becomes more difficult and is prone to error, particularly with regards to the correct naming and use of channels [12]. The time taken to develop and verify the CSP algorithm for a complex system can often be rivaled by the time taken to convert and debug the program written for one of the above mentioned CSP implementations [12]. Clearly this is not ideal and a means for translating the original CSP directly to executable code is more desirable.

3. Hydra Framework

3.1. *Compiler Generators*

A number of parser generators and parsing frameworks for Python were investigated. The strengths and weaknesses of each compiler generator were assessed and ANTLR [18] was chosen as being most the suitable candidate for use in Hydra.

3.1.1. *ANTLR*

ANTLR (ANother Tool for Language Recognition) is a parser generator that automates the construction of lexers and parsers [18]. ANTLR generates language recognisers that use a fast and powerful *LL(*)* parsing technique, which is an extension to *LL(k)* that uses arbitrary lookahead to make decisions. Ambiguity is handled by ANTLR's backtracking functionality, which allows the parser to work out the correct course of action during runtime, and partial memoization of results means that this can be achieved with linear time complexity [18].

The code generation features of ANTLR are also quite advanced, with formal abstract syntax tree construction rules allowing for custom ASTs to be constructed. Additionally, ANTLR's tight integration with StringTemplate enables the generation of structured text such as source code [18]. These features make the code generator easily retargetable with minimal changes to the front-end. ANTLR also has a grammar development IDE, named ANTLRWorks, which allows for the visualisation and debugging of parsers generated in any of ANTLR's supported target languages, which include Python among others [18]. ANTLR is also actively supported with ongoing development, mailing lists, updated project website, and plenty of documentation and examples.

3.2. *Concurrent Framework Modules*

A code generator was required to interpret the source language and produce an equivalent version in the target language [3]. The complexity of the code generator is often dependent on the complexity of the target language or architecture.

One approach involves developing all the necessary constructs and underlying framework from scratch. A more practical approach is to find and use existing frameworks for the target architecture, and add custom code only for the functionality that is missing or incomplete [10]. Therefore, the back-end concurrent framework for Hydra is built on top of two existing Python frameworks, namely PYRO and PyCSP.

3.2.1. *Python Remote Objects*

Python Remote Objects (PYRO) is a simple yet powerful framework for working with distributed objects written in Python. PYRO essentially handles all the network communication between objects, allowing remote objects to appear as local ones [11]. Additionally, PYRO provides remote method invocation functionality, which allows for methods from remote objects to be called locally. PYRO can be used over a network, allowing processes to be distributed between a number of separate computers, or it can be used purely on the local machine to provide a convenient inter-process communication mechanism [11].

PYRO consists of a special nameserver component that provides functionality for registering and retrieving remote objects. Client code is then able to register named objects with the PYRO nameserver and retrieve these objects using the specified name [11]. This remote object framework provides all the necessary functionality to implement CSP channels. Each communication *channel* between *processes* can be implemented as a remote `Channel` object with `read` and `write` methods. As such, PYRO plays a critical role in the implementation of the concurrent Hydra back-end.

3.2.2. *PyCSP*

PyCSP is a Python module that provides a number of CSP constructs such as channels, channel poisoning, basic guards, skip guards, input guards, processes, and the alternative, parallel and sequential constructs [17]. The biggest drawback of PyCSP is that the current implementation (version 0.3.0 at the time this research was conducted) makes use of Python's threading library, which is limited by the GIL [8,17]. One solution to this problem is to make use of network channels for communication between multiple local or remote operating system processes, which is the approach we have taken.

The PyCSP *Process* construct is implemented by simply instantiating an object of the `Process` class, which extends from Python's `Thread` class [17]. The instantiated Process object does not begin execution until it is used in either the *Parallel* or *Sequential* constructs [17]. Communication via *Channels* is handled by simply passing the read and write methods of a Channel object as arguments in a Process's constructor. PyCSP Channels allow for any object to be sent across them, including Processes [17]. This is a useful feature that allows for easy distribution of work, as well as the relaxation of type limitations.

PyCSP provides network channel functionality using PYRO [19]. With the appropriate custom framework code, this functionality can be leveraged to overcome PyCSP's reliance on Python's threading library. PyCSP has already implemented Python versions of most of the CSP constructs, such as the *process*, *channel*, *guard* and *alternative* commands, thus alleviating the need to develop these from scratch.

3.3. *CSP Grammar*

While Hoare indicated that programs expressed in the original notation should be implementable, he also made it clear that the notation was not suitable for use as a programming language [4]. However, CSP provides a convenient notation for defining the architecture and communication channels of the processes used by the program. For this reason, a number of compromises and changes have been made to the grammar to allow for the integration of CSP into Python as a means of describing parallel communication.

The grammar that was used in the Hydra prototype is presented below. A number of simplifications have been made to ease the construction of the prototype as mentioned in Section 2.1. The goal production or starting point of the grammar is the *program* production, which accepts zero or more Python import statements and a list of commands as represented further on in the grammar. There is only one method for defining parallel processes, but this is expanded upon in Section 3.4.2.

```
program       = ( PYIMPRT )* command_list;
parallel      = '[[' process ( '||' process )* ']]';
process       = ( proc_label )? command_list;
proc_label    = ID '::';
declaration   = ID ( ',' ID )* ':' type ';';
int_const     = simple_expr;
range         = ( ID ':' )? int_const '..' int_const;
type          = ( '(' INT '..' INT ')' basictype ) | basictype;
basictype     = 'integer' | 'boolean' | 'char';
command_list = declaration* command+;
command       = ( simple_cmd | struct_cmd ) ';';
simple_cmd    = assignment | input_cmd | output_cmd | 'SKIP' | PYEXPR;
struct_cmd    = alternative | repetitive | parallel;
assignment    = target_var ':=' expression;
subscripts    = simple_expr ( ',' simple_expr )*;
target_var    = ID ( '[' int_const ']' )? | struct_target;
struct_targ   = ID? '(' var_list? ')';
var_list      = target_var ( ',' target_var )*;
simple_expr   = ID ( '[' int_const ']' )? | INT | BOOL | CHR | PYEXPR;
struct_expr   = ID? '(' expr_list? ')';
expr_list     = expression ( ',' expression )*;
expression    = simple_expr | struct_expr;
input_cmd     = ID '?' target_var;
output_cmd    = ID '!' expression;
repetitive    = '*' alternative;
alternative   = '[' guarded ( '[]' guarded )* ']';
guarded       = ( '(' range ')' )? guard '->' command_list;
guard         = guardlist | input_cmd | nullcmd;
guardlist     = guard_elem ( ';' guard_elem )* ( ';' input_cmd )?;
guard_elem    = simple_expr | declaration;
```

A significant change that warrants discussion is the removal of expression operators such as the arithmetic and Boolean operators. In their place, the ability to use Python expressions has been added, allowing for much greater flexibility when it comes to expressions. The Python code is enclosed in braces and can be any valid Python expression. To support functionality from Python's vast module collection, the ability to add Python import statements to the beginning of the program was added. These import statements are preceded by "_include" and are enclosed in braces. The rationale behind this rather significant change is that Python has the ability to evaluate expressions and implementing them in the grammar would just duplicate existing functionality. This removes the burden of parsing and evaluating expressions and essentially gets the Python interpreter to do this on behalf of the parser. This feature also allows for the use of all of Python's data types, bypassing the limited data type support natively provided by the parser. Examples of the use of these constructs are given in Section 4.

The Hydra CSP grammar supports single-line comments, starting with a double hyphen and ending in a newline. Since there is no symbol for the $\rightarrow$ and $\Box$ symbols used by the *guarded* statement on common keyboards, "->" and "[]" were used in their place. The Hydra lexer supports four basic expression types, namely identifiers, characters, integers and booleans. Identifiers start with a lowercase letter of the alphabet, and can be followed by any combination of uppercase and lowercase letters, digits and the underscore character. Characters can be any valid ASCII character, denoted between single-quotes. Integers are simply defined as a series of digits. And finally, Boolean expressions are denoted by either "True" or "False" and are case-sensitive.

3.4. *Hydra Framework Implementation*

3.4.1. *Using Hydra*

Before going into the implementation of the framework, it would be beneficial to describe the manner in which CSP is used within a Python program. The mechanism chosen is fairly simple, although somewhat less than ideal. The process is described below, along with a very simple example, which defines a process, declares an integer variable and assigns it a value.

```
from Hydra.csp import cspexec

code = """[[
  prod ::
      data : integer;
      data := 4;
]];
"""
cspexec(code, progname='simple')
```

Firstly, the Hydra `csp` module must be imported. The `csp` module provides the `cspexec` method, which takes the string containing the CSP algorithm as an argument and an optional program name argument. The `cspexec` method is responsible for converting the algorithm and managing the execution of the processes. Since the CSP algorithm is represented as a string, it is possible to specify the algorithm inline as a triple-quoted string or the algorithm could be specified in a separate file which could then be read in and supplied to the `cspexec` method. The program is then run by simply executing the Python program as usual.

3.4.2. *Implementation Decisions*

The `parallel` production, although very simple in its appearance, is of paramount importance as it defines the concurrent architecture of the program. It takes a list of one or more *processes* to be executed in parallel. During execution, these *processes* are spawned asynchronously and may execute in parallel, thus achieving one of the project goals: execution of code over multiple processors. However, the prototype implementation exhibits two distinct behaviours. For the top-level *parallel* construct, it generates the appropriate code for executing over multiple Python interpreters, but for any nested *parallel* statements, PyCSP's `Parallel` method is used. The rationale behind this is that current desktop computers have at most eight processor cores, therefore, implementing every process in a new Python interpreter instance is not likely to scale adequately. In the future, this distinction could be made explicitly controllable by the programmer.

Another important set of CSP constructs is the *input* and *output* commands. These essentially define the *channels* of communication between *processes* and provide a synchronisation mechanism in the form of a rendezvous. *Channels* are named according to their source and destination *processes* and are carefully tracked and recorded so that they can be correctly registered by the PYRO nameserver before execution. The *input* and *output* commands generate simple `read` and `write` method calls on the appropriate `Channel` objects. While it is possible to communicate with external code through these *channels* by acquiring the appropriate *Channel* object via PYRO, such actions are not recommended as they can affect the correct operation of the Hydra program.

The `process` construct is represented as a PyCSP `Process`. The necessary care is taken to retrieve the relevant `Channel` objects from the PYRO nameserver using the `getNamedChannel` method, based on the recorded *channels*. Since CSP allows for the definition of anonymous *processes*, a technique for handling and defining these methods was devised. The technique is fairly simple and involves giving each *process* an internal name. It

is worth noting that since anonymous *processes* have no user-defined name, it is not possible to use *input* and *output* commands within these processes.

The `repetitive`, *alternative* and `guarded` statements are implemented using appropriately constructed Python `while` and `if-else` statements. To simplify code generation, all `while` blocks start with an `if` statement with the condition always false. This means that only `elif` statements need to be generated when translating the *alternative* construct. *Input guards* are implemented using PyCSP's `Alternative` class and the `priSelect` method that uses order of appearance as an indicator of priority.

Array declarations need to be treated specially by the declaration code as CSP permits the declaration of array bounds, which can lead to out-of-bounds errors if the programmer attempts to reference an uninitialised list variable. Therefore, arrays are declared as a Python `list` with the appropriate number of elements all set to `None`.

Python has a number of keywords that cannot be used as method or variable names. Therefore, all user-defined identifiers are sanitised by simply prefixing an underscore to any identifiers that clash with known keywords. Python expressions and statements embedded in the CSP are handled by simply expressing them as normal Python code. This allows the programmer to use external modules and leverage the power of Python within their CSP code.

3.5. Process Distribution and Execution

Once the programmer has defined their Hydra CSP-based program within a Python file, this file can be executed as a normal Python program (this runs in its own Python interpreter), which then calls the `cspexec` method of the Hydra framework. The `cspexec` method controls the execution process from receiving the CSP algorithm, having the ANTLR-based translator convert it, registering the appropriate channels, to finally starting the execution of the concurrent program.

A relatively simple approach was taken to bootstrapping and executing the relevant processes once code generation was complete. One of the problems encountered with using PyCSP's network channel functionality is that all *channels* need to be registered with the PYRO nameserver before the *processes* are able to retrieve the remote `Channel` objects. There is no easy way to add this registration process to the generated program without encountering situations where one *process* requests a *channel* that has not yet been registered. This problem was addressed by registering all the necessary *channels* beforehand in the `cspexec` method of the `Hydra.csp` module. The translation process returns a list of *channels* and *processes* that need to be configured and executed, which is then processed by a simple loop that registers the appropriate *channel* names with the PYRO nameserver. This can be seen in the Python code snippet below.

```
# Iterate through required channels
for i, chan in enumerate(outpt.channels):
    cn = One2OneChannel()   # Create new channel
    chans.append(cn)        # Keep track of the channel

    # Register the channel with the PYRO nameserver
    registerNamedChannel(chans[i], chan)
```

Since this happens before *process* execution, there is no chance of *channels* being unregistered or multiple registrations occurring for the same *channel* name, thus breaking interprocess communication.

Once the *channels* have been registered, the *processes* are asynchronously executed by spawning a new Python interpreter using a loop and Python threads. The `cspexec` method then waits for the *processes* to finish executing and allows the user to view the results before ending the program. This process can be seen in the following Python code snippet.

```
class runproc(Thread):
    def __init__ (self, procname, progname):
        Thread.__init__(self)
        self.procname = procname
        self.progname = progname
    def run(self):
        os.system('python ' + self.progname + '.py ' + self.procname)

def cspexec(cspcode, progname='hydraexe', debug=False):
    # Translation and channel registration occurs here

    proclist = []
    # Iterate through the list of defined processes
    for proc in outpt.procs:
        # Create a new Python interpreter for each top-level
        # process and start it in a new thread.
        newproc = runproc(proc, progname)
        proclist.append(newproc)
        newproc.start()

    # Wait for processes to finish before terminating
    for proc in proclist:
        proc.join()
```

An overview of the Hydra translation and execution process is shown in Figure 1.

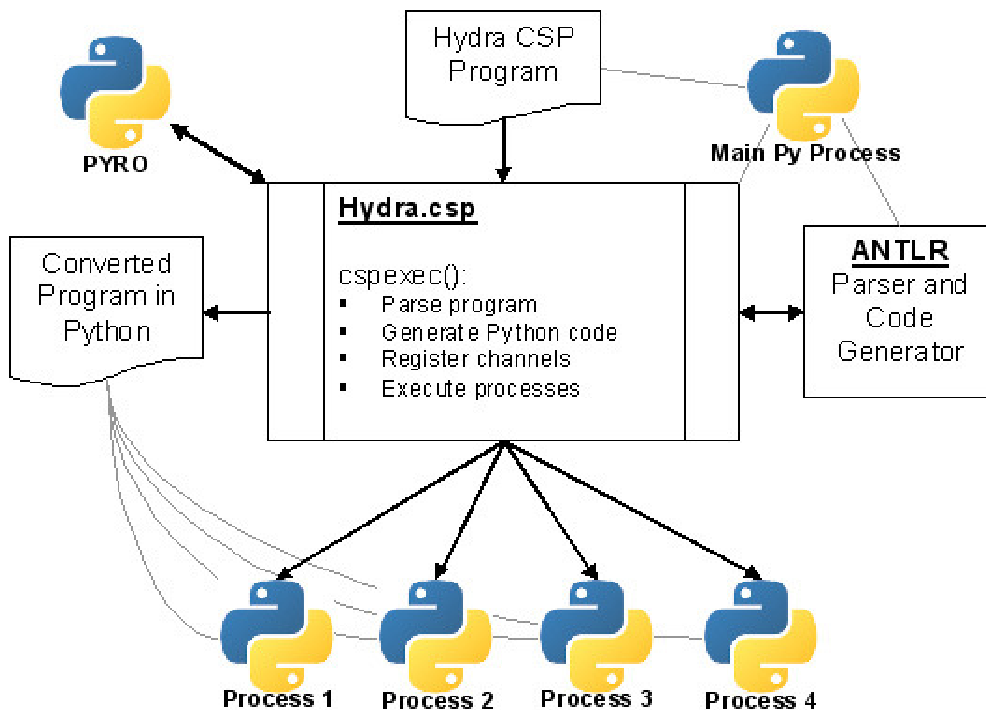

Figure 1. Hydra translation and execution process.

Since all the *processes* are defined within the same Python file, it was necessary to provide some means for the new Python interpreters to execute the correct *process* method. During code generation, simple command-line argument handling is added to the output file that allows for the correct method to be executed based on the supplied argument. This is shown in the Python code snippet below.

```
# Snippet from generated Python code
def __program(_proc_):
    # PyCSP processes omitted

    # Process selection
    if _proc_ == "producer":
        Sequence(producer())
    elif _proc_ == "consumer":
        Sequence(consumer())
    else:
        print 'Invalid process specified.'

__program(sys.argv[1])
```

In the above snippet, PyCSP's `Sequence` construct is used to start the appropriate *Process*. This is because PyCSP Processes are only executed when they are used in a PyCSP Parallel or Sequence construct.

4. Analysis

Two forms of testing were performed on the Hydra prototype. The system was tested with a number of sample programs; the resulting output code was then manually inspected to determine if it is an accurate representation of the CSP algorithm.

Secondly, the code was executed and the operating system's process and CPU load monitoring tools were used to determine whether or not the program was executing over multiple process cores.

Since this implementation is merely a prototype for investigating the feasibility of using CSP within Python, the actual performance of the framework was not considered. The primary focus was on enabling multiprocessor execution in Python. Once that had been achieved, further work can focus on refining the framework and optimising for performance.

All testing was performed on the system configuration specified in Table 1.

Table 1. Testing platform configuration.

Component	Specification
CPU	AMD Opteron 170 (2 cores @ 2.0GHz)
Motherboard	ASUS A8R32-MVP Deluxe
Memory	2x1GB G.Skill DDR400
Hard Disk	Seagate 320GB 16MB Cache
Network	Marvel Gigabit On-board Network Card
Operating System	Microsoft Windows 2003 Server SP2
Python	Version 2.5.2

4.1. Generated Code Analysis

One of the example Hydra CSP algorithms can be seen in the code listing below. This is a simple program with two processes (only the producer is shown). The `producer` process outputs the value of `x` to the `consumer` process 10000 times and the `consumer` process simply inputs the value received from `producer` and stores it in `y`. The value of `x` is updated with the current time expressed in seconds from a fixed starting time. The example also highlights the use of Python *import* statements and the use of Python statements within CSP.

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

One of the resulting Python processes can be seen in the listing below.

```
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
```

Looking at the output code, it is clear that Hydra has generated both *processes* and defined them correctly, with correct *channel* initialisation and variable declarations. The repetitive command is present in the form of a `while` loop with the appropriate control variable and *alternative* code. The *guarded* commands can also be seen in the form of the `elif` statements, with expressions and statement blocks correctly represented. The *output* command can be seen with the `write` method call on the *channel* object. This example, while simple, is able to show many of the CSP constructs and their respective representations in Python using Hydra.

The resulting Hydra program was then executed and the Windows Task Manager was used to monitor the `python.exe` interpreter processes and overall CPU usage. All unnecessary programs and services were closed to ensure the least possible interference with the CPU

Table 2. Average CPU loads.

Core	Average Load
CPU 0	83%
CPU 1	79%
Combined	81%

load measurements. To demonstrate the parallel execution effectively, the *guard* conditions for the `producer` and `consumer` processes were changed to True, thus creating infinite loops. Additionally, complex mathematical calculations were added to the producer and consumer processes to increase CPU load. This provided enough time to effectively demonstrate multi-core usage. Table 2 shows the resulting average CPU loads, which clearly indicate multi-core execution. The number of Python processes was also verified. Four processes were found, which include two for the CSP processes, one for the Hydra framework and one for the PYRO nameserver. Therefore, the correct number of processes are being spawned. However, further testing is required on larger more complex programs to confirm their successful functioning and the correct functioning of the communication channels as well.

5. Conclusions

The goal of the Hydra project is the creation of a concurrent framework for Python based on CSP. This framework is responsible for converting CSP code into concurrent Python code. The process involved the development of a parser for CSP using ANTLR and the creation of a code generator using ANTLR and StringTemplate, which takes the AST produced by the parser and generates the required Python code. Finally, basic testing was conducted to determine whether or not the Hydra framework was capable of meeting its objectives and it was confirmed that the Hydra prototype appears to execute the target program correctly, along with correct channel initialisation and communication. It also spawns the correct number of processes and executes over multiple processors. However, as stated previously, more rigorous testing and evaluation is required to validate the correctness of the Hydra program with certainty.

The Hydra prototype has demonstrated that it is possible to take a CSP algorithm and convert it into concurrent Python code, using the method described in this paper, and have the concurrent program execute over multiple CPU cores. The objective of developing a flexible parser and translator was also achieved thanks to ANTLR's powerful parsing and code generation functionality.

A recent update to the PyCSP project (version 0.6.1) has added support for creating CSP processes as operating system processes instead of threads, similar to what is done in this prototype, although the underlying implementation differs.

Acknowledgements

The authors would like to acknowledge the financial support of Telkom, Comverse, Stortech, Tellabs, Amatole Telecom Services, Mars Technologies, Bright Ideas 39, and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

References

[1] Brian Hayes. Computing in a Parallel Universe. *American Scientist*, 95:476–480, 2007.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, 2/E.* Addison-Wesley, 2nd edition, 2006.

[3] Pat Terry. *Compiling with C# and Java*. Addison-Wesley, 2005.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[5] Alex Martelli. *Python in a Nutshell*. O'Reilly & Associates, Inc., 2003.

[6] Python.org. About Python. Online, May 2008.

[7] D. Beazley and P. Lomdahl. Feeding a Large Scale Physics Application to Python. In *Proceedings of the 6 th International Python Conference*, San Jose, California, October 1997.

[8] Python.org. Python Library and Extension FAQ. Online, January 2008.

[9] Gregory Benson, Alexey Fedosov, Joe Gutierrez, Brian Hardie, Tony Ngo, Jennifer Reyes, and Yiting Wu. River - A Python-based Framework for Rapid Prototyping of Reliable Parallel Run-time Systems. Online, May 2008.

[10] Gregory Benson and Alexey Fedosov. Python-based Distributed Programming with Trickle. In Hamid R. Arabnia, editor, *PDPTA*, pages 30–36, Las Vegas, Nevada, USA, June 25-28 2007. CSREA Press.

[11] Irmen de Jong. PYRO - Python Remote Objects. Online, May 2008.

[12] V. Raju, L. Rong, and G.S. Stiles. *Automatic Conversion of CSP to CTJ, JCSP, and CCSP*. IOS Press, 2003.

[13] Abhijit Belapurkar. CSP for Java Programmers. Online, June 2005.

[14] Gerald Hilderink, Jan Broenink, Wiek Vervoert, and Andre Bakkers. Communicating Java Threads. In *Proceedings of the 20th World Occam and Transputer User Group Technical Meeting*, pages 48–76, The Netherlands, 1997. IOS Press.

[15] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[16] Neil C.C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, pages 183–205. IOS Press, July 2007.

[17] John Markus Bjørndalen, Brian Vinter, and Otto Anshus. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, pages 229–248. IOS Press, July 2007.

[18] Terence J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, Raleigh, North Carolina, 2007.

[19] John Markus Bjørndalen. PyCSP. Online, May 2008.