Communicating Process Architectures 2008
Susan Stepney, Fiona Polack, Alistair McEwan, Peter Welch, and Wilson Ifill (Eds.)
IOS Press, 2008

CSPBuilder - CSP based Scientific Workflow Modelling

Rune Møllegård FRIBORG and Brian VINTER

Department of Computer Science, University of Copenhagen,
DK-2100 Copenhagen, Denmark

{runef , vinter} @diku.dk

Abstract. This paper introduces a framework for building CSP based applications, targeted for clusters and next generation CPU designs. CPUs are produced with several cores today and every future CPU generation will feature increasingly more cores, resulting in a requirement for concurrency that has not previously been called for.

The framework is CSP presented as a scientific workflow model, specialized for scientific computing applications. The purpose of the framework is to enable scientists to exploit large parallel computation resources, which has previously been hard due of the difficulty of concurrent programming using threads and locks.

Keywords. CSP, Python, eScience, computational science, workflow, parallel, concurrency, SMP

Introduction

This paper presents a software development framework targeted for clusters and tomorrow's CPU designs. CPUs are produced with multiple cores today and every future CPU generation will feature increasingly more cores. To fully exploit this increasingly parallel hardware, more concurrency is required in developed applications.

The framework is presented as a scientific workflow model, specialized for scientific computing. The purpose of the framework is to enable scientists to gain access to large computation resources, which have previously been off limits, because of the difficulty of concurrent programming — the *threads-and-locks* approach does not scale well.

The major challenges faced in this work include creating a graphical user interface to create and edit CSP [1] networks, design a component system that works well with CSP and Python, create an execution model of the designed CSP networks and run experiments on the framework to find the possibilities and limitations. CSPBuilder can be downloaded from [2].

1. Background

Over the past few decades, companies producing CPUs have consistently increased processor speeds in each new edition by decreasing the size of transistors and increasing the complexity of the processor. The number of transistors on a chip have doubled every 2 years over the last 40 years, as declared by Moore's Law [3]. However, doubling the number of transistors does not automatically lead to faster CPU speeds, and requires additional control logic to manage these. Speed and throughput have typically been increased by adding more control logic and memory logic, in addition to increasing the length of the processor pipeline. Unfortunately more pipelines mean more branch-prediction logic, with the effect that it becomes very expensive to flush the pipeline when a branch is wrongly predicted. Many other extensions and complexities, e.g. SIMD pipelines, have been added to the CPU design during the past 40 years to increase CPU performance.

Today, numerous *walls* have been hit. The amount of transistors is still doubled every two years, so Moore's Law still applies. However, three problems have been raised: the *power wall*, the *frequency wall* and the *memory wall*. According to Intel [4], heat dissipation and power consumption increase by 3 percent for every 1 percent increase in processor performance. Intel also explain that because of bigger relative difference between memory access and CPU speeds, memory also becomes a bottleneck. Furthermore, the pipeline has become too long, so the cost of flushing outweighs the performance gained by increasing the pipeline length. All of these mean that we can go no further with current designs, and Intel suggest in [4] that the next step is parallel computation.

With several processing units, the *power wall*, *frequency wall* and *memory wall* are avoided, since there is no longer a need to increase the processor performance for a single unit. Instead you must be aware of communication and synchronization between threads, which can cause overhead, deadlocks, livelocks and starvation if used wrongly.

Computers of tomorrow are getting more and more processing units, which can be utilized by creating concurrent applications that will scale towards many processors. We are already at 128+ cores in graphic processors, 9 cores in the CELL-BE processor from IBM, SONY and TOSHIBA and recently Intel announced that they are experimenting with an 80-core CPU [5].

1.1. Motivation

Many scientists (chemists, physicists, etc.) are not experienced programmers, but are able to do scientific computing by programming sequential applications. So far they have been relying on the hardware manufactures to produce hardware which has improved the performance of their applications — allowing for more sophisticated and computationally intensive science.

Due to the limitations of sequential computing already discussed, scientists must now develop *concurrent* applications, in order to take advantage of parallel hardware and to advance the science. The amount of difficulty involved in creating concurrent applications, depends on the programming language and methodology. Traditional concurrent programming, with *threads* and *locks*, makes it difficult to program even simple applications — adding more parallelism to an already threaded program tends to result in problems, not solutions. As a direct result, concurrent programming is seen as *hard*, and is generally avoided by the majority of programmers.

We want to encourage scientists to develop concurrent programs using a CSP [6] based approach, where applications are built as layered networks of communicating processes. Such an approach is *reliable*, no unexpected surprises; *scalable*, to different numbers of processes and processors; and *compositional*, enabling processes to be 'glued' together to build increasingly complex functionality.

A feature of CSP based designs is that every process can be completely isolated from the global namespace, only interacting with other processes through well-defined mechanisms such as channel inputs and outputs — processes are *not* context sensitive. This in turn permits a high level of code reuse within scientific communities, as previously built components can be connected in different ways, corresponding to the data-flow of a particular computation.

Recent reports of using the GPU[1] and CELL-BE for scientific computing, have reported performance increases of up to 100-fold for some scientific algorithms. However, the difficulty of programming on a GPU or the CELL-BE is evident, and we desire a high level of

[1]*Graphics Processing Unit* – general-purpose graphics hardware found in high-end workstations, e.g. the NVidia GeForce2.

code reuse — i.e. algorithms written should be able to run on a number of different architectures, without a significant porting effort. This includes within a single-processor system, heterogeneous multi-core systems, and distributed over networks of machines. A CSP based design, of communicating processes, allows us to mix and match processing architectures — selecting the best performing implementations of processes for particular architectures.

While architectures have differing performance characteristics, programming in different languages can also affect performance. Development in a high-level language such as Python is usually faster, but produces code that runs slower than a similar implementation in a low-level language, such as C. By programming the computation intensive parts in C, and using Python as the ‘glue’, we optimize the execution time and avoid having to program the entire application in C, saving development time.

When doing scientific work, which often relies on particular mathematics libraries to do the “number crunching”, the functions provided are not necessarily all implemented in the same language. By using tools such as SWIG [7] and F2PY [8] we hope to address this issue, making it possible to use code from C, C++ and Fortran in a single scientific application.

Our solution is to provide a framework, written in Python, that assists scientists in creating concurrent applications based on a CSP design. The framework uses a graphical user interface similar to other *flow-based* programming environments already available, and as such, we hope that scientists will find our framework useful and accessible.

1.2. PyCSP

PyCSP [9] is the CSP [1] library for Python used in this paper. It is a new implementation and is currently evolving into a stable library. At the moment it supports four different channel types, that can be used for connecting parallel processes: *one-to-one*, *one-to-any*, *any-to-one* and *any-to-any*. Similar to occam, support for guarded choices is only available on the reading ends of *one-to-one* and *any-to-one* channels. When more than one process is attached to the *any* end of a channel, only one process at that end is involved in the communication, and queue in a FIFO. Communication on channels is synchronous — a channel output will not complete until the inputting process has accepted the data. In the future, we hope to support all types of guards for channel communication, as well as having full support for networked channels, and the easy distribution of CSPBuilder applications across computer networks.

The syntax of PyCSP is fairly simple and works well in Python. When executing a CSP network using PyCSP, all processes are created as kernel threads, though performance on *shared-memory* architectures is limited by the *Global Interpreter Lock* (see section 3.1.4).

1.3. Scientific Workflow Modelling and CSP

The purpose of a scientific application is usually to calculate a result based on input data. This data flows through the application and is the basis of sub-problems and sub-solutions until eventually a result, or several results, are found. With this in mind we use the term “workflow” for the data-flow of a scientific application. We use the term “scientific workflow” for the workflow of eScience applications, where “eScience” is used to describe computationally intensive science applications, normally run on shared-memory multi-processor hardware or in distributed network environments.

A typical eScience application might be anything from complex climate modelling to a simple n-body simulation. Generally, any application that does a large number of computations to produce a result within a particular scientific field.

Only a few [10,11] have previously looked at CSP and thought that this might be a good description for scientific workflows. In this paper we will produce an application that uses some of the ideas from CSP algebra and the projects mentioned above, combined in a framework that allows CSP based applications to be designed in a visual tool, and executed in a variety of ways (depending on the hardware available). We stipulate that CSP is ideal for reasoning about the dataflow of eScience applications, particularly when the target environment is concurrent execution. The compositional structure of a CSP network enables application developers to reuse networks of components as top-level components themselves.

In section 5 we cover some of the other frameworks available. Some of these are very popular today, and at the PARA ’08 event there was an entire day of workshops devoted to scientific workflow modelling. The scientists there argued that they are able to understand flow-based programming environments, and use them to develop scientific applications. The future users of CSPBuilder are the same as for other frameworks, and by making CSPBuilder operate in a similar fashion, we expect that those users will be able to use the CSPBuilder framework to construct applications.

One of the reasons for working with scientific workflows is to enable access to large computation resources. The model presented in this paper, in addition to support for remote channels, will make it possible to divide scientific workflow applications from a small number of CPU cores, to hundreds of nodes on different LANs — provided that the application is designed in a way that supports this; a design method that is promoted by the CSPBuilder framework.

1.4. Summary of Contributions

A new framework is implemented, tested and benchmarked in this paper. This framework consists of a visual tool to build applications and a tool to execute the constructed applications. The framework is implemented in Python and supports to use C, C++ and Fortran code by providing ‘wizards’ to access these languages. The framework is called CSPBuilder and incorporates extensive use of the CSP algebra.

The visual tool provides an “easy to use” graphical user interface, enabling users to construct applications using the ideas of flow-based programming [12] to produce a CSP [1] network. In our experiments we show that the visual tool is capable of handling large and complex applications.

Applications that are constructed with CSPBuilder can be executed successfully on a single computer, combining routines from a number of different programming languages. With the future introduction of remote channels in PyCSP it will be possible to execute the applications on any number of hosts.

The framework encourages code reuse by constructing applications from reusable components. This has proven very useful during the experimentation phase.

The primary advantages of this framework lie in code reuse and constructing complex scientific applications focusing on the workflow. CSP ideas underpin the concurrency mechanisms employed in constructed applications, enabling the automatic deconstruction of whole systems into individual concurrent components.

2. The Visual Tool

This section describes a user-friendly application that can model a CSP network using a layout similar to flow-based programming [12]. This layout is required to resemble the CSP network for a scientific workflow model. Figure 1 shows an application modelled using our visual tool.

In CSPBuilder every application starts with a blank canvas, where processes and channels can be inserted. Processes appear as named boxes, with their external connections labelled. Channels are shown as lines connecting the processes. To simplify things, any inbound or outbound connection will only accept one channel going in or out, depending on the connection type.

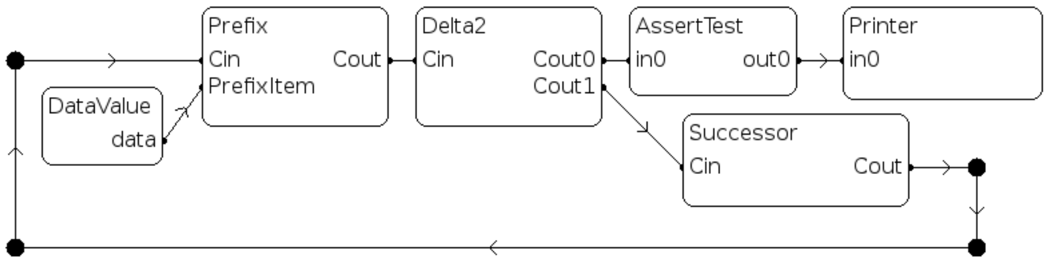

Figure 1. A CSPBuilder application that generates incrementing natural numbers.

A number of connected processes are known as a process network, as shown in figure 1. This network could be used as a component in another application, described in section 2.1.

The remainder of this section describes the component system, connecting components with channels and connection points. Saving and loading CSP applications to and from files are then described, followed by details on component configuration and replication. These parts are necessary to construct an application, and are parts of the framework that make it possible to build CSP networks that can be run efficiently in a distributed environment.

2.1. Component System

The design of the component system is based on the following requirements:

- We need to be able to link the Python code of each process in an easy to understand framework, to make it simple to add or remove components.
- The organization of the process network needs to be scalable, which means that the user should be able to handle large and complex applications, without losing control or an overview of the whole system.
- The user should quickly and easily be able to group parts of the process network into components, that appears and function like other processes.
- Components should be stored in a library for reuse.
- An application built with CSPBuilder must be targetable to different hardware, and have a performance better than or equal to an equivalent application written entirely in Python.

These requirements are examined in more detail in the following sections.

2.1.1. Scalable Organization

Consider a network of 2000 processes. To handle this many processes, and even more channels, it is necessary to group parts of the network into smaller compositional processes. This can be done by allowing the user to select a group of connected processes and condense them into a single component. If this new component has unconnected inbound or outbound connections, these are added to its interface, in addition to channels that already cross the group boundary. From an external perspective, this new component looks like any other component in the system.

Collecting together components in groups, and using these to form other components, leads naturally to a tree structure, whose leaves are component implementations. Each level of the tree is assigned an increasing *rank* number, with leaf processes having a rank of 1. This is used to prevent cyclic structures.

2.1.2. Components

Components are the most important part of CSPBuilder. A component is a CSPBuilder application that has been stored in the component library. These stored components are available for use in other applications, and come in two different forms:

1. The component is a process network consisting entirely of process instances of other components and includes no actual code implementations.
2. The component includes at least one process that contains a process implementation. This process implementation has a link to a Python function that implements the process. A simple example of a process in CSPBuilder is “IDProcess”, shown in listing 1, that simply forwards data received on its input channel to its output channel.

```
from common import *

def CSP_IdProcessFunc(cin, cout):
    while 1:
        t = cin()
        cout(t)
```

Listing 1. Example CSP process implementation – the IDProcess

To make it as easy as possible for the user to create components, we specify that to create a component, you just have to copy or move your CSPBuilder application to a “Components” directory. When the CSPBuilder application reloads the library, it discovers this new component and makes it available for use in new applications.

Functions specific to building components are also incorporated. These include naming unconnected channel-ends and naming the main application. When creating components, the application name is used for the new component. Unconnected channel-ends for the component’s input and output are named in similar ways.

2.1.3. Component Library

To aid in component management, each component requires a package name. This is to make it easier to find the desired component, for example, a “statistics” package containing relevant statistical components. For CSPBuilder to be an effective tool, it will need a wide variety of components, offering a range of different functionalities.

2.1.4. A Wizard for Building Components

A developer should be able to reuse code made by others, or reude code made earlier in another application. Reusing older code is made easier with components and the component library, so to increase the ease of creating new components a ‘wizard’ has been implemented that guides the developer through the process of creating a component.

A quick search on the Internet will show that large online archives of scientific code are available for free use. It is desirable to be able to easily use a function written in any language, and currently it could be argued that it is possible just by having the components implemented in Python. The developer can use SWIG [7] to import code from C or C++, and most programming languages are able to build libraries that can be used from C or C++. This therefore makes it possible to extend Python with code written in all kinds of languages. A project named F2PY [8] can import Fortran 77 and Fortran 90 code into Python.

The wizard guides the user through the process of creating components written in Python, C, C++, Fortran 77 and Fortran 90. These languages were chosen because of the numerous scientific libraries that use these. As mentioned earlier, most languages can build a library that is accessible from C or C++.

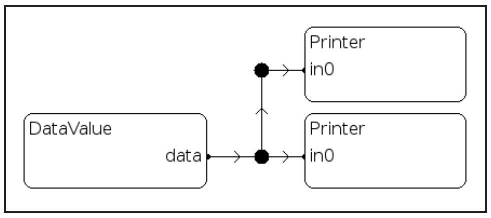

Figure 2. One2AnyChannel formed by connecting three processes to a single connection point, single outputting process, multiple inputters.

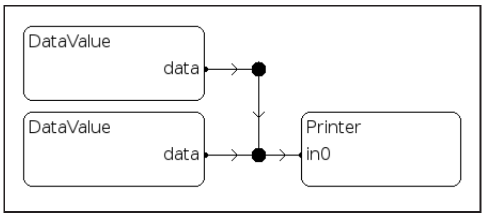

Figure 3. Any2OneChannel formed by connecting three processes to a single connection point, single inputting process, multiple outputters.

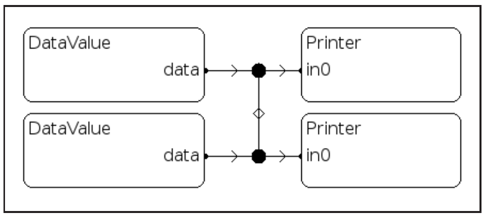

Figure 4. Any2AnyChannel formed by connecting four processes to a single connection point, multiple inputting and outputting processes.

The inclusion of other programming languages is expected to have a positive effect on application performance in CSPBuilder. Python uses the *Global Interpreter Lock* (see section 3.1.4) to access Python objects. This means that only one Python thread is allowed to access Python objects at any one time, limiting any advantage of running threads that are not dependent on each other in parallel. This lock can be freed when executing external code imported into Python, making it efficient to have certain parts written in other languages. Also, compiled languages are typically faster than interpreted languages, which further improves performance.

2.2. Channels and Connection Points

Processes connected by channels form a process network. The different types of channels available and how they work in PyCSP were introduced earlier. The types of channels are One2OneChannel, One2AnyChannel, Any2OneChannel and Any2AnyChannel.

The One2OneChannel is simple, because it can be represented by a single line going from one process to another. Representing the other types of channel is more complex. To address this issue, we introduce connection points. These can have any number of inbound and outbound connections, to processes or other connection points, enabling visualisation of all channel types and for the ‘bending’ of channels. Examples of these can be seen in figures 1, 2, 3 and 4.

Before any code can be generated, or process networks constructed, the connection graph for each channel is reduced to contain at most one connection point. Starting with each connection point, or node, that node’s neighbours are examined. If that neighbour is another node, as opposed to a process, the connections there are moved to the current node. This is done recursively, until only single connection points remain, and runs in $O(n)$ time, where n is the number of connection points.

The visual tool does not currently indicate the type of data carried on a channel, but the channels are typed (in Python). When trying to execute a mis-connected network, the tool will generate an error.

2.3. Configuring a Component

When working with the visual tool some components will need to be configured. These components should have their individual configuration functionality specialised for their specific purpose. A method is provided for the user to configure the component and save this setting in the `.csp` file, for later execution. A typical example of component configuration is something that allows the user to specify the name of a data file. To handle this, a structure is defined that a component has to implement in order to provide a configuration functionality.

We will now focus on the three issues of configuring a component:

1. Activate the configuration process.
2. Save the new configuration.
3. Load saved or default configuration on execution.

As mentioned in section 2.1.2, the Python implementation of a component is a file that we import, with its own name-space. If this name-space has a function named `setup()`, we call this function when the user configures the component. If the function does not exist, the user will not be able to configure the component. To save the configuration, any structure returned by this `setup()` function is serialized and saved in the component's `.csp` file. When executed, the component's top-level function is provided with the previously saved unserialized data structure. An example of a small configurable component is shown in listing 2.

It is left to the individual component programmer to decide what user interface will be used to configure the component. In the example shown in listing 2, a `wxWindows` file dialog is used to acquire input from the user.

The configuration data may be saved on several levels. When working with CSPBuilder a configuration can be saved on the working level or on any lower level, down to the rank where the process implementation is located. As standard all saved information from setting up components is saved in the working process and not in the process with the implementation. This gives the possibility for different setups for every application, and necessary to create components that are as general as possible. Saved configurations are attached to the process instance.

Configuration data with a higher rank will override any configuration data with a lower rank. This has the desired effect: that any configured process instance of a component will use the most recent configuration, as long as it is activated in the main application, and not as part of any other component.

2.4. Process Replication

When building applications for concurrent scientific computing, a common way to organize the calculations, if the algorithms allow it, is to divide the calculation into different jobs and process these concurrently with workers. An application that use 50 workers would quickly become cumbersome in CSPBuilder because of the 50 process instances in the visual tool. To address this issue, a process multiplier is created. When enabling the process multiplier on a process instance, the user must enter the desired number of replications.

Any channels connected to a process instance where a multiplier has been set, can be thought of as being multiplied by the corresponding amount. The addition of extra channels and processes is handled in the execution step.

```python
configurable = True
from common import *
import pylab

default_data = None

# Configuration (called from builder.py)
def setup(data = default_data):
    import wx
    import os
    wildcard = "PNG (*.png)|*.png|"      \
               "All files (*.*)|*.*"

    saveDir = os.getcwd()

    dlg = wx.FileDialog(
        None, message="Choose an image file, containing the data",
        defaultDir=os.getcwd(),
        defaultFile="",
        wildcard=wildcard,
        style=wx.OPEN | wx.CHANGE_DIR
        )

    if dlg.ShowModal() == wx.ID_OK:
        paths = dlg.GetPaths()
        data = paths[0].replace(saveDir + '/', '')

    os.chdir(saveDir)
    dlg.Destroy()
    return data

# CSP Process (called from execute.py)
def ReadFileFunc(out0 , data = default_data):
    img = pylab.imread(str(data))
    out0(img)
```

Listing 2. An example of a component that has configuration enabled

On execution, a multiplier x will cause the specified process instance to be created in x exact copies. If the process instance is an instance of a process network this network will be multiplied in x exact copies, creating x times the number of processes and channels in the process network. When a process is multiplied, all connections are multiplied as well and will be turned into One2AnyChannels, Any2OneChannels or Any2AnyChannels.

3. Concurrent Execution

In this section we describe how a data structure, constructed by the visual tool and saved to `.csp` files, is executed successfully. This is done by converting the data structure into a structure resembling a CSP process network. The PyCSP library is used to construct processes and their connections, and finally to execute those processes.

All functionality presented by the visual tool in section 2 must be handled in the execution step. Here we will focus on the requirements relevant when executing on a single system. The non-trivial functionalities required include: channel poisoning; multiplication of components and their connections; importing external code; and releasing the *Global Interpreter Lock*.

3.1. Building and Executing a Process Network

The overall goal is to build a network that will have a performance similar to a network implemented entirely in Python using PyCSP. This means that all parsing and network building needs to be done before execution and cannot be done on demand. To improve performance, the tree data-structure describing processes is first flattened, as shown in figure 5.

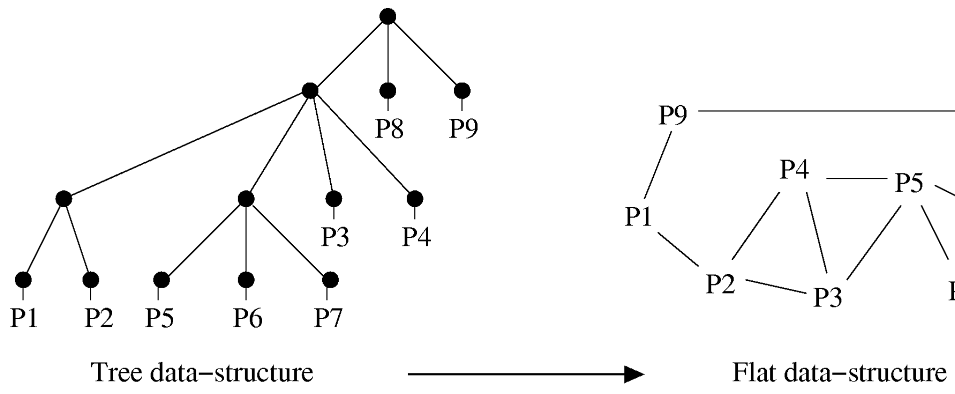

Figure 5. Data structures. In the left figure the tree data-structure is illustrated, which represents the structure of the CSP network when the `.csp` files are parsed. The black dots are a process structure and the lines represent any number of connection structures. This data-structure is converted into the flat data-structure illustrated in the right figure. This is a one-way conversion and can not be reversed.

An important feature in the construction of CSPBuilder has been to resemble the CSP algebra in the visual tool. During execution it is equally important to execute the CSPBuilder application exactly as it was built, and to ensure that everything is executed correctly. Here we focus on guards, channel poisoning, importing external code and releasing the *Global Interpreter Lock*, which comprise the difficult parts of executing a CSPBuilder application.

3.1.1. Multiplying Processes

Multiplying a process only makes sense in cases where a computation is embarrassingly parallel, meaning that the problem state can be sent to a process and the process can compute a result using this state data, with no dependencies, and send the partial result to a process that collects all partial results into a final result. This design is usually called a producer-worker or a producer-worker-collector setup and works best with embarrassingly parallel problems. A dynamic orchestration of processes is used where the amount of workers can be varied easily and you can have many more jobs than workers, making it easier to utilize all processes. If a computation can not be done in a dynamic orchestration design, then it does not make sense to use this multiplier flag. Instead a static design can be built with specialized components for doing a parallel computation with $2, 4, 8, ...$ processes.

Another design where multiplying processes will be applicable is in process networks handling streams. Imagine 4 processes connected in serial, doing different actions on a stream. If one of these steps is more time-consuming than any of the others, it will slow down the entire process. Multiplying this process is simple and if hardware is available for the extra process, it improves the overall performance of the process network.

3.1.2. Channel Poisoning

In CSP, without channel poisoning, a process can only terminate once it has fulfilled its task. This creates a problem when a process does not know when it has fulfilled its task. When constructing a network of communicating processes most of the processes will be the kind that will never know when they have fulfilled their task. They will read from their input channels, compute and send the resulting data to their output channels. These processes combined will compute advanced problems and loop forever. One might add a limit saying that a process will do 500 loops and it can consider its task fulfilled. In some applications this is possible, but most applications can not define the needed loops prior to execution. Also one might construct an extra set of channels that will communicate a signal to the processes letting them know that their task is fulfilled, and initiate a shut-down. Channel poisoning is a clever method to do just that, but uses communication the channels that already exist. PyCSP has support for channel poisoning, which is based on channel poisoning in JCSP [13,14].

Channel poisoning is implemented in PyCSP by raising an exception in process execution, when a channel connected to this process is poisoned. The exception is caught by the PyCSP library and poisons all other channels connected to this process. After poisoning all channels connected to the process, the process terminates. This will eventually terminate all processes and cause the entire application to exit as desired.

If a process is currently waiting on a non-poisoned channel, then nothing will happen in the process until it reads or writes from one of its poisoned channels. This might happen if a process is waiting for an action and it is another process that has poisoned the network and desires that the application terminates. The application will stall until the action happens and the process writes or reads to the poisoned network.

For this reason when constructing CSPBuilder applications it is important to consider how an application is poisoned if the user wants the application to terminate at some point.

3.1.3. Importing External Code

The wizard for CSPBuilder described in section 2.1.4 provides an easy method for building a component that calls into C, C++ or Fortran code. In this section the framework for using external code in CSPBuilder is described.

Using the import statement in Python it is possible to import modules. A module can be a Python script, package or it can be a binary shared library, as in this case where we want to use code from other programming languages.

For importing Fortran code the F2PY [8] project is used, which is capable of compiling Fortran 77/90/95 code to a binary shared library, making it accessible for Python. To import C or C++ code the SWIG [7] project is used to compile to binary shared libraries, similar to F2PY. Both projects are wrappers that make it relatively easy to handle data conversion between Python and other languages.

All external code will reside in the `External` folder in the CSPBuilder directory. A module name specifies a sub-directory in `External`, where all source and interface files are located. When compiled, the generated module will be saved as a ‘`.so`’ file with the module name as its file name in the `External` directory. A *Makefile* is created for every component and for the entire `External` directory, so that all modules can be compiled by executing `make` in the `External` directory. This is necessary when applications are moved to different machines, where the architecture and shared library dependencies may vary.

3.1.4. Releasing the GIL

PyCSP [9] uses the Python *treading.Thread* class to handle the execution of processes in a CSP network. This class uses kernel threads to implement multi-threading which should enable PyCSP to run concurrently on SMP systems. Unfortunately concurrent execution of threads is prohibited by the GIL. The GIL (Global Interpreter Lock) is a lock that protects access to Python objects. It is described in the documentation of Python threads [15]. Accessing Python objects is not thread-safe and as such cannot be done concurrently.

To be able to utilize the processors in an SMP system we will release the GIL while doing computations outside the domain of Python. In section 3.1.3 it was explained how external code can be imported into Python. When calling into Fortran code using F2PY the GIL is released automatically and acquired again when returning to Python. With C and C++ the situation is different, because here it is possible to access Python objects by using the API declared in `python.h`. It is the responsibility of the component developer to not access

Python objects while the GIL is released. Releasing and acquiring is done with the following macros defined in `python.h`:

```
// Release GIL
Py_BEGIN_ALLOW_THREADS

// Acquire GIL
Py_END_ALLOW_THREADS
```

The effects of releasing the GIL can be seen in section 4.1 where experiments are carried out on an SMP system. We have now covered relevant issues in the building and execution of a process network and can construct a CSP network from the `.csp` files created in the visual tool.

3.2. Performance Evaluation

A classic performance test for CSP implementations includes the Commstime [16] test, which is commonly used for benchmarking CSP frameworks. This computes the time spent on a single channel communication. In this test we will compare the performance of the Commstime test written in “Python with PyCSP”, with the CSPBuilder created “Commstime” application shown in figure 6. The CSPBuilder Commstime creates a CSP network in PyCSP and should perform the same, with perhaps only a slight overhead of having to create the extra *DataValue* process. In table 1 the result of the tests are shown. When comparing, there is a slight difference where the *DataValue* process is concerned, but this process is necessary to initialise the network and cannot be removed from the application. In “Python with PyCSP” this data-value is a simple integer.

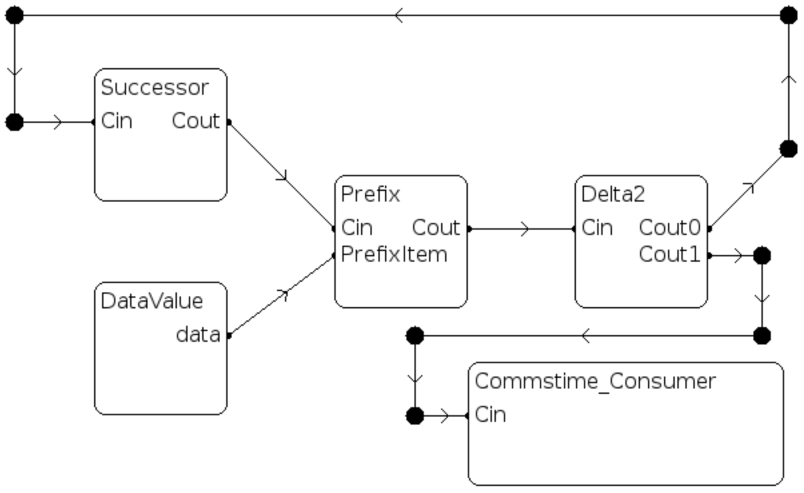

Figure 6. Commstime. A CSPBuilder application that resembles the Commstime performance test.

Table 1. CSPBuilder Commstime. A comparison of the channel communication time when using CSPBuilder vs. only Python and PyCSP. The Commstime tests were executed on a Pentium 4 2Ghz CPU.

Test	Avg. time per. chan (μs)
Python and PyCSP	91.43
CSPBuilder	96.30

The results of CSPBuilder are as expected. The performance of Python and PyCSP are not competitive to many other CSP implementations, especially compilable languages. However, Python has many other advantages that in our case outweigh the poor performance:

- Easy to use and very flexible.
- Can interact with most languages.
- Many scientists already know Python.
- Faster development cycle.
- Encourages programmers to write readable code.
- Compute intensive parts can be written in compilable languages.

4. Experiments

In this section we test the performance of CSPBuilder using a simple *Prime Factorization* experiment. The tests will be performed with a varied amount of workers in the application. Workers are the processes that, because of the design of the process network, are meant to be identical, run concurrently and compute sub-problems of a larger problem.

The experiments show that CSPBuilder is capable of executing applications on an 8 core SMP system. On the 8 core SMP system the GIL is released to be able to utilize all cores successfully.

4.1. Prime Factorization

As a test case for executing applications in CSPBuilder, *Prime Factorization* was chosen. It is simple and the computation problem can easily be changed to run for varying times. In the book by Donald Knuth [17], 5 different algorithms for doing prime factorization are explained. The simple one is the least effective and is based on doing *trial division*[2]. *Trial division* is used in the *direct search factorization*[3] algorithm. The simple prime factorization algorithm was chosen for the following reasons:

- Parts of the algorithm can to be written in both C and Python. The simplicity of the algorithm is an advantage here.
- The nature of the algorithm makes it possible to use the *multiplier* functionality in CSPBuilder. The algorithm is easy to divide into jobs that can be computed by workers.
- With a simple algorithm it will be easier to identify the aspects that do not perform well.
- The algorithm has limited communication, but still enough to test various cases, e.g. distributed vs. one machine.

A serialized Python implementation of the *direct search factorization* algorithm can be found at PLEAC[4] (the Programming Language Examples Alike Cookbook). This implementation is extended and adapted to a parallel version that we implement in the CSPBuilder framework.

4.1.1. Implementation Details

The *prime factorization* problem is built as a component reading a number as input and outputting a result. Since *direct search factorization* is an embarrassingly parallel problem, the processing can be divided into jobs and handed over to a set of workers as illustrated in figure 7.

On initialisation, the worker process sends an empty result to the controller, to indicate that it is ready for more work. The controller loops until all primes have been found, sending jobs to and collecting results from workers. If a non-empty result is received, the controller

[2]Trial division: `http://mathworld.wolfram.com/TrialDivision.html`
[3]Direct search factorization: `http://mathworld.wolfram.com/DirectSearchFactorization.html`
[4]PLEAC: `http://pleac.sourceforge.net/pleac_python/numbers.html`

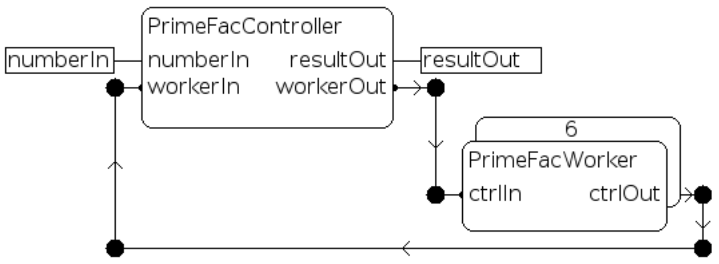

Figure 7. PrimeFac Component, consisting of a controller and a worker multiplied 6 times.

waits for all workers to finish and, if any other workers also had a non-empty result, the best result is picked and the computation resumes.

If n is the number we are factorizing into primes, then all primes have been found when $d >= \sqrt{n}$, where $[2 \dots d]$ are the divisors tested. All the prime factorizations of n can be found in $[2 \dots \sqrt{n}]$.

Numbers that are particularly interesting to factorize into primes are those larger than the representation available generally in compilers (e.g. 32-bit and 64-bit). To work with unsigned integers larger than 18446744073709551615, which is the limit for $64bit$ registers, some special operations are needed. Numbers larger than this need software routines for doing basic operations such as addition, subtraction, multiplication and division.

Python has internal support for large numbers which makes the task of implementing prime factorization in Python much simpler. Creating the C version is a bit more tricky. An external component is created using the wizard described in section 2.1.4. To test the implementation, a version working with numbers less than $64bits$ is created. All basic mathematical operations are then replaced with function calls to the library “LibTomMath”[5], which handles large numbers. For transferring large numbers between Python and C a decimal string format is used.

Finally we add a release for the GIL as described in section 3.1.4, which enables us to maximize concurrent execution in the application.

4.1.2. Performance Evaluation

For our experiments the Mersenne[6] number $2^{222} - 1$ is used. This number was picked by trial and error, with the purpose to find a number where the prime factorizations could be computed within 30 minutes for the least effective run. All tests have solved the problem:

$$\begin{aligned} n &= 2^{222} - 1 \\ &= 6739986666787659948666753771754907668409286105635143120275902562303 \\ &= 3^2 * 7 * 223 * 1777 * 3331 * 17539 * 321679 * 25781083 \\ &\quad * 26295457 * 319020217 * 616318177 * 107775231312019 \end{aligned}$$

In the performance test we compare the two implementations, one with the worker written in Python and one with the worker as an external component written in C which also releases the GIL. In the C implementation we use the large number library *LibTomMath*. This large number implementation is actually slower than the large number implementation in Python, shown in the tests where the “Python only” version outperforms the “Python and C” version for the case with only one worker. We base this conclusion on the fact that the sequential test for “Python and C” finishes in 1547 minutes, while the “Python only” version

[5]LibTomMath: `http://math.libtomcrypt.com/`

[6]Mersenne number: `http://mathworld.wolfram.com/MersenneNumber.html`

finishes in 1005 minutes. Both implementations spend all of the execution time in the worker loop with very little communication between processes.

To compare the effects of adding more workers we examine tests with 1, 2, 4, 6 and 8 workers, shown in figure 8. The “Python and C” version performs well, and by looking at the speedup in figure 9, we see that performance scales almost linearly. This means that adding double the amount of workers on a system with double the capacity doubles the performance and halves the run-time. The speedup shown in figure 9 is not quite linear. The drop in performance is caused by having to flush the workers every time a result is found. Time is then spent sending new jobs to workers. This overhead increases with the number of workers, but is largely acceptable given the advantages and benefits of this approach. All benchmarks were run on an 8 core SMP system.

The increase in run-time, when adding workers to the “Python only” version in figure 8, is caused by the unnecessary context-switching and communication, since the added workers will only steal CPU time from the first worker. The reason that the run-time only increases by a little even though many workers are added, is that the other workers are starved and therefore will never ask for a job to compute.

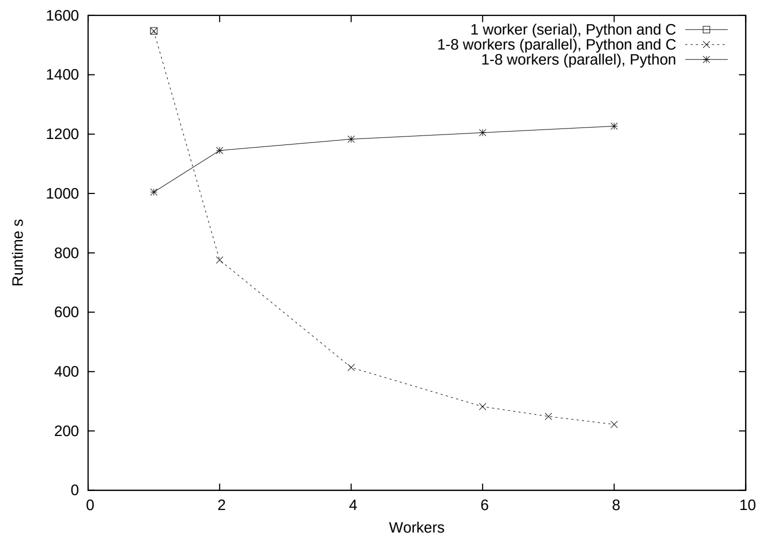

Figure 8. Prime Factorization of the Mersenne number $2^{222} - 1$.

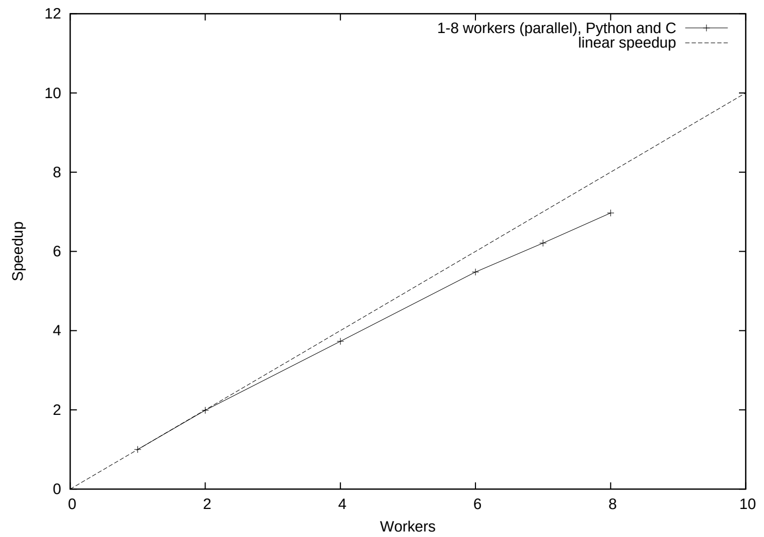

Figure 9. Speedup of prime factorization of the Mersenne number $2^{222} - 1$.

The sequential benchmark is based on single worker execution. This is arranged by setting the job size to 10^{16} iterations, which causes only one job to be sent to the single worker waiting. This benchmark provides a baseline reference for sequential execution speed in CSPBuilder, and is used as the basis when calculating the speedup of the parallel benchmark shown in figure 9.

These results show us that when constructing a scientific workflow in CSPBuilder, it is possible to get a reasonable performance and avoid the GIL, by programming the computationally intensive components in compilable languages. CSPBuilder is usable for both coarse-grained and fine-grained construction of whole systems. With a coarse-grained process network, we require the computation intensive components to execute concurrently internally, if a reasonable performance is desired. With a fine-grained process network, internal concurrency in the components is not necessary. The *prime factorization* implementation is somewhere in between a coarse-grained and fine-grained network.

5. Related Work

Several different frameworks exist that can handle scientific workflows in different ways. To mention some of the more common, there are *The Kepler Project*[7] [18], *Knime*[8], *LabVIEW*[9], *FlowDesigner*[10] and *Taverna*[11]. The graphical tool of CSPBuilder is a quite similar to these frameworks, though currently less functionality is available in CSPBuilder. CSPBuilder differs by having a basic graphical tool, that assists in constructing a CSP network and manages a component library. The power of the CSPBuilder framework lies in the communication model based on CSP.

On the CSP side, Hilderink [19] has created a graphical modelling language, GML, in which CSP networks can be defined.

6. Conclusions and Future Work

In this paper we have presented a graphical framework for designing and building concurrent applications based on CSP. Ideally suited to current and future multi-processor and multi-core machines, CSPBuilder provides a simple and intuitive means for designing concurrent applications. The graphical tool compiles directly to Python using PyCSP, and supports transparent integration of C, C++ and Fortran functions. Experiments have shown that near linear speedup can be obtained on embarrassingly parallel applications, which demonstrates that the CSPBuilder tool dos not impose any significant overheads.

This paper has hinted at the distribution of CSPBuilder applications on networks of workstations and other distributed memory architectures. Although PyCSP does support networked channels, some modifications to the basic channel code in PyCSP have been made as part of the work presented here. Similar changes will need to be made to the network channel code in PyCSP before CSPBuilder is able to target these architectures.

It might also be interesting and useful to add more descriptive visual representations of channels, inspired by Hilderink, such as identifying guarded choice on channel inputs to a process.

[7]The Kepler Project: `http://www.kepler-project.org/`
[8]Knime: `http://www.knime.org/`
[9]LabVIEW: `http://www.ni.com/labview/`
[10]FlowDesigner: `http://flowdesigner.sourceforge.net/`
[11]Taverna: `http://taverna.sourceforge.net/`

Although CSPBuilder is at a relatively early stage of development, we hope that it will grow and flourish, eventually becoming a useful tool to aid scientists in constructing scientific workflows, as well as for the programming of CSP based concurrent applications generally.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, june 21, 2004 edition, 2004.

[2] The CSPBuilder Framework. http://www.migrid.org/vgrid/CSPBuilder/.

[3] Description of Moores Law. http://www.intel.com/technology/mooreslaw/. Viewed Online January 2008.

[4] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. *Intel White Paper*, 2005.

[5] Annoncement: 80 core CPU. http://www.intel.com/pressroom/archive/releases/20070204comp.htm. Viewed online september 2007.

[6] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[7] Simplified Wrapper and Interface Generator (SWIG). http://www.swig.org. Viewed online january 2007.

[8] F2PY - Fortran to Python interface generator. http://www.scipy.org/F2py. Viewed online January 2008.

[9] Otto J. Anshus, John Markus Bjørndalen, and Brian Vinter. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

[10] Peter Y. H. Wong and Jeremy Gibbons. A Process-Algebraic Approach to Workflow Specification and Refinement. In *Proceedings of 6th International Symposium on Software Composition*, March 2007.

[11] Peter Y. H. Wong. Towards A Unified Model for Workflow Processes. In *1st Service-Oriented Software Research Network (SOSoRNet) Workshop*, Manchester, United Kingdom, June 2006.

[12] Flow-Based Programming. http://en.wikipedia.org/wiki/Flow-based_programming. Viewed online september 2007.

[13] Communicating Sequential Processes for Java. http://www.cs.kent.ac.uk/projects/ofa/jcsp/. Viewed online january 2008.

[14] Berhnard H.C Sputh and Alastair R. Allan. JCSP-Poison: Safe Termination of CSP Process Networks. *Communicating Process Architectures 2005*, pages 71–107, 2005.

[15] Thread State and the Global Interpreter Lock. http://docs.python.org/api/threads.html. Viewed online january 2008.

[16] Neil C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, sep 2003.

[17] Donald E. Knuth. *The Art of Computer Programming - Volume 2 - Seminumerical Algorithms*. Addison-Wesley, third edition, 1998.

[18] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.

[19] G.H. Hilderink. Graphical Modelling Language for Specifying Concurrency Based on CSP. *IEE Proceedings - Software*, 150(2):108–120, 2003.