Distributed Scientific Computing in Python

Master Thesis

February 2008
Supervisor Brian Vinter

Rune Møllegård Madsen <downey@diku.dk>

Department of Computer Science
University of Copenhagen
February 2008

Abstract

The manufactorers of processing units are moving towards multi-processor systems to avoid the *memory-wall*, *power-wall* and *frequency-wall*. The 8-core CPUs and the CELL available on the market today proves this tendency. To program for higher concurrency more synchronization is necessary than when programming for a few CPUs.

Working with eScience involves scientific computing and requires large computation resources to run even simple simulations. This thesis designs and implements a framework to assist in developing highly concurrent applications using a scientific workflow. The scientific workflow in the developed framework is based on the CSP algebra and is able to construct and execute CSP networks. Code reuse is encouraged as applications are constructed by reusable components. The developed framework makes it simpler to construct complex scientific applications focusing on the workflow and letting the CSP algebra find the available concurrency in the application.

The framework developed here is capable of mixing different hardware and different programming languages in the same application making it relevant for scientific applications. The performance tests proves the potential of this framework.

Contents

1 Introduction

This thesis presents a framework targeted for clusters and the CPU designs of tomorrow. CPUs are produced with several cores today and every next CPU generation is improved with more cores. This calls for more concurrency in the applications developed.

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 has been off limits, because of the difficulty of concurrent programming.

The major challenges faced in this thesis includes creating a graphical user interface to create and edit CSP networks, design a component system that works well with CSP and Python, extend PyCSP with support for networked channels, design a method to group processes on nodes and run experiments on the framework to find the possibilities and limits.

In section 1.1 we will explain the background and the motivation for doing this thesis. A short introduction to the scientific workflow model and communicating sequential processes is then followed. The summary of contributions is listed in section 1.4. Finally an outline of the thesis is available in section 1.5 and a short list of often used terms in section 1.6

1.1 Background and Motivation

The companies producing CPUs have over the past decades been making processing units that are faster for every new edition. The faster speeds have been reached by decreasing the size of transistors and increasing the complexity of processors. The number of transistors on a chip have been doubled every 2 years over the past 40 years, as declared by Moore's Law [3]. Doubling the amount of transistors is not enough to gain faster CPU speeds, so control logic and memory logic is added to utilize the many transistors. More pipelines are added increasing the throughput. Unfortunately more pipelines mean more branch-prediction logic and this has the effect that it becomes very expensive to flush the pipeline, when a branch is predicted wrong. Many other extensions and complexities have been added to the CPU design during the past 40 years to gain faster CPUs.

Today a wall has been hit. The amount of transistors is still doubled every second year, so Moore's Law still holds. However three problems have been raised, the *power wall*, *frequency wall* and *memory wall*. The heat dissipation and power consumption increase with 3 percent for every 1 percent increase in processor performance, this according to Intel[12]. Also Intel [12] explains that because of bigger differences between memory access speeds and CPU performance, memory becomes a bottleneck. The last wall is that the pipeline has become too long, so the cost of flushing outweighs the performance increase of increasing the pipeline to attain higher CPU frequencies. This all means that we can go no further with the current designs, and Intel suggest in the referenced paper that the next step is parallel computations.

With several processing units, you avoid the *power wall*, *frequency wall* and *memory wall* since you do not have to increase the processor performance for a single unit. Instead you must be aware of communication and synchronization between processes, which can cause overhead and deadlocks if used wrongly.

Today computers are getting more and more processing units, which can be utilized by creating concurrent applications that will scale towards many processors. Recently Intel announced that they are experimenting with an 80-core CPU[1].

To get effective computing power for scientific computing we have to program for concurrency and for as many nodes as possible. This means handling communication and synchronization in our programming, making it a lot more difficult to program even simple applications. Also time has to be spent identifying parallel parts of our problem.

These issues are addressed in this thesis by creating a framework, that should be usable by most people capable of programming in Python.

The idea is that CSP (see section 1.3) would be a natural way to build parallel scientific computing applications. Typically you read data, run different functions analyzing and processing the data and then write the result. This design would be simple to create with CSP and also convey the idea to the people working with science.

As a programming language for our tool we are selecting among the languages that have CSP available at its core or as a library. Among those, we have Occam[7], Erlang[4], C++[19], Python[11] and Java[2]. Occam and Erlang are removed as possibilities because they have limited libraries available for them, which would make it hard to create all kinds of applications in these languages. Java and Python are similar in performance compared to C++ which is much faster. Python is easier to extend and works better with other languages than Java, so Java is removed as a possibility. Looking at the advantages and disadvantages of Python and C++ for framework.

Features relevant to CSPBuilder (the framework)	**Python**	**C++**
Works with CSP	Yes (in testing)	Yes (in testing)
Fast development time for prototypes	Yes	No
Large code-base	Yes	Yes
Can interface with almost any language	Yes	Yes
Good performance	No	Yes
Works well with threads	No	Yes

Table 1: **Comparing Python and C++.**

We choose Python, even though it is slower and does not run threads concurrently. Python does not perform well with threads because of the GIL (Global Interpreter Lock)[10]. Only one thread can access Python objects, which eliminates the possibility of running two Python threads in parallel which both access Python objects. It is possible to avoid the lock

by calling into another language, releasing the lock, doing the work there and returning, attaining the lock again. While executing outside of Python it is possible to run the threads simultaneously. We plan that processes containing CPU-intensive work are programmed in another desired language that also might be able to do the work more effectively. This should solve the problem with threads and since we are doing the CPU-intensive work in languages with better performance, the performance of Python is no longer an issue. Whether this is true will be tested in the thesis.

Another advantage of using implementations in other languages is that it is possible to reuse complex functions, improving development time. The framework users should be able to use different very fast libraries in the same application, without having to put much thought into the concurrent part. The CSPBuilder framework, developed in this project, will make the application run concurrently providing an easy use of already built components eg. "Fast Fourier implementation".

When doing scientific work and relying on already built libraries to do the number crunching, the functions you need to do math are not necessarily implemented in the same programming language. This can sometimes be difficult to solve and we want to address this issue by using SWIG[9] and F2PY[5], that makes it possible to use code from C / C++ and Fortran in the same program and make it work together.

When executing a distributed application, the available nodes may be different in many ways, like the type of architecture, amount of CPU, memory and other hardware differences. Some components may be executed faster on some nodes than on others, and we would like to take advantage of this fact in order to optimize the execution, by moving executing parts of the application to nodes optimal for this kind of computing.

The original idea for this project was presented by Brian Vinter at DIKU[1].

1.2 A Scientific Workflow Model

The purpose of a scientific application usually lands in the area of calculating a result based on 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 the application when it is a scientific application. We can now use the term scientific workflow.

Scientific workflow is in this thesis used for the workflow of scientific applications in the eScience field. A scientific application might be anything from climate modelling to prime factorization. Any application that does a large number of computations to produce a result within a scientific field.

Several different applications exist that can handle scientific workflows in different ways. To mention a few of the very common:

[1] Datalogisk Institut Koebenhavns Universitet

LabView `http://www.ni.com/labview/`
"With LabVIEW, engineers and scientists can interface with real-world signals; analyze data for meaningful information; and share results through intuitive displays, reports, and the Web.

FlowDesigner `http://flowdesigner.sourceforge.net/`
"FlowDesigner is a free (GPL/LGPL) data flow oriented development environment. It can be used to build complex applications by combining small, reusable building blocks."

Taverna `http://taverna.sourceforge.net/`
"The Taverna project aims to provide a language and software tools to facilitate easy use of workflow and distributed compute technology within the eScience community."

Only a few[21, 20] have previously looked at the design of the CSP algebra and thought that this might be a very good description of a scientific workflow. In this thesis we will produce an application that use some of the ideas from CSP algebra, *LabView*, *FlowDesigner* and *Taverna* and combine these to create a framework where CSP applications can be designed in a visual tool and then executed on any number of nodes. This has not been done before.

One reason for working with scientific workflows is to enable access to large computation resources for people who have not earlier had the ability to access those. The model produced in this thesis makes it possible to divide the computation from a small number of CPU-cores to hundreds of nodes on different networks. Combined with a visual tool to design CSP networks, this is a scientific workflow model.

1.3 Communicating Sequential Processes

CSP[13] is a mathematical theory known as process algebras. CSP was introduced in 1978 by C.A.R. Hoare. It has since then evolved substantially and is now available in several programming languages either as an external library or the programming languages themselves have been based on the CSP algebra. C.A.R. Hoare published a book[14] on *Communicating Sequential Processes* in 1985 which is the bible on the subject. Here we will very briefly mention the constructs from CSP that we will use in this thesis.

Processes are created and can either run successfully or fail. Depending on the outcome other processes are run. Processes can be created as a parallel or sequential group of processes. These can interact with channels, but every end of a channel can only be interacted with from one process. The possible actions on a channel are:

read Performing a blocking read.

write Performing a blocking write, that blocks until the result has been received with a read.

guard Used in the *Alternative* construct to listen on several channels. Only one channel is read from.

An *Alternative* construct is also blocking and will block until one process writes to one of the channels connected. If several processes write at the same time, only one process will be allowed to finish the write while the other is blocking until a read occurs.

PyCSP[11] the CSP library for Python used in this thesis has the functionality just mentioned. In PyCSP processes are kernel threads. PyCSP has 4 channel types:

One2OneChannel Connects two processes.

One2AnyChannel One process is able to write on this channel and any number of processes are able to issue a read. Only one process will receive the result and it can be any of the reading processes. All other reading processes will be blocked until a result is ready for them.

Any2OneChannel One process is able to read from this channel and any process can issue a write. When several processes write concurrently, one is picked and the other are left as is.

Any2AnyChannel Any process is picked for writing and any process is picked for reading the result. All processes block until their read or write is carried out.

This is a small subset of CSP and we will not use the CSP syntax since the constructs used are related to what is possible in PyCSP and relevant for this thesis. For this purpose it is simpler to call the constructs by the names they have been given in PyCSP. Using CSP is an obvious choice because of the combination of processes and channels, that can easily be visualized.

1.4 Summary of Contributions

A new framework is implemented, tested and benchmarked in this thesis. 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 enables users to use C / C++ and Fortran code easily by providing a wizard to enable these programming languages. The framework is called CSPBuilder and also incorporates extensive use of the CSP algebra.

The visual tool provides an "easy to use" graphical user interface, that enables users to construct applications using the ideas of flow-based programming[6] with the target to produce a CSP[14] network. In the experiments it is shown that the visual tool is capable to handle large and complex applications.

A CSP library for Python is extended with complete support for networked channels. These are tested and benchmarked. This is the first working CSP library for Python that has complete support for networked channels.

Applications constructed with CSPBuilder can be executed succesfully on a machine and combine different programming languages in one application. With the use of the CSP library for Python (PyCSP) it is possible to execute the applications on any number of hosts. It is shown that it is possible to get performance to scale linearly or better.

The CSPBuilder framework is able to handle anything from small amounts of communication in the byte range to large amounts in the Gbyte range. During tests it was possible to transfer data with rates of $30Mbyte/s$ using networked channels developed in this thesis.

The framework benefits code-reuse because applications are constructed by reusable components. This has proven very useful during the experimenting phase.

The real power of this framework lies in code reuse and constructing complex scientific applications focusing on the workflow and letting the CSP algebra find the available concurrency in the application.

1.5 Outline of Thesis

This is a short description of the contents of each major section in this thesis. In the sections 2, 3 and 4 the CSPBuilder framework is documented. The framework consists of a visual tool which is documented in section 2, a construction of the actual CSP network in section 3 and enabling the distributed execution of the CSP network in section 4. To test the new CSPBuilder framework experiments are carried out in section 5. This includes a *Succesive Over-Relaxation* experiment and a *Prime Factorization* experiment that illuminates possibilities and bottlenecks within the framework.

1.6 Terms Used in Thesis

- **The tool** is developed for this thesis and is named "CSPBuilder"
- **The developer** is the person developing components for "CSPBuilder"
- **The user** is the person building "CSPBuilder applications" by connecting components

2 The Visual Tool

This section describes a user-friendly application that can model a CSP network using a layout like flow-based programming[6]. This layout is required to resemble the CSP network for the scientific workflow model in section 1.2. Figure 1 shows an application modelled using "the visual tool" and developed for this thesis. We will explain the non-intuitive parts in the development of this tool.

In section 2.8 the overall test results are that the visual tool performs as expected and can produce the CSP applications that are needed as input for the concurrent execution in section 3.

In the next sections we will explain: The component system. Connecting the components with channels and connection points. Saving and loading CSP applications to or from files. Configuring a component. Multiplying components. Debugging CSP applications.

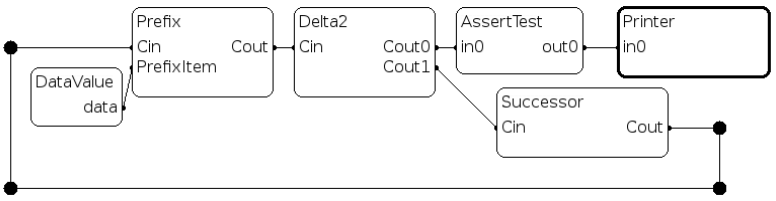

Figure 1: **`Generating Numbers`** A CSPBuilder application that generates incrementing natural numbers. The components Prefix, Delta2 and Successor are common CSP processes.

These parts are necessary to construct an application and are part of the framework which will make it possible to build CSP networks that can be run efficiently in a distributed environment.

2.1 The Basics

The purpose of this tool is to provide a fast and simple method to develop CSP applications that will run in a distributed environment. Some requirements for the graphical user interface and usability are defined, without any experiments determining the need, but purely based on the ideas of what would be "nice to have", and what would be "need to have" when using a graphical tool to construct CSP networks. The remaining requirements concerning the visual tool are based on what data is needed to construct and execute a CSP application.

In CSPBuilder every application will start with a blank stage, where processes and channels can be inserted. Processes from CSP will appear as squares with name and connections written onto them. Channels will be shown as lines connecting the squares (processes). A process can then have for example 2 connections going in and one connection going out. This means that the user can connect 2 channels from processes which have connections going out and 1 channel from a process with an inbound connection. To simplify things any inbound or outbound connection will only accept one channel going in or out, depending on the connection type.

When having connected a number of processes, we have a process network. This network could be used as a component in another application, which is further described in the next section 2.2.

2.2 Component System

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

- We need to link the Python code of each process in an easy to understand framework, to make it simple to add or remove components.

- The building and organization of the process network needs to be scalable, which means that the user should be able to handle very big and complex applications, without losing control or overview.
- The user should be able to group parts of the process network into components, looking and functioning like processes.
- It has to be quick and easy to build components consisting of process networks.
- Components should be stored in a library for reuse.
- The entire application built in CSPBuilder has to be transferable into a scalable CSP process network, that runs just as fast as an equivalent application written entirely in Python.

These requirements are addressed in the next sections.

2.2.1 Scalable Organization

Imagine a huge process network of 2000 processes. To handle this many processes and even many more channels, it is necessary to group parts of the network into smaller items. This could be done by enabling the user to select a group of connected processes and turn them into an item. If this item has unconnected inbound or outbound connections it would be similar to a black box and from an outside view be identical to other components. When the functionality of this process network was desired in an application, it could be inserted like any other component. Doing it like this would solve the problem of reuse and complexity, since any application can be built by a small number of components that again are built by a small number of components. Any component can then be reused in other components, creating a tree data-structure, where the leafs are the component implementations.

This tree data-structure solution is chosen for CSPBuilder. If a component includes a process instance of the same component or one in a higher rank a cyclic dependency arises. The rank being the number of components that include a given component. Handling and protection against cyclic dependencies is out of the scope of this thesis.

In section 2.2.5 it is explained how to maintain the structural integrity and create a CSP process network that is as flexible as possible and that executes the developed application just as fast as an equivalent application written entirely in Python.

2.2.2 A Component

The most important part of CSPBuilder are the components. A component is a CSPBuilder application that has been moved to the component library. The new component is then available and can have two different forms:

1. The component is a process network consisting entirely of process instances of other components and includes no implementations of any CSP processes.

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 CSP process. A very simple example of a CSP IDProcess (process which forwards input onto its output channel) is available in listing 1.

Listing 1: **EXAMPLE** An example of a CSP IDProcess implementation.

```
1 from common import *
2
3 def CSP_IdProcessFunc(cin, cout):
4     while 1:
5         t = cin()
6         cout(t)
```

To make it as easy as possible for the user to create components we define that to create a component, you just have to copy/move your CSPBuilder application to a Components folder. When the CSPBuilder application reloads the library it discovers this new component and makes it available for insertion into new applications.

Functions are developed that are specific for building components. This includes naming unconnected channel-ends and giving the main application a name. The name is used as the component name, and the naming of unconnected channel-ends is used for describing the inbound / outbound connections for our new component.

2.2.3 Component Library

To be able to organize components every component will need a package name. The reason for this is to make it easier to find the desired component. For the CSPBuilder to be an effective tool, it will need as many components in the component library as possible.

2.2.4 A Wizard for Building Components

A developer should be able to reuse code made by others or 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 is 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 is available for free use. It is desired 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[9], to import code from C / C++ and most

programming languages are able to build libraries that can be used from C / C++, thus making it possible to extend Python with code written in all kinds of languages. A project named F2PY[5] can import Fortran 77/90 code into Python.

With the wizard we will guide the user through the process of creating components written in Python, C / C++ and Fortran 77/90. These languages were chosen because numerous scientific libraries are available in these. and as mentioned earlier most languages can build a library, which is accessible from C / C++.

The inclusions of other programming languages is expected to have a positive effect on the performance of applications in CSPBuilder. Python uses a lock *Global Interpreter Lock* (see section 3.1.6) to access Python objects. This means that only one Python thread is allowed to access Python objects, removing any advantage of running threads not dependent of each other. This lock can be freed when executing external code imported into Python, making it efficient to have certain parts written in other languages. Also executing compiled languages vs. interpreted languages is usually faster, again improving performance.

2.2.5 Structural Integrity

The component system describes components and networks of components. The description of components includes all the necessary details to construct a CSP network. The actual implementation of the CSP process, the channels connecting the CSP processes and the types of channels used in the CSP network.

The visual tool will work with the component system, to visualize processes and channels, such that the user can expect that the appplication constructed in CSPBuilder is executed exactly like it appears in the visual tool. The CSP algebra[13, 14] is used as the model for the construction of the component system and the visualization of processes and channels in the visual tool. When executing a CSPBuilder application the entire CSP network is constructed and initialized using PyCSP. This guarantees that an application constructed in CSPBuilder will run equally fast as a similar application constructed with Python and PyCSP.

The basics and requirements for the "Component System" have now been addressed. The next section describes how to construct and visualize channels in the visual tool.

2.3 Channels and Connection Points

Channels connect processes creating a process network. The different types of channels available and how they work in PyCSP is introduced in section 1.3. The types of channels are One2OneChannel, One2AnyChannel, Any2OneChannel and Any2AnyChannel.

The One2OneChannel is simple, because it can be represented by a line going from one process to another, while representing the other channel types is more complex. First it was thought that the functionality could be available through accepting several connections to

the same inbound or outbound connection on a process. The idea is pictured in figure 2, where the first channel is a One2AnyChannel and the last channel is a Any2OneChannel. Figure 3 shows the problem with this concept. This looks like an ambiguous representation, which can be translated into three different channel setups:

1. A One2AnyChannel connecting three processes and a One2OneChannel connecting the last.
2. A One2OneChannel connecting two processes and an Any2OneChannel connecting the three.
3. An Any2AnyChannel connecting all processes.

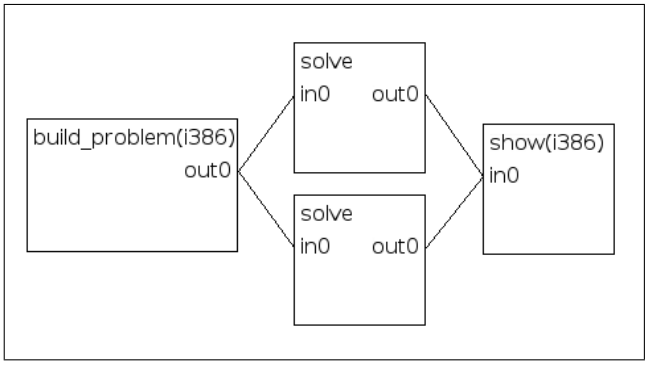

Figure 2: **`Channel Example`** The first idea for representing a One2AnyChannel and an Any2OneChannel.

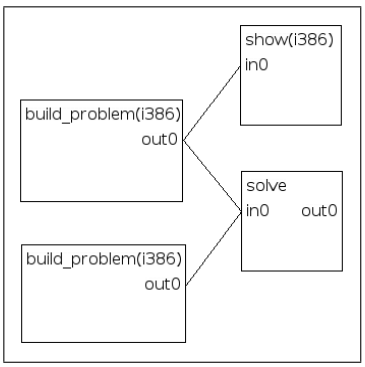

Figure 3: **`Channel Example`** The first idea for representing an Any2AnyChannel.

It is concluded that this representation would create too many problems because it would not be obvious to the user which kind of channel was used.

To address this issue we introduce connection points (figure 4). These have an unlimited amount of inbound and outbound connection points. This makes it possible to connect any

process to any other process and to bend channels. What determines which channel type a connection point is part of, is decided by the following rules:

If more than one connection point exists in a channel, as shown in the bottom example in figure 4, then we can also reduce it to a channel with one connection point collecting all connections to all processes connected by this channel.

Using the same method to reduce all redundant connection points, it is possible to simplify the process of detecting the channel type. We can now calculate the amount of inbound and outbound connections to processes. These values identify the channel type that is being represented. Examples of the different channel types can be viewed in figure 4, 5, 6 and 7.

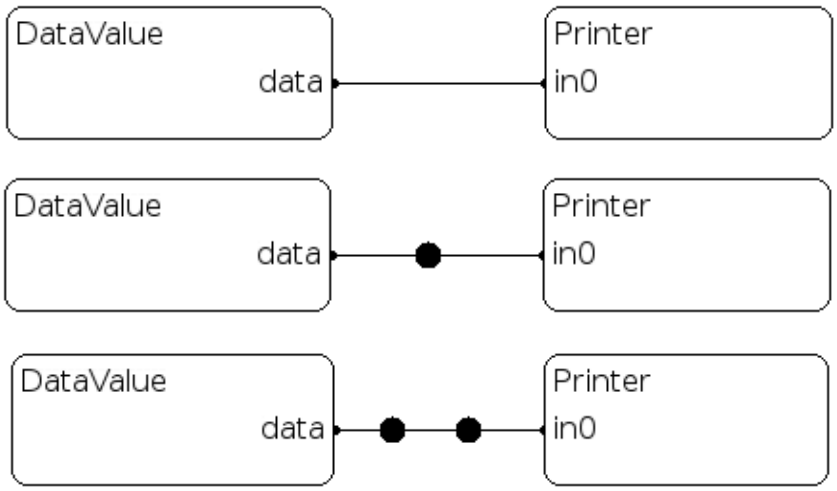

Figure 4: **`One2OneChannel`** 2 processes connected in 3 different ways, all valid. 1 outbound connection and 1 inbound connection makes a One2OneChannel. Note though, that the bottom example will be reduced to the middle example when calculating the channel type.

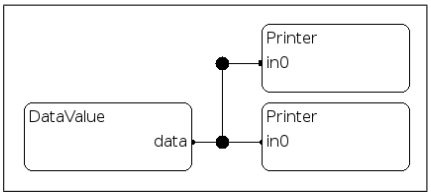

Figure 5: **`One2AnyChannel`** Three processes connected by a single connection point. 1 outbound connection and >1 inbound connections makes a One2AnyChannel.

An added bonus of reducing a channel to one connection point is to remove unnecessary cycles between connection points in the channel. A user might create a cycle between three connection points. This would just get eliminated since all connected connection points can

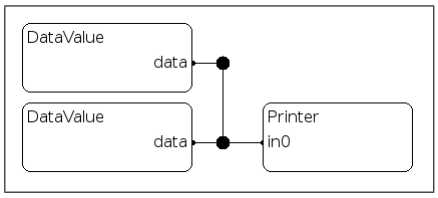

Figure 6: **`Any2OneChannel`** Three processes connected by a single connection point. >1 Outbound connections and 1 inbound connection makes an Any2OneChannel.

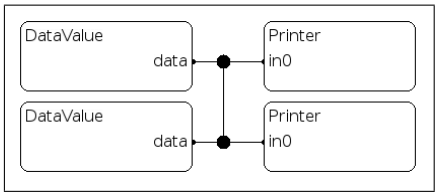

Figure 7: **`Any2AnyChannel`** Four processes connected by a single connection point. >1 Inbound connections and >1 Outbound connections makes an Any2AnyChannel

be viewed as being part of an invisible cycle with every other connection point. The problem of reducing is divided into sub-problems that can be solved recursively. The sub-problem consists of finding the neighbours of a connection point and checking whether they can be eliminated. The neighbour can be eliminated if it is a connection point. When eliminating a neighbour connection point all connections are moved to the existing connection point. This sub-problem is solved recursively for all connection points until only one is left. It runs in $O(n)$ time where n is the amount of connection points. When running the algorithm recursively on all neighbours we have the advantage that it can have any connection point as its starting point.

2.4 File Format and Structure

After constructing a network of components and channels we save the network to a file, to be able to load it at a later time.

CSPBuilder consists of two parts; the graphical tool which helps the user develop and connect processes into an application and a separate tool for executing a CSPBuilder application. Some times the user will not be enable to run the graphical user interface at the machine where the application is executed, which is why CSPBuilder has been separated into the two parts. To design a working data structure for an application the following requirements have to be met:

- All saved applications will need to be in a format that corresponds to the format for a component, as mentioned in section 2.2.2.

- The file format has to be able to handle a tree data-structure (section 2.2.1), where the rank height of the tree has no limits.
- A process can contain a process network or a process implementation, section 2.2.2.
- The design should be easily converted to a flat data-structure, which contains an unfolded process network, with a minimum amount of channels.
- The file format needs to be easy to be build, parse, edit, save and load.

As the file type, the simple choice would be to serialize an object structure to a file, using the Python module *pickle*[2]. This however would make it impossible to edit the application in a standard text editor and also make it harder to debug any errors in the data structure. Instead XML is chosen, since a library for building and parsing structures is available, thus saving development time. Another reason for choosing XML is its wide acceptance. If at some point someone would like to replace the graphical tool for another tool that could generate CSPBuilder applications this would probably be easier with XML, because XML is a well known format.

Next is to describe the items and their relations to see what is required of the XML design. The items that need to be saved are processes, process instances, channel-ends, code, connection points and channel items.

To illustrate the relations between items a design is declared in Backus-Naur form and then later translated to an XML design. The conversion of this structure to XML is simple and will not be documented in this thesis.

```
<doc> := <process>
<process-list> := <process> | <process> <process-list>
<process> := <name>
             <package>
             <channel-end-list>
             <channel-list>
             <code>
             <process-list>
             <processlink-list>
<code> := <import> <mainFunc>
<channel-end-list> := <channel-end> | <channel-end> <channel-end-list>
<channel-end> := <direction> <name> <target>
<direction> := 'in' | 'out'
<channel-list> := <channel> | <channel> <channel-list>
<channel> := <name> <in-connection> <out-connection>
<in-connection> := 'in' <target>
<out-connection> := 'out' <target>
<processlink-list> := <processlink> | <processlink> <processlink-list>
<processlink> := <name> <instance-name> <cspfile> <w> <h> <x> <y>
```

[2]Standard module in Python, that is able to serialize and unserialize Python data objects.

```
<connection-point-list> := <connection-point> | <connection-point> <connection-point-list>
<connection-point> := <name> <x> <y>

<cspfile> := filename of component .csp file
<name> := text
<package> := package name for grouping processes
<instance-name> := Unique name of process. Used for identification.
<target> := path to existing process in process hieraki
```

With this structure it is possible to save an application built in CSPBuilder and later reload it recreating all structures, placing these at the correct locations with the right settings.

2.5 Configuring a Component

When working with the visual tool some components will need to be configured. These components should have their individual configuration functionality specialized 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. An example of a configuration could be to enter the file name of a data file. To address this issue, a structure is defined that a component has to comply with in order to implement a configuration functionality.

We will now focus on the 3 issues of configuring a component:

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

As explained in 2.2.2, the Python implementation of a component is a file that we import, which has its own name-space. If this name-space has a function named `setup()`, we will call this function when a user wants to configure a 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 `.csp`-file, by the `builder.py` application. On execution a function named `load(data)` will be called with the previously saved unserialized data structure. An example of a miniature configurable component can be seen in listing 2.

It will be up to the individual component programmer to decide what user interface will be used to configure the component. In the example in listing 2, `raw_input()` is used to get input from the user.

The configuration data can 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, which is necessary

to create components that are as general as possible. Saved configurations are attached to the process instance.

The data structure from section 2.4 is extended with the following:

```
<process> := <name>
             <package>
             <channel-end-list>
             <channel-list>
             <code>
             <process-list>
             <processlink-list>
             <config>
<processlink> := <name> <instance-name> <cspfile> <w> <h> <x> <y> <config>
<config> := <config-data> <configured-process>
<code> := <import> <mainFunc> <configurable>

<configurable> := 'yes' | 'no'
<config-data> := serialized base64 encoded configuration data
<configured-process> := path to configured process in process hiarki
```

Any configuration data with highest rank will overwrite configuration data with lower rank. This has the desired effect that any configured process instance of a component will use the recent configuration, as long as it is activated in the main application and not in any components.

2.6 Process Multiplier

When building applications for doing 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 using 50 workers would quickly become very cumbersome in CSPBuilder because of 50 process instances floating around. To address this issue, a process multiplier is created. When enabling the process multiplier on a process instance the user can enter the amount of processes wanted. In the previous case the amount would be 50.

The data structure of a process instance is extended with a multiplier:

```
<process> := <name>
             <package>
             <channel-end-list>
             <channel-list>
             <code>
             <process-list>
             <processlink-list>
             <config>
             <multiplier>
<processlink> := <name> <instance-name> <cspfile> <w> <h> <x> <y>
                 <config>
```

```
                <multiplier>
<multiplier> := integer
```

Any connected channels to a process instance where a multiplier has been set, is thought of as multiplied in the corresponding amount. The adding of channels and processes will be done in the execution step.

On execution the 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 a One2Any or a One2All Channel. In section 3.1.2 a broadcast flag is introduced which implies whether a One2Any or a One2All Channel is created.

2.7 Debugging Applications

CSPBuilder is a framework designed to help build applications. This framework produces code for communication and does some work behind the scenes, which could make it difficult to detect the errors existing in an application. To assist the user / developer a set of debugging tools are designed.

Developing good debugging functionality is difficult. To be useful they need to be robust even though the executed applications deadlock or fault. Limited debugging tools will be implemented in CSPBuilder and are explained here, as well as suggestions for further development of debugging tools.

Debugging channels can be used for detecting an error and localizing this error. Also it can help to eliminate other components from suspicion and as such, more time can be spent focusing on the faulty parts. Deadlocks and faults in channel poisoning can be particularly difficult to identify, since no output can be received from a network which has deadlocked.

2.7.1 Current Debugging

To be able to identify and localize an error, it is possible to monitor a channel by selecting it for debug in the graphical user interface. This prints out all data sent over the channel. To identify which channels are enabled for debugging, the data structure from section 2.4 is extended:

```
<channel> := <name> <in-connection> <out-connection> <debug>
<connection-point> := <name> <x> <y> <debug>
<debug> := 'yes' | 'no'
```

With this extension it can be determined which channels have debugging enabled on execution. Since the execution is handled by PyCSP the debugging is implemented in PyCSP.

To avoid that the added code will affect performance when executing with no debugging enabled, two versions of PyCSP is created. One with debug functionality and a release version that has no debug functionality.

The monitoring of channels stops when an application deadlocks. If enabled, a debug command-shell is run that listens to commands from the keyboard. This debug command-shell will be accessible on execution and has two main functions. It can output all information about processes, showing whether its connections are active or not, where active means that a process is either waiting to write or waiting to read. The other function is to output information on all channels, displaying which channels have been poisoned and how many processes that are active on a channel.

2.7.2 Better Debugging

It is difficult to handle listening to more than one channel at a time, since output is mixed and it is not easy to identify channel names with the channels in CSPBuilder. A map of the process network with all channel names displayed would make it a lot easier to identify where data belongs. With the debugging tools developed in this thesis, it is recommended not to debug more than one channel at a time.

2.8 Correctness Test

Now the visual tool is tested whether it provides the expected functionality explained in past sections. If only a small part of a test fails and does not make the tool unusable, but merely inconvenient, the test is marked as "Partial OK", otherwise "OK" or "FAILED". The test results can be viewed in table 2. Three tests was partially ok and commented below:

Comment 1 Is not able to correctly notice whether the "Save" menu item should be disabled or not, depending on whether any changes have been made to the application.

Comment 2 Selecting lines between processes and connection points can be very difficult, since the area for selecting them equals the size of the line.

Comment 3 Tabs can not be entered correctly in the edit window.

The comments are mere inconveniences in the usage of CSPBuilder and will not affect any results in this thesis. The overall test is concluded a success.

2.9 Summary

A tool `builder.py`, that enables users to design CSPBuilder applications and save them to `.csp` files, has been developed for this thesis and design decisions have been explained.

The work required to produce this tool has been extensive since we needed a robust application. It is used to construct the `.csp` files for the correctness and performance tests

Listing 2: **EXAMPLE** An example of a component that has configuration enabled

```
text = 'Default'
number = 0

def setup():
    text = raw_input('Input string:')
    number = input('Input number:')
    return [text, number]

def load(data):
    text = data[0]
    number = data[1]

def process(out0, out1):
    out0(text)
    out1(number)
```

Test case	Result	Comment
New / Open / Close / Save / Save As / Quit	Partial OK	1
Delete / Reload	OK	
Toggle Grid	OK	
Set Package Name, Process Name, Preferred Arch.	OK	
Set / Clear Icon	OK	
Resize, Minimize, Maximize Window	OK	
Insert Component or Connection Point	OK	
Create Channel	OK	
Select and Delete Channel / Process / Connection Point	Partial OK	2
Moving Processes or Connection Points	OK	
Selecting Multiple Objects	OK	
Setting Alternative, Broadcast and Debug Channel State	OK	
Reduce Connection Points and Channels	OK	
Detect Channel Type on Right-Clicking Connection Point	OK	
Multiply Processes	OK	
Lock Process to Node	OK	
Set Workload of Process	OK	
Edit Name of Process and Connections	OK	
Configure Process	OK	
Edit CSP XML Window	OK	
Edit Python Code Window	Partial OK	3
Create New Component with the Wizard	OK	

Table 2: **Testing correctness of CSPBuilder `./builder.py`** The graphical user interface of CSPBuilder and construction of csp files. Tests are done to the extent that each functionality is tried and the result checked.

during this thesis. Making it robust has cost many hours, but just as many hours have been spend using the tool. It has always worked correctly and proved simple to build the applications that tested the ideas presented in the thesis.

3 Concurrent Execution

In this section a data structure constructed by the visual tool and saved to `.csp` files is executed succesfully. This is done by converting the data structure to a structure resembling a CSP network, constructing this CSP network with the PyCSP library by starting the process network and finally executing the processes in the network.

Every functionality presented by the visual tool in section 2 must be handled in the execution step. We will focus on the requirements relevant when executing on a single system. The non-trivial functionalities required include channel poisoning, simulating channels for the CSP *Alternative* construct, multiplication of components and their connections, importing external code and releasing the *Global Interpreter Lock*.

3.1 Building and Executing a Process Network

The goal is to build a network that will have similar performance 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 from the tree data-structure available in the data-file and shown in figure 8, a flat data-structure is constructed where all processes exist on the same level connected with the smallest possible amount of channels. A component containing a process network with several connections will then be unfolded and the processes in the process network connecting to the outside will be connected directly to processes in other components depending on how the CSPBuilder application is designed.

In the construction of CSPBuilder an important feature has been to resemble the CSP algebra in the visual tool. When executing it is equally important to execute the CSPBuilder application exactly as it was built, to make sure everything is executed correctly. For these purposes we will in this section focus on enabling guards for all channels, channel poisoning, importing external code and releasing the *Global Interperter Lock*, which designate the difficult parts of executing a CSPBuilder application. Also in the visual tool we have introduced some elements that do not exist in a CSP network. Among these are multiplying processes and One2AllChannels (broadcasts). In this section we will describe their part in CSPBuilder.

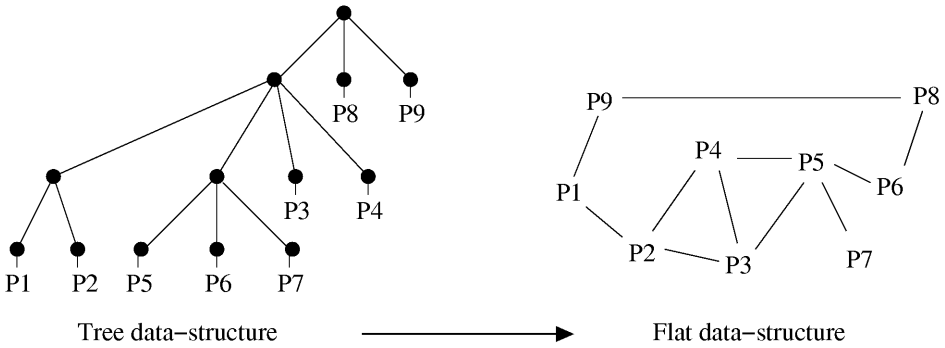

Figure 8: **`Data structures`** In the left figure the tree data-structure is illustrated, which represents the structure of the CSP network when the `.csp` files are parsed. The black dots are a process structure and the lines represent any number of connection structures. This data-structure is converted into the flat data-structure illustrated in the right figure. This is a one-way conversion and can not be reversed.

3.1.1 Multiplying Processes

On execution, the 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, then this network will be multiplied in x exact copies, creating a lot of processes and channels.

When a process is multiplied all connections are multiplied as well and will be turned into either One2AnyChannel, One2AllChannel or a Any2OneChannel depending on direction and whether the broadcast flag is set.

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 orchestration design can be built with specialized components for doing a parallel computation with 2/4/8/x processes. The *Successive Over-Relaxation* in section 5.2 is an example of this.

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 Adding a One2AllChannel

A One2AllChannel allows a broadcast functionality in the CSP network. This functionality is very easily created with a process that reads from one channel and outputs on all other channels. In section 3.1.3 this method is used to simulate the functionality of a One2AllChannel.

It was decided to implement a One2AllChannel in PyCSP even though it can be simulated. The reason was that a small performance gain is achievable, because an extra process is added to the network when simulating vs. using an actual channel. Also if 1000 processes were connected to this One2AllChannel process then 1000 One2OneChannels are created which is avoided with the actual channel.

The One2AllChannel does not support guards, since it was based on the implementation of the One2AnyChannel, that also does not support guards. To use guards, channels can be simulaleted using the method explained in the next section 3.1.3.

3.1.3 Simulating One2Any and One2All Channels

The current PyCSP version does not support using One2Any or One2All channels as a guard in the *Alternative* construct. In CSPBuilder we need to be able to use all channel types as guards. Components in CSPBuilder will be used in different applications and if a component used a channel as guard, this component would then be limited in its use in CSPBuilder. To avoid this issue a simple solution is chosen.

The One2Any channel is simulated by One2One channels and a process reading from a channel and then selecting a One2OneChannel to write data to. When data is written to a regular One2Any channel, data will be sent to any process that reads from the channel. To be able to support this a polling functionality is added to the One2One channel. Polling a channel returns a value telling whether a process is ready to read or not. The process simulating the One2Any channel is now able to select a channel that a process is ready to read from and send data to this process. The simulated One2All channel is similar, but broadcasts the read data to all One2One channels connected.

3.1.4 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 channels, compute and send data to a channel. These processes combined will compute advanced problems and loop forever waiting to read from channels or send to channels. 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. Channel poisoning is a clever method to do just that, but by communicating the signal across the channels that already exist. PyCSP has support for channel poisoning, which is based on channel poisoning in JCSP[2, 17].

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 it terminates the thread. This will eventually terminate all threads and cause the entire application to exit as wanted.

If a process is currently waiting on another channel but the one that is poisoned then nothing will happen until the process writes or reads to the poisoned channel. 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 think about how an application is poisoned if the user wants the application to terminate at some point.

3.1.5 Importing External Code

The wizard for CSPBuilder described in section 2.2.4 provides an easy method to build 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[5] 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 / C++ code the SWIG [9] project can compile to a binary shared library just as F2PY. Both projects are wrappers that make it easier to handle data conversion between Python and the external code.

All external code will reside in the `External` folder in the CSPBuilder directory. A module name specifies a directory in `External` where all source and interface files are located. When compiled the generated module will be saved as a `.so` file with 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 the applications are moved to another machine since the architecture and the shared libraries can vary between machines.

3.1.6 Releasing the GIL

PyCSP[11] uses the Python *treading.Thread* class to handle the many 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 documented in the documentation of Python threads[10]. Accessing Python objects is not thread-safe and as such cannot be executed concurrently.

To be able to utilize the processes in an SMP system we will release the GIL while doing computations outside the domain of Python. In section 3.1.5 it was explained how to go outside the domain of Python by importing external code. When calling into Fortran code using F2PY the GIL is released automatically and acquired again when returning to Python. With C / 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, not to access Python objects while the GIL is relased. 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 5.1 where experiments are carried out on an SMP system.

We have now covered relevant issues in the builing and execution of a process network and can construct a CSP network from the `.csp` files created in the visual tool. In the next section it is tested whether the functionalities implemented works correctly.

3.2 Correctness Test

To test for correctness several simple process networks are built, each demonstrating a functionality of CSPBuilder and testing whether it works as expected. To help check for correct execution the AssertTest component is developed.

3.2.1 The AssertTest Component

Inspired by Unit testing CCUnit[3] and the assert command, an *AssertTest* component is built. Unit testing is where tests are built for specialized functions and run after an implementation change to test whether all functions still operate correctly. The assert command is a quick

[3]CCUnit: `http://ccunit.sourceforge.jp/`

way to test if an expression is true or false, and if false, quits execution and writes out an assertion error, helping the developer to find and correct the error.

The component *AssertTest* is defined to have three functionalities: Record, Play and Test. The component will have one in-bound connection and one out-bound connection. It can be configured with a state file variable and a mode. Mode can be one of the three modes below, and state file will be the file containing recorded data.

- **Record** all data going through the process to the state file. It will work as a gateway and send all data out on its out-bound connection.
- **Play** all data in state file to its out-bound connection. The in-bound connection will accept data.
- **Test** all data on the in-bound connection with the recorded data-set in the state file. If any differences occur between the data it will exit and state that the test has failed.

The above three modes are to be used in process networks, to check that data traversing the *AssertTest* component is identical in every run. In this thesis it is used in the execution tests to test that built process networks still work correctly after changes have been made to the implementation.

3.2.2 Execution Tests

Tests are done by executing `execute.py` with different CSP applications utilizing the different functionalities available. If a test is executed as expected it is marked "OK" otherwise "FAILED". The test results are viewed in table 3.

All tests in table 3 performed as expected and thus we can expect that `execute.py` will handle any CSPBuilder application that we would like to execute.

3.3 Performance Evaluation

A classic performance test for CSP implementations include the Commstime[19] 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" and a Commstime test showed in figure 9 constructed in CSPBuilder. 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 4 the result of the tests are shown. When comparing, there is a slight difference where the *DataValue* process is to blame, but this process is necessary to initialize the network and can not be removed from the application. In the "Python with PyCSP" this datavalue is a simple integer. The results of CSPBuilder are as expected.

Test case	Result
Simple application that uses One2OneChannels	OK
Simple application with component written in C	OK
Simple application with component written in Fortran 77	OK
Simple application that uses a One2AllChannel	OK
Simple application that uses a One2AnyChannels and an Any2OneChannel	OK
Simple application with a process network inserted as a component	OK
Commstime	OK
Simulated One2AnyChannel to support guards	OK
Simulated One2AllChannel to support guards	OK
Multiply process instances according to the multiply setting	OK
Load the configured data of a process instance	OK
Debugging channel	OK
Debug configured data of process instances	OK
Debugging channels and processes with the debug shell	OK
Printing CSP statistics	OK

Table 3: **Testing correctness of CSPBuilder `./execute.py`** The execution of CSP files. Tests are carried out to the extent that each functionality is tried and the result checked. The output for the tests are printed in appendix C.1.

Test	Avg. Time per. chan
Python and PyCSP	91.43μs
CSPBuilder	96.30μs

Table 4: **CSPBuilder Commstime** A comparison of the channel communication time when using CSPBuilder vs. only Python and PyCSP. The difference is caused by the extra *DataValue* process. The Commstime tests were executed on a Pentium 4 2Ghz CPU and the output is printed in appendix D.1.

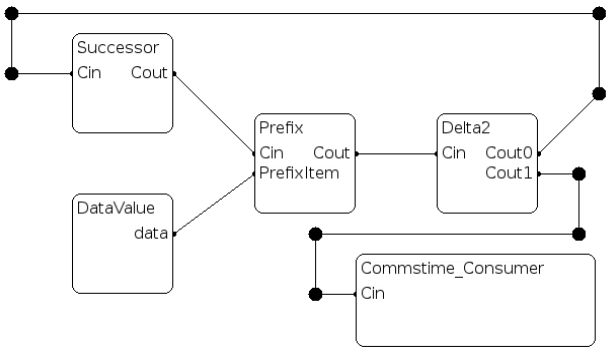

Figure 9: **`Commstime`** A CSPBuilder application that resembles the Commstime performance test.

3.4 Summary

In section 2 we have described the visual tool. This produces a data structure which is saved in `.csp` files. The execution of the process network which is described in the data structure is made possible by the work described in this section.

Applications can be created with the visual tool and they can be executed with the execute tool, that will create a threaded CSP network, capable of running concurrently, provided the *Global Interpreter Lock* does not prevent it. It is identified how to avoid the effects of the *Global Interpreter Lock* and how to multiply processes for more concurrency. This is tested in the *Prime Factorization* experiment in section 5.

All tests were run successfully and the overhead of CSPBuilder is very small compared to running a similar application purely with Python and PyCSP. Using the features documented in this section it is possible to construct applications that will run concurrently on a SMP machine utilizing all cpu-cores available.

4 Distributing the CSP Network

In this section CSPBuilder is improved with the possibility to extend execution across several nodes removing the limitations of the GIL (see section 3.1.6) while also making a lot more processing power available, since the amount of available processors in one machine is no longer a limitation.

The requirements for running a distributed version of CSPBuilder are explained in section 4.1. Then PyCSP is extended with support for networked channels and a method for grouping processes on nodes is presented. This section successfully implements the needed functionality. The advantage of distributing the CSP network is tried in the experiments section 5.

4.1 Requirements

For CSPBuilder to be useful, we require that any application has to always run correctly no matter where the processes are located. Even in the case where n processes in an application are located on n machines, having only one process on each machine, making every channel a networked channel, we require that the application is executed correctly.

Adding support for running on different architectures is expected to boost performance in the applications which have components that perform better on specialized hardware.

An example of this could be to build components of the *Succesive Over-Relaxation* or *Matrix Multiplication* implementations done by Mohammad Jowkar for the CELL architecture[15]. This would enable the CSPBuilder application to use the power of the CELL architecture while using standard i486 architecture for other tasks, increasing performance of the entire application while improving development time. Because of the multi-core organization and

the memory organization on the CELL architecture, it is expected that developing applications is harder and more time-consuming than for the standard i486 architecture. Being able to implement only the computation-heavy parts on the CELL architecture should save development time.

To be able to control the location of specific processes we require that processes can be locked to specific nodes. This is important for the following reasons: If in-data is lying on a specific host or if out-data has to be produced on a specific host. When a library required by a component is only available on a specific host. If a component includes an interactive dialog it is important that this dialog is run on the host where the dialog can be shown and interacted with.

When creating a CSPBuilder application it is organized into process instances of components. Some of these components consist of networks of process instances themselves. Pictured, it could look like figure 10. The 9 processes on the lowest level have to be grouped into n groups corresponding to n nodes. It is preferred to have the lowest amount of networked channels, since these are expected to perform worse than regular channels.

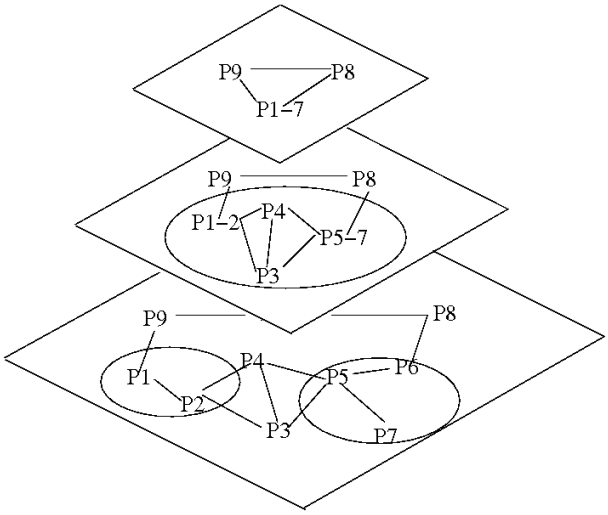

Figure 10: **`Process network structure`** This illustration visualises how the process network appears at the different levels, when looking up components containing smaller process networks. It is interesting because it shows how groups have internal connections. At level 1 only 3 channels are visible, at level 2 there are 8 channels visible and at the lowest level 11 channels are visible.

When distributing processes to nodes, often it will happen that scientific process networks that use static or dynamic orchestration in the processing will have a number of processes connected closely and all having a very heavy workload. To have any advantage of running on several nodes these heavy processes need to be spread across nodes. This means

that they have to be identified as processes with a heavy workload, which introduces the heavy workload tag in CSPBuilder.

To handle communication between processes placed on different nodes but connected by a networked channel we rely on PyCSP to correctly communicate and simulate One2OneChannels, Any2OneChannels, One2AnyChannels, One2AllChannels and guards. Also PyCSP has to be able to handle communicating the same data with networked channels as with regular channels.

4.2 Extending PyCSP with Network Channels

PyCSP is currently under heavy development and has reached version 0.2.4 at the time of this writing. It is becoming more and more mature by each release and now has working support for remote channels. The implementation uses the library Pyro[8] to communicate objects and channel-ends by providing proxy objects that redirect method calls to the correct machines where the original channel-end object is located. The input ends of the remote channels support the *Alternative* construct and can work as guards when used with the channel types One2One and Any2One. This version was not available at the time when CSPBuilder was implemented, and as such was not a valid choice when having to choose the CSP implementation for executing CSP networks.

The performance of PyCSP version 0.2.4 is tested and compared with the performance of the chosen CSP implementation in section 4.2.3.

At the time when CSPBuilder was implemented a PyCSP version 0.2.3 was available. This version had serious problems with the *Alternative* construct and remote channels only worked correctly for One2OneChannels. This had to do with the way Pyro[8] was used, which created problems with locking of processes because a wait called on a remote location did not affect the process thread running on the local location like it should. The *Alternative* construct was disabled on the RemoteChannel because of limitations in the implementation.

Another solution that was tried was the *CSP Library for Python*, developed by Mustafa Temiz[18]. This had working support for the *Alternative* construct with remote channels but only supported One2OneChannels and used Pyro[8] for its communication like the forementioned PyCSP versions.

Either of the two options PyCSP 0.2.3 or CSP Library for Python can not be used for executing the CSP network in CSPBuilder, since they do not support the functionality required. They are lacking support for One2Any, Any2One and Any2Any channels and also working guards for these channels.

Since no other implementations of the CSP algebra exist for Python, the solution could be to lessen the requirements for the CSP implementation by accepting that only One2One channels can be networked. This would put limitations to how processes could be organized

on nodes and by this limit the available concurrency in the CSPBuilder application. Since CSPBuilder is targeted for people with less experience in concurrent applications it is important that we do not require people to keep with a "special" process structure to be able to run an application on a number of nodes. This leaves us with the only choice of adding the support for networked channels. To make this as simple as possible the PyCSP version 0.2.2 will be used since this is also the version that we have extended with support for One2All channels and debugging information. Also I find the structure of version 0.2.2 simpler than the newer structure, which utilises classes more extensively.

The PyCSP version 0.2.2 implements all processes by extending the threading class, and all channels have implementations that rely on locks that put process threads on wait queues or notify them when needed. Needed is when all ends of a channel are ready to send / receive data. The *Alternative* construct and guards make it a bit more complicated because when a process is waiting on the *Alternative* construct it do not wait to read on all channels, but just on the first to arrive. This requires messages to be passed back and forth to release the waits on all the other guards that were not selected.

In planning the extension of PyCSP it is decided that functionality is more important than performance, and a solution is found that requires more communication, threads and data handling but is simpler and solves most cases for CSPBuilder.

To handle communication in the new networked channels Pyro seems like the obvious choice. However Pyro is a huge library which can do a lot more than what is needed for this CSP implementation. In simple terms, Pyro can in our case be replaced with a TCP connection and serializing / unserializing Python data. There is no need for any Base Name Server that maintains shared objects. At creation of the CSP network the location of all processes is known, as is the TCP ports that are available on different nodes.

We have now explained the reason why an implementation of networked channels is developed for this thesis and that raw TCP connections are used for network communication.

4.2.1 Implementation Details

The implementation of network channels resemble the functionality in regular One2OneChannel operations. We require that every end of a channel has to always know the state of the opposite end, to be able to work as effectively as the regular One2OneChannels and also support the *Alternative* construct and guards. To transform a regular channel into a networked channel it is replaced by a One2NetChannel and a Net2OneChannel, which will have separated threads for handling network communication. This will make sure that both ends of the simulated One2OneChannel are at the same state in the communication process. The network synchronization commands needed for handling this and other channel types like One2AnyChannels and Any2OneChannels bring us to the following set of data communication:

Sent from the writing end to the reading end

"WRITEready" A process is ready to write. Used for implementing support for the *Alternative* construct and for the AnyNet2OneChannel

"poisonCLI_" A process has poisoned the channel. This poisons the connected process.

"64LLLLLLLL" The length of the data package. The reading is now aware that the next data will be the serialized objects and knows how many bytes to read.

"<64 byte pickled data>" The data, which is serialized objects or other Python data types.

"GET_states" Requests an update of whether "READ_ready" or "ALT_enable" are set at the reading end. This is necessary for a poll functionality, used with simulating the *Alternative* constructs for the One2AnyChannel mentioned in section 3.1.3.

Sent from the reading end to the writing end

"READ_ready" A process is ready to read.

"DATAisREAD" The serialized objects have been read. All states are reset.

"READ_RESET" A process is no longer waiting to read. Used in Any2OneChannels.

"ALT_enable" Informs that the reading end is being used as a guard.

"ALTdisable" Informs that the reading end is no longer used as a guard.

"poisonSRV_" A process has poisoned the channel. This poisons the connected process

"S_R_ready1" Answers a "GET_states" request. Reading end ready to read.

"S_R_ready0" Answers a "GET_states" request. Reading end not ready to read.

"S_A_ready1" Answers a "GET_states" request. Reading end used as guard.

"S_A_ready0" Answers a "GET_states" request. Reading end not used as guard.

For optimization reasons commands except the one containing serialized objects are in raw text and have a length of 10 bytes. This is for simplicity and to save the need for a header in our communication. Now we can always settle with reading 10 bytes. If the 10 bytes consist of a data length command ("nnnnLLLLLL") the next data read will be serialized objects with the length of *nnnn* bytes. The data containing serialized objects is serialized by using the Python module *pickle* to guarantee a safe transfer of any object type.

If support for guards was not necessary the extra threads ServerThread / ClientThread could be avoided and communication could be handled entirely by the One2Net / Net2One

classes, but since we need guards we need the threads to handle waking up the process with the *Alternative* construct.

When a One2OneChannel is replaced with a networked version, there are 4 threads and 2 sockets that have to be synchronized to provide the proper safety against deadlocks. For this we use 4 monitors, a threadMonitor for every thread controlling the sockets and a rwMonitor for every process thread calling the methods `.read` or `.write` on the channel class. Figures 11 and 12 display the synchronization needed when doing a normal read and a read with a guard. Guards are implemented exactly like with the regular channels, only adding the logic for maintaining the necessary state variables on all ends of the networked channel. The synchronization logic is based on the logic used in PyCSP regular channels.

Whether the synchronization logic is implemented correctly is tested in the correctness test, but since it involves threads we can not be 100 percent certain that no flaws are in the code, unless we proved the conditions for all cases. Proving the implementation for all cases is out of scope for this thesis, which is why we settle for the correctness test.

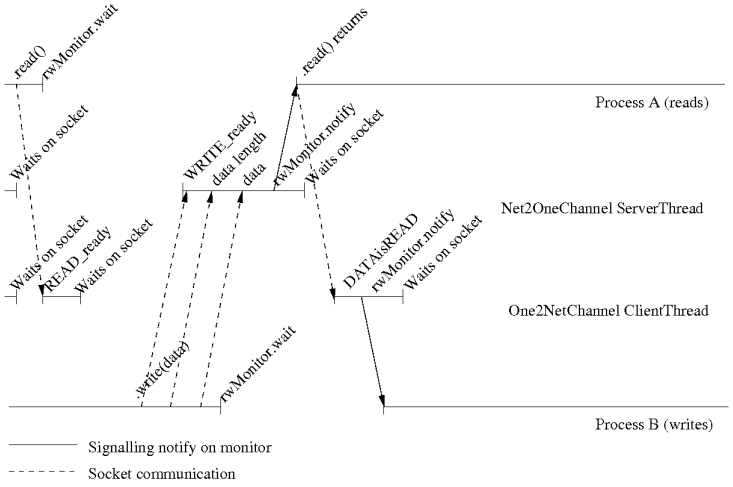

Figure 11: **`One2Net / Net2One channel synchronization`** Process A reads from Net2OneChannel, then Process B writes, synchronization happens and the channel transmission has completed.

When the process network is initiated, the `ServerThread` and the `ClientThread` connect to each other and these socket connections are maintained until the channel is poisoned.

In figure 12 a typical communication sequence is shown. The socket communication is visualized as dotted lines and looking at these, we see that Process A communicates to the `ClientThread` and Process B communicates to the `ServerThread`. Because of the synchronization logic it is guaranteed that communication at no time happens in both directions at the same time. One TCP socket should be enough to support 2-way communication and in the following it is tested whether this is recommended.

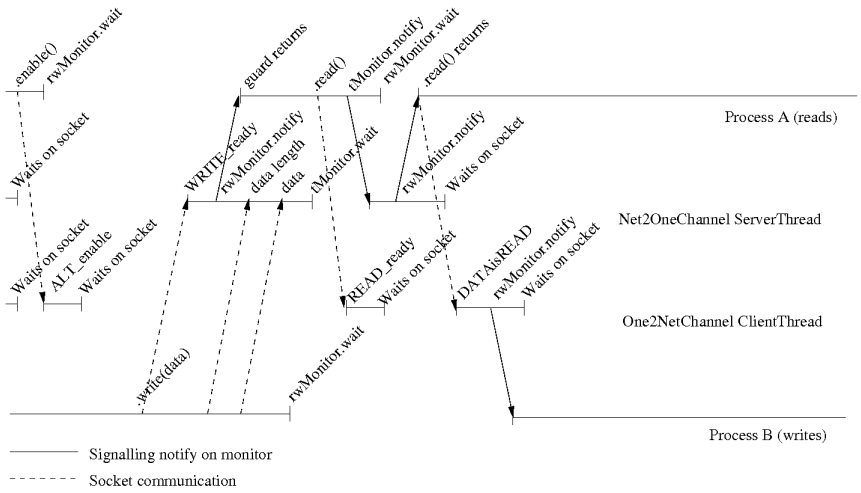

Figure 12: **One2Net / Net2One channel synchronization with guards** Process A initialises an Alternative construct with the Net2OneChannel as a guard and then calls `.priSelect`. This calls `.enable` and communication begins. Process B now writes which wakes Process A and a case which is similar to the situation in figure 11, but in the situation where Process B wrote data before Process A was ready to read. Process A can now choose to read from the channel or not. In this case Process A reads the data, synchronization / communication happens and the channel transmission has completed.

Since TCP connections are blocking 2-way communication, these properties can be used in the development of blocking channel ends. Some exchange of messages is also necessary, like poison messages, ready to read messages, ready to write messages and other control messages. These messages and data go back and forth the TCP connection, but not at the same time. They are nicely serialized since data can only be read or sent at one time.

During development tests have shown that 1 TCP connection has significant performance limitations. To test the extent of these limitations we do a test where we compare the use of 1 TCP connection with the use of 2 TCP connections for doing 2-way communication.

We expect that with 2 TCP connections, one for each direction instead of just one, a better performance should be reachable. To test this, communication times are measured for an implementation with 1 vs. 2 channels. These tests are performed by running the Commstime application and by running a simple 2-way communication test. In table 6 with the Commstime tests, the results showed that the case with just one TCP connection slowed down communication, which illustrated the performance limitations during development. It is possible that the communication could be done with another orchestration of communicating threads, which would have made the case with 1 TCP connection as fast as with 2 TCP connections. This is showed in table 5, where a simple 2-way communication test shows the 1 and 2 TCP connections performing equally well. The Commstime tests in table 6 show an obvious reason for choosing 2 TCP connections over 1 in our case. In the test with CSPBuilder Commstime (3 networked channels) the performance is 41 times faster for 2 TCP connections vs. 1 TCP connection.

2-way communication using	**Avg. of 3 runs**
1 TCP connection	2.23 s
2 TCP connections	2.10 s

Table 5: **Simple Communication Test 1 vs. 2 TCP Connections** Tests were carried out by doing concurrent communication with several threads, to test if 1 TCP connection vs. 2 TCP connections would have any difference on performance. There is a small difference in the performance which can possibly grow if more concurrent communication is added. The data communication done in this test is not a valid representation of the communication done in the Commstime tests in table 6. The tests were executed on a Pentium 4 2Ghz and output is printed in appendix D.2.

The performance limitations of using 1 TCP connection are expected to be caused by the socket module in Python or the socket operations in the operating system. To find the actual cause, a test case could be implemented in C-code, which would uncover whether this was a problem with Python or with handling of sockets in the operating system.

When a channel-end now has the control of a thread it is no longer possible to shutdown a process and the associated channels just by ending the function of the process, since the thread of the channel-end still exists. To be able to exit a process network it is necessary to poison all channels in the CSP network and avoid the case where all processes have finished executing but channel-end threads still exist.

To avoid developing the entire thread communication for every channel type, the One2Net (`ClientThread`) / Net2One (`ServerThread`) channels are able to function as channel-ends in simulating networked versions of the regular One2Any, Any2One and One2All channel. The Any2Any channel is not supported, but there is no reason why it should not work. An example of how it could be implemented is suggested in figure 16.

The implementations in figure 14, 15 and 16 are quite cumbersome when counting the number of threads and socket connections. In the case that only some connections to a One2AnyChannel were on the same node, we could optimize the application by not using TCP connections for these and instead implement another communication protocol equal to the one used in regular PyCSP channels, but with support for working together in a setup like the one in figure 14. It is decided that to keep it simple all channel implementations are based on the One2OneChannel in figure 13. As with regular channels, guards are only supported for One2OneChannels and Any2OneChannels.

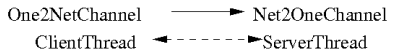

Figure 13: **`Networked One2OneChannel`** The dotted line is the socket communication and the solid line illustrates the connection.

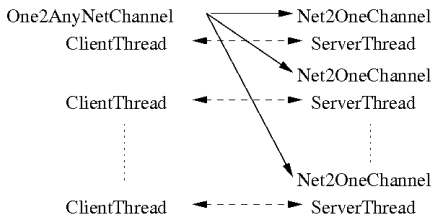

Figure 14: **`Networked One2AnyChannel`** The dotted lines are the socket communication and the solid lines illustrate the connections making it up for a regular One2AnyChannel.

With the extended PyCSP supporting networked channels, we can now position CSP processes on different hosts, increasing the potential performance. Any CSP processes can be positioned anywhere, regardless of the channel types used, because all regular channels now have a corresponding networked edition.

Benchmark	TCP conn per chan.	Networked chan.	Regular chan.	Avg. time per chan.
Commstime standard	NA	0	4	92.02 μs
Commstime net	2	1	3	210.95 μs
Commstime net	1	1	3	1171.05 μs
CSPBuilder Commstime	2	3	1	489.70 μs
CSPBuilder Commstime	1	3	1	20194.34 μs

Table 6: **Commstime Tests 1 vs. 2 TCP connections** The Commstime tests measures the time for doing one communication on a single channel. All have been done using the PyCSP version with the implemented NetChannel extension. A remarkable difference in performance is shown between using 1 vs- 2 TCP connections. The reason for the low performance in the case of 1 TCP connection is the blocking of TCP connections which conflicts with the concurrent communication. The tests were executed on a Pentium 4 2Ghz and the results are the average of 3 runs. Output is printed in appendix D.2.

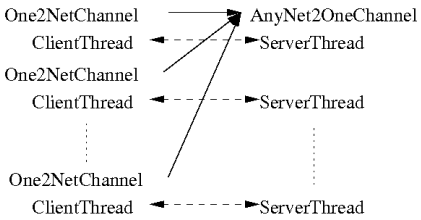

Figure 15: **`Networked Any2OneChannel`** The dotted lines are the socket communication and the solid lines illustrate the connections making it up for a regular Any2OneChannel.

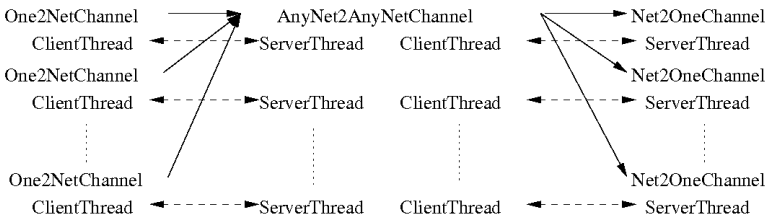

Figure 16: **`Networked Any2AnyChannel`** The dotted lines are the socket communication and the solid lines illustrate the connections making it up for a regular Any2AnyChannel.

4.2.2 Correctness Test

We are testing correctness by running small tests that utilize the different channels. The tests are written in Python and use the extended PyCSP-net to test communication. The conclusions of the test runs are available in table 7.

Testing	Result
One2OneChannel	OK
One2AnyChannel	OK
One2AllChannel	OK
Any2OneChannel	OK
One2OneChannel guards	OK
Any2OneChannel guards	OK
poisonChannel	FAILED

Table 7: **Testing correctness of the extended PyCSP** The table shows the conclusions on the state of the test items based on the test runs in appendix C.2.

Network poisoning fails in the tests in table 7. The problems of channel poisoning are caused by the following 2 reasons:

1. If a process exits without having poisoned all the connected channels, the ones that are networked will have threads running. Since no Python application will exit before all threads have terminated this will cause the application to deadlock.

2. When a channel is poisoned all processes in a blocked read or write need to be released from any monitors, so that they can be shut down. In the implementation of the networked channels there is a race condition when notifying monitors, in the event of channel poisoning.

It is decided that these issues will not be fixed in this thesis. It is acceptable that poisoning channels does not work for networked channels and has the consequence that to terminate applications we will sometimes have to kill the Python process. This will not affect the result of any performance tests carried out in this thesis.

4.2.3 Performance Evaluation

PyCSP version 0.2.2 has been extended with network support, and to be able to detect where possible performance gains or bottlenecks are located the performance of the extended PyCSP is now evaluated and compared with the performance of the standard PyCSP version 0.2.2 and the PyCSP version 0.2.4 with network support. Both will tell us a little about what we can expect when using CSPBuilder with the extended version of PyCSP. The extended version is called PyCSP-net while the other standard versions are called PyCSP ver.

0.2.2 and ver. 0.2.4-net. For all tests we will run an implementation of the Commstime[19] test, which is commonly used for benchmarking CSP frameworks. The Commstime calculates the time spent on channel communication, and since the changes made in PyCSP have been to the channel implementation this will suffice to compare the different versions.

Another purpose of the test is to identify the overhead of replacing a regular channel with a networked channel. This is very relevant for CSPBuilder since applications will often be run with more networked channels than regular channels.

A note about the PyCSP ver. 0.2.4-net is that as we see in table 8, the performance is quite good and the implementation of the networked channel using Pyro is very clean. The downside is that it was difficult to get working when we needed to construct a CSP network with several networked channels and eventually we had to give up, because of this error "PyroError: not allowed to share proxy among multiple threads". Because of this, it was not possible to do network tests with more than 1 out of 4 possible channels changed from regular to networked.

Commstime Test	**Version**	**Avg. Time per. chan**	**Network Overhead**
Regular channels	PyCSP ver. 0.2.2	90.94μs	
Regular channels	PyCSP ver. 0.2.4-net	158.36μs	
Regular channels	PyCSP-net	91.43μs	
Network (localhost)	PyCSP ver. 0.2.4-net (Pyro)	328.00μs	169.64μs
Network (localhost)	PyCSP-net (TCP)	201.07μs	109.64μs
Network (2 hosts)	PyCSP ver. 0.2.4-net (Pyro)	263.12μs	104.76μs
Network (2 hosts)	PyCSP-net (TCP)	171.80μs	80.37μs

Table 8: **PyCSP Commstime** A comparison of the network overhead in extended PyCSP-net and the official PyCSP versions. The overhead is calculated by subtracting the timing for a run with regular channels from the run with regular channels and one networked channel. This provides the network overhead for one networked channel. The Commstime tests were executed on 1 or 2 hosts installed with a Pentium 4 2Ghz CPU. Output from tests are printed in appendix D.3

The different PyCSP versions all perform very similar as seen in table 8. The reason for the small performance difference in running on 2 hosts vs. 1 host is that the channel implementation requires some computation, and as such has the advantage of 2 CPUs vs. 1 CPU. We see that the overhead of using networked channels vs. regular channels is the same as the time spent on a regular channel. Having a communication time of twice the time of a regular channel is very acceptable, though the measured timings are only for a small amount of data and the results are likely to be worse for networked channels transferring larger data sizes.

To test the performance of the network overhead for different data sizes the Commstime application is changed, so that it transfers a specified amount of bytes across the networked channel. The results of the tests are plotted in figure 17. The low performance for PyCSP-net in the range from 600 bytes to 1450 bytes is caused by buffering in the TCP layer. The TCP layer does not guarantee that data is sent immediately, but buffers data to increase the size of each ethernet package sent. The typical MTU (Maximum Transmission Unit) of an ethernet package is 1500. The TCP layer will buffer data until it has enough for an ethernet package or until a timeout of a few milliseconds interrupts and sends the ethernet package. By using the tool *Wireshark*[4] it is possible to see the packages sent during the tests. Monitoring the ethernet traffic shows us that the TCP layer bundles the header data "`1000LLLLLL`" with the actual data and sends this in one ethernet package. This is done for the tests with data sizes from 600 bytes to 1450 bytes. The other tests, with good performance, sends the header data and the actual data in seperate packages. When the header data and the actual data is bundled it interferes with the synchronization shown in figure 11 and consequently delays the transmission as in figure 17.

To test this theses, a small hack to PyCSP is made, where 1400 bytes are added to the header data "`1000LLLLLL`". This make it impossible for the TCP layer to bundle the header data with the actual data. Tests are run for data sizes of 500, 1000 and 1200 and the results are plotted in figure 17. This shows us that we avoid the delay in transmission with this hack and proves that buffering in the TCP layer is to blame for the big rise in network overhead in figure 17.

It is not obvious why this delay only happens for data sizes above 600, but the TCP layer is definitely the reason. The TCP protocol provides a "TCP_NODELAY" option for sockets, which disables the nagle algorithm responsible for some of the buffering on the TCP layer. Using this option solves the problem as displayed in figure 17, but adds a small overhead for larger data sizes.

The PyCSP versions both successfully transferred 5 Gbyte during tests with a data size per channel communication of 1Mbyte. Every test did 5000 channel transmissions, which amounts to 5 Gbyte. This was done in 142 seconds for PyCSP ver. 0-2-4-net and in 168 seconds for PyCSP-net. The throughput of the data communication can then be calculated to $5000Mbyte/142s = 35.2Mbyte/s$. The 2 hosts are connected by a GBit Ethernet which has a theoretical throughput of $125Mbyte/s$. In practice throughputs higher than $50Mbyte/s$ are rarely achieved and we find $35.2Mbyte/s$ and $29.8Mbyte/s$ are very good results.

We expected that PyCSP ver. 0.2.4-net would perform much worse in the network tests, since they rely on Pyro for communication. When the official PyCSP at some point reaches a maturity where networked channels are supported as extensively as in the now extended PyCSP-net, it might be an advantage to switch to the official PyCSP for CSPBuilder, since the implementation will be easier to maintain.

[4] A network traffic analyzer for Unix operating systems. Formerly named Ethereal

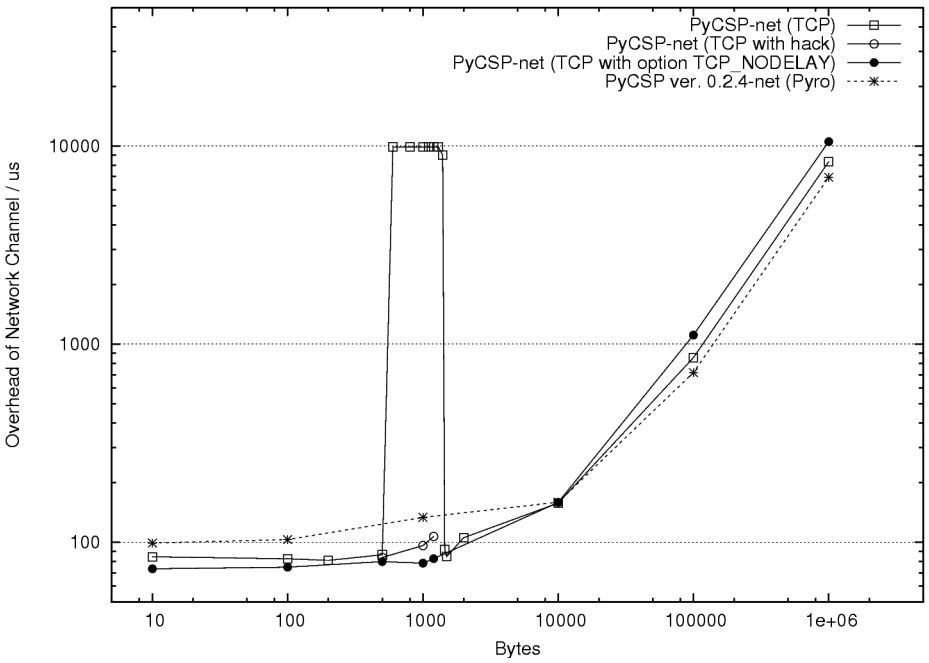

Figure 17: **`Network Performance`** A performance measurement on the network overhead for different data sizes. PyCSP-net is the implementation extended for this thesis. The big rise in overhead of the PyCSP-net implementation in the range 600 bytes to 1450 is explained in section 4.2.3. The tests are executed on 2 hosts communicating across an ethernet.

With the extended PyCSP-net we now have a CSP implementation where any CSP network can be distributed on any number of machines. In the worst case it is expected that a distributed CSP network will run in twice the time than it would on a single machine because the overhead of changing regular channels to network channels has shown to be twice the communication time of a regular channel. Unless data sizes communicated exceed 10Kbyte, then the network overhead will rise according to figure 17. The implementation of networked channels used in this thesis also has lower performance with data sizes from 600 to 1450 bytes. Solutions have been suggested but have not been added in the implementation used for the experiments in section 5. In the next section we will improve the average case by using a method to group processes.

4.3 Organizing Processes on Nodes

To achieve good performance from CSPBuilder we have to organize processes on nodes from a set of rules. The rules are defined in this section, and based on these rules a method is designed and implemented.

When executing a CSPBuilder application we have a single node which parses the `.csp` file. To keep the node servers as simple as possible they receive the data structure of the csp network as late as possible in the execution. This is the moment before the CSP network is initiated and threads are created, since these would not be suitable to send from the main execution node to the node servers. The node servers are dumb, in the sense that they just initiate and execute the CSP network that they are sent, without any knowledge of what other nodes the processes might communicate with. The node server terminates when all processes have been poisoned.

The entire process network is described in a data structure. This data structure is cut into parts, each part containing the processes and channels that are to run on one node. Each part of the data structure is then sent to its node, after which the node creates the process network. The application should run as if it ran on a single node. The procedure explained here will require the same installation and components available on all nodes, for processes which are not locked to a specific host or preferred architecture.

If a process network is distributed on a number of nodes, regardless of the amount of communication and workload, the entire application will possibly run slower than if it was executed on a single node.

If the application has a small number of processes that contain most of the computation and these processes are located near to each other e.g. located in the same components, then it is important to identify these components. Otherwise they will be distributed to the same node depending on how many nodes are available.

From figure 10 we saw that a simple method for grouping processes is to group on the topmost level of the tree data-structure, by dividing process instances of components into

n groups where n is the amount of nodes available. Since most of the process instances on the topmost level will likely consist of smaller process networks, the process networks are unfolded and divided if more nodes are available. In reality this is implemented as an upside-down method, where the grouping is done from the lowest level and up. Doing it upside down makes it easier to implement. The details is explained in section 4.3.1.

To group processes into an optimal setup it is necessary to know the relative amount of computation and communication in every process and across every channel. With this information processes can be positioned on different nodes if they are doing heavy computations. Communicating processes should be positioned close to each other and preferably on the same node, as long as none are doing heavy computations. A method to locate processes that contain heavy computations would be to run the application in a mode on a smaller problem, where a profiler is recording the communication between processes and the time every process is active on a CPU. The application is then run on the real problem and fed with the recorded data to be able to organize and group processes for better performance.

A potentially very effective method for grouping processes and run the application with the best performance would be to do the following: Create a graph mapping all processes with weights and connected with weighted lines. This graph is constructed from the data produced by the profiler. The goal is then to group processes in this graph into n groups where n is the number of nodes while providing the lowest possible sum of all weights connecting groups and an even sum of weights in each group. Finding this solution is very likely an NP-complete problem and is out of scope for this thesis.

We base our grouping on the simple method grouping processes on the topmost level in the tree data-structure. To improve performance we add a property to the CSPBuilder component. This provides the component developer the possibility to tag a component likely to do heavy computation. When this component is used in an application it is prioritized upon execution to position it on a node with no other “heavy” processes.

A property for setting a preferred architecture for a component, and a property for locking a component to one host is added. Locking a component makes it possible to have a component read input data only available on a specific host, and also having component displaying visual results on a specific host. The reason for being able to prefer one architecture is that some algorithms can be executed much faster on special hardware if the implementations of the algorithms allow it. With this the developer can tag these components with this preferred architecture, and the grouping method will try to organize the network in a manner that positions processes on their preferred architecture.

4.3.1 Implementation Details

The method for grouping processes implemented in this section is expected to perform better than the case where processes is divided equally onto nodes, one by one. To implement the

grouping method there are 3 steps to consider:

- How are the processes given IDs, so that it is easy to find processes close by?
- How do we walk through the IDs and organize them on nodes?
- How do we handle and prioritize the processes that are locked to a node, prefers a special architecture or has a heavy workload?

In the following we define a group as the process network which a process can be part of. This group is contained within a component. All processes are given group IDs from their respective location in the tree data-structure. When this method has been run all processes are associated with a list of group IDs, identifying which groups a process is part of. A process can be part of several groups, since components can contain process networks of other components. The identification of group IDs is implemented with a recursive function that parses the tree data-structure. The identification step is shown in a simplified form below:

```
i = 0
PARSE_PROCESS(p, g)
  global i
  i += 1
  parsing p
  if p contains other processes y
    for each y
      g2 = g + [i]
      PARSE_PROCESS(y, g2)

  create new process from p and associate group list g


To perform identification step.
PARSE_PROCESS(processNetwork, [])
```

After this step we can find the groups of all processes by selecting processes with the highest group ID, one by one. Processes are then picked out and positioned onto nodes until all nodes have e processes.

```
Set g = find highest group id
Set n = 1
Set e = number of processes / number of nodes

while more processes
  while processes in group g
    pick process in g and move to node n
    if e processes on node n,
      increase n
  decrease g
```

With the method shown above none of the tags ("locked to node", "preferred architecture", "heavy workload") have been taken into account. We deal with these in the prioritized list below. The entire procedure of organizing processes on nodes is carried out according to this list.

1. Lock processes to a node, because if processes are not run on the correct node the required libraries and data might not be available.
2. Position the processes with the heavy workload, to have as many nodes as possible available to spread these processes across. They are positioned one by one on nodes, making sure no node has more than 1 process more than any other node. The preferred architecture is used to position processes on nodes.
3. Fill up nodes with processes, until e processes are positioned. This is done in an extended version of the method listed above where the preferred architecture is matched as well.
4. Finally the rest of the processes, where the preferred architecture could not be found, are positioned on the rest of the nodes using the method listed above.

In the end channels are calculated based on the position of processes with the following method:

```
for every channel
  if all ends are located on same node, then move to node.
  else create Remotechannel and send to nodes.
```

This method should help us avoid the worst case of having all the heavy processes located on one node, while having the highest number of networked channels possible.

4.3.2 Correctness Test

In this section we describe and show the results of a few tests in CSPBuilder that check the grouping of processes onto nodes. To test where processes are located, a *Beacon* component is used which prints out a configured value together with the hostname of the host where the instance of the component is running. All tests are run with 4 nodes available to group on. Output from the tests are available in appendix C.3.

OK **Case 1 (Lock to node)** Application with 6 *Beacon* processes. 4 locked to one node and 2 locked to 2 different nodes.

OK **Case 2 (Heavy workload)** A process instance of a component containing 4 *Beacon* processes, with the heavy workload tag set. 3 *Beacon* processes are placed together with this process instance, which challenges the grouping of the heavy workload processes.

OK **Case 3 (Preferred architecture)** Application with a process instance of a component containing 2 *Beacon* processes preferring an "Itanium" architecture. One of the nodes is configured for this architecture. 3 other *Beacon* processes are put in the application with no preference for other architectures.

OK **Case 4 (General grouping)** Application with 8 *Beacon* processes. Application with 9 *Beacon* processes. Application with 10 *Beacon* processes. Application with 11 *Beacon* processes.

FAILED **Case 5 (Combined)** Creating an application that has a process instance of each of the process networks built in case 1-4. Checking the grouping that all processes are placed correctly.

The combined test (case 5) was unsuccessful. The failed test is not caused by a flaw in the implementation, but because of the method design. The test made a suboptimal placement when 4 "heavy workload" processes were positioned on 3 nodes and not across 4 nodes for maximum performance. The reason for the suboptimal placement is that the method sets the limit that the maximum processes positioned on a node is p/n, where p is number of processes and n number of nodes. The positioning of "heavy workload" processes are prioritized lower than the positioning of "locked to node" and "preferred architecture". 4 processes was locked to the node with "Itanium" architecture and 2 processes had "Itanium" as their preferred architecture. 6 processes was the limit for every node, which was the reason the test failed.

The conclusion for the tests conducted in this section shows that CSPBuilder is able to position processes according to the method implemented in section 4.3.1. This method is not the best solution, but it does provide an better solution than if no method was used. Besides the problems for the method just mentioned, another issue is that the amount of channel communication is not used as a parameter for the grouping method, which can have an unknown effect on performance. Results in section 4.2.3 showed that the overhead of using networked channels compared to regular channels is surprisingly low and as such we can expect the negative impact on performance to be relatively small.

4.4 Summary

The possible performance attainable by CSPBuilder has been improved by this section. PyCSP has been extended with support for networked channels. The performance of network channels has been proved high for the implementation developed for this thesis and for the new PyCSP ver. 0.2.4. The extended PyCSP is the only CSP implementation for Python that supports One2One, One2Any, One2All and Any2One channels completely. For the One2One and Any2One channel support for guards is also supported succesfully. These were the requirements for a CSP implementation for Python that would be able to execute a CSP network across several nodes.

The network performance is tested and has proved to be able to handle a high throughput of $30Mbyte/s$ and a latency below $200\mu s$. With this performance it should be possible to execute almost any CSP network in a distributed environment and get a performance increase from the concurrency available in the CSP network.

Support for channel poisoning does not work in this CSP implementation. It should be expected that when using the extended PyCSP implementation, it might be necessary to terminate processes by hand after the CSP network has fulfilled its purpose.

To gain better concurrency from a CSP network, a simple method for dividing a CSP network into groups has been developed. It improves on the average case, but not on the worst case.

With this section CSPBuilder is now a complete application and framework with the purpose of developing CSPBuilder applications that can be executed in a distributed environment. The performance of CSPBuilder applications is tested in the next section 5.

5 Experiments

In this section we test the performance of CSPBuilder. We have chosen 2 experiments that will test areas of CSPBuilder, and based on the results we will conclude on the capabilities of CSPBuilder. The experiments are *Prime Factorization* and *Successive Over-Relaxation*, where one parameter we will be changing is the 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, and by this help solve the larger problem.

The test environments in table 9 are used in the performance evaluation sections. On the 8 core SMP system, tests are run with the amount of workers ranging from 1 to 8. On the cluster we run 1 node on each of the 13 hosts and 3 nodes on the master host. The master host has 2 CPUs and these compute faster than the node hosts. This means that we get 16 nodes and can run tests with workers ranging from 1 to 16.

The experiments prove that CSPBuilder is capable of executing CSPBuilder applications on an 8 core SMP system and on a 16 node cluster. On the 8 core SMP system the GIL is released to be able to utilize all cores succesfully. On the cluster the CSP network is distributed to nodes for better performance.

5.1 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 any period wanted. In the book of Donald Knuth [16], 5 different algorithms for doing prime factorization are

explained. The simple one is the least effective and is based on doing *Trial Division*[5]. *Trial Division* is used in the algorithm *Direct Search Factorization*[6]. The simple prime factorization algorithm was chosen due to the following reasons:

- Parts of the algorithm will have to be written in both C and Python. The simplicity of the algorithm is an advantage here.
- It is desired to use a dynamic orchestration design which will enable us 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 what 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* is found at PLEAC[7] (Programming Language Examples Alike Cookbook). This implementation is extended and divided into a parallel version that is implemented in the CSPBuilder framework.

5.1.1 Implementation Details

The *Prime Factorization* 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 18.

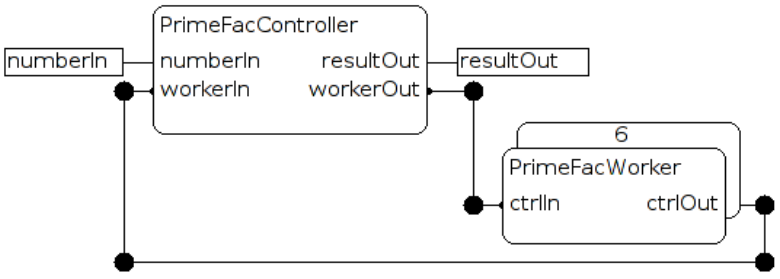

Figure 18: **`PrimeFac Component`** consisting of a controller and a worker multiplied 6 times.

Every worker will tell the controller when it is vacant, by sending a result. On initialize a worker will send an empty result to the controller. The controller runs a loop until all

[5]Trial Division: `http://mathworld.wolfram.com/TrialDivision.html`
[6]Direct Search Factorization: `http://mathworld.wolfram.com/DirectSearchFactorization.html`
[7]PLEAC: `http://pleac.sourceforge.net/pleac_python/numbers.html`

primes have been found. This loop receives a result from a worker and sends a new job. If a non-empty result is received, the controller 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 $[2 \dots d]$ is all the divisors tested. All the factorizations of n can be found in $[2 \dots \sqrt{n}]$.

The numbers that are interesting to factorize in primes are the numbers that are larger than the number representation available in general compilers. Normally to work with integers larger than 18446744073709551615, which is the limit for $64bit$ registers, special operations are needed. Numbers larger than this need functions for doing basic operations like addition, subtraction, multiplication and division. The number is no longer a number from the machine point of view, but instead represented by a block of data in main memory.

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 in section 2.2.4. To test the implementation, a version working only with numbers less than $64bits$ is created. All basic math operations are then replaced with function calls to the library LibTomMath[8], which handles large numbers. For transferring large numbers between Python and C a radix 10 string format is used.

Finally we add a release for the GIL as in section 3.1.6, which enables us to use the available concurrency in the application.

5.1.2 Performance Evaluation

For our experiments the Mersenne[9] number $2^{222} - 1$ is used. This number was picked by trial and error, with the purpose to find a number where the prime factorization 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 and releasing the GIL. In the C implementation we use a large number library *LibTomMath*. This large number implementation is slower than the large number implementation in Python, which

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

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

System	Hardware
8 core SMP	2 CPU: Intel Xeon QuadCore E5310 1.60GHz, cache size 4x4096Kb Memory: 8Gb
A Cluster with 13 working hosts and a master host	The master host: 2 CPU: Intel Xeon CPU 2.40GHz, cache size 512Kb Memory: 1Gb The 13 node hosts: 1 CPU: Intel Pentium 4 CPU 2.00GHz, cache size 512Kb Memory: 512Mb

Table 9: **Test Environments** These are the available test environments used for the experiments in this section.

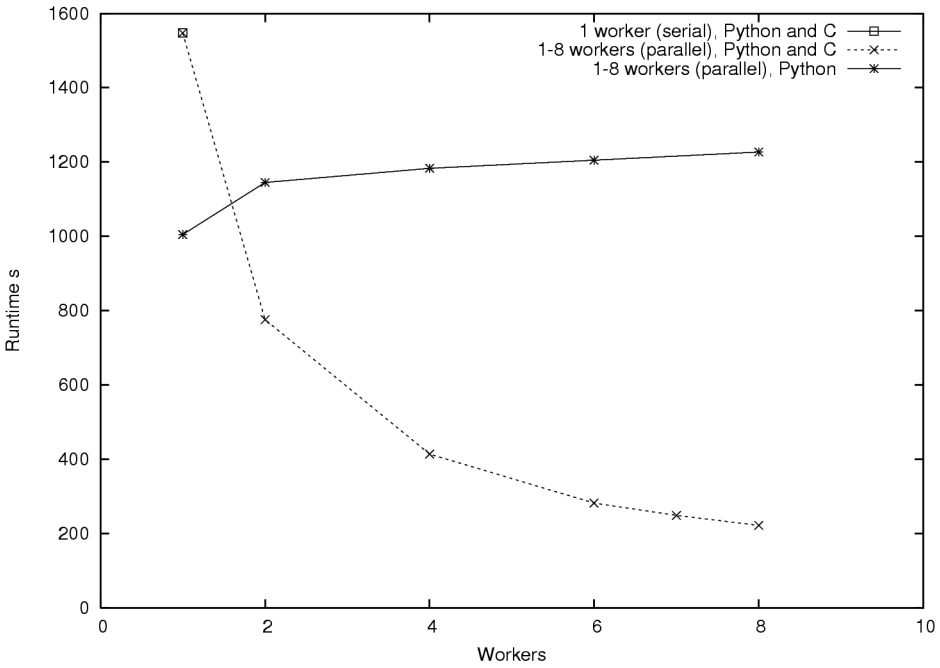

Figure 19: **`Prime Factorization`** Doing prime factorization of the Mersenne number $2^{222} - 1$. The combination of Python and C can utilize the threads while the "Python only" is limited to only one thread, because of the GIL. Also in the "Python only" example the overhead of context switches increases when the amount of workers increases. All benchmarks are run on the 8 core SMP system. Output from the test is printed in appendix D.5.

is also shown in the tests where the "Python only" outperforms the "Python and C" for the case with only one worker. To compare the effects of adding more workers we examine tests with 1, 2, 4, 6 and 8 workers and plot them in figure 19. The performance of the "Python and C" version does very well, and looking at the speedup in figure 20 the performance scales linear, which means that adding the double amount of workers on a system with double capacity increases the performance to the double, reducing the run-time to half. The speedup is not entirely a straight line in figure 20. This is caused by having to flush the workers every time a result is found. Time is then spent sending new jobs to workers. This negative effect increases with the amount of workers.

The increase in run-time, when adding workers in the "Python only" version in figure 19, 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.

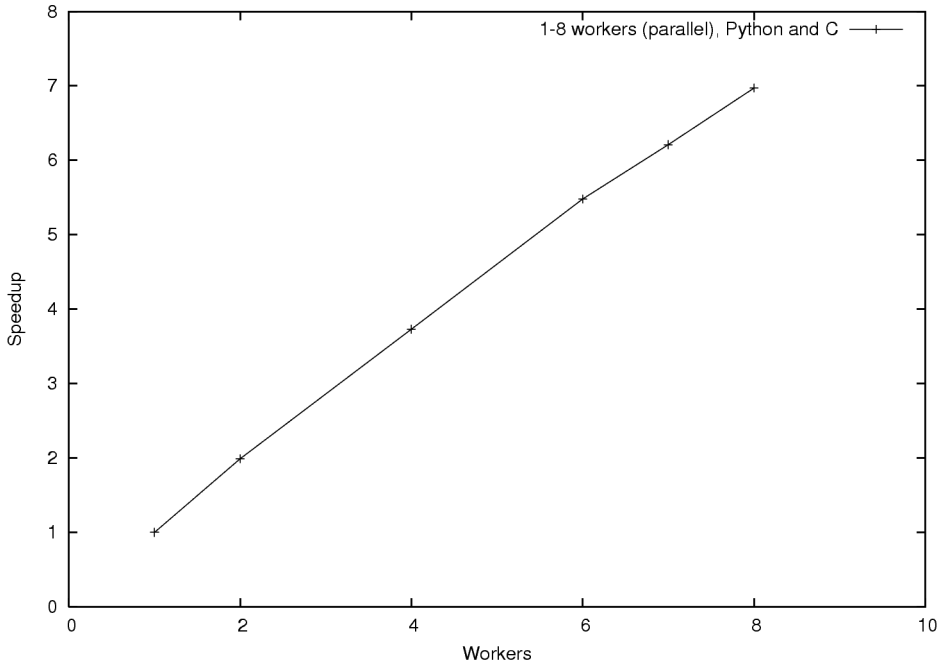

Figure 20: **`Speedup of Prime Factorization`** Doing prime factorization of the Mersenne number $2^{222} - 1$. Combining Python and C can improve the utilization of threads tremendously and linear speedup is almost reached for this benchmark, which is very acceptable. We did not expect better results than this. All benchmarks are run on the 8 core SMP system. Output from the test is printed in appendix D.5.

The plotted speedup is based on the simulated serial execution. The serial benchmark has a job size of 10^{16}, which causes only 1 job to be sent to the 1 worker waiting. This benchmark equals the reference of an serial execution with CSPBuilder and as such is our reference when calculating the speedup of the parallel benchmark in figure 20.

The next tests are carried out to see the effect of not releasing the GIL. We expect to see three plots almost identical in figure 21. For comparison the test result from figure 19 with the GIL released is plotted. There are two things to notice in figure 21. One is the plots being so irregular, the other is that they do not line up close to each other. The reason for the two graphs being so irregular is the grouping of processes. Whenever the grouping is not good, a huge rise in run-time can be seen. The two plots for the version not releasing the GIL perform very differently because of the nodes available. One is run with 8 nodes available, while the other has the same amount of nodes as workers available. With the same amount of nodes as workers, the controller must be located on a node together with a worker. This would starve the controller process though this is not the case in the situation with 6 workers. It is to difficult to say anything about the effect of the grouping, because of the low performance even though all 8 nodes are available and processes are grouped perfectly on nodes.

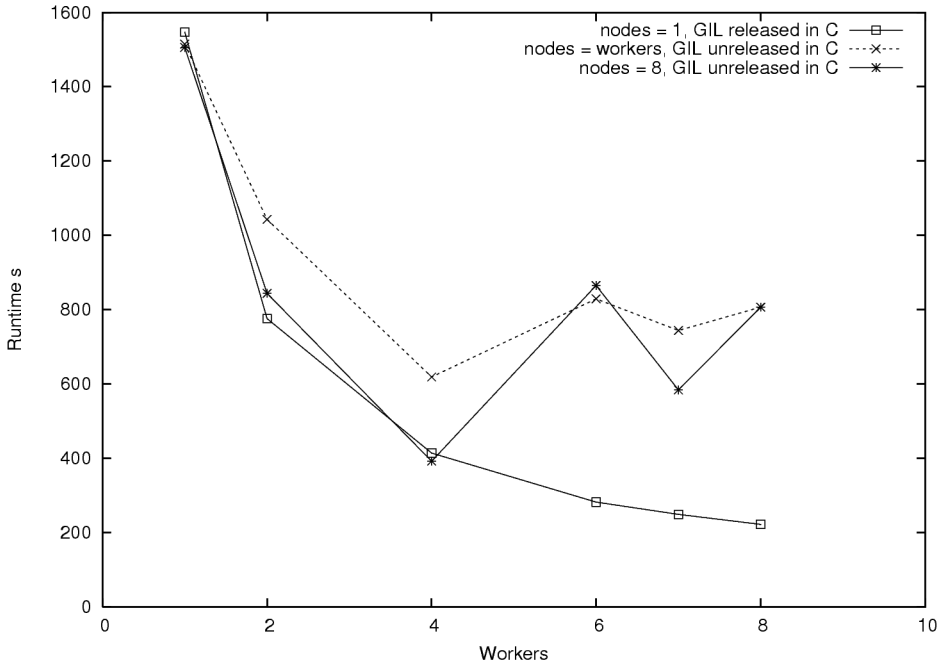

Figure 21: **`Testing with GIL`** When testing performance without releasing the GIL in C, we expect to be able to gain the same performance by running the CSP processes on nodes, thus avoiding the limits set by the GIL. Looking at the results, running the processes on nodes does not achieve the performance that was expected. All benchmarks are run on the 8 core SMP system. Output from the test is printed in appendix D.6.

When viewing the execution with the tool `top`[10] on the host, we are able to see that only 1 or 2 nodes are utilized by 100% while others are idle. Our theses is that the NetChannels are somehow starved by nodes running 100%, since the GIL is unreleased. Examining the plots in figure 21 tells us that 4 workers is the limit, before they start starving each other thus

[10] A UNIX application that shows the current processes or threads running on a UNIX system

having an overall negative effect on performance.

To compare the performance of networked channels vs. regular channels, we run an experiment where the size of every job is changed and by this the amount of jobs and communication changes. We use both channels in the possible situation by running the tests for regular channels on the 8 core SMP and the tests for network channels on the cluster. The results are plotted in figure 22. When the number of jobs comes below about 500 the run-time will rise until it reaches the run-time of running with one worker. The less jobs, the less concurrency in the application, since the flushing of jobs becomes excessively expensive and there is also a higher probability that the result will be available in the first job tried, again making workers superfluous.

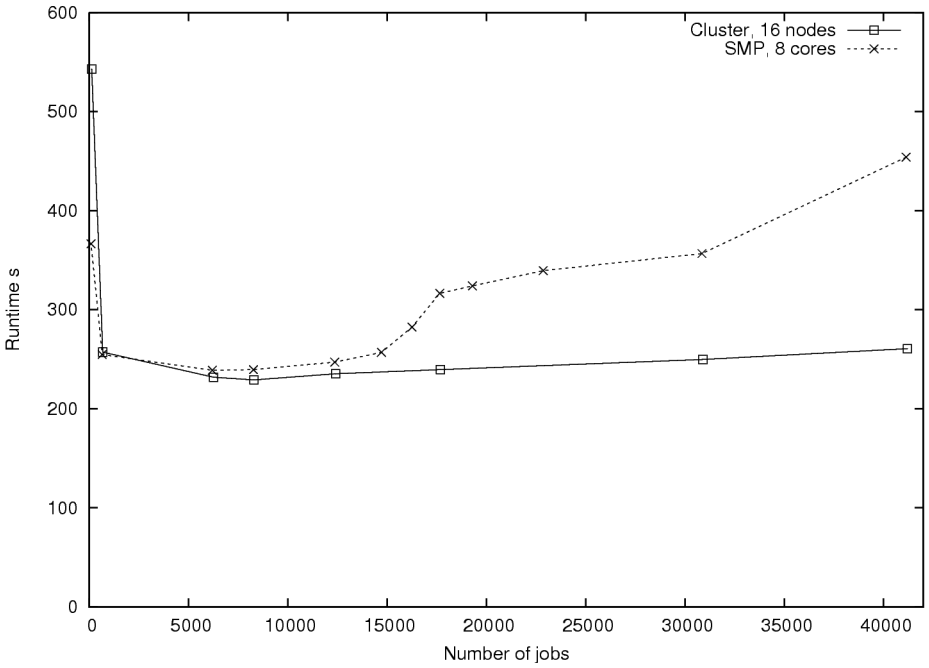

Figure 22: **`Changing the job size`** The number of jobs positioned along the X-scale is calculated based on the job size, which ranges from 15000 to 10^7. An interesting effect in this figure is that the performance of the cluster does not degrade much when the number of jobs increases. The two plotted values on the left are run with a jobcount of around 100. Output from the test is printed in appendix D.7.

Based on the results in figure 22, we can conclude that using networked channels on a cluster vs. regular channels on an SMP machine has much lower latencies during communications. When increasing the number of jobs, the frequency of communication increases as well and the latency of channel communication becomes apparent in the run-time. Since both runs use the same amount of workers as the available CPUs, it is certain that the rise in run-time when increasing the number of jobs must arise come from channel communication overhead.

This section has focused on testing the performance of running a CSPBuilder application on a SMP system. The tests has showed the importance of releasing the GIL to be able to gain acceptable performance results. In the final tests plotted in figure 22 a comparison is made between running on a cluster and running on an SMP system. The results of this test shows that running on a cluster using nodes with CSPBuilder has a huge potential, because it scales remarkably well when increasing the amount of communication.

5.2 Successive Over-Relaxation

This experiment will have an added amount of communication compared to the experiment described in section 5.1. It will not release the GIL and it will be written entirely in Python. This will test the performance when running on a cluster while changing the amount of concurrency in the application.

A *Successive Over-Relaxation*[22] component is built that can process a matrix of 2 dimensions. *Successive Over-Relaxation* is a numerical method. It has several applications in calculating weather predictions and for solving linear systems of equations. Here it is used to calculate the temperatures around hot objects and find the ending state of the space surrounding the objects. The purpose of the test is to test CSPBuilder and may not calculate temperatures correctly. *Successive Over-Relaxation* has been chosen for the following reasons:

- It is simple to implement and test. It is a common computation problem when doing scientific computing. This has shown that CSPBuilder can be used as a framework when implementing a *Successive Over-Relaxation*.
- For every iteration it has a step where all workers are synchronized. This can possibly have a negative effect on performance, since with CSP all communication is blocking, which means that communication cannot be hidden. Tests will show whether the CSP network constructed by CSPBuilder will be able to communicate effectively enough to provide an acceptable speedup.
- The size of data communicated is bigger than in the *Prime Factorization* experiment. Communicating across a physical Ethernet has a negative effect on performance because of the possibly higher latencies. This will help to identify the negative effect.
- The orchestration is static since every worker needs to know about the other workers. The *Prime Factorization* experiment used a dynamic orchestration enabling it to use the *multiplier* functionality of CSPBuilder. This is not possible in *Successive Over-Relaxation* and shows that implementation is still possible using CSPBuilder as the framework.

Doing successive over-relaxation involves calculating an average for every point in a matrix of 2 dimensions. The average of all 4 neighbours including the point itself is calculated and the value of the point is replaced with the new average:

$$\forall x \in \{1, 2, \ldots, width - 1\}, \forall y \in \{1, 2, \ldots, height - 1\}$$
$$(x, y) = 0.2 * ((x, y) + (x - 1, y) + (x + 1, y) + (x, y - 1) + (x, y + 1))$$

This is done for every point in the matrix until $\delta < \epsilon$ where δ is the sum of all changes. At this time the matrix is considered stable, if a small enough ϵ has been chosen.

5.2.1 Dividing Into a Parallel Problem

In its original form any point is dependent on any other point, which makes it very hard to do in parallel. However *Successive Over-Relaxation* has the characteristic that you can use points from an older iteration than the previous and you will still get a correct result. Not the exact same result, but still correct. Doing this has the disadvantage that it takes longer to converge to a stable state, but being able to parallellize the problem quickly outweighs this disadvantage.

According to Chang, Yan and Le[23], a common and effective method is to divide points into red and black points swapping between calculating one set and the other set. This method is illustrated in figure 23 for sharing processing between two processes. The matrix containing the original state is divided into equally sized row-blocks, where the amount of row-blocks corresponds to the amount of parallel processes. Every parallel process will then have to communicate its neighbour points in every iteration. Using data calculated in a previous iteration increases the number of iterations necessary to reach a stable state.

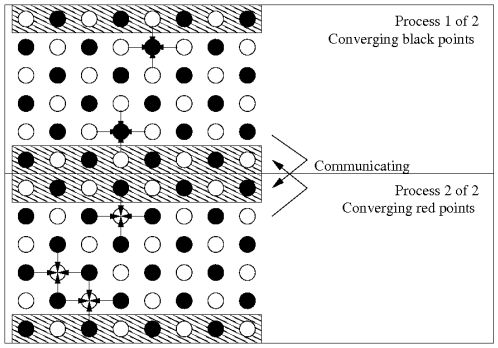

Figure 23: **`Parallel Successive Over-Relaxation`** is here shared between two processes. In this iteration process 1 computes black points, by using white points from the previous iteration. This eliminates the data-dependency between process 1 and 2. In every iteration border points are communicated to neighbour processes.

The decision of dividing into row-blocks is based on this being the easiest to handle and the easiest method to identify the neighbour points to be communicated. Alternatives for row-blocks is to divide into columns or into squares. Whether any performance can

be gained by dividing in rows or columns depends on what dimensions the matrix has. Changing the dimensions changes the size of communication between processes. In this thesis we are using *Successive Over-Relaxation* to test the performance of CSPBuilder, and will not test the performance of dividing the matrix in other blocks than rows.

5.2.2 Implementation Details

When constructing the application it would seem obvious to use the *multiplier* functionality of CSPBuilder, but this is limited to dynamic orchestration designs as described in section 3.1.1. If we wanted to do *Successive Over-Relaxation* in a producer-worker setup, we would have to send the entire state of every block through a set of processes that would synchronize and copy border communication. This would require a huge overhead on communication compared to only communicating the borderlines as in the suggested design.

A *ParallelSOR* component is created that has 3 inputs and 3 outputs. 2 inputs for receiving the top and bottom row, 2 outputs for sending the top and bottom row and finally 1 input for receiving the problem and 1 output for sending the result. As you see in figure 23, the top border is not communicated by the first process and the bottom border is also not communicated by the process handling the bottom block. CSPBuilder can only handle a static number of inputs and outputs, so we connect dummy processes to these to avoid any deadlocks. We could create a specialized version of the component to be a top-component and one to be a bottom-component, but it was decided than one general component was preferred. The design of a 4 process parallel setup can be seen in figure 25. This design is saved as a component and used in the application in figure 24.

Figure 24: **`4-Process-Parallel-SOR Application`** consisting of 3 process instances. LoadImageMatrix loads an image file from disk and outputs it as a gray-tone matrix. 4-ParallelSOR is the component shown in figure 25 and could for example be replaced with a 16-ParallelSOR if wanted. The final process displays the image in an window using a library: Matplotlib (pylab).

All processes are iterating until $\delta < \epsilon$. This δ is based on all points, but we have in this thesis decided to base it on the local points in a block. This means that some processes will detect that the system has reached a stable state before this is true for the entire problem. To solve this problem an extra component and an extra communication would be needed to calculate a global δ, which then would decide whether the processes should stop iterating. For simplicity, we have chosen not to implement this extra process and communication step. Instead all processes continue iterating until all have reached an iteration where $\delta < \epsilon$. This has the effect that a test with one worker will calculate a δ based on all points, and as we add workers the tests will run for a few more iterations until all workers have $\delta < \epsilon$.

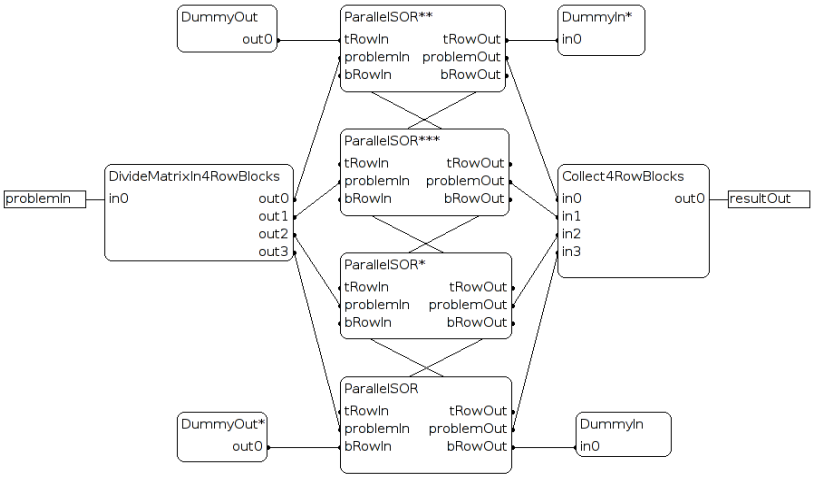

Figure 25: **`4-ParallelSOR Component`** is constructed of 4 ParallelSOR process with a design that makes them work effectively in a setup like this. To feed the processes we use a component DivideMatrixIn4RowBlocks, which divides the matrix and adds the extra border on top and bottom as is shown in figure 23. These extra lines are used to store the top / bottom line of the neighbour processes. The Collect4RowBlocks component receives the results, strips the blocks of the extra top and bottom and outputs the result as a single matrix.

With this implementation we want to test the communication overhead when doing communication in every iteration, and this can still be tested without having a global δ. Also to simplify the implementation, it can only work correctly with a matrix where the width and height are a multiple of 2. This is easily fixed, but has not been a priority since the tests can be run correctly with this limitation.

5.2.3 Performance Evaluation

In this section we will test the performance of 4 different problems, plot the result of each test and compare these plots to show the effects when varying the problem. Every test is run with $1, 2, 4, 8 and 16$ workers in order to be able to plot results that will show us how the performance changes when the computation is divided among more workers. Changing the amount of workers does also change the amount of synchronization and communication necessary to compute the result. All *Successive Over-Relaxation* tests are run on the cluster, using from 1 to 16 nodes. In all tests the workers are spread across the nodes and 2 workers will never be located on the same node.

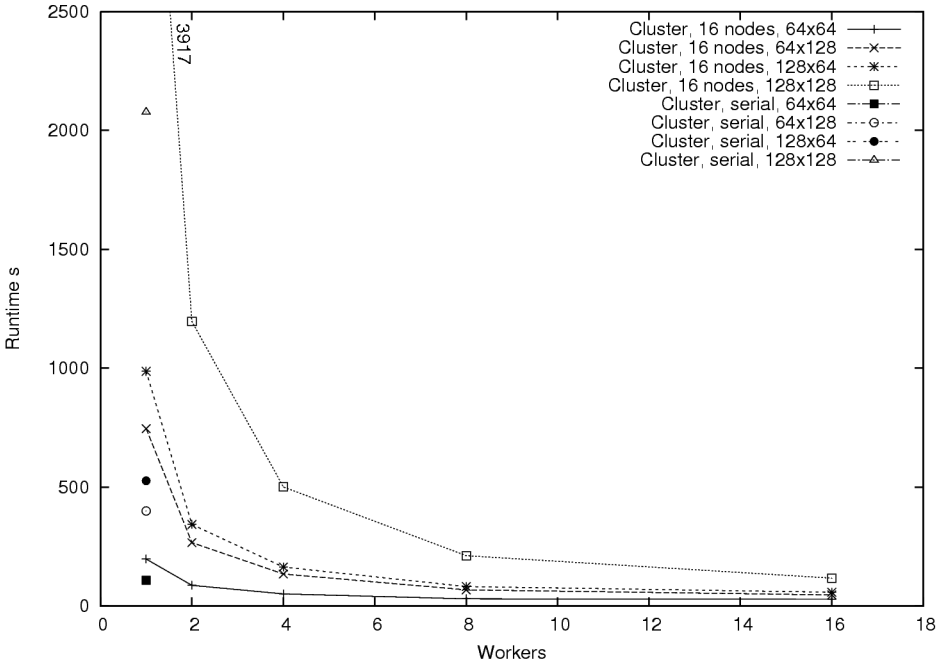

Figure 26: **`Successive Over-Relaxation`** The results from the distributed tests of the CSPBuilder SOR application. To generate realistic speedup data, tests have also been carried out with a serial version of SOR written in Python without the use of CSPBuilder and PyCSP. The results of the serial runs are plotted in the graph at the 1 worker position. A sample of the input data, which has been used in the tests, is displayed in figure 27. Output from the test is printed in appendix D.8.

The results from distributed tests on four input matrices is shown in figure 26. The pur-

pose of this test is to investigate how the SOR application scales in a distributed environment and whether the scaling differs depending on the size of the input matrix. The four input matrices vary in size from $64 * 64$ to $128 * 128$. Changing the width of the matrix changes the size of data that is communicated to neighbours in every iteration. Changing the size $x * y$ changes the size of the local matrix and the computation in every worker.

The input and output matrix of the $64 * 64$ is displayed in figure 27. When running the tests all points larger than zero or points at borders are locked to their value and will not converge. They will instead affect all other points in the matrix. This input matrix was chosen because there are no big areas which require more iterations to converge to a stable state. When testing on other sizes the input matrix in figure 27 is multiplied to cover the area of the new size. In the case with a input matrix of size $128 * 128$ the matrix in figure 27 would be multiplied four times covering the entire input matrix. The alternative could be to scale the size, but this would create larger "empty areas" which take longer to converge to the stable state. Scaling would also produce a problem that would be very different from the one we are solving and possibly have other characteristics that would affect the results.

In figure 26 we have 8 plotted results for the case with 1 worker and 4 plotted results for all other cases. We are comparing the results of the parallel SOR application to the results of a serial SOR application, since in the case of 1 worker, this would be the one to execute. The major difference in run-time between the parallel SOR on 1 worker and the serial SOR is the iterations. The serial SOR uses exactly half the iterations than the parallel SOR, because it can converge the entire matrix without having to swap between red and black points. Avoiding any overhead produced by CSPBuilder and PyCSP also saves a bit of run-time, though this is a constant factor and not influenced by the size of the input matrix.

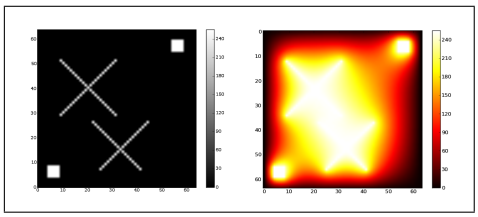

Figure 27: **`Input and Output`** The left image shows a sample of the input data used in the performance tests. The SOR application calculates all points with the value of zero, which results in the image on the right. This has required about 2700 iterations to converge to a stable state.

Next we are going to calculate the speedup of the 4 tests carried out in figure 26. This is done based on the serial SOR application, since the speedup is measured relevant to the speed of running on 1 worker and using an implementation without the overhead of the parallel implementation. The speedup of the SOR in figure 28 performs best with larger

problem sizes and performs poorly with a small problem size.

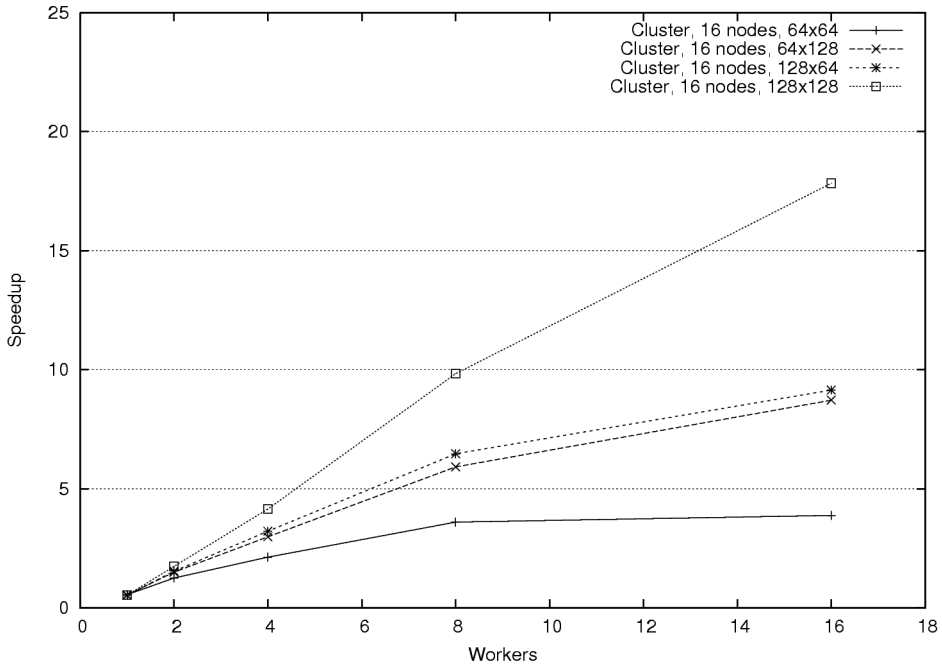

Figure 28: **`Speedup of Successive Over-Relaxation`** This graph is based on the results from figure 26. The speedup is calculated relative to the serial SOR. The results of the parallel SOR are diverting in iterations, when the number of workers is increased. A speedup corrected for this diversion is available in figure 29, which shows a better speedup for all tests.

As mentioned earlier the tests have a small increase in iterations when the number of workers is increased. With more iterations, more work is being done, which is identical to the situation where a higher ϵ is used with fewer workers. If we wanted to investigate the performance of our *Successive Over-Relaxation* implementation we would use the speedup in figure 28. But the SOR implementation is merely a tool to test the performance of running an application in CSPBuilder with the characteristics of a SOR. With this purpose it is more important to compare modified test results that resemble the work needed to do an exact number of iterations, than comparing the actual test results.

We correct the tests to resemble an equal amount of iterations and compare their speedup in figure 29.

The effect we see where we can reach speedups larger than the number of workers is caused by avoiding cache-misses in the CPU. The cache-misses occurs when running a large matrix on a small number of workers, because then the worker cannot have the same amount of values in the CPU cache as in the case where more workers are used. The problem sizes we are testing on are relatively small ($64 * 64$ to $128 * 128$), but because we are using Python and Python objects to handle the local data, the problem with cache-misses occurs even with

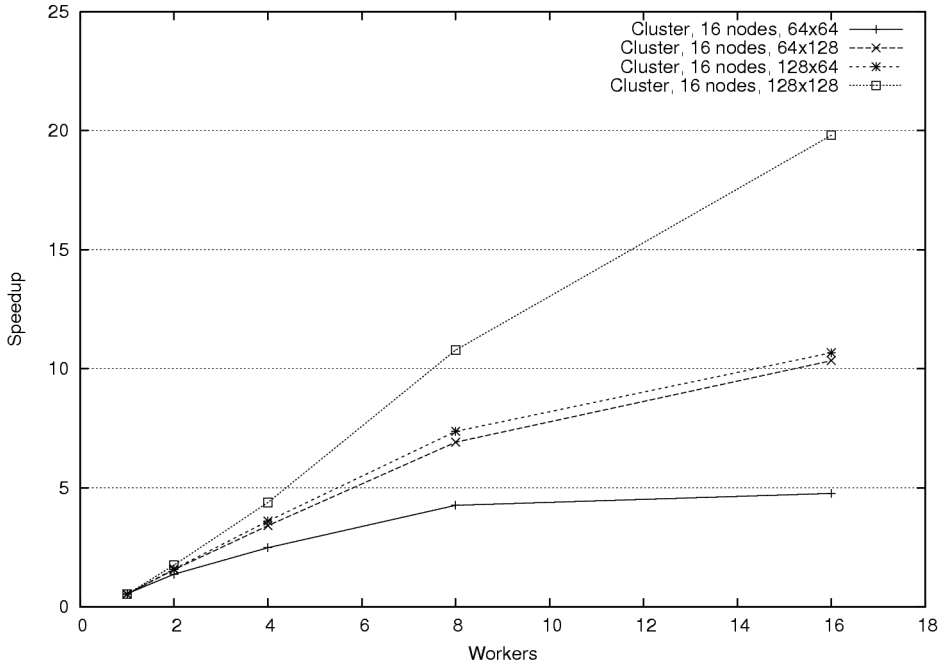

Figure 29: **`Corrected Speedup of Successive Over-Relaxation`** This graph is the corrected speedup of the results in figure 26, where the results have been corrected for the diverting iterations when the number of workers is increased. This results in a better speedup for all tests.

these problem sizes.

Based on the speedup results we can conclude that the networked PyCSP channels provide a good solution for communicating large amounts of data. Otherwise it would not be possible to get a speedup of almost 20 for the case with 16 workers and a $128 * 128$ matrix.

In the bottom of the speedup results we have the test with the $64*64$ matrix. This problem size is too small to provide a good speedup because the overhead of the communication becomes too big, and when running on a few nodes it does not have significant problems with cache-misses like the $128 * 128$ matrix.

The overhead of communication does not depend on the overall problem size, but on the width of the problem size, due to the implementation dividing the matrix in rows. To investigate how the time spent on an iteration is affected by the amount of communication, we construct 5 problem sizes which all have different width, but same overall size. The matrix sizes are $16 * 256$, $32 * 128$, $64 * 64$, $128 * 32$ and $256 * 16$, all with 4096 points. The input matrix is configured to have 3 borders with the values of 255 and 1 border with values of 0. This is the case for all tests. The tests will differ in the amount of iterations used, since a narrow problem size will only need a few iterations to converge to a stable state, while producing the output matrix in figure 30 will need the most iterations.

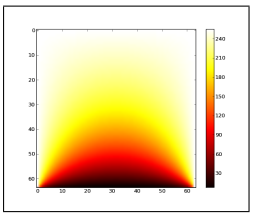

Figure 30: **`Output From Communication Test`** This is the result of a test with the SOR application. The input matrix has the left, top and right set to 255 and the bottom set to 0. It has required 7300 iterations to converge to this state.

To be able to compare these tests we compare the time spent in a single iteration. In every iteration there are two steps: Communication and Computation. The more communication the less time for computation and in effect longer run-times.

The 5 tests are run on the cluster for the parallel SOR application with 4 workers and for the serial SOR application for comparison. The tests with parallel SOR will show us the tendency of an iteration when changing the amount of communication while keeping the amount of computation constant. The serial SOR should show us that the amount of computation is constant.

Looking at the result in figure 31 we see that the serial SOR tests are close to constant. The small added cost for the $32 * 128$, $64 * 64$ and $128 * 32$ matrices is caused by all border values that are locked, and the amount of border values differs when changing the dimensions. This effect explains why the results for the parallel SOR curves a bit. From the results we

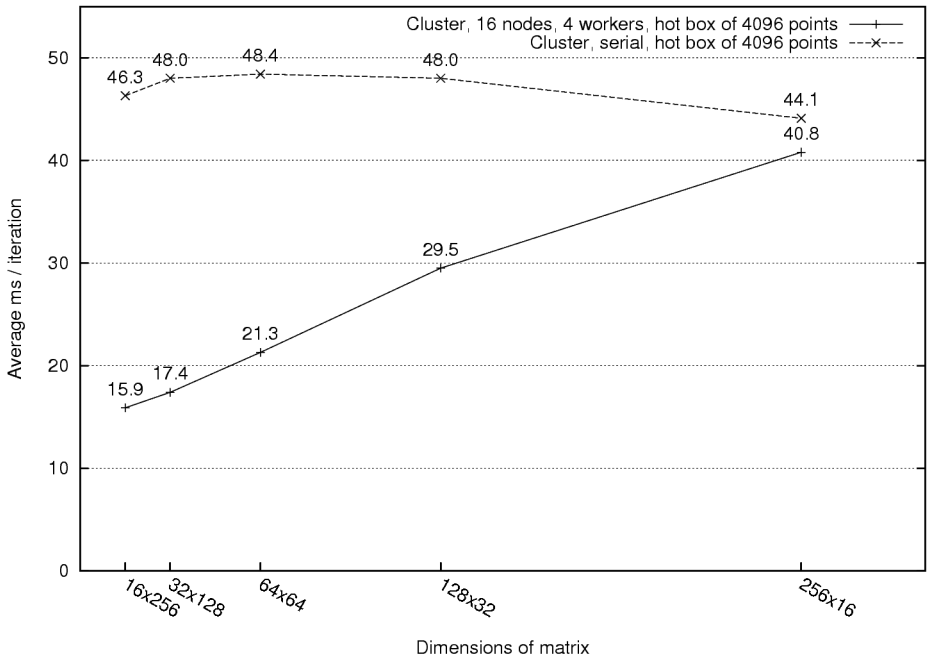

Figure 31: **`Communication Test`**. Shows the time spent on doing communication and computation in an iteration when changing the amount of communication by varying the width of the input matrix. Output from the test is printed in appendix D.9.

can see the overhead of communication and the drop in CPU utilization, due to the added communication. Most importantly this shows us that the overhead of communication is proportional to the size of the data communicated across the networked channels. We expect that the graph will level out in the bottom, where the width becomes below 16, because of the synchronization necessary to do communication. We can conclude that in this case the networked channels scale very well.

5.3 The Limits of CSPBuilder

In the experiments with *Successive Over-Relaxation* (section 5.2) and *Prime Factorization* (section 5.1) the possibilities and limits of CSPBuilder combined with the extended PyCSP have been challenged.

In this section we will focus on the limits of CSPBuilder. Finding the limit to how many processes we can put in an application and how much time it takes to initialize CSP networks of different sizes. Most importantly it will be tested whether CSPBuilder parsing CSP files and initializing CSP networks scales proportionally with the increasing amount of processes.

The limits of grouping processes on nodes is not tested, since this has already been tested in other sections (4.3.2, 5.1 and 5.2) and it is concluded that it performs reasonably in most situations, but does not guarantee a perfect grouping.

During tests with running on nodes we have experienced problems when working with more than 500 networked channels. The problems were caused by limitations in the amount of TCP sockets on the hosts. For this reason we choose to run the tests of CSPBuilder on a single host using regular channels.

We use the *Prime Factorization* application and configure it with different multipliers to test the performance when working with large applications. The amount of processes in every test is based on the multiplier m and the constant multiplier 512. It can be calculated with these formulas:

$$\begin{aligned} CSPBuilderProcesses &= (((PrimeFacController + 1) + (PrimeFacWorker + 1) * m) + 1) \\ &\quad * \ 512 + 5 * (process + 1) + 1 \\ CSPProcesses &= (PrimeFacController + PrimeFacWorker * m) * 512 \\ &\quad + \ 5 * process \end{aligned}$$

In table 10, the process counts have been calculated to show where the number of processes in the test results originates from. There is a difference in numbers between CSPBuilder and CSP, because CSPBuilder has all processes encapsulated in an extra process structure to support that any single process can contain an entire process network, consisting of processes connected by channels.

m	**CSPBuilder Processes**	**CSP Processes**
1	2571	1029
2	3595	1541
4	5643	2565
8	9739	4613
16	17931	8709
24	26123	12805
32	34315	16901

Table 10: **Number of Processes for Testing** Inserted m in the formulas and calculated the test cases. When plotting the test results, they are arranged in ranges of these process counts.

When running "`execute.py -i file.csp`", the file.csp is loaded and the process network is parsed. Including parsing this file, the components included in the data-structure are parsed until a data-structure describing the entire process network is created. The timings for this procedure are plotted in figure 32 for different application sizes. The test shows that CSPBuilder can handle parsing very large applications and scales proportional to the amount of processes in the parsed application.

The plotted timings in figure 33 increase at an exponential rate. This is the step where the CSP process network is initialized in the extended PyCSP version. The run-time required to

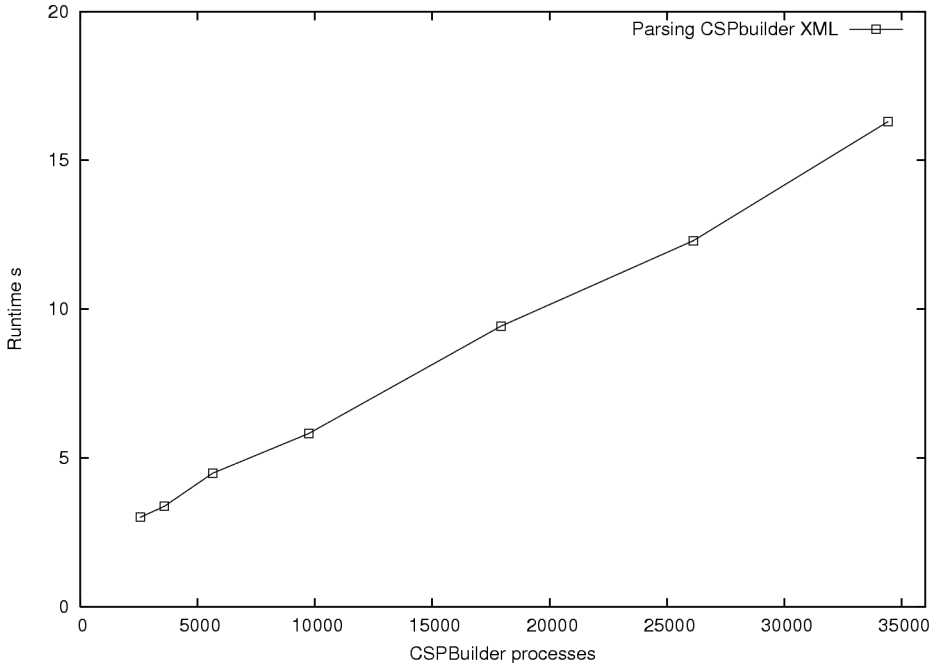

Figure 32: **`Parsing CSPBuilder XML`** A performance measurement on how much time CSPBuilder spends parsing XML and calculating the necessary structures for an execution. The measurements are run for different process network sizes. All benchmarks are run on the 8 core SMP system. Output from the test is printed in appendix D.10.

start the CSP network is many times larger than the times required for parsing the CSP data. Since the initialization of the CSP network increases at an exponential rate, while the time spent parsing CSP data increases at an linear rate. This makes the time spent parsing CSP data insignificant compared to the time needed to initialize the CSP network.

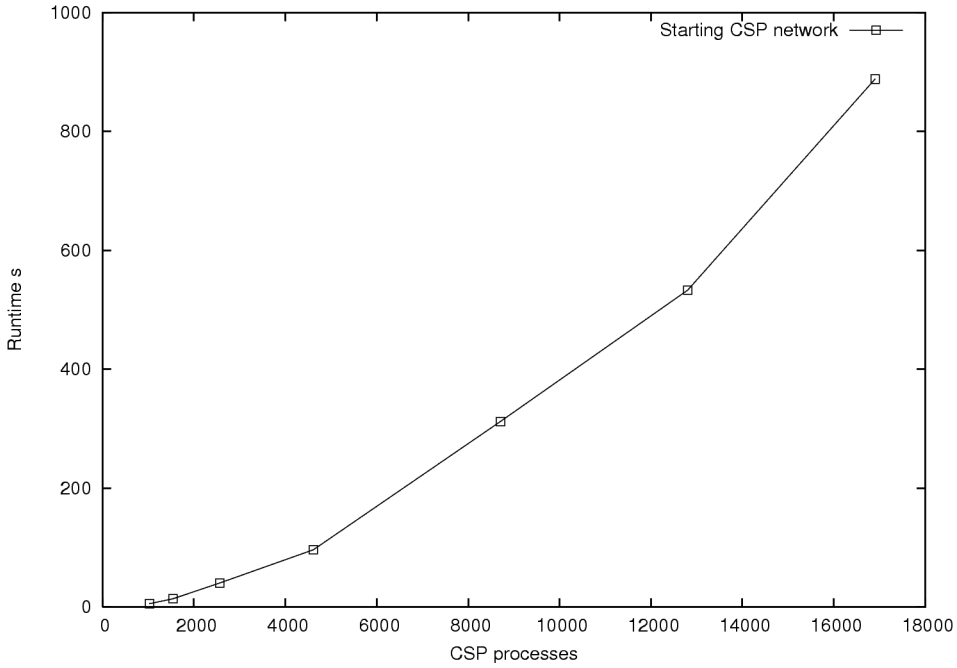

Figure 33: **`Initializing CSP network`** A performance measurement on how much time it takes to initialize large CSP networks. The initialization includes creating processes (threads) and regular channels. The measurements are run for different process network sizes. All benchmarks are run on the 8 core SMP system. Output from the test is printed in appendix D.10.

The exponential increase in figure 33 might be the operating system or Python causing an increased computation for every thread or class created, since every CSP process is created as a thread. Based on the test results, we conclude that CSPBuilder is capable of handling most applications with a reasonable time spent on initialization. If an application consists of a very large number of processes, it is recommended that the application runs for a time period long enough to make the needed time spent initializing the process network insignificant. Running an application with this many threads has a lot of other consequences, such as the time spent context-switching which would be high with 10000 threads.

5.4 Summary

The CSPBuilder framework have now been tested extensively through the experiments carried out in this section. One of the tests did not perform as expected. This test was carried out where nodes was used on an SMP system, to increase the concurrency in an applica-

tion that did not release the GIL. The test resulted in a performance that was unpredictable. When using 4 workers the performance was high, while when using 6 and 8 workers the performance dropped below the performance of using only 2 workers. No explanation for the unpredictable performance in this specific case was found.

The rest of the experiments in this section have all been carried out successfully. In a test the performance of the cluster was compared to the performance of the SMP system and especially the overhead of channel communication when increasing the amount of communication was interesting. The performance of running on the cluster compared to running on the SMP system was very impressive when the amount of communication was increased. We also have found that applications doing heavy computations in concurrent processes are able to utilize nodes or CPU-cores and scale performance linearly, provided that enough concurrency is available in the application.

All features developed for the CSPBuilder framework have been in use in the tests run in the experiments. All features but the poisoning of networked channels have worked correctly.

The performance of CSPBuilder is dependent on the structure of applications. Based on the experiments we find a number of recommendations when building applications for SMP systems and clusters:

> *When running on an SMP system, components with a heavy workload should be implemented outside of Python to be able to release the GIL.*
>
> *Use nodes to be able to achieve concurrency without releasing the GIL.*
>
> *Divide computations into as many processes as possible to increase the available concurrency in the application.*
>
> *Do not read from a channel before the read data is needed. Waiting to read until data is needed will increase the amount of concurrency in an application.*

With the experiments in this section the basic possibilities with CSPBuilder have been tested. From the results we can conclude that more advanced CSPBuilder applications with many more processes and channels will be able to run on large clusters utilizing nodes.

6 Conclusions

The CSPBuilder framework has been designed, implemented, tested and benchmarked. The framework consists of a visual tool to build applications and a tool to execute the constructed applications. The framework is implemented in Python and enables users and developers to use C / C++ and Fortran code easily by providing a wizard to enable these programming languages.

The performance of CSPBuilder is tested extensively in experiments. These experiments provide information on how CSPBuilder will perform when working on more complex applications. The overall results are that CSPBuilder can scale performance linearly with the amount of nodes, if enough concurrency is available in an application. Enough concurrency, is to have at least the same amount of non-dependent processes as nodes available.

If an application is run on a SMP system it is identified how to avoid the effects of the *Global Interpreter Lock* and how to multiply processes for more concurrency. With this information a *Prime Factorization* experiment is run and is able to utilizing all cpu-cores available in the SMP system.

To provide network support for CSPBuilder a CSP library for Python (PyCSP) is extended with complete support for networked channels. The network performance is tested and has proved to be able to handle a high throughput of $30Mbyte/s$ and a latency below $200\mu s$. With these results we can expect a very small overhead from channel communication when the size of the CSP network is increased.

The CSPBuilder framework currently has 2 major issues. One is that the networked channels cannot be poisoned correctly, because networked channels have a thread for every channel end. When the process is terminated by a poison, the networked channel ends still exist. The other is a problem with the amount of TCP sockets used by the framework. When using networked channels every channel will use 2 TCP connections, if many networked channels are connected to processes on the same nodes, these nodes will possibly exhaust the number of available TCP sockets. Both of these issues are related to the implementation of networked channels in PyCSP.

We tested the latest release of the official PyCSP ver. 0.2.4 that comes with basic network support for channels. We expected that PyCSP ver. 0.2.4 would perform much worse in the network tests, since they rely on Pyro for communication. The performance of the extended PyCSP and PyCSP ver. 0.2.4 was equally high. When the official PyCSP at some point reaches a maturity where networked channels are supported as extensively as the implementation in this thesis, it might be an advantage to switch to the official PyCSP for CSPBuilder. This would solve the 2 major issues mentioned above.

Suggestions for improvements on the current CSPBuilder:

- *The search for the perfect grouping of processes on nodes is likely an NP-complete problem. A method that would find a close to perfect solution for this problem would increase the overall performance of CSPBuilder. Finding a better grouping becomes more important when the number of nodes increases. The current method developed in this thesis is far from optimal.*

- *Another method to obtaining an optimal execution is to move CSP processes between nodes during execution. The node servers should monitor the processes and if there is not enough resources available it sends one of the CPU-intensive processes to another node. This is done by saving the state of the node, sending it across the network and to the destination node restoring*

the state and thus resuming execution. All channels would have to support that the channel-ends could be moved around to any node.

- *When running a distributed execution we concurrently start all nodes and then start the main execution, which will spread processes across the nodes. This requires that we know the hostname and port number of all nodes. If this procedure was reversed, the main execution could start by waiting for n nodes. Then every node would be started one by one and when n nodes are up and running the processes are spread across the nodes and distributed execution commenses. Using this method we would only need to know the hostname and port number of the main execution, which would be listening for nodes.*

- *Create stable node servers, that are able to receive processes, execute a CSP network and shut it down. These node servers should be able to receive several different CSP process networks concurrently. This would enable us to dedicate a pool of machines for node servers and then let all the users of CSPBuilder on the network utitilize these node servers and share the resources available.*

The work required to produce the CSPBuilder framework has been extensive. It was desired to produce a complete framework, able to run scientific applications. With CSPBuilder we have a good first version of a scientific workflow model. Improvements can be made, but real work can be done with the current framework as it is today.

References

[1] Annoncement from intel: Working on 80 core cpu.
internet:`http://www.intel.com/pressroom/archive/releases/20070204comp.htm`.
Viewed online september 2007.

[2] Communicating sequential processes for java.
internet:`http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.
Viewed online january 2008.

[3] Description of moore's law.
internet:`http://www.intel.com/technology/mooreslaw/`.
Viewed online January 2008.

[4] Erlang programming language.
internet:`http://www.erlang.org/`.
Viewed online january 2008.

[5] F2py - fortran to python interface generator.
internet:`http://www.scipy.org/F2py`.
Viewed online January 2008.

[6] Flow-based programming.
internet:`http://en.wikipedia.org/wiki/Flow-based_programming`.
Viewed online september 2007.

[7] occam-pi: blending the best of csp and the pi-calculus.
internet:`http://www.cs.kent.ac.uk/projects/ofa/kroc/`.
Viewed online january 2008.

[8] Python remote objects library.
internet:`http://pyro.sourceforge.net/`.
Viewed online January 2008.

[9] Simplified wrapper and interface generator (swig).
internet:`http://www.swig.org`.
Viewed online january 2007.

[10] Thread state and the global interpreter lock.
internet:`http://docs.python.org/api/threads.html`.
Viewed online january 2008.

[11] Otto J. Anshus, J. Markus Bjoerndalen, and B. Vinter.
PyCSP - Communicating Sequential Processes for Python.
In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages –, jul 2007.

[12] 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.

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

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

[15] M. Jowkar.
Exploring the Potential of the Cell Processor for High Performance Computing.
University of Copenhagen DIKU, 2007.

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

[17] Berhnard H.C Sputh and Alastair R. Allan.
Jcsp-poison: Safe termination of csp process networks.
Communicating Process Architectures 2005, pages 71–107, 2005.

[18] M. Temiz.
CSP Library for Python.
University of Copenhagen DIKU, 2007.

[19] Peter Welch and Neil Brown.
An introduction to the kent c++csp library.
Communicating Process Architectures 2003, 2003.

[20] 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.

[21] 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.

[22] David M. Young.
Iterative Methods for Solving Partial Difference Equations of Elliptic Type.
Harvard University, Cambridge, Mass, 1950.

[23] Chaoyang Zhang, Hong Lan, Yang Ye, and Brett D. Estrade.
Parallel sor iterative algorithms and performance evaluation on a linux cluster.
Naval Reseach Lab, 2005.

A User Guide

This is a short guide on how to use CSPBuilder. The purpose of CSPBuilder is to enable developers to construct concurrent programs using a visual tool that simplifies complex applications and handles execution of applications, exploring the power of CSP[14]. CSPBuilder is an application and a framework that handles reuse of components from a component library.

A.1 Requirements

Depending on what features are used, more or less libraries are needed.

The minimum software requirements for basic operation of CSPBuilder are the following:

Common

- Python 2.5
  `http://www.python.org`
- Python standard modules.
  time, thread, threading, types, os, sys, xml.dom.minidom, base64, pickle or cPickle, socket
- Psyco 1.5.2
  `http://psyco.sourceforge.net`
  psyco

builder.py Building applications.

- Python standard modules.
  cStringIO, shutil
- wxPython 2.8.4
  `http://www.wxpython.org`
  wx, wx.wizard, wx.lib.buttons, wx.lib.editor

execute.py Executing applications.

- Python standard modules.
  random
- NumPy 1.0.4 (used in CSP components)
  `http://numpy.scipy.org`
  numpy
- Matplotlib / pylab (used in CSP components)
  `http://matplotlib.sourceforge.net`
  pylab

/External External Components in C / C++ or FORTRAN.

- GNU Make 3.81
  `http://www.gnu.org/software/make`
- GCC 4.1.2
  `http://www.gnu.org/software/gcc`
- SWIG 1.3.31
  `http://www.swig.org`
- F2py 2.3472
  `http://www.scipy.org/F2py`
- GNU Fortran
  `http://gcc.gnu.org/fortran`
- LibTomMath (used in External CSP components)
  `http://math.libtomcrypt.com/`

For CSPBuilder to be fully functional the following default ports and port-ranges are required to be available.

21567 (UDP) Timing server

12001-16001 (TCP) Channel communication

Ports used in NODES.py

A.2 Installation

To install CSPBuilder, unpack the tar.gz file. It will be necessary to compile the External components after unpacking. To do this, change to `/External` folder and edit `Rules.mk` to match the available executables on your system. After this, run `make` compiling the C / C++ and Fortran components packaged with CSPBuilder. If later on an external component is created, it is necessary to run `make` to compile the newly created component.

CSPBuilder is now ready to use, provided that the needed libraries / modules are available (see section A.1).

A.3 Build Applications

Start building applications by running `builder.py`. When finished, save the application as a `.csp` file. This file can be executed by `execute.py`, which is explained in the next section A.4.

When using CSPBuilder the main features that are available for use are shown in figures 34, 35, 36, 37 and 38.

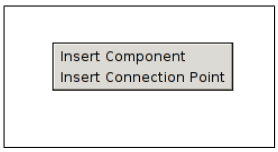

Figure 34: **`Right click on an empty area`** This menu pops up where a component or a CPoint can be inserted.

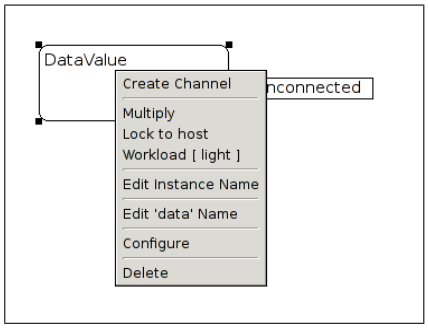

Figure 35: **`Right click on process`**

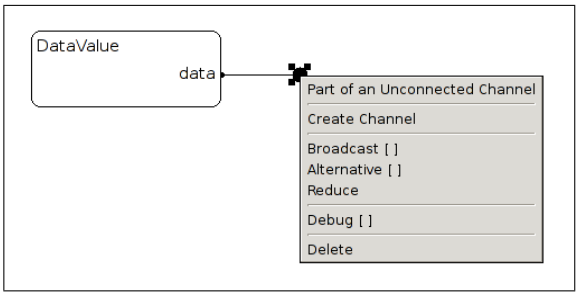

Figure 36: **`Right click on CPoint`**

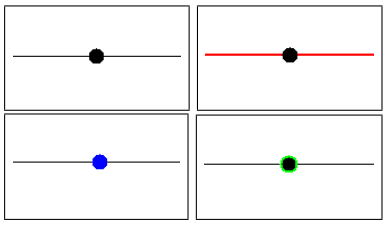

Figure 37: **`Channels with a CPoint`** Top-left: Regular channel. Top-right (red lines): Debug enabled. Bottom-left (blue circle): Alternative enabled. Bottom-right (green circle outline): Broadcast enabled.

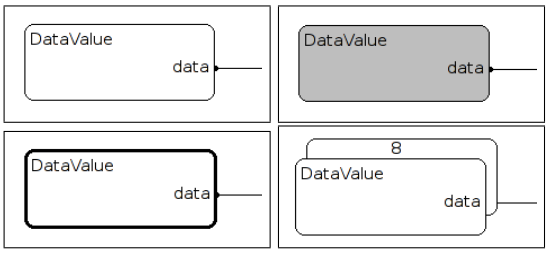

Figure 38: **Processes** Top-left: Regular process. Top-right: Workload set to heavy. Bottom-left: Locked to specific node in a network run. Bottom-right: Multiplied to 8 processes.

A.4 Execution

To execute an application use `execute.py`.

```
> ./execute.py --help

./execute.py [arguments] file.csp
Part of the CSPBuilder application.

Copyright (c) 2007 Rune Madsen, rune@wagawaga.dk.

Usage:
  -ddd, ---debug-all       Enable all debugging and output
  -dp, --debug-pycsp       Enable debugging and debug SHELL
  -dc, --debug-channels    Enable channel debug output
  -dl, --debug-config      Print all configured components
  -i, --info               Print info on processes, channels

  -n, --network            Run network execution. Requires NODES.py file
                           This option disables -dp and -dc

  -fn, --force-netchannels All channels will be network channels
  -uh, --disable-host      Disables run on specific host distribution.
  -ua, --disable-arch      Disables run on preferred arch distribution.
  -uw, --disable-workload  Disables heavy workload distribution.


debug SHELL commands:
  c   Prints all channels and their current state
  p   Prints all processes and their current state
  q   Quits shell (Will not quit application, if frozen)
```

```
> ./node_execute.py --help

./node_execute.py [arguments] hostname port
Part of the CSPBuilder application.

Copyright (c) 2007 Rune Madsen, rune@wagawaga.dk.
```

If doing a network run, edit `NODES.py` to resemble the number of nodes. Start nodes by running "`node_execute <host> <port>`" on each node. Finally run "`execute.py -n <csp file>`".

A small script `start_dist_execute.py` is available that performs the forementioned actions. To kill an execution use the script `kill_dist_execute.py`.

B Applications and Source Code

The CSPBuilder framework, the CSPBuilder applications and tests are available on the CD accompanying the thesis. The content of the CD can also be downloaded from this location: `http://diku.rm.wagawaga.dk/thesis.tgz`

The content of the CD:

`/Thesis.pdf` A copy of the thesis.

`/CSPBuilder.tgz` A packaged copy of the CSPBuilder folder

`/CSPBuilder/` The CSPBuilder framework and applications.

C Output from Correctness Tests

C.1 Concurrent Execution

Simple application that uses One2OneChannels

```
> ./execute.py Tests/Simple-CSP-Network-Numbers.csp
0123456789
```

Simple application with component written in C

```
> ./execute.py Tests/Simple-CSP-Network-Numbers-C.csp
0123456789
```

Simple application with component written in Fortran 77

```
> ./execute.py Tests/Simple-CSP-Network-Numbers-Fortran.csp
0123456789
```

Simple application that uses a One2AllChannel

```
> ./execute.py Tests/Simple-CSP-Network-Numbers-One2All.csp
0123456789
```

Simple application that uses a One2AnyChannels and an Any2OneChannel

```
> ./execute.py Tests/Simple-CSP-Network-Numbers-One2Any-Any2One.csp
0123456789
```

Simple application with a process network inserted as a component

```
> ./execute.py Tests/Simple-CSP-Network-Fibonacci.csp
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,
46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,
14930352,24157817,39088169,63245986,102334155,165580141,267914296,433494437,701408733,
1134903170,1836311903,2971215073,4807526976,7778742049,
```

Commstime

```
> ./execute.py Applications/Commstime. csp
 = 2.385458.
Time per ch : 2.385458/(4*5000) = 0.000119 s = 119.272900 us
consumer done, poisoning channel
```

Simulated One2AnyChannel to support guards

```
> ./execute.py Tests/One2Any-Alternative.csp
Thread(17595.0): Init!
Thread(27547.0): Init!
Thread(47694.0): Init!
Thread(20631.0): Init!
Thread(27547.0): OK! (0)
Thread(47694.0): DELAY TOO BIG >0.5s (1) DELAY=4.99874091148 s
Thread(17595.0): OK! (2)
Thread(47694.0): OK! (3)
Thread(20631.0): OK! (4)
Thread(17595.0): OK! (5)
Thread(27547.0): OK! (6)
Thread(47694.0): OK! (7)
Thread(27547.0): OK! (8)
Thread(17595.0): OK! (9)
Thread(27547.0): OK! (10)
Thread(20631.0): OK! (11)
Thread(47694.0): OK! (12)
Thread(47694.0): OK! (13)
Thread(20631.0): OK! (14)
Thread(47694.0): OK! (15)
Thread(17595.0): OK! (16)
Thread(27547.0): OK! (17)
Thread(27547.0): OK! (18)
Thread(20631.0): OK! (19)
Thread(27547.0): OK! (20)
Thread(17595.0): OK! (21)
Thread(47694.0): OK! (22)
Thread(20631.0): OK! (23)
Thread(17595.0): OK! (24)
Thread(27547.0): OK! (25)
Thread(47694.0): OK! (26)
Thread(20631.0): OK! (27)
Thread(17595.0): OK! (28)
Thread(20631.0): OK! (29)
```

```
Thread(27547.0): OK! (30)
Thread(47694.0): OK! (31)
Thread(17595.0): OK! (32)
Thread(20631.0): OK! (33)
Thread(20631.0): OK! (34)
Thread(27547.0): OK! (35)
Thread(47694.0): OK! (36)
Thread(17595.0): OK! (37)
Thread(47694.0): OK! (38)
Thread(47694.0): OK! (39)
Thread(20631.0): OK! (40)
Thread(27547.0): OK! (41)
Thread(47694.0): OK! (42)
Thread(27547.0): OK! (43)
Thread(17595.0): OK! (44)
Thread(20631.0): OK! (45)
Thread(27547.0): OK! (46)
Thread(17595.0): OK! (47)
Thread(17595.0): OK! (48)
Thread(27547.0): OK! (49)
Thread(17595.0): OK! (50)
```

Simulated One2AllChannel to support guards

```
> ./execute.py Tests/One2All-Alternative.csp
2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072,262144,
524288,1048576,
```

Multiply process instances according to the multiply setting

```
> ./execute.py Tests/Multiplier.csp
Hello World!Hello World!Hello World!Hello World!Hello World!
```

Load the configured data of a process instance

```
> ./execute.py Tests/Configuration.csp
Hello World!
```

Debugging channel

```
> ./execute.py -dc Tests/Simple-CSP-Network-Numbers-Debug.csp
0main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 1
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 1
1main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 2
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 2
2main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 3
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 3
3main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 4
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 4
main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 5
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 5
4main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 6
5main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 6
main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 7
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 7
67main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 8
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 8
8main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 9
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 9
9main.Simple-CSP-Network-Numbers.chan_cpoint4 write_debug 10
main.Simple-CSP-Network-Numbers.chan_cpoint4 read_debug 10
```

Debug configured data of process instances

```
> ./execute.py -dl Tests/Simple-CSP-Network-Numbers-Debug.csp
main.Simple-CSP-Network-Numbers.Printer.Printer : ['', '', 10]
main.Simple-CSP-Network-Numbers.DataValue.DataValue : 0
main.Simple-CSP-Network-Numbers.AssertTest.AssertTest :
                              [1, 'Tests/Simple-CSP-Network-Numbers-state.data']
```

Debugging channels and processes with the debug shell

```
> ./execute.py -dp Tests/Simple-CSP-Network-Numbers-Debug.csp
0123456789
c
One2One {'read': 0, 'write': 0} poisoned:True
  main.Simple-CSP-Network-Numbers.Successor.CSP_Successor.CSP_SuccessorFunc.out0.write
  main.Simple-CSP-Network-Numbers.Prefix.Prefix.PrefixFunc.in0.read
One2One {'read': 0, 'write': 0} poisoned:True
  main.Simple-CSP-Network-Numbers.Prefix.Prefix.PrefixFunc.out0.write
  main.Simple-CSP-Network-Numbers.Delta2.CSP_Delta2.CSP_Delta2Func.in0.read
One2One {'read': 0, 'write': 0} poisoned:True
  main.Simple-CSP-Network-Numbers.Delta2.CSP_Delta2.CSP_Delta2Func.out1.write
```

```
  main.Simple-CSP-Network-Numbers.Successor.CSP_Successor.CSP_SuccessorFunc.in0.read
One2One {'read': 0, 'write': 0} poisoned:True
  main.Simple-CSP-Network-Numbers.DataValue.DataValue.DataValueFunc.out0.write
  main.Simple-CSP-Network-Numbers.Prefix.Prefix.PrefixFunc.in1.read
One2One {'read': 0, 'write': 0} poisoned:True
  main.Simple-CSP-Network-Numbers.Delta2.CSP_Delta2.CSP_Delta2Func.out0.write
  main.Simple-CSP-Network-Numbers.AssertTest.AssertTest.AssertTestFunc.in0.read
One2One {'read': 0, 'write': 0} poisoned:True
  main.Simple-CSP-Network-Numbers.AssertTest.AssertTest.AssertTestFunc.out0.write
  main.Simple-CSP-Network-Numbers.Printer.Printer.SYS_PrinterFunc.in0.read
p
main.Simple-CSP-Network-Numbers.Prefix.Prefix
  One2One.read Active:0
  One2One.read Active:0
  One2One.write Active:0
main.Simple-CSP-Network-Numbers.Delta2.CSP_Delta2
  One2One.read Active:0
  One2One.write Active:0
  One2One.write Active:0
main.Simple-CSP-Network-Numbers.Successor.CSP_Successor
  One2One.read Active:0
  One2One.write Active:0
main.Simple-CSP-Network-Numbers.Printer.Printer
  One2One.read Active:0
main.Simple-CSP-Network-Numbers.DataValue.DataValue
  One2One.write Active:0
main.Simple-CSP-Network-Numbers.AssertTest.AssertTest
  One2One.read Active:0
  One2One.write Active:0
q
```

Printing CSP statistics

```
> ./execute.py -i Tests/Simple-CSP-Network-Numbers-Debug.csp
-
  ('ProcessCount', 13)
  ('CSPProcessCount', 6)
-
  ('One2OneChannel', 6)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
```

C.2 Network support for PyCSP

One2One, One2Any, One2All and Any2One Channel

```
[@amigos24 pycsp_test]> python2.5 channeltest_net.py
-----------------------
Testing One2One Channel
Reader and writer should both report as done
  [<Process(Thread-3, started)>] Writing 0
  [<Process(Thread-4, started)>] Reading 0
  [<Process(Thread-3, started)>] Writing 1
  [<Process(Thread-4, started)>] Reading 1
  [<Process(Thread-3, started)>] Writing 2
  [<Process(Thread-4, started)>] Reading 2
  [<Process(Thread-3, started)>] Writing 3
  [<Process(Thread-4, started)>] Reading 3
  [<Process(Thread-3, started)>] Writing 4
  [<Process(Thread-4, started)>] Reading 4
Writer [<Process(Thread-3, started)>] wrote all
Reader [<Process(Thread-4, started)>] got all
' poisonSRV_ '
' poisonCLI_ '
-----------------------
Testing One2Any Channel
Writer should report as done, none of the readers should
  [<Process(Thread-9, started)>] Writing 0
  [<Process(Thread-11, started)>] Reading 0
  [<Process(Thread-9, started)>] Writing 1
  [<Process(Thread-11, started)>] Reading 1
  [<Process(Thread-9, started)>] Writing 2
  [<Process(Thread-11, started)>] Reading 2
  [<Process(Thread-9, started)>] Writing 3
  [<Process(Thread-11, started)>] Reading 3
  [<Process(Thread-9, started)>] Writing 4
  [<Process(Thread-11, started)>] Reading 4
Reader [<Process(Thread-11, started)>] got all
' poisonSRV_ '
Writer [<Process(Thread-9, started)>] wrote all
```

```
’  ’

[@amigos24 pycsp_test]> python2.5 channeltest_net.py
-----------------------
Testing One2All Channel
Writer should report as done, none of the readers should
  [<Process(Thread-5, started)>] Writing 0
  [<Process(Thread-6, started)>] Reading 0
  [<Process(Thread-7, started)>] Reading 0
  [<Process(Thread-5, started)>] Writing 1
  [<Process(Thread-6, started)>] Reading 1
  [<Process(Thread-7, started)>] Reading 1
  [<Process(Thread-5, started)>] Writing 2
  [<Process(Thread-6, started)>] Reading 2
  [<Process(Thread-7, started)>] Reading 2
  [<Process(Thread-5, started)>] Writing 3
  [<Process(Thread-6, started)>] Reading 3
  [<Process(Thread-7, started)>] Reading 3
  [<Process(Thread-5, started)>] Writing 4
  [<Process(Thread-6, started)>] Reading 4
  [<Process(Thread-7, started)>] Reading 4
Reader [<Process(Thread-6, started)>] got all
Reader [<Process(Thread-7, started)>] got all
Writer [<Process(Thread-5, started)>] wrote all
’ poisonSRV_ ’
’ poisonSRV_ ’
’ poisonCLI_ ’
’  ’
-----------------------
Testing Any2One Channel
Reader should report as done, none of the writers should
  [<Process(Thread-13, started)>] Writing 0
  [<Process(Thread-12, started)>] Writing 0
  [<Process(Thread-14, started)>] Reading 0
  [<Process(Thread-14, started)>] Reading 0
  [<Process(Thread-12, started)>] Writing 1
  [<Process(Thread-13, started)>] Writing 1
  [<Process(Thread-14, started)>] Reading 1
  [<Process(Thread-12, started)>] Writing 2
  [<Process(Thread-14, started)>] Reading 1
  [<Process(Thread-13, started)>] Writing 2
  [<Process(Thread-14, started)>] Reading 2
  [<Process(Thread-12, started)>] Writing 3
Reader [<Process(Thread-14, started)>] got all
’ poisonSRV_ ’
’ poisonSRV_ ’
```

One2One and Any2One Channel Guards

```
[@amigos24 pycsp_test]> python2.5 alt_test_net.py
Bip 2
Bip 1
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 0
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 1
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 2
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 3
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 4
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 5
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 6
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 7
ding 1
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type ’instancemethod’> foo 8
ding 1
```

```
p1: got from select: <bound method Net2OneChannel._call of
<pycsp.NetChannels.Net2OneChannel instance at 0xb7cac44c>>
<type 'instancemethod'> foo 9
' poisonSRV_ '
'  '
Bip 2
Bip 2
Bip 1
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 0
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 1
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 2
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 10
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 3
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 4
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 11
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 5
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 6
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 7
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 12
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 8
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 13
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 9
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 14
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 15
ding 1
p1: got from select: <bound method AnyNet2OneChannel._call of
<pycsp.NetChannels.AnyNet2OneChannel instance at 0xb7cc2fac>>
<type 'instancemethod'> foo 16
ding 1
' poisonCLI_ '
'  '
```

C.3 Organizing Processes on Nodes

Case 1 (Lock to node)

```
Locked to node 2 compute-0-1.local
Locked to node 3 compute-0-2.local
Locked to node 1 amigos24.diku.dk
Locked to node 1 amigos24.diku.dk
```

```
Locked to node 1 amigos24.diku.dk
Locked to node 1 amigos24.diku.dk
```

Case 2 (Heavy workload)

```
Heavy Workload compute-0-3.local
Heavy Workload compute-0-2.local
Heavy Workload compute-0-1.local
Heavy Workload amigos24.diku.dk
Normal compute-0-1.local
Normal compute-0-2.local
Normal amigos24.diku.dk
```

Case 3 (Preferred architecture)

```
Normal compute-0-2.local
Normal compute-0-1.local
Normal compute-0-3.local
Itanium arch amigos24.diku.dk
Itanium arch amigos24.diku.dk
```

Case 4 (General grouping)

```
Grouping-General 8 processes
General compute-0-1.local
General compute-0-1.local
General compute-0-2.local
General compute-0-2.local
General compute-0-3.local
General compute-0-3.local
General amigos24.diku.dk
General amigos24.diku.dk


Grouping-General 9 processes
General compute-0-1.local
General compute-0-1.local
General compute-0-2.local
General compute-0-2.local
General compute-0-3.local
General compute-0-3.local
General amigos24.diku.dk
General amigos24.diku.dk
General amigos24.diku.dk


Grouping-General 10 processes
General compute-0-2.local
General compute-0-2.local
General compute-0-3.local
General compute-0-3.local
General compute-0-1.local
General compute-0-1.local
General compute-0-1.local
General amigos24.diku.dk
General amigos24.diku.dk
General amigos24.diku.dk


Grouping-General 11 processes
General compute-0-3.local
General compute-0-3.local
General compute-0-1.local
General compute-0-1.local
General compute-0-1.local
General compute-0-2.local
General compute-0-2.local
General compute-0-2.local
General amigos24.diku.dk
General amigos24.diku.dk
General amigos24.diku.dk
```

Case 5 (Combined)

```
Locked to node 1 amigos24.diku.dk
Locked to node 1 amigos24.diku.dk
Locked to node 1 amigos24.diku.dk
Locked to node 1 amigos24.diku.dk
Itanium arch amigos24.diku.dk
Itanium arch amigos24.diku.dk

Heavy Workload compute-0-3.local
Heavy Workload compute-0-3.local
Normal compute-0-3.local
Normal compute-0-3.local
```

```
Normal compute-0-3.local
Normal compute-0-3.local

Locked to node 2 compute-0-1.local
Heavy Workload compute-0-1.local
General compute-0-1.local
General compute-0-1.local
General compute-0-1.local
General compute-0-1.local
General compute-0-1.local

Locked to node 3 compute-0-2.local
Heavy Workload compute-0-2.local
General compute-0-2.local
General compute-0-2.local
General compute-0-2.local
Normal compute-0-2.local
Normal compute-0-2.local
```

D Output from Performance Tests

D.1 Concurrent Execution

Commstime - Python and PyCSP

```
[@compute-0-1 pycsp_test]> python2.5 commstime.py
----------- run 1/10 -------------
Running commstime test
DT = 1.824518.
Time per ch : 1.824518/(4*5000) = 0.000091 s = 91.225898 us
consumer done, posioning channel
----------- run 2/10 -------------
Running commstime test
DT = 1.819889.
Time per ch : 1.819889/(4*5000) = 0.000091 s = 90.994453 us
consumer done, posioning channel
----------- run 3/10 -------------
Running commstime test
DT = 1.841657.
Time per ch : 1.841657/(4*5000) = 0.000092 s = 92.082846 us
consumer done, posioning channel
```

Commstime - Python and PyCSP

```
[@compute-0-1 CSPBuilder]> ./execute.py -i Applications/Commstime.csp
-
  ('ProcessCount', 11)
  ('CSPProcessCount', 5)
-
  ('One2OneChannel', 5)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
DT = 1.902147.
Time per ch : 1.902147/(4*5000) = 0.000095 s = 95.107353 us
consumer done, poisoning channel
[@compute-0-1 CSPBuilder]> ./execute.py -i Applications/Commstime.csp
-
  ('ProcessCount', 11)
  ('CSPProcessCount', 5)
-
  ('One2OneChannel', 5)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
DT = 1.918872.
Time per ch : 1.918872/(4*5000) = 0.000096 s = 95.943594 us
consumer done, poisoning channel
[@compute-0-1 CSPBuilder]> ./execute.py -i Applications/Commstime.csp
-
  ('ProcessCount', 11)
  ('CSPProcessCount', 5)
-
  ('One2OneChannel', 5)
```

```
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
DT = 1.936915.
Time per ch : 1.936915/(4*5000) = 0.000097 s = 96.845746 us
consumer done, poisoning channel
```

D.2 1 vs. 2 TCP Channels

1 TCP Connection

```
[@compute-0-2 TestsInReport]> python2.5 1_TCP_Channels_2.py
Testing 1 TCP channels
starting loop
Time: 2.26608610153
[@compute-0-2 TestsInReport]> python2.5 1_TCP_Channels_2.py
Testing 1 TCP channels
starting loop
Time: 2.31251311302
[@compute-0-2 TestsInReport]> python2.5 1_TCP_Channels_2.py
Testing 1 TCP channels
starting loop
Time: 2.13008999825

Average = 2.23 seconds
```

2 TCP Connections

```
[@compute-0-2 TestsInReport]> python2.5 2_TCP_Channels_2.py
Testing 2 TCP channels
starting loop
Time: 2.15294194221
[@compute-0-2 TestsInReport]> python2.5 2_TCP_Channels_2.py
Testing 2 TCP channels
starting loop
Time: 2.0948779583
[@compute-0-2 TestsInReport]> python2.5 2_TCP_Channels_2.py
Testing 2 TCP channels
starting loop
Time: 2.0757830143

Average = 2.10 seconds
```

Commstime standard

```
[@compute-0-3 pycsp_test]> python2.5 commstime.py
---------- run 1/10 -------------
Running commstime test
DT = 1.844168.
Time per ch : 1.844168/(4*5000) = 0.000092 s = 92.208397 us
consumer done, posioning channel
---------- run 2/10 -------------
Running commstime test
DT = 1.844696.
Time per ch : 1.844696/(4*5000) = 0.000092 s = 92.234802 us
consumer done, posioning channel
---------- run 3/10 -------------
Running commstime test
DT = 1.832641.
Time per ch : 1.832641/(4*5000) = 0.000092 s = 91.632056 us
consumer done, posioning channel


Average = 92.02 us
```

Commstime Net with 2 TCP Connections

```
[@compute-0-3 pycsp_test]> python2.5 commstime_net.py
---------- run 1/10 -------------
Running commstime test
DT = 4.185449.
Time per ch : 4.185449/(4*5000) = 0.000209 s = 209.272444 us
consumer done, posioning channel
' poisonSRV_ '

[@compute-0-3 pycsp_test]> python2.5 commstime_net.py
---------- run 1/10 -------------
Running commstime test
DT = 4.279868.
```

```
Time per ch : 4.279868/(4*5000) = 0.000214 s = 213.993394 us
consumer done, posioning channel
' poisonSRV_ '

[@compute-0-3 pycsp_test]> python2.5 commstime_net.py
---------- run 1/10 -------------
Running commstime test
DT = 4.191973.
Time per ch : 4.191973/(4*5000) = 0.000210 s = 209.598649 us
consumer done, posioning channel
' poisonSRV_ '

Average = 210.95 us
```

Commstime Net with 1 TCP Connection

```
[@compute-0-3 pycsp_test]> python2.5 commstime_net.py
---------- run 1/10 -------------
Running commstime test
DT = 22.257356.
Time per ch : 22.257356/(4*5000) = 0.001113 s = 1112.867796 us
consumer done, posioning channel
' poisonSRV_ '

[@compute-0-3 pycsp_test]> python2.5 commstime_net.py
---------- run 1/10 -------------
Running commstime test
DT = 26.475750.
Time per ch : 26.475750/(4*5000) = 0.001324 s = 1323.787498 us
consumer done, posioning channel
' poisonSRV_ '

[@compute-0-3 pycsp_test]> python2.5 commstime_net.py
---------- run 1/10 -------------
Running commstime test
DT = 21.530176.
Time per ch : 21.530176/(4*5000) = 0.001077 s = 1076.508796 us
consumer done, posioning channel
' poisonSRV_ '

Average = 1171.05 us
```

CSPBuilder Commstime with 2 TCP Connections

```
DT = 9.710605.
Time per ch : 9.710605/(4*5000) = 0.000486 s = 485.530245 us
consumer done, poisoning channel

DT = 9.951881.
Time per ch : 9.951881/(4*5000) = 0.000498 s = 497.594059 us
consumer done, poisoning channel

DT = 9.719879.
Time per ch : 9.719879/(4*5000) = 0.000486 s = 485.993946 us
consumer done, poisoning channel

Average = 489.70 us
```

CSPBuilder Commstime with 1 TCP Connection

```
DT = 403.970155.
Time per ch : 403.970155/(4*5000) = 0.020199 s = 20198.507750 us
consumer done, poisoning channel

DT = 403.607193.
Time per ch : 403.607193/(4*5000) = 0.020180 s = 20180.359650 us
consumer done, poisoning channel

DT = 404.083394.
Time per ch : 404.083394/(4*5000) = 0.020204 s = 20204.169703 us
consumer done, poisoning channel

Average = 20194.34 us
```

D.3 Network support for PyCSP

PyCSP ver. 0.2.2 - Regular Channels

```
[@compute-0-1 test]> python2.5 commstime.py
---------- run 1/10 -------------
Running commstime test
DT = 1.829689.
Time per ch : 1.829689/(4*5000) = 0.000091 s = 91.484451 us
consumer done, posioning channel
---------- run 2/10 -------------
```

```
Running commstime test
DT = 1.809707.
Time per ch : 1.809707/(4*5000) = 0.000090 s = 90.485358 us
consumer done, posioning channel
---------- run 3/10 -------------
Running commstime test
DT = 1.817511.
Time per ch : 1.817511/(4*5000) = 0.000091 s = 90.875554 us
consumer done, posioning channel
```

PyCSP ver. 0.2.4-net - Regular Channels

```
[@compute-0-1 test]> python2.5 commstime.py
---------- run 1/10 -------------
Running commstime test
DT = 3.159140.
Time per ch : 3.159140/(4*5000) = 0.000158 s = 157.956994 us
consumer done, posioning channel
---------- run 2/10 -------------
Running commstime test
DT = 3.164077.
Time per ch : 3.164077/(4*5000) = 0.000158 s = 158.203852 us
consumer done, posioning channel
---------- run 3/10 -------------
Running commstime test
DT = 3.178653.
Time per ch : 3.178653/(4*5000) = 0.000159 s = 158.932650 us
consumer done, posioning channel
```

PyCSP-net - Regular Channels

```
[@compute-0-1 pycsp_test]> python2.5 commstime.py
---------- run 1/10 -------------
Running commstime test
DT = 1.824518.
Time per ch : 1.824518/(4*5000) = 0.000091 s = 91.225898 us
consumer done, posioning channel
---------- run 2/10 -------------
Running commstime test
DT = 1.819889.
Time per ch : 1.819889/(4*5000) = 0.000091 s = 90.994453 us
consumer done, posioning channel
---------- run 3/10 -------------
Running commstime test
DT = 1.841657.
Time per ch : 1.841657/(4*5000) = 0.000092 s = 92.082846 us
consumer done, posioning channel
```

PyCSP ver. 0.2.4-net (Pyro) - Network Channels on localhost

```
[@compute-0-1 test]> python2.5 commstime_net_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
---------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 6.552993.
Time per ch : 6.552993/(4*5000) = 0.000328 s = 327.649653 us
consumer done, posioning channel

[@compute-0-1 test]> python2.5 commstime_net_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
---------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 6.544076.
Time per ch : 6.544076/(4*5000) = 0.000327 s = 327.203798 us
consumer done, posioning channel


[@compute-0-1 test]> python2.5 commstime_net_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
---------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 6.600840.
Time per ch : 6.600840/(4*5000) = 0.000330 s = 330.042005 us
consumer done, posioning channel
```

PyCSP-net (TCP) - Network Channels on localhost

```
[@compute-0-1 pycsp_test]> python2.5 commstime_net.py
---------- run 1/10 -------------
Running commstime test
DT = 4.010570.
Time per ch : 4.010570/(4*5000) = 0.000201 s = 200.528502 us
consumer done, posioning channel
' poisonSRV_ '
'  '
---------- run 2/10 -------------
Running commstime test
DT = 4.038543.
Time per ch : 4.038543/(4*5000) = 0.000202 s = 201.927149 us
consumer done, posioning channel
' poisonSRV_ '
'  '
---------- run 3/10 -------------
Running commstime test
DT = 4.015660.
Time per ch : 4.015660/(4*5000) = 0.000201 s = 200.783002 us
consumer done, posioning channel
' poisonSRV_ '
'  '
```

PyCSP ver. 0.2.4-net (Pyro) - Network Channels on 2 hosts

```
[@compute-0-1 test]> python2.5 commstime_net.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
---------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [192.38.109.154 (amigos24.diku.dk : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 5.258536.
Time per ch : 5.258536/(4*5000) = 0.000263 s = 262.926805 us
consumer done, posioning channel


[@compute-0-1 test]> python2.5 commstime_net.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
---------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [192.38.109.154 (amigos24.diku.dk : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 5.258982.
Time per ch : 5.258982/(4*5000) = 0.000263 s = 262.949109 us
consumer done, posioning channel
---------- run 2/2 -------------
Registering obj as service name :pycsp_test/chans/d1
Running commstime test


[@compute-0-1 test]> python2.5 commstime_net.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
---------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [192.38.109.154 (amigos24.diku.dk : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 5.269891.
Time per ch : 5.269891/(4*5000) = 0.000263 s = 263.494551 us
consumer done, posioning channel
```

PyCSP-net (TCP) - Network Channels on 2 hosts

```
[@compute-0-1 pycsp_test]> python2.5 commstime_net_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.481231.
Time per ch : 3.481231/(4*5000) = 0.000174 s = 174.061549 us
consumer done, posioning channel


[@compute-0-1 pycsp_test]> python2.5 commstime_net_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.432745.
Time per ch : 3.432745/(4*5000) = 0.000172 s = 171.637249 us
consumer done, posioning channel
```

```
[@compute-0-1 pycsp_test]> python2.5 commstime_net_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.394069.
Time per ch : 3.394069/(4*5000) = 0.000170 s = 169.703448 us
consumer done, posioning channel
```

D.4 Network Bandwidth with PyCSP

```
PyCSP-net : Commstime 1 networked channel
Data: 10 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.518560.
Time per ch : 3.518560/(4*5000) = 0.000176 s = 175.928009 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 100 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.478126.
Time per ch : 3.478126/(4*5000) = 0.000174 s = 173.906291 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 200 bytes
2 Hosts


[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.448380.
Time per ch : 3.448380/(4*5000) = 0.000172 s = 172.419000 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 500 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.556259.
Time per ch : 3.556259/(4*5000) = 0.000178 s = 177.812946 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 600 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 199.612684.
Time per ch : 199.612684/(4*5000) = 0.009981 s = 9980.634201 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 800 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 199.691358.
Time per ch : 199.691358/(4*5000) = 0.009985 s = 9984.567904 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 1000 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
```

```
Running commstime test
DT = 199.416407.
Time per ch : 199.416407/(4*5000) = 0.009971 s = 9970.820355 us
consumer done, posioning channel

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 199.417375.
Time per ch : 199.417375/(4*5000) = 0.009971 s = 9970.868754 us
consumer done, posioning channel

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 199.416407.
Time per ch : 199.416407/(4*5000) = 0.009971 s = 9970.820343 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 1100 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 199.456285.
Time per ch : 199.456285/(4*5000) = 0.009973 s = 9972.814250 us


PyCSP-net : Commstime 1 networked channel
Data: 1200 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 199.493951.
Time per ch : 199.493951/(4*5000) = 0.009975 s = 9974.697542 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 1300 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 199.537005.
Time per ch : 199.537005/(4*5000) = 0.009977 s = 9976.850247 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 1400 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 199.574669.
Time per ch : 199.574669/(4*5000) = 0.009979 s = 9978.733444 us
consumer done, posioning channel

PyCSP-net : Commstime 1 networked channel
Data: 1450 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.661843.
Time per ch : 3.661843/(4*5000) = 0.000183 s = 183.092153 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 1500 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.524317.
Time per ch : 3.524317/(4*5000) = 0.000176 s = 176.215851 us
consumer done, posioning channel
```

```
PyCSP-net : Commstime 1 networked channel
Data: 2000 bytes
2 Hosts


[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.935161.
Time per ch : 3.935161/(4*5000) = 0.000197 s = 196.758056 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 10 Kbytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 4.998906.
Time per ch : 4.998906/(4*5000) = 0.000250 s = 249.945307 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 100 Kbytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 18.903736.
Time per ch : 18.903736/(4*5000) = 0.000945 s = 945.186806 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel
Data: 1 Mbytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 168.338040.
Time per ch : 168.338040/(4*5000) = 0.008417 s = 8416.902006 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (hack)
Data: 500 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.504455.
Time per ch : 3.504455/(4*5000) = 0.000175 s = 175.222754 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (hack)
Data: 1000 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.713313.
Time per ch : 3.713313/(4*5000) = 0.000186 s = 185.665643 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (hack)
Data: 1200 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.966907.
Time per ch : 3.966907/(4*5000) = 0.000198 s = 198.345351 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (TCP_NODELAY)
Data: 10 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
```

```
---------- run 1/10 -------------
Running commstime test
DT = 3.298065.
Time per ch : 3.298065/(4*5000) = 0.000165 s = 164.903247 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (TCP_NODELAY)
Data: 100 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.323100.
Time per ch : 3.323100/(4*5000) = 0.000166 s = 166.155005 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (TCP_NODELAY)
Data: 500 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.426185.
Time per ch : 3.426185/(4*5000) = 0.000171 s = 171.309257 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (TCP_NODELAY)
Data: 1000 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.396749.
Time per ch : 3.396749/(4*5000) = 0.000170 s = 169.837451 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (TCP_NODELAY)
Data: 1200 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 3.479023.
Time per ch : 3.479023/(4*5000) = 0.000174 s = 173.951149 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (TCP_NODELAY)
Data: 10000 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 5.003991.
Time per ch : 5.003991/(4*5000) = 0.000250 s = 250.199544 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (TCP_NODELAY)
Data: 100000 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 24.027360.
Time per ch : 24.027360/(4*5000) = 0.001201 s = 1201.367998 us
consumer done, posioning channel


PyCSP-net : Commstime 1 networked channel (TCP_NODELAY)
Data: 1000000 bytes
2 Hosts

[@compute-0-1 pycsp_test]> python2.5 commstime_net_data_t1.py
---------- run 1/10 -------------
Running commstime test
DT = 212.100452.
Time per ch : 212.100452/(4*5000) = 0.010605 s = 10605.022597 us
consumer done, posioning channel
```

```
PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 10 bytes
1 Host

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 6.670795.
Time per ch : 6.670795/(4*5000) = 0.000334 s = 333.539760 us
consumer done, posioning channel


PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 100 bytes
1 Host

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 6.693113.
Time per ch : 6.693113/(4*5000) = 0.000335 s = 334.655654 us
consumer done, posioning channel


PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 1000 bytes
1 Host

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 6.862480.
Time per ch : 6.862480/(4*5000) = 0.000343 s = 343.123996 us
consumer done, posioning channel


PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 10 Kbytes
1 Host

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 7.474496.
Time per ch : 7.474496/(4*5000) = 0.000374 s = 373.724806 us
consumer done, posioning channel


PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 100 Kbytes
1 Host

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 16.727751.
Time per ch : 16.727751/(4*5000) = 0.000836 s = 836.387551 us
consumer done, posioning channel
```

```
PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 1 Mbytes
1 Host

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 111.658246.
Time per ch : 111.658246/(4*5000) = 0.005583 s = 5582.912302 us
consumer done, posioning channel


PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 10 bytes
2 Hosts

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 5.150338.
Time per ch : 5.150338/(4*5000) = 0.000258 s = 257.516897 us
consumer done, posioning channel



PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 100 bytes
2 Hosts

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 5.232043.
Time per ch : 5.232043/(4*5000) = 0.000262 s = 261.602151 us
consumer done, posioning channel


PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 1000 bytes
2 Hosts

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 5.840739.
Time per ch : 5.840739/(4*5000) = 0.000292 s = 292.036951 us
consumer done, posioning channel


PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 10 Kbytes
2 Hosts

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 6.356986.
Time per ch : 6.356986/(4*5000) = 0.000318 s = 317.849302 us
consumer done, posioning channel
```

```
PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 100 Kbytes
2 Hosts

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 17.525563.
Time per ch : 17.525563/(4*5000) = 0.000876 s = 876.278150 us
consumer done, posioning channel


PyCSP-net : PyCSP ver. 0.2.4-net : Commstime 1 networked channel
Data: 1 Mbytes
2 Hosts

[@compute-0-1 test]> python2.5 commstime_net_data_t1.py
WARNING: please change 'localhost' to the host you run the pyro
nameserver on in pyrocom.py
----------- run 1/2 -------------
Initializing server
Pyro Server Initialized. Using Pyro V3.7
Found Name server at  [10.255.255.253 (compute-0-1.local : 9090)]
Registering obj as service name :pycsp_test/chans/d0
Running commstime test
DT = 142.036725.
Time per ch : 142.036725/(4*5000) = 0.007102 s = 7101.836252 us
consumer done, posioning channel
```

D.5 Prime Factorization - 1 to 8 workers

```
The Number: (2^222) - 1
Workers: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
main.PrimeFacCBigNum.DataValue.DataValue :
67399866667876599486667537717549076684092861056351431202759025623 03
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 11)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
  '26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
> 0.022027s Main::ParseXML
> 0.000178s Main::OrganizeNetwork
> 0.001138s Main::SetupCSPNetwork
> 221.690906s Main::Execution

The Number: (2^222) - 1
Workers: 7
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
main.PrimeFacCBigNum.DataValue.DataValue :
67399866667876599486667537717549076684092861056351431202759025623 03
-
  ('ProcessCount', 21)
  ('CSPProcessCount', 10)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
> 0.021542s Main::ParseXML
> 0.000181s Main::OrganizeNetwork
> 0.000957s Main::SetupCSPNetwork
> 249.277848s Costum::Calculation done


The Number: (2^222) - 1
```

```
Workers: 6
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
main.PrimeFacCBigNum.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 19)
  ('CSPProcessCount', 9)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
> 0.021809s Main::ParseXML
> 0.000172s Main::OrganizeNetwork
> 0.000876s Main::SetupCSPNetwork
> 282.138925s Main::Execution


The Number: (2^222) - 1
Workers: 4
Job_size: 10^6


@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
main.PrimeFacCBigNum.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 15)
  ('CSPProcessCount', 7)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
> 0.020231s Main::ParseXML
> 0.000175s Main::OrganizeNetwork
> 0.000897s Main::SetupCSPNetwork
> 413.562633s Costum::Calculation done


The Number: (2^222) - 1
Workers: 2
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
main.PrimeFacCBigNum.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 11)
  ('CSPProcessCount', 5)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
> 0.018573s Main::ParseXML
> 0.000181s Main::OrganizeNetwork
> 0.000631s Main::SetupCSPNetwork
> 776.128018s Costum::Calculation done


The Number: (2^222) - 1
Workers: 1
Job_size: 10^6


@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
main.PrimeFacCBigNum.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 9)
  ('CSPProcessCount', 4)
-
  ('One2OneChannel', 4)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']
```

```
> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
> 0.017658s Main::ParseXML
> 0.000176s Main::OrganizeNetwork
> 0.000535s Main::SetupCSPNetwork
> 1546.573442s Costum::Calculation done


The Number: (2^222) - 1
Workers: 1
Job_size: 10^16

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
main.PrimeFacCBigNum.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 9)
  ('CSPProcessCount', 4)
-
  ('One2OneChannel', 4)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
> 0.019437s Main::ParseXML
> 0.000188s Main::OrganizeNetwork
> 0.000547s Main::SetupCSPNetwork
> 1548.193005s Costum::Calculation done



The Number: (2^222) - 1
Workers: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFac_DoBigNumber.csp
main.testName.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 11)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFac_DoBigNumber.csp
> 0.022339s Main::ParseXML
> 0.000180s Main::OrganizeNetwork
> 0.001033s Main::SetupCSPNetwork
> 1226.931243s Costum::Calculation done


The Number: (2^222) - 1
Workers: 6
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFac_DoBigNumber.csp
main.testName.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 19)
  ('CSPProcessCount', 9)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFac_DoBigNumber.csp
> 0.020858s Main::ParseXML
> 0.000184s Main::OrganizeNetwork
> 0.000900s Main::SetupCSPNetwork
> 1204.991989s Costum::Calculation done


The Number: (2^222) - 1
Workers: 4
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFac_DoBigNumber.csp
main.testName.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
```

```
-
  ('ProcessCount', 15)
  ('CSPProcessCount', 7)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFac_DoBigNumber.csp
> 0.047404s Main::ParseXML
> 0.000186s Main::OrganizeNetwork
> 0.000895s Main::SetupCSPNetwork
> 1183.380027s Costum::Calculation done


The Number: (2^222) - 1
Workers: 2
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFac_DoBigNumber.csp
main.testName.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 11)
  ('CSPProcessCount', 5)
-
  ('One2OneChannel', 2)
  ('Any2OneChannel', 1)
  ('One2AnyChannel', 1)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFac_DoBigNumber.csp
> 0.018292s Main::ParseXML
> 0.000185s Main::OrganizeNetwork
> 0.000650s Main::SetupCSPNetwork
> 1145.101460s Costum::Calculation done


The Number: (2^222) - 1
Workers: 1
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i -dl Applications/PrimeFac/PrimeFac_DoBigNumber.csp
main.testName.DataValue.DataValue :
6739986666787659948666753771754907668409286105635143120275902562303
-
  ('ProcessCount', 9)
  ('CSPProcessCount', 4)
-
  ('One2OneChannel', 4)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

> Output from timing server
> 0.000000s Main::Init Applications/PrimeFac/PrimeFac_DoBigNumber.csp
> 0.018744s Main::ParseXML
> 0.000332s Main::OrganizeNetwork
> 0.000606s Main::SetupCSPNetwork
> 1004.795276s Costum::Calculation done
```

D.6 Prime Factorization - 1 to 8 nodes

```
The Number: (2^222) - 1
Workers: 8
Nodes: 8
CPU count: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport9910 ./node_execute.py localhost 9910
screen -L -dmS nodeport9911 ./node_execute.py localhost 9911
screen -L -dmS nodeport9912 ./node_execute.py localhost 9912
screen -L -dmS nodeport9913 ./node_execute.py localhost 9913
screen -L -dmS nodeport9914 ./node_execute.py localhost 9914
screen -L -dmS nodeport9915 ./node_execute.py localhost 9915
```

```
screen -L -dmS nodeport9916 ./node_execute.py localhost 9916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 23)
  ('CSPProcessCount', [('localhost', -1, 2), ('localhost', 9910, 2),
('localhost', 9911, 2), ('localhost', 9912, 1), ('localhost', 9913,
  1), ('localhost', 9914, 1), ('localhost', 9915, 1), ('localhost',
  9916, 1), ('ALL', 11)])
-
  ('One2OneChannel', 0)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 10)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 1)
  ('Net2OneChannel', 3)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']
'  '
' poisonSRV_ '

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.022398s Main::ParseXML
0.004564s Main::OrganizeNetwork
0.025907s Main::SetupCSPNetwork
207.966113s Costum::Calculation done


The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 7
Nodes: 7
CPU count: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19911 ./node_execute.py localhost 19911
screen -L -dmS nodeport19912 ./node_execute.py localhost 19912
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19914 ./node_execute.py localhost 19914
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 21)
  ('CSPProcessCount', [('localhost', -1, 3), ('localhost', 19910, 2),
('localhost', 19911, 2), ('localhost', 19912, 1), ('localhost', 19913,
  1), ('localhost', 19914, 1), ('localhost', 19916, 1), ('ALL', 11)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 15)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 8)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.023960s Main::ParseXML
0.004791s Main::OrganizeNetwork
0.106630s Main::SetupCSPNetwork
743.946676s Costum::Calculation done


The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 6
Nodes: 6
CPU count: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19911 ./node_execute.py localhost 19911
screen -L -dmS nodeport19912 ./node_execute.py localhost 19912
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
```

```
  ('ProcessCount', 19)
  ('CSPProcessCount', [('localhost', -1, 3), ('localhost', 19910, 2),
('localhost', 19911, 2), ('localhost', 19912, 1), ('localhost', 19913, 1),
('localhost', 19916, 1), ('ALL', 10)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 13)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 7)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.021579s Main::ParseXML
0.003786s Main::OrganizeNetwork
0.103363s Main::SetupCSPNetwork
828.573166s Costum::Calculation done


The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 4
Nodes: 4
CPU count: 8
Job_size: 10^6
@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 15)
  ('CSPProcessCount', [('localhost', -1, 2), ('localhost', 19910, 2),
('localhost', 19913, 2), ('localhost', 19916, 2), ('ALL', 8)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 10)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 6)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']
0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.020396s Main::ParseXML
0.002538s Main::OrganizeNetwork
0.084914s Main::SetupCSPNetwork
618.681084s Costum::Calculation done



The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 2
Nodes: 2
CPU count: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 11)
  ('CSPProcessCount', [('localhost', -1, 4), ('localhost', 19910, 2), ('ALL', 6)])
-
  ('One2OneChannel', 3)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 4)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 2)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']
```

```
0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.036176s Main::ParseXML
0.001149s Main::OrganizeNetwork
0.052109s Main::SetupCSPNetwork
1043.017717s Costum::Calculation done



The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 1
Nodes: 1
CPU count: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 9)
  ('CSPProcessCount', 4)
-
  ('One2OneChannel', 4)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.018381s Main::ParseXML
0.000319s Main::OrganizeNetwork
0.000521s Main::SetupCSPNetwork
1515.319419s Costum::Calculation done



The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 8
Nodes: 8
CPU count: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19911 ./node_execute.py localhost 19911
screen -L -dmS nodeport19912 ./node_execute.py localhost 19912
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19914 ./node_execute.py localhost 19914
screen -L -dmS nodeport19915 ./node_execute.py localhost 19915
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 23)
  ('CSPProcessCount', [('localhost', -1, 3), ('localhost', 19910, 2),
('localhost', 19911, 2), ('localhost', 19912, 1), ('localhost', 19913, 1),
('localhost', 19914, 1), ('localhost', 19915, 1), ('localhost', 19916, 1),
('ALL', 12)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 17)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 9)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.028040s Main::ParseXML
0.005765s Main::OrganizeNetwork
0.119569s Main::SetupCSPNetwork
807.183675s Costum::Calculation done


The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 7
Nodes: 8
CPU count: 8
```

```
Job_size: 10^6

@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19911 ./node_execute.py localhost 19911
screen -L -dmS nodeport19912 ./node_execute.py localhost 19912
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19914 ./node_execute.py localhost 19914
screen -L -dmS nodeport19915 ./node_execute.py localhost 19915
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 21)
  ('CSPProcessCount', [('localhost', -1, 2), ('localhost', 19910, 2),
('localhost', 19911, 2), ('localhost', 19912, 1), ('localhost', 19913, 1),
('localhost', 19914, 1), ('localhost', 19915, 1), ('localhost', 19916, 1),
('ALL', 11)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 15)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 8)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.022481s Main::ParseXML
0.004948s Main::OrganizeNetwork
0.090371s Main::SetupCSPNetwork
583.913124s Costum::Calculation done


The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 6
Nodes: 8
CPU count: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19911 ./node_execute.py localhost 19911
screen -L -dmS nodeport19912 ./node_execute.py localhost 19912
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19914 ./node_execute.py localhost 19914
screen -L -dmS nodeport19915 ./node_execute.py localhost 19915
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 19)
  ('CSPProcessCount', [('localhost', -1, 2), ('localhost', 19910, 2),
('localhost', 19911, 1), ('localhost', 19912, 1), ('localhost', 19913, 1),
('localhost', 19914, 1), ('localhost', 19915, 1), ('localhost', 19916, 1),
('ALL', 10)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 13)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 7)

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.054718s Main::ParseXML
0.004141s Main::OrganizeNetwork
0.076882s Main::SetupCSPNetwork
864.889510s Costum::Calculation done

['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']


The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 4
Nodes: 8
CPU count: 8
Job_size: 10^6
```

```
@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19911 ./node_execute.py localhost 19911
screen -L -dmS nodeport19912 ./node_execute.py localhost 19912
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19914 ./node_execute.py localhost 19914
screen -L -dmS nodeport19915 ./node_execute.py localhost 19915
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 15)
  ('CSPProcessCount', [('localhost', -1, 2), ('localhost', 19910, 1),
('localhost', 19911, 1), ('localhost', 19912, 1), ('localhost', 19913, 1),
('localhost', 19914, 1), ('localhost', 19915, 1), ('localhost', 19916, 0),
('ALL', 8)])
-
  ('One2OneChannel', 0)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 11)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 7)

['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.020691s Main::ParseXML
0.003461s Main::OrganizeNetwork
0.033654s Main::SetupCSPNetwork
391.797900s Costum::Calculation done


The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 2
Nodes: 8
CPU count: 8
Job_size: 10^6

@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19911 ./node_execute.py localhost 19911
screen -L -dmS nodeport19912 ./node_execute.py localhost 19912
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19914 ./node_execute.py localhost 19914
screen -L -dmS nodeport19915 ./node_execute.py localhost 19915
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 11)
  ('CSPProcessCount', [('localhost', -1, 2), ('localhost', 19910, 2),
('localhost', 19911, 2), ('localhost', 19912, 0), ('localhost', 19913, 0),
('localhost', 19914, 0), ('localhost', 19915, 0), ('localhost', 19916, 0),
('ALL', 6)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 1)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 5)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 3)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.020007s Main::ParseXML
0.002885s Main::OrganizeNetwork
0.042765s Main::SetupCSPNetwork
843.733746s Costum::Calculation done



The Number: (2^222) - 1
Release of GLOBAL INTERPRETER LOCK REMOVED!!
Workers: 1
Nodes: 8
CPU count: 8
Job_size: 10^6
```

```
@amigos50:~/CSPBuilder> ./start_dist_execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
screen -L -dmS nodeport19910 ./node_execute.py localhost 19910
screen -L -dmS nodeport19911 ./node_execute.py localhost 19911
screen -L -dmS nodeport19912 ./node_execute.py localhost 19912
screen -L -dmS nodeport19913 ./node_execute.py localhost 19913
screen -L -dmS nodeport19914 ./node_execute.py localhost 19914
screen -L -dmS nodeport19915 ./node_execute.py localhost 19915
screen -L -dmS nodeport19916 ./node_execute.py localhost 19916
./execute.py -n -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp

-
  ('ProcessCount', 9)
  ('CSPProcessCount', [('localhost', -1, 2), ('localhost', 19910, 2),
('localhost', 19911, 0), ('localhost', 19912, 0), ('localhost', 19913, 0),
('localhost', 19914, 0), ('localhost', 19915, 0), ('localhost', 19916, 0),
('ALL', 4)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 3)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 3)
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.018223s Main::ParseXML
0.002115s Main::OrganizeNetwork
0.027433s Main::SetupCSPNetwork
1505.554158s Costum::Calculation done
```

D.7 Prime Factorization - Jobsizes

```
The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 15000 (job count = 41137)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 41137
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.026565s Main::ParseXML
0.000212s Main::OrganizeNetwork
0.001757s Main::SetupCSPNetwork
453.875342s Costum::Calculation done



The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 20000 (job count = 30852)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
```

```
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 30852
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.026916s Main::ParseXML
0.000340s Main::OrganizeNetwork
0.001790s Main::SetupCSPNetwork
356.607609s Costum::Calculation done


The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 27000 (job count = 22859)
@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 22859
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.025906s Main::ParseXML
0.000204s Main::OrganizeNetwork
0.001652s Main::SetupCSPNetwork
339.423906s Costum::Calculation done


The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 32000 (job count = 19302)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 19302
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']
0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.024750s Main::ParseXML
0.000192s Main::OrganizeNetwork
0.001589s Main::SetupCSPNetwork
324.035781s Costum::Calculation done



The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 35000 (job count = 17655)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
```

```
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 17655
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.025808s Main::ParseXML
0.000212s Main::OrganizeNetwork
0.001660s Main::SetupCSPNetwork
316.370706s Costum::Calculation done

The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 38000 (job count = 16256)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 16256
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']
0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.023133s Main::ParseXML
0.000300s Main::OrganizeNetwork
0.001701s Main::SetupCSPNetwork
282.545650s Costum::Calculation done


The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 42000 (job count = 14715)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 14715
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']
0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.024923s Main::ParseXML
0.000301s Main::OrganizeNetwork
0.001644s Main::SetupCSPNetwork
256.862142s Costum::Calculation done



The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 50000 (job count = 12363)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
```

```
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 12363
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.025159s Main::ParseXML
0.000324s Main::OrganizeNetwork
0.001630s Main::SetupCSPNetwork
247.071531s Costum::Calculation done


The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 75000 (job count = 8268)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 8268
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.025047s Main::ParseXML
0.000200s Main::OrganizeNetwork
0.001610s Main::SetupCSPNetwork
239.466595s Costum::Calculation done


The Number: (2^222) - 1
Workers: 8
CPU count: 8
Job_size: 10^5 (job count = 6198)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 6198
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.027178s Main::ParseXML
0.000448s Main::OrganizeNetwork
0.001809s Main::SetupCSPNetwork
238.843195s Costum::Calculation done


The Number: (2^222) - 1
Workers: 8
CPU count: 7
Job_size: 10^6 (job count = 650)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
```

```
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 650
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.024729s Main::ParseXML
0.000189s Main::OrganizeNetwork
0.001587s Main::SetupCSPNetwork
254.686638s Costum::Calculation done


The Number: (2^222) - 1
Workers: 8
CPU count: 7
Job_size: 10^7 (job count = 86)

@amigos50:~/CSPBuilder> ./execute.py -i Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
-
  ('ProcessCount', 23)
  ('CSPProcessCount', 12)
-
  ('One2OneChannel', 11)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)
Number of jobs done: 86
['3**2', '7', '223', '1777', '3331', '17539', '321679', '25781083',
'26295457', '319020217', '616318177', '107775231312019']

0.000000s Main::Init Applications/PrimeFac/PrimeFacCBigNum_DoBigNumber.csp
0.024887s Main::ParseXML
0.000196s Main::OrganizeNetwork
0.001608s Main::SetupCSPNetwork
366.605933s Costum::Calculation done
```

D.8 Successive Over-Relaxation - 1 to 16 workers

64x64

```
Workers: 16
Image: 64x64
Iterations: 7298

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/16-Process-Parallel-SOR.csp

-
  ('ProcessCount', 51)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 2),
('amigos24', 9980, 2), ('c0-1', 9981, 2), ('c0-2', 9982, 2),
('c0-3', 9983, 2), ('c0-4', 9984, 2), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 1),
('c0-9', 9989, 1), ('c0-10', 9990, 1), ('c0-11', 9991, 1),
('c0-12', 9992, 1), ('c0-13', 9993, 1), ('ALL', 24)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 66)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 66)
iterations:7298 7
iterations:7298 6
iterations:7298 5
iterations:7298 4
iterations:7298 3
iterations:7298 2
iterations:7298 0
iterations:7298 1
iterations:7298 15
iterations:7298 14
```

```
iterations:7298 13
iterations:7298 12
iterations:7298 11
iterations:7298 10
iterations:7298 8
iterations:7298 9

0.000000s Main::Init Applications/SOR/16-Process-Parallel-SOR.csp
0.929978s Main::ParseXML
0.166721s Main::OrganizeNetwork
0.054429s Main::SetupCSPNetwork
67.544931s Costum::Calculation done


Workers: 8
Image: 64x64
Iterations: 7166


[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/8-Process-Parallel-SOR.csp

-
  ('ProcessCount', 34)
  ('CSPProcessCount', [('amigos24', -1, 2), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 1),
('c0-9', 9989, 1), ('c0-10', 9990, 1), ('c0-11', 9991, 1),
('c0-12', 9992, 1), ('c0-13', 9993, 0), ('ALL', 16)])
-
  ('One2OneChannel', 0)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 36)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 36)
iterations:7166 0
iterations:7166 2
iterations:7166 3
iterations:7166 4
iterations:7166 5
iterations:7166 6
iterations:7166 7
iterations:7166 1

0.000000s Main::Init Applications/SOR/8-Process-Parallel-SOR.csp
0.829195s Main::ParseXML
0.085504s Main::OrganizeNetwork
0.032963s Main::SetupCSPNetwork
80.162837s Costum::Calculation done


Workers: 4
Image: 64x64
Iterations: 7018

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp
-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
iterations:7018 3
iterations:7018 2
iterations:7018 0
iterations:7018 1

0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.796540s Main::ParseXML
0.037497s Main::OrganizeNetwork
0.038311s Main::SetupCSPNetwork
147.426358s Costum::Calculation done
```

```
Workers: 2
Image: 64x64
Iterations: 6578

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/2-Process-Parallel-SOR.csp

-
  ('ProcessCount', 21)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 2),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 0),
('c0-6', 9986, 0), ('c0-7', 9987, 0), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 10)])
-
  ('One2OneChannel', 3)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 9)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 9)
iterations:6578 0
iterations:6578 1
0.000000s Main::Init Applications/SOR/2-Process-Parallel-SOR.csp
0.760335s Main::ParseXML
0.027610s Main::OrganizeNetwork
0.034296s Main::SetupCSPNetwork
225.848393s Costum::Calculation done


Workers: 1
Image: 64x64
Iterations: 6444

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/1-Process-SOR.csp

-
  ('ProcessCount', 15)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 0), ('c0-4', 9984, 0), ('c0-5', 9985, 0),
('c0-6', 9986, 0), ('c0-7', 9987, 0), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 7)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 4)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 4)
iterations:6444 0
0.000000s Main::Init Applications/SOR/1-Process-SOR.csp
0.753159s Main::ParseXML
0.019574s Main::OrganizeNetwork
0.021402s Main::SetupCSPNetwork
253.581841s Costum::Calculation done
```

64x128

```
Workers: 16
Image: 64x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/16-Process-Parallel-SOR.csp
-
  ('ProcessCount', 51)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 2),
('amigos24', 9980, 2), ('c0-1', 9981, 2), ('c0-2', 9982, 2),
('c0-3', 9983, 2), ('c0-4', 9984, 2), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 1),
('c0-9', 9989, 1), ('c0-10', 9990, 1), ('c0-11', 9991, 1),
('c0-12', 9992, 1), ('c0-13', 9993, 1), ('ALL', 24)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 66)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
```

```
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 66)
iterations:3072 6
iterations:3072 5
iterations:3072 4
iterations:3072 3
iterations:3072 2
iterations:3072 0
iterations:3072 1
iterations:3072 15
iterations:3072 14
iterations:3072 13
iterations:3072 12
iterations:3072 11
iterations:3072 10
iterations:3072 8
iterations:3072 9
iterations:3072 7
0.000000s Main::Init Applications/SOR/16-Process-Parallel-SOR.csp
0.927206s Main::ParseXML
0.154453s Main::OrganizeNetwork
0.020654s Main::SetupCSPNetwork
45.836219s Costum::Calculation done


Workers: 8
Image: 64x128


[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/8-Process-Parallel-SOR.csp
-
  ('ProcessCount', 34)
  ('CSPProcessCount', [('amigos24', -1, 2), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 1),
('c0-9', 9989, 1), ('c0-10', 9990, 1), ('c0-11', 9991, 1),
('c0-12', 9992, 1), ('c0-13', 9993, 0), ('ALL', 16)])
-
  ('One2OneChannel', 0)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 36)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 36)
iterations:3020 1
iterations:3020 0
iterations:3020 2
iterations:3020 3
iterations:3020 4
iterations:3020 5
iterations:3020 6
iterations:3020 7
0.000000s Main::Init Applications/SOR/8-Process-Parallel-SOR.csp
0.824309s Main::ParseXML
0.076019s Main::OrganizeNetwork
0.027668s Main::SetupCSPNetwork
67.411385s Costum::Calculation done


Workers: 4
Image: 64x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp
-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
iterations:2970 3
```

```
iterations:2970 2
iterations:2970 0
iterations:2970 1
0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.797266s Main::ParseXML
0.027980s Main::OrganizeNetwork
0.019430s Main::SetupCSPNetwork
134.149324s Costum::Calculation done


Workers: 2
Image: 64x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/2-Process-Parallel-SOR.csp
-
  ('ProcessCount', 21)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 2),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 0),
('c0-6', 9986, 0), ('c0-7', 9987, 0), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 10)])
-
  ('One2OneChannel', 3)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 9)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 9)
iterations:2702 0
iterations:2702 1
0.000000s Main::Init Applications/SOR/2-Process-Parallel-SOR.csp
0.764739s Main::ParseXML
0.023388s Main::OrganizeNetwork
0.040463s Main::SetupCSPNetwork
266.383484s Costum::Calculation done


Workers: 1
Image: 64x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/1-Process-SOR.csp

-
  ('ProcessCount', 15)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 0), ('c0-4', 9984, 0), ('c0-5', 9985, 0),
('c0-6', 9986, 0), ('c0-7', 9987, 0), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 7)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 4)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 4)
iterations:2590 0
0.000000s Main::Init Applications/SOR/1-Process-SOR.csp
0.744532s Main::ParseXML
0.021536s Main::OrganizeNetwork
0.019938s Main::SetupCSPNetwork
745.908109s Costum::Calculation done
```

128x64

```
Workers: 16
Image: 128x64

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/16-Process-Parallel-SOR.csp

-
  ('ProcessCount', 51)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 2),
('amigos24', 9980, 2), ('c0-1', 9981, 2), ('c0-2', 9982, 2),
('c0-3', 9983, 2), ('c0-4', 9984, 2), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 1),
('c0-9', 9989, 1), ('c0-10', 9990, 1), ('c0-11', 9991, 1),
('c0-12', 9992, 1), ('c0-13', 9993, 1), ('ALL', 24)])
-
```

```
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 66)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 66)
iterations:3808 7
iterations:3808 6
iterations:3808 5
iterations:3808 4
iterations:3808 3
iterations:3808 2
iterations:3808 0
iterations:3808 1
iterations:3808 15
iterations:3808 14
iterations:3808 13
iterations:3808 12
iterations:3808 11
iterations:3808 10
iterations:3808 8
iterations:3808 9
0.000000s Main::Init Applications/SOR/16-Process-Parallel-SOR.csp
0.931830s Main::ParseXML
0.115277s Main::OrganizeNetwork
0.019214s Main::SetupCSPNetwork
57.528427s Costum::Calculation done


Workers: 8
Image: 128x64


[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/8-Process-Parallel-SOR.csp
-
  ('ProcessCount', 34)
  ('CSPProcessCount', [('amigos24', -1, 2), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 1),
('c0-9', 9989, 1), ('c0-10', 9990, 1), ('c0-11', 9991, 1),
('c0-12', 9992, 1), ('c0-13', 9993, 0), ('ALL', 16)])
-
  ('One2OneChannel', 0)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 36)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 36)
iterations:3720 0
iterations:3720 2
iterations:3720 3
iterations:3720 4
iterations:3720 5
iterations:3720 6
iterations:3720 7
iterations:3720 1
0.000000s Main::Init Applications/SOR/8-Process-Parallel-SOR.csp
0.824224s Main::ParseXML
0.038590s Main::OrganizeNetwork
0.019422s Main::SetupCSPNetwork
81.289447s Costum::Calculation done


Workers: 4
Image: 128x64

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp

-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
```

```
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
iterations:3654 3
iterations:3654 2
iterations:3654 0
iterations:3654 1
0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.790646s Main::ParseXML
0.025399s Main::OrganizeNetwork
0.016555s Main::SetupCSPNetwork
163.519650s Costum::Calculation done


Workers: 2
Image: 128x64

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/2-Process-Parallel-SOR.csp
  ('ProcessCount', 21)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 2),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 0),
('c0-6', 9986, 0), ('c0-7', 9987, 0), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 10)])
-
  ('One2OneChannel', 3)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 9)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 9)
iterations:3374 0
iterations:3374 1

0.000000s Main::Init Applications/SOR/2-Process-Parallel-SOR.csp
0.758496s Main::ParseXML
0.026208s Main::OrganizeNetwork
0.036285s Main::SetupCSPNetwork
342.463881s Costum::Calculation done


Workers: 1
Image: 128x64

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/1-Process-SOR.csp
-
  ('ProcessCount', 15)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 0), ('c0-4', 9984, 0), ('c0-5', 9985, 0),
('c0-6', 9986, 0), ('c0-7', 9987, 0), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 7)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 4)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 4)
iterations:3266 0
0.000000s Main::Init Applications/SOR/1-Process-SOR.csp
0.747223s Main::ParseXML
0.022035s Main::OrganizeNetwork
0.029395s Main::SetupCSPNetwork
986.936965s Costum::Calculation done
```

128x128

```
Workers: 16
Image: 128x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/16-Process-Parallel-SOR.csp
-
  ('ProcessCount', 51)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 2),
('amigos24', 9980, 2), ('c0-1', 9981, 2), ('c0-2', 9982, 2),
```

```
('c0-3', 9983, 2), ('c0-4', 9984, 2), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 1),
('c0-9', 9989, 1), ('c0-10', 9990, 1), ('c0-11', 9991, 1),
('c0-12', 9992, 1), ('c0-13', 9993, 1), ('ALL', 24)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 66)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 66)
iterations:4214 6
iterations:4214 5
iterations:4214 4
iterations:4214 3
iterations:4214 2
iterations:4214 0
iterations:4214 1
iterations:4214 14
iterations:4214 15
iterations:4214 13
iterations:4214 12
iterations:4214 10
iterations:4214 11
iterations:4214 8
iterations:4214 9
iterations:4214 7
0.000000s Main::Init Applications/SOR/16-Process-Parallel-SOR.csp
0.925097s Main::ParseXML
0.160199s Main::OrganizeNetwork
0.019313s Main::SetupCSPNetwork
116.447961s Costum::Calculation done


Workers: 8
Image: 128x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/8-Process-Parallel-SOR.csp
-
  ('ProcessCount', 34)
  ('CSPProcessCount', [('amigos24', -1, 2), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 1),
('c0-9', 9989, 1), ('c0-10', 9990, 1), ('c0-11', 9991, 1),
('c0-12', 9992, 1), ('c0-13', 9993, 0), ('ALL', 16)])
-
  ('One2OneChannel', 0)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 36)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 36)
iterations:4160 0
iterations:4160 2
iterations:4160 3
iterations:4160 4
iterations:4160 5
iterations:4160 6
iterations:4160 7
iterations:4160 1
0.000000s Main::Init Applications/SOR/8-Process-Parallel-SOR.csp
0.826696s Main::ParseXML
0.042550s Main::OrganizeNetwork
0.018719s Main::SetupCSPNetwork
211.230812s Costum::Calculation done


Workers: 4
Image: 128x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp
-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
```

```
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
iterations:4002 3
iterations:4002 2
iterations:4002 0
iterations:4002 1

0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.797216s Main::ParseXML
0.062491s Main::OrganizeNetwork
0.040434s Main::SetupCSPNetwork
500.356726s Costum::Calculation done


Workers: 2
Image: 128x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/2-Process-Parallel-SOR.csp
-
  ('ProcessCount', 21)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 2),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 0),
('c0-6', 9986, 0), ('c0-7', 9987, 0), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 10)])
-
  ('One2OneChannel', 3)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 9)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 9)
iterations:3818 0
iterations:3818 1
0.000000s Main::Init Applications/SOR/2-Process-Parallel-SOR.csp
0.758250s Main::ParseXML
0.020338s Main::OrganizeNetwork
0.035390s Main::SetupCSPNetwork
1196.103896s Costum::Calculation done

Workers: 1
Image: 128x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/1-Process-SOR.csp

-
  ('ProcessCount', 15)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 0), ('c0-4', 9984, 0), ('c0-5', 9985, 0),
('c0-6', 9986, 0), ('c0-7', 9987, 0), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 7)])
-
  ('One2OneChannel', 2)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 4)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 4)
iterations:3796 0
0.000000s Main::Init Applications/SOR/1-Process-SOR.csp
0.744242s Main::ParseXML
0.018833s Main::OrganizeNetwork
0.014721s Main::SetupCSPNetwork
3916.994337s Costum::Calculation done
```

Serial Tests

```
Image: 64x64 2 crosses and squares
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_2crossesC.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 1088
```

```
Solved in 107.91519618 seconds

Image: 128x128 8 crosses and squares
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_8crossesBig.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 1898
Solved in 2076.67063498 seconds

Image: 128x64 4 crosses and squares
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_2crossesCwide.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 1633
Solved in 526.16346097 seconds

Image: 64x128 4 crosses and squares
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_2crossesChigh.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 1291
Solved in 399.206455946 seconds
```

D.9 Successive Over-Relaxation - Communication Test

Parallel Tests

```
Image: 16x256

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp
-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
iterations:1110 3
iterations:1110 2
iterations:1110 0
iterations:1110 1
0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.800603s Main::ParseXML
0.039797s Main::OrganizeNetwork
0.059569s Main::SetupCSPNetwork
17.578172s Costum::Calculation done


Image: 32x128

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp
-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
iterations:3738 3
iterations:3738 2
iterations:3738 0
iterations:3738 1
0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.792177s Main::ParseXML
0.087147s Main::OrganizeNetwork
0.028113s Main::SetupCSPNetwork
64.884599s Costum::Calculation done
```

```
Image: 64x64

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp
-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
iterations:7018 3
iterations:7018 2
iterations:7018 0
iterations:7018 1
0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.796258s Main::ParseXML
0.082067s Main::OrganizeNetwork
0.022554s Main::SetupCSPNetwork
149.460411s Costum::Calculation done


Image: 128x32

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp
-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
iterations:3628 3
iterations:3628 2
iterations:3628 0
iterations:3628 1
0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.793366s Main::ParseXML
0.056297s Main::OrganizeNetwork
0.017790s Main::SetupCSPNetwork
107.040007s Costum::Calculation done


Image: 256x16

[@amigos24 CSPBuilder]> ./start_dist_execute.py -i Applications/SOR/4-Process-Parallel-SOR.csp
-
  ('ProcessCount', 26)
  ('CSPProcessCount', [('amigos24', -1, 3), ('amigos24', 9979, 1),
('amigos24', 9980, 1), ('c0-1', 9981, 1), ('c0-2', 9982, 1),
('c0-3', 9983, 1), ('c0-4', 9984, 1), ('c0-5', 9985, 1),
('c0-6', 9986, 1), ('c0-7', 9987, 1), ('c0-8', 9988, 0),
('c0-9', 9989, 0), ('c0-10', 9990, 0), ('c0-11', 9991, 0),
('c0-12', 9992, 0), ('c0-13', 9993, 0), ('ALL', 12)])
-
  ('One2OneChannel', 1)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 0)
  ('One2NetChannel', 19)
  ('One2AnyChannel', 0)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 19)
```

```
iterations:1218 3
iterations:1218 2
iterations:1218 0
iterations:1218 1

0.000000s Main::Init Applications/SOR/4-Process-Parallel-SOR.csp
0.796923s Main::ParseXML
0.036528s Main::OrganizeNetwork
0.049670s Main::SetupCSPNetwork
49.664563s Costum::Calculation done
```

Serial Tests

```
Image: 256x16 hotbox
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_hotbox_256x16.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 492
Solved in 21.73756814 seconds


Image: 128x32 hotbox
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_hotbox_128x32.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 1590
Solved in 76.3857741356 seconds


Image: 64x64 hotbox
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_hotbox.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 3216
Solved in 155.916387081 seconds


Image: 32x128 hotbox
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_hotbox_32x128.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 1762
Solved in 84.5692579746 seconds


Image: 16x256 hotbox
[@amigos24 CSPBuilder]> cp Applications/SOR/SOR_test_hotbox_16x256.png Applications/SOR/SOR_test.png
[@amigos24 CSPBuilder]> python2.5 SOR-serial.py
Iterations: 541
Solved in 25.08177495 seconds
```

D.10 Testing CSPBuilder

```
(PrimeFacController+PrimeFacWorkerx32)x512+5processes = 16901

@amigos50:~/CSPBuilder> ./execute.py -i Tests/Benchmark-PrimeFac.csp
-
  ('ProcessCount', 34315)
  ('CSPProcessCount', 16901)
-
  ('One2OneChannel', 4)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 513)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 513)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)

[Actual prime factorization killed]

0.000000s Main::Init Tests/Benchmark-PrimeFac.csp
16.349173s Main::ParseXML
0.000371s Main::OrganizeNetwork
887.654045s Main::SetupCSPNetwork

(PrimeFacController+PrimeFacWorkerx24)x512+5processes = 12805

-
  ('ProcessCount', 26123)
  ('CSPProcessCount', 12805)
-
  ('One2OneChannel', 4)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 513)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 513)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
```

```
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)

[Actual prime factorization killed]

0.000000s Main::Init Tests/Benchmark-PrimeFac.csp
12.256621s Main::ParseXML
0.000348s Main::OrganizeNetwork
533.414399s Main::SetupCSPNetwork


(PrimeFacController+PrimeFacWorkerx16)x512+5processes = 8709

-
  ('ProcessCount', 17931)
  ('CSPProcessCount', 8709)
-
  ('One2OneChannel', 4)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 513)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 513)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)

[Actual prime factorization killed]

0.000000s Main::Init Tests/Benchmark-PrimeFac.csp
9.427083s Main::ParseXML
0.000361s Main::OrganizeNetwork
311.687921s Main::SetupCSPNetwork


(PrimeFacController+PrimeFacWorkerx8)x512+5processes = 4613

-
  ('ProcessCount', 9739)
  ('CSPProcessCount', 4613)
-
  ('One2OneChannel', 4)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 513)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 513)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)

[Actual prime factorization killed]

0.000000s Main::Init Tests/Benchmark-PrimeFac.csp
5.816334s Main::ParseXML
0.000343s Main::OrganizeNetwork
96.154220s Main::SetupCSPNetwork



(PrimeFacController+PrimeFacWorkerx4)x512+5processes = 2565


-
  ('ProcessCount', 5643)
  ('CSPProcessCount', 2565)
-
  ('One2OneChannel', 4)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 513)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 513)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)

[Actual prime factorization killed]

0.000000s Main::Init Tests/Benchmark-PrimeFac.csp
4.489342s Main::ParseXML
0.000354s Main::OrganizeNetwork
40.261509s Main::SetupCSPNetwork


(PrimeFacController+PrimeFacWorkerx2)x512+5processes = 1541
```

```
-
  ('ProcessCount', 3595)
  ('CSPProcessCount', 1541)
-
  ('One2OneChannel', 4)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 513)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 513)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)

[Actual prime factorization killed]

0.000000s Main::Init Tests/Benchmark-PrimeFac.csp
3.384199s Main::ParseXML
0.000347s Main::OrganizeNetwork
13.560546s Main::SetupCSPNetwork


(PrimeFacController+PrimeFacWorkerx1)x512+5processes = 1029

-
  ('ProcessCount', 2571)
  ('CSPProcessCount', 1029)
-
  ('One2OneChannel', 1028)
  ('AnyNet2OneChannel', 0)
  ('Any2OneChannel', 1)
  ('One2NetChannel', 0)
  ('One2AnyChannel', 1)
  ('One2AllNetChannel', 0)
  ('One2AllChannel', 0)
  ('Any2AnyChannel', 0)
  ('One2AnyNetChannel', 0)
  ('Net2OneChannel', 0)

[Actual prime factorization killed]

0.000000s Main::Init Tests/Benchmark-PrimeFac.csp
3.008836s Main::ParseXML
0.000350s Main::OrganizeNetwork
5.285216s Main::SetupCSPNetwork
```