CSP FOR EXECUTABLE SCIENTIFIC WORKFLOWS

Ph.D. Dissertation

Rune Møllegaard Friborg

eScience Center, Department of Computer Science
The PhD School of Science
University of Copenhagen
Copenhagen, Denmark

Submission date: August 31th, 2011

Acknowledgements

I would like to thank all the people who have been there during my PhD and helped to make these past 3 years as interesting and fun as they have been. I especially wish to thank my advisor Brian Vinter who, in addition to always being able to come up with constructive ideas and believing in me all the way, has been a great colleague.

I wish to thank John Markus Bjørndalen for hosting my stay at the University of Tromsø, Norway. It was six pleasant months with interesting ideas and research. During this stay I had the joy of sharing an office with Lars Tiede, whom I would also like to thank for many interesting discussions related to our research.

Thanks go to my close colleagues for the cosy and heartwarming environment that we had in the corner office (the small one).

Also, to the many students who have worked on PyCSP and CSPBuilder: thank you for giving me the opportunity to discuss research topics closely related to my thesis.

Thanks to Christian Storm Pedersen for providing me with an office space in Aarhus, Denmark during the final period of the PhD.

And finally, my dear wife Marie and my two kids Therese and Bjørn, who have been patient with me when I have been busy working on the PhD. I know that it has not been easy, especially the finishing phase.

The presented research is supported by NABIIT grant 2106-06-0059 from The Danish Council for Strategic Research.

Abstract

This thesis presents CSP as a means of orchestrating the execution of tasks in a scientific workflow. Scientific workflow systems are popular in a wide range of scientific areas, where tasks are organised in directed graphs. Execution of such graphs is handled by the scientific workflow systems and can usually benefit performance-wise from both multiprocessing, cluster and grid environments.

PyCSP is an implementation of Communicating Sequential Processes (CSP) for the Python programming language and takes advantage of CSP's formal and verifiable approach to controlling concurrency and the readability of Python source code. Python is a popular programming language in the scientific community, with many scientific libraries (modules) and simple integration to external languages. This thesis presents a PyCSP extended with many new features and a more robust implementation to allow scientific applications to run on heterogenous hardware, combining multiple hardware architectures. This is especially important in scientific computing as the performance of computational tasks may be orders of magnitude faster depending on the hardware architecture used.

To ensure the robustness of the PyCSP library the internal synchronisation model has been model-checked successfully using the SPIN Model Checker. This has checked the synchronisation model for the presence of deadlocks, livelocks, starvation, race conditions and correct channel communication behaviour.

The use of PyCSP for scientific workflows is demonstrated through examples. By providing a robust library for organising scientific workflows in a Python application I hope to inspire scientific users to adopt PyCSP. As a proof-of-concept this thesis demonstrates three scientific applications: kNN, stochastic minimum search and McStas to scale well on multi-processing and cluster computing using PyCSP. Additionally, McStas is demonstrated to utilise grid computing resources using PyCSP.

Finally, this thesis presents a new dynamic channel model, which has not yet been implemented for PyCSP. The dynamic channel is able to change the internal synchronisation mechanisms on-the-fly, depending on the location and number of channel-ends connected. Thus it may start out as a simple local pipe and evolve into a distributed channel spanning multiple nodes. This channel is a necessary next step for PyCSP to allow for complete freedom in executing CSP processes on local and remote resources.

Contents

Chapter 1

Introduction

Computers are providing an enormous potential for scientific advancement in almost all scientific fields. Computational science is a tool for scientists, and as a tool one of its purposes is to assist in reducing the time from idea to scientific discovery. Another purpose is to allow virtual experiments of natural phenomenons, that are impossible or impractical to create on demand in nature.

The driving motivation for the presented work is to give scientists that are not Computer Science majors the possibility to build correct, efficient, modular, reusable and scalable applications. It is a well-known challenge to maintain scientific code; many applications are written by scientists without any formal computer science or software engineering qualifications and have often grown organically from a small kernel to hundreds of thousands of code-lines. Such applications have traditionally targeted simple single-core systems and have still grown to a complexity where the cost of maintaining the codes is prohibiting, and where the continued correctness of the code is often questionable. This problem is being addressed today by training scientists in design patterns and good practice in program development. However, emerging architectures, which are massively parallel and often heterogeneous, may again raise the complexity of software development to a level where scientific users (non-computer scientists) are no longer able to produce reliable scientific software.

In this thesis, the term scientific users is used quite loosely to mean programmers that are scientists, but not computer scientists. Scientific users of computing is in itself a diverse group, one extreme is scientists that use existing applications, commercial or community codes, where they change configurations and input data, but in general do not change the code itself. The other extreme is scientists that primarily do program development, often the persons behind large community codes. The approach I promote here targets the set in between the two extremes, scientists that express the model they work on directly as a computer program, but where the programming is still a means to an end, and not the primary focus of the research. This kind of computational-scientist typically changes the code frequently, and the code is most often shared with a small number of co-researchers, typically within a research group. I target scientific users of computers that do their own programming and change their program frequently to match the development in their research. They are interested in better performance of their applications, but productivity and time to solution are the critical measures in their program development.

The classic approach to parallel programming involves threads, shared memory and locks, but this requires that the programmer identifies all critical regions and dependencies correctly without resulting in serialised executions, caused by large computational parts in critical regions. Scientific users usually avoid this kind of programming and use OpenMP [31] to add parallelisation. OpenMP is mostly used for loop parallelisation and requires identifying all critical regions within a loop. Another approach is to use parallelised libraries such as BLAS [26], but these are often not enough and are limited to shared memory systems. Scientific users must be encouraged to write maintainable and well-structured code. Several tools such as the Intel Parallel Studio [3] or Microsoft Parallel Computing [8] exist that aim to aid programmers in producing parallel programs, but these tools are not flexible enough for scientific applications and do not interface well with code produced during the past 30 years which is still in use. Most scientists have access to many different kinds of computing resources, thus the produced code must also be portable and must run on single-core, multi-core, cluster and grid systems. Graphical workflow systems as Knime [23, 88] and Taverna [49, 59] are helping scientists to structure code, but they lack sufficient support for parallel execution.

PyCSP, a CSP library for Python is the main contribution of this thesis. It has been designed to fit with the needs of the scientific users, which also means that hard choices have been made. PyCSP has been optimised towards transparency for the scientific user. This approach is to accommodate a more flexible and forgiving use of CSP than what is currently available in occam-pi, JCSP, C++CSP2, CHP, and others. Two of the hard choices were: 1. Choosing an any-to-any channel over explicit separation in one-to-one, any-to-one, one-to-any and any-to-any channels. 2. Python is dynamically typed and it was chosen to also have dynamically typed channels. The down-side of these choices is that errors that would have been caught at compile-time are not discovered until run-time or not at all.

PyCSP is optimised to allow computational scientists to be productive and create code that can grow organically – other CSP variants, i.e. JCSP, targets software engineering and thus requires the programmer to explicitly choose the nature of a channel. While PyCSP and JCSP appear (and are) very similar at first look, the target users for each CSP system have dictated the hard choices and at their core the CSP systems thus become quite different.

1.1 Contributions

The central contributions of this thesis is the work on PyCSP; a CSP library for Python, useful for experimentation, research and teaching in educational contexts. This thesis also presents a number of related contributions, where some make use of PyCSP. Two contributions that do not relate directly to PyCSP are:

> Implementation of an optimised and parallel version for shared memory architectures of a scientific Java application. The new implementation have made it possible to perform clustering for at least ten times more ChIP profiles, even though the algorithm has a time and space-complexity of $O(n^2)$. The results are presented in the paper ”CATCHprofiles: Clustering and Alignment Tool for ChIP profiles”.
>
> Implementation of a GPGPU version for finding likelihoods of stereo-based articulated tracking. It is now possible to achieve speedups of more than two orders of magnitude when running this application on a GPU compared to a single CPU core. The work is described in the paper ”GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking”.

The paper ”CSPBuilder - CSP based Scientific Workflow Modeling” was published in 2008 and based on work done in my Master’s Thesis. It presents CSPBuilder - a new framework that consists of a visual tool to build applications and a tool to execute the constructed applications. The framework is implemented in Python and incorporates extensive use of the CSP algebra. The primary advantages of this framework lie in code reuse and construction of complex scientific applications focusing on the workflow. CSP ideas underpin the concurrency mechanisms employed in constructed applications, enabling the automatic deconstruction of whole systems into individual concurrent components.

The original PyCSP which was closely related to JCSP, has been redone with a new feature set. The work is published in ”PyCSP Revisited” and ”Three Unique Implementations for Processes in PyCSP”. This was a necessary next step to provide students and scientific users with a flexible CSP library. The result is a much simpler PyCSP with only one channel type, delayed termination through retiring channels, support for output guards, and an external choice that is closer to that of occam than JCSP. The contributed PyCSP has introduced features that have not yet been seen in any other CSP library.

The new PyCSP has been demonstrated in ”PyCSP - Controlled Concurrency”. The use of CSP for executing scientific workflows on distributed systems is validated through experiments in ”Rapid development of scalable scientific software using a process oriented approach”. Tracing of PyCSP networks and a tool for visualising traces was also added to the PyCSP package, with the purpose of providing a better understanding for a constructed workflow.

The final contribution is the design and automatic verification of a dynamic channel model that can start out as a simple local pipe and evolve dynamically into a distributed channel spanning multiple nodes. This work was published in ”Verification of a Dynamic Channel Model using the SPIN Model Checker”. With the results from this paper I can also conclude that the synchronisation mechanism in the contributed PyCSP can be model-checked successfully by SPIN. The channel uses a two-phase locking approach with global ordering of locks, which is shown to work correctly for both a shared memory model and a distributed model.

1.2 Publications

List of published peer-reviewed papers:

Rune Møllegaard Friborg, Søren Hauberg, Kenny Erleben: GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking
Paper presented at Computer Vision GPU (CVGPU 2010), ECCV 2010 Workshop, Hersonissos, Heraklion, Crete, Greece, September 11, 2010

Rune Møllegaard Friborg, Brian Vinter: CSPBuilder - CSP based Scientific Workflow Modeling
Communicating Process Architectures 2008, WoTUG-31, Proceedings of the 31st WoTUG Technical Meeting, University of York, Yorkshire, UK, September 7-10, 2008
ISBN: 978-1-58603-907-3, Concurrent Systems Engineering 66, IOS Press, pp. 347 – 369

Brian Vinter, John Markus Bjørndalen, Rune Møllegaard Friborg: PyCSP Revisited
Communicating Process Architectures 2009, WoTUG-32, Proceedings of the 32nd WoTUG Technical Meeting, Technische Universiteit Eindhoven, The Netherlands, November 1-4, 2009
ISBN: 978-1-60750-065-0, Concurrent Systems Engineering 67, IOS Press, pp. 263 – 276

Rune Møllegaard Friborg, John Markus Bjørndalen, Brian Vinter: Three Unique Implementations of Processes for PyCSP
Communicating Process Architectures 2009, WoTUG-32, Proceedings of the 32nd WoTUG Technical Meeting, Technische Universiteit Eindhoven, The Netherlands, November 1-4, 2009
ISBN: 978-1-60750-065-0, Concurrent Systems Engineering 67, IOS Press, pp. 277 – 292

Rune Møllegaard Friborg, Brian Vinter, John Markus Bjørndalen: PyCSP - controlled concurrency
IJIPM: International Journal of Information Processing and Management, Vol. 1, No. 2, pp. 40 – 49, 2010
DOI: 10.4156/ijipm.vol1.issue2.6

Rune Møllegaard Friborg and Brian Vinter: Rapid Development of Scalable Scientific Software Using a Process Oriented Approach
JOCS: Journal of Computational Science, In Press, Corrected Proof
DOI: 10.1016/j.jocs.2011.02.001

Rune Møllegaard Friborg, Brian Vinter: Verification of a Dynamic Channel Model using the SPIN Model Checker
Communicating Process Architectures 2011, WoTUG-33, Proceedings of the 33rd WoTUG Technical Meeting, University of Limerick, Ireland, June 19-22, 2011
ISBN: 978-1-60750-773-4, Concurrent Systems Engineering 68, IOS Press, pp. 35 – 54

Paper in submission:

Fiona G. G. Nielsen, Kasper Galschiøt Markus, Rune Møllegaard Friborg, Lene Monrad Favrholdt, Hendrik G. Stunnenberg, Martijn Huynen: CATCHprofiles: Clustering and Alignment Tool for ChIP profiles
Submitted to PLoS One.

1.3 Organisation

This thesis is organised into chapters providing the background for the main work (Chapters 2, 3, 4 and 5), the presentation of the main work (Chapter 6), future work (Chapter 7) and finally conclusions (Chapter 8). The published papers listed in the previous section are included in appendix A.

Chapter 2

Scientific Workflows

The concept of workflows emerged during the 1980's in the business community and was used to describe a sequence of tasks and the dependencies between such tasks. The scientific community have since picked it up, as there was a need to organise tasks involved in applications within computational science. The computational tools for scientists involve complex data analysis, visualization steps, data aggregation and data handling in general. A common usage scenario in bio-science involves interfacing with online gene databases or performing sequence analysis through online services. For many scientists, such tasks also include the repetitive cycle of moving data between tasks.

Scientific workflows systems are systems that are optimized towards handling scientific workflows, enabling users to built and execute workflows. The need for such systems emerged in the mid 1990's [89]. During execution, all necessary data is automatically moved between tasks. Currently many different scientific workflow systems exist and as many different approaches to executing a scientific workflow. Curcin [32] looks into whether it is possible to design a scientific workflow system that may satisfy all needs. He investigates six difference scientific workflow systems: Discovery Net, Taverna, Triana, Kepler, Yawl and BPEL. His conclusion is that conformity between the systems is highly unlikely, due to the differences in requirements for each scientific domain. The overview of workflow system features in [36] shows a diversity in features and provides an overview to assist the scientific user in the selection of the right scientific workflow system for a project.

Scientific workflow systems provide non-computer scientists with the means of creating large computational-oriented applications, partially by the organisation of tasks and the dependencies between them. An executing scientific workflow is defined as a workflow instance. In workflow instances, often a central administration unit is handling the dependencies. Data flows through the workflow instance and is the basis of sub-problems and sub-solutions until eventually a result or several results are found. A scientific workflow models the data-flow of a scientific application.

2.1 Cases of Scientific Computing

For two scientific applications, I have assisted scientists by producing high performance implementations replacing parts of their original application. Scientific workflows for these examples are presented, to show how a scientific application can be mapped to a workflow, though the applications have not been composed in scientific workflow systems. The initial code for the first application was a complete implementation in the Java programming language, with no particular modularised design.

2.1.1 Clustering and Alignment Tool for ChIP profiles

The purpose of the first application, CATCHprofiles (**C**lustering and **A**lignment **T**ool for **CH**ip **profiles**), is to fully explore the spatial and combinatorial patterns in ChIP-profiling data and detect potentially meaningful patterns. In this process, the binding patterns must be aligned and clustered which is an algorithmically and computationally challenging task. Figure 2.1 shows an example of a matched alignment and the following merge by average. The clustering is a hierarchical composition of the profile patterns with the exhaustive alignment at each step. The work presented in this section has been published in [73] (Appendix A.2).

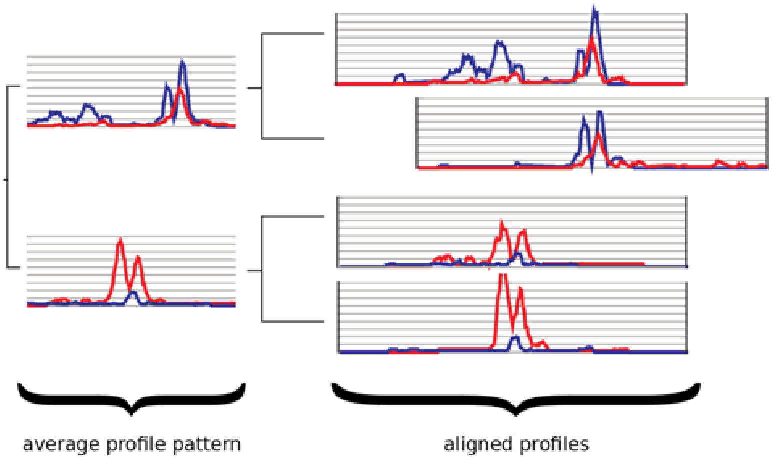
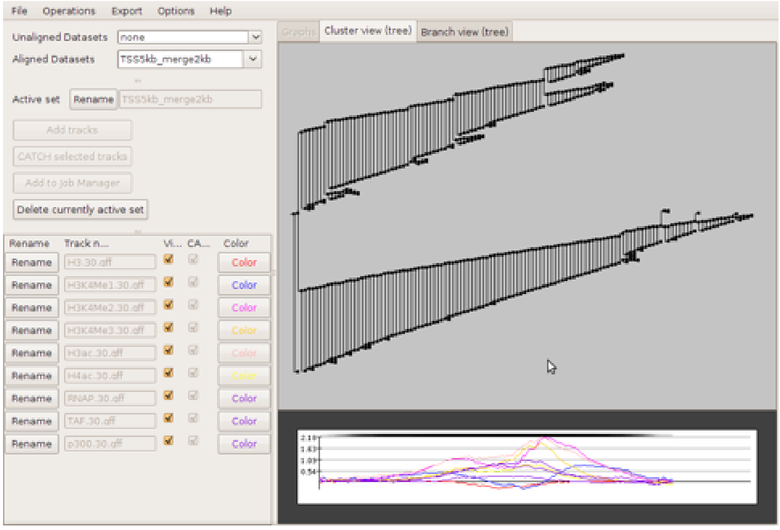


Figure 2.1: Conceptual illustration of the CATCH clustering algorithm, which shows the clustering of four profiles with two tracks of ChIP data, plotted in red and blue respectively. All pairs of profiles are aligned to find the alignment with the highest similarity.



Unaligned Datasets: none

Aligned Datasets: TSS5kb_merge2kb

Active set: TSS5kb_merge2kb

Rename	Track n...	Vi...	CA...	Color
Rename	H3.30.gff	☑	☑	Color
Rename	H3K4Me1.30.gff	☑	☑	Color
Rename	H3K4Me2.30.gff	☑	☑	Color
Rename	H3K4Me3.30.gff	☑	☑	Color
Rename	H3ac.30.gff	☑	☑	Color
Rename	H4ac.30.gff	☑	☑	Color
Rename	RNAP.30.gff	☑	☑	Color
Rename	TAF.30.gff	☑	☑	Color
Rename	p300.30.gff	☑	☑	Color

Figure 2.2: Screenshot of CATCHprofiles cluster view.

The application has a user-friendly graphical interface (Figure 2.2) for examination and browsing of the clustering results. CATCHprofiles requires no prior knowledge about functional sites, detects known binding patterns ”ab initio”, and enables the detection of new patterns from ChIP data at a high resolution. CATCHprofiles’ capability for exhaustive analysis combined with its ease-of-use makes it an invaluable tool for explorative research based on ChIP profiling data.

Upon activation of a clustering of N selected profiles from the CATCHprofiles Java application, the entire job description of data and user-selected parameters are compiled in JSON format and used as input to the high performance CATCH engine written in C. The complete clustering result is then loaded back into the Java application for visualisation.

The CATCHprofiles’ clustering algorithm has four main components: the initial comparison and similarity score computation for all profile pairs, the selection of the highest scoring profile pair, the merging of the selected pair into a representative profiles and the updating of the similarity score table. The CATCH engine written in C is implemented with care to ensure that there are no conflicts when accessing shared memory from subroutines. The OpenMP [33] directives are then used to implement a platform-independent threading for the pairwise score computations and the selection of the highest scoring profile pair. The computations are contained in loops and are simple to parallelise using OpenMP, as all shared memory conflicts have been removed from the subroutines. The OpenMP directives are translated to pthread calls by the compiler. The compiler creates the necessary code to handle creation and termination of threads, synchronisation between threads, shared and local variables and orchestration of the workload between the running threads.

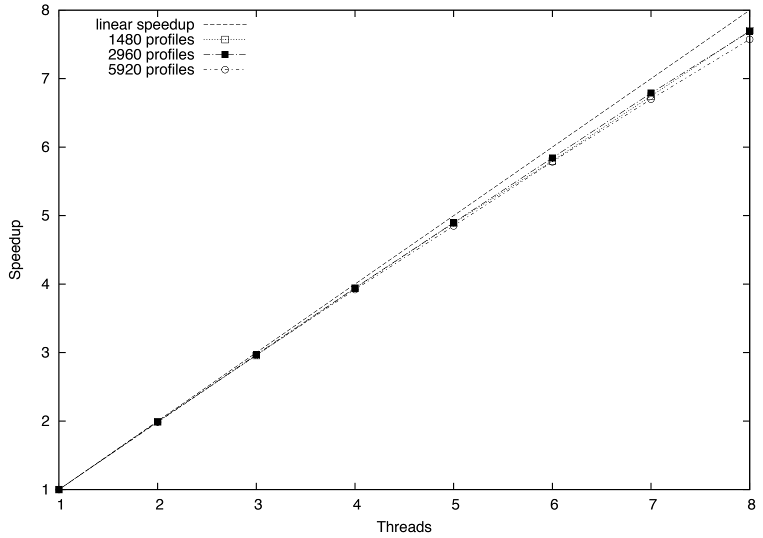


Figure 2.3: The speedup plot of the relative performance increase in the CATCHprofiles clustering engine. The parallel implementation of the CATCHprofiles clustering engine results in a near-linear speedup of computation time with increased number of threads. The y-axis shows the speedup and the x-axis the number of threads used.

The benchmarks are executed on a multi-core system with 8 cores: two Intel Xeon E5310 Quad Core processors and 8 GB RAM running Ubuntu 9.04. The speedup plot (Figure 2.3) shows a close to linear speedup for up to 8 cores when executing our benchmarks. The benchmark data set consisted of 5920 profiles, 8 tracks and a track length of 52. On the 8-core machine the CATCH algorithm required approximately 57 minutes to finish, illustrating the algorithm capacity for clustering larger data sets within a reasonable running time.

The paper in appendix A.2 includes results that were computed on a SGI Altix 3700 supercomputer running CATCHprofiles successfully on 128 cores and using close to 128 GB of memory.

2.1.1.1 Scientific Workflow

Figure 2.4 shows a constructed scientific workflow which is based on the structure of the computational part of the CATCHprofiles application. Later, in Chapter 4 I will discuss how the CSP algebra can be used to define the relations shown in figure 2.4. In the previous section, single tasks of this workflow have been changed to be able to utilise a multi-core processor efficiently.

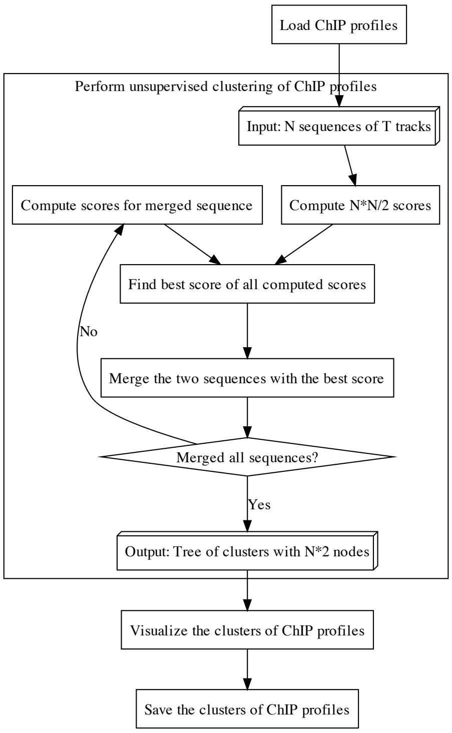

Load ChIP profiles
Perform unsupervised clustering of ChIP profiles
Input: N sequences of T tracks
Compute scores for merged sequence
Compute N*N/2 scores
Find best score of all computed scores
No
Merge the two sequences with the best score
Merged all sequences?
Yes
Output: Tree of clusters with N*2 nodes
Visualize the clusters of ChIP profiles
Save the clusters of ChIP profiles

Figure 2.4: Scientific workflow of main tasks in CATCHprofiles

2.1.2 Likelihoods for Stereo-Based Articulated Tracking

The next application is within the scientific field of computer vision. For many years articulated tracking has been an active research topic within the computer vision community. While working solutions have been suggested, computational time is still problematic. This section presents a GPU implementation that is orders of magnitude faster than a traditional CPU implementation of a ray-casting based likelihood model.

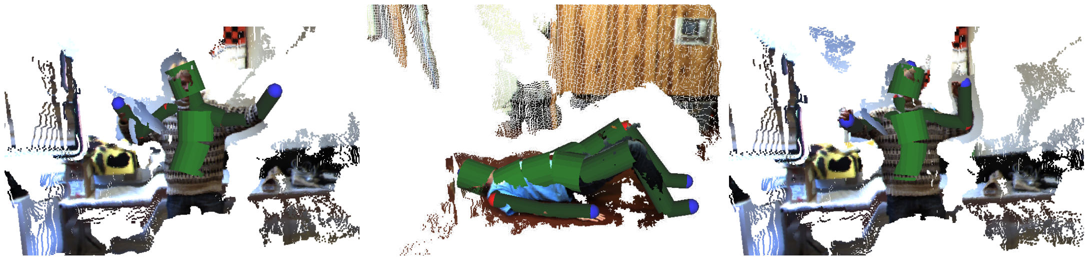


Figure 2.5: The images show stereo points with a super-imposed illustration of a skin model.

Three-dimensional articulated human motion tracking is the process of estimating the configuration of body parts over time from sensor input [78]. One approach to this estimation is to use motion capture equipment where e.g. electromagnetic markers are attached to the body and then tracked in three dimensions. While this approach gives accurate results, it is intrusive and cannot be used outside laboratory settings. Alternatively, computer vision systems can be used for non-intrusive analysis such as the one shown in Figure 2.5. One standard approach is to use a particle filter [30] for finding a sequence of poses that match the observed data well. From a practical point of view this means making many random guesses of the current pose and comparing these to the observed data. In terms of performance, the critical part is comparing each guess to the data.

At the heart of the articulated tracker is a particle filter, which is concerned with estimating an unobserved state of a system from observations. In terms of articulated tracking it is concerned with estimating the pose $\vec{\theta}_t$ at each frame in a video sequence. In terms of statistics, it seeks $p(\vec{\theta}_t|\mathcal{X}_{1:t})$, where the subscript denotes time and $\mathcal{X}_{1:t} = \{\mathcal{X}_1, \ldots, \mathcal{X}_t\}$ denotes all observations seen at time t. This distribution is crudely represented as a set of samples that are propagated through time by sampling from $p(\vec{\theta}_t|\vec{\theta}_{t-1})$. Each sample $\vec{\theta}_t^{(j)}$ is assigned a weight according to its likelihood $p(\mathcal{X}_t|\vec{\theta}_t^{(j)})$.

Thus, at each time step t the following must be computed:

```
for $j = 1$ to $J$ do
    Sample $\vec{\theta}_t^{(j)}$ from $p(\vec{\theta}_t|\vec{\theta}_{t-1}^{(j)})$ ;
    $w_j \leftarrow p(\mathcal{X}_t|\vec{\theta}_t^{(j)})$ ;
end for
```

It is expensive to evaluate the likelihood $p(\mathcal{X}_t|\vec{\theta}_t^{(j)})$, but the loop can be executed in parallel as each sample is treated completely independently. Executing this loop in parallel on a GPU requires optimising the data access patterns, otherwise the performance is likely to be worse than on a CPU. Once new samples are drawn and assigned weights, the current pose can be estimated as the mean value of $p(\vec{\theta}_t|\mathcal{X}_{1:t})$.

The algorithm presented in appendix A.1 achieves a major speedup when implemented on the GPU. However, it requires careful planning in designing for the massive parallelism in the GPU architecture. The first problem to be addressed is how to block data and computations efficiently with respect to performance. The task is to minimise data communication and maximise the amount of computations done by one block of threads. The targeted GPU architectures are the CUDA [9] enabled Nvidia GPUs with compute capability from 1.1 to 1.3.

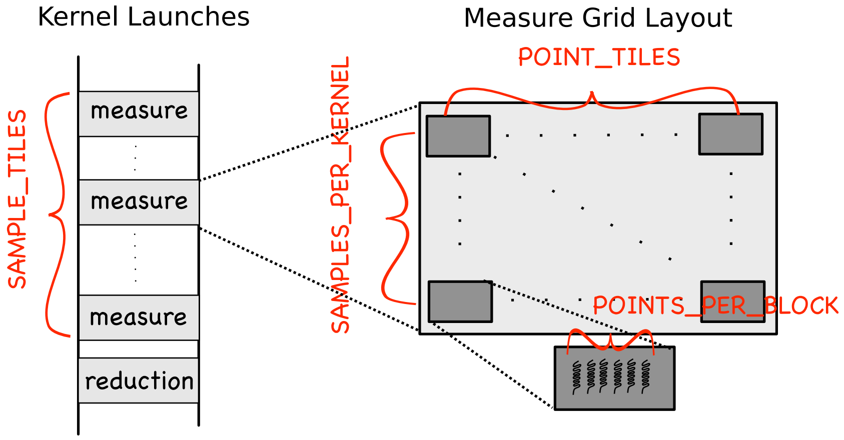


Figure 2.6: Illustration of the grid layout and kernel launches for a single GPU. A sequence of measure kernel launches is executed: one for each tile of samples. Only a single reduction kernel is launched prior to returning to the CPU thread handling the GPU.

The orchestration of data results in the grid layout illustrated in Figure 2.6. The grid of thread blocks is organised in such a way that each thread block corresponds to one sample and one tile of stereo points. A measure kernel is then launched on this grid. During execution the measure kernel will loop over samples in consecutive launches to avoid stalling the GPU hardware. Additionally, support for multiple GPU devices is performed by dividing the samples into one chunk for each GPU. If multiple GPUs are available the same number of CPU worker threads is created and then given a GPU to control. The overhead of launching CPU threads is small and the effect is only visible for very small problem sizes that are not relevant for this problem domain.

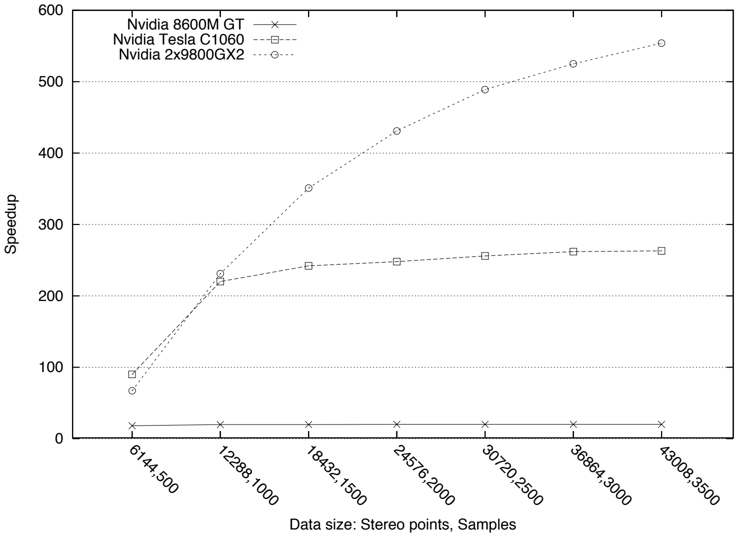


Figure 2.7: The speedup achieved when computing a data set of the specified size on a GPU vs. the CPU. The referenced sequential CPU implementation is measured on an Intel Core 2 Duo @ 2.4Ghz.

The speedup plot in figure 2.7 uses a CPU implementation as reference. The same input data set has been used for the CPU and the GPU benchmarks. The measure function used in the CPU implementation is identical to the measure function used in the GPU implementation, but the invocation of the measure function is purely sequential and thus only utilise one core. Since the problem is memory bound, the one thread will have to wait on memory. It is expected that an optimised CPU implementation could execute twice as fast, compared to the reference CPU implementation. On the GPU, the memory latency has been successfully hidden which becomes apparent when looking at the speedup numbers in figure 2.7.

2.1.2.1 Scientific Workflow

Figure 2.8 shows a constructed scientific workflow based on the structure of the computational part of the presented application. Later, in Chapter 4 I will discuss how the CSP algebra can be used to define the relations shown in figure 2.8.

In the previous section, the task ”Compute likelihood value” in the workflow has been made to run on the GPU efficiently.

A purpose for this algorithm is to provide a quick match for a pose recorded by cameras. The match can then be used to generate feedback for the user in front of the cameras. The response time for the implementation must be fast enough, such that the feedback result is still relevant for the user. Additionally, it must have a high throughput, and from the scientific workflow in figure 2.8 it is clear that the generation of the next set of 3D coordinates can be computed in parallel with the search for the current best likelihood value. The composed task of finding the best skeleton, may continue with the next set of coordinates every time the best likelihood value have been found. Such optimisations are not necessarily simple to find, which is why the execution of a scientific workflow must adopt naturally.

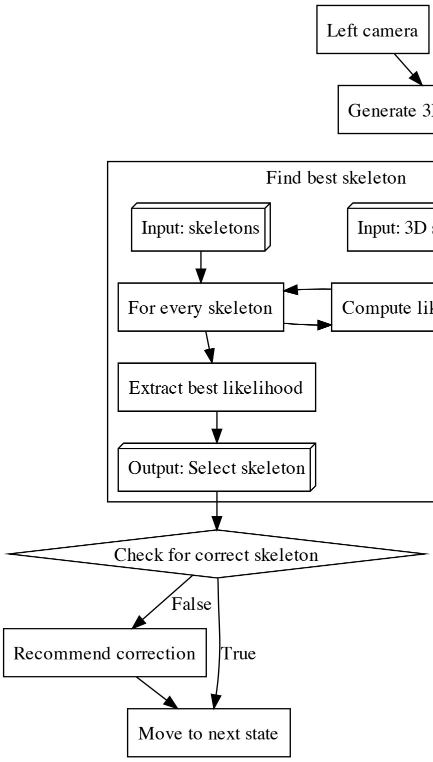


Figure 2.8: Scientific workflow of articulated tracking

2.2 Scientific Workflow Systems

This section provides a short overview of a few popular scientific workflow systems and focuses on their capability of executing workflow instances in a parallel environment. The systems presented in this section all provide a desktop authoring environment and enactment engine for scientific workflows. Figure 2.9 shows a screenshot (downloaded from http://www.knime.org) of an example data analysis workflow in Knime, the overall GUI is similar across the different systems. Features common for most of the presented systems are:

Sharing of workflows to a component repository for allowing easy access and reuse.

Recording provenance information, in order for a workflow instance to be reproduced.

Through components, direct access to larger number of scientific data repositories.

Hierarchy in workflows, where complex tasks may be composed of simpler components.

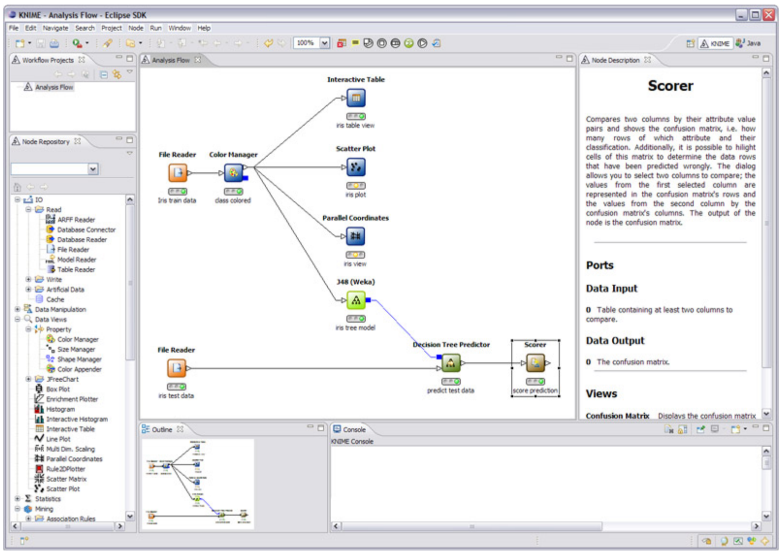
Scientific workflows are represented as a directed graph.

A command-line execution tool for executing scientific workflows.

Applications that are built using scientific workflow systems may be data intensive or computational intensive. The systems should optimise for both cases. Data-parallelism is used to improve the computational intensive parts where data may be streamed or e.g. large arrays are divided into smaller chunks. The data-parallelism is a fine-grained approach to better performance. The coarse-grained approach for computational intensive systems is the off-loading of tasks to other available processors, preferably where the data already resides. Data intensive applications may often be improved through parallel computing, but rather than running computations in parallel the data is fetched in parallel to increase the available bandwidth and, more importantly, reduce the latency of waiting for the next batch of data.

2.2.1 Taverna

Taverna [49] is oriented towards execution of eScience applications in grid environments. It is easily extendable and written in Java. Code written in other programming languages than Java must be run in separate processes. Distributed execution of tasks is handled through a broker service which manages the execution.



Scorer

Compares two columns by their attribute value pairs and shows the confusion matrix, i.e. how many rows of which attribute and their classification. Additionally, it is possible to hilight cells of this matrix to determine the data rows that have been predicted wrongly. The dialog allows you to select two columns to compare; the values from the first selected column are represented in the confusion matrix's rows and the values from the second column by the confusion matrix's columns. The output of the node is the confusion matrix.

Ports

Data Input

0 Table containing at least two columns to compare.

Data Output

0 The confusion matrix.

Views

Confusion Matrix Displays the confusion matrix

Figure 2.9: Screenshot of an example data analysis workflow in Knime

2.2.2 Knime

Knime[88, 24] is built on the eClipse framework and is written in Java. It has a similar feature set to Taverna. Knime is especially popular for chemistry and bio-science related applications. Control flows can be created by adding special components that add a conditional loop or a conditional branch.

2.2.3 LabVIEW

LabVIEW [51] is designed for interfacing with lab equipment and analysing the received data for meaningful information. Data can be processed and results shared through displays, reports and the Web. It differs from the other systems by encouraging a more fine-grained flow-based programming. It also supports realtime systems and is able to export applications to hardware, e.g. FPGAs.

2.2.4 Kepler

Kepler [65] differs from the other workflow systems in that it separates the structure of the workflow model from its model of computation. Different engines may be used for executing a given workflow graph. The Kepler scientific workflow system is written in Java and has a feature set similar to Taverna and Knime.

2.2.5 P-Grade / P-Grade Portal

This system is particularly different from the others. The P-GRADE [55] project uses a CSP-inspired parallel environment for executing distributed scientific applications. The project combines low-level programming and a graphical workflow tool with a CSP-like communication model. Applications created with this tool are compiled to MPI or PVM and provide basic support for grid systems. A later project named P-GRADE portal [56] presents an online Web platform for handling dependencies between grid jobs and enables a single application to use multiple grid middleware architectures. The P-GRADE portal has left the CSP-style programming used in the first P-Grade project and is now using directed acyclic graphs for its workflows.

2.3 Fine-grained and Coarse-grained Orchestration

Workflows are traditionally composed at the coarse-grained level as the individual tasks in a workflow consist of one or more algorithms.

The work in this thesis is a follow-up on the scientific workflow systems and does not replace current scientific workflow systems. Many scientists are using these systems and making it possible for them to create full eScience applications, with only little programming experience. Other scientists may use MATLAB or Python NumPy for their eScience projects. In reality these systems does not exclude each other but may as well benefit from a combined application. A combined application, where the coarse grained orchestration is handled by the scientific workflow system and the fine-grained is handled by NumPy / MATLAB. As these scientists are not well-trained in structured programming development there is a need for tools that make it simple to organise the coarse-grained orchestration. The workflow may for many applications represent the coarse-grained orchestration, while the fine-grained orchestration of an application consists of the math or routines in the individual tasks.

Chapter 3

Hardware Architectures for Scientific Workflows

The performance of scientific workflow instances differs depending on how suited the hardware architecture is for the different tasks. Most scientists have access to a range of hardware architectures, where some are optimized towards handling large data sets and others more on computations. This chapter provides an overview of the different architectures used for executing scientific workflows. I use the four classifications by Flynn [37]: SISD, SIMD, MISD and MIMD. The MISD (Multiple Instruction, Single Data) classification is not relevant for scientific workflows, as it is used for fail-over or redundancy in critical systems.

3.1 SISD

SISD (Single Instruction, Single Data) is a term referring to a computer architecture in which a single processor; the uniprocessor, executes a single instruction stream to operate on data stored in a single memory system. SISD can have concurrent processing characteristics within the microprocessor. Instruction fetching and pipelined execution of instructions are common examples found in most of the modern SISD computers. Such computers may execute instructions out-of-order, while committing them in-order. Executing instructions out-of-order allows for a better utilisation and thereby a higher IPC (instructions per clock). A higher IPC usually results in an improved performance to power usage ratio.

In scientific computing the SISD hardware architecture is rarely used, but a corresponding software architecture can be seen in many places. Single tasks in a scientific workflow can be viewed as single applications communicating with other tasks. The tasks may run on individual nodes and communicate through standard network protocols. The tasks may even share data, but only through communication, never through implicit access to shared or distributed memory. Every task in such a system is completely isolated and behaves similarly to the SISD architecture.

Similarly, web services are often used in scientific workflows and are from the user point-of-view a SISD architecture, as it is a separate application with a single instruction pointer and memory system. The use of web services is common in the fields of bioinformatics, astronomy, chemo-informatics, health informatics and others. Using the WSDL (Web Service Description Language) it is possible to connect to a large amount of existing services. WSDL is an XML format and the interface to many web services. It is the machine-readable description of the operations (or functions) offered by a service. Internally the web services are run on the MIMD architecture to be able to serve a large amount of requests from a single database.

3.2 MIMD

Each processor fetches its own instructions and operates on its own data. MIMD (Multiple Instructions, Multiple Data) architectures are the most versatile systems for parallel computing and can be organized into shared-memory and distributed memory systems. This division is made on how the MIMD processors access memory. Shared memory systems may be of the bus-based, extended or hierarchical type. Distributed memory machines may have hypercube or mesh interconnection schemes.

The multi-core processor is a MIMD architecture and consists of multiple homogeneous cores with access to a shared memory system. The scheduling of threads onto cores is handled by the operating system. To utilise multiple cores in a single application it is necessary to organise tasks, such that there are tasks that do not depend on each other.

Several scientific workflow systems use a thread pool to execute independent tasks in parallel on multi-core processors. Shared memory systems in 2011 exist with up to 256 cores, but are extremely expensive at that size. Off-the-shelf hardware is available with up to 12 cores per processor. Distributed memory systems, such as supercomputers or clusters, can scale to a large amount of cores. Depending on the processor design and the bandwidth of the inter-connection they range from extremely expensive (top 500 supercomputers [67]) to inexpensive clusters of standard workstations [91].

The existing scientific workflow systems that are able to execute on clusters use a main control node to handle the scheduling of tasks onto nodes. A single main control node is a huge bottle-neck and limits the execution of scientific workflows to small clusters. Clusters are very common in the scientific community, as many scientists have computational-intensive problems to solve. Re-occurring single tasks in a scientific workflow may require large computations, thus a specialised high-performance implementation for a cluster could be implemented and re-used in scientific workflows. This approach is discussed in Section 3.4.

3.3 SIMD

SIMD (Single Instruction, Multiple Data) architectures execute the same instruction by multiple processors using different data streams. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor which fetches and dispatches instructions. Multimedia extensions in modern processors are a limited form of SIMD parallelism. Vector architectures are the largest class of processors of the SIMD architectures.

Vector architectures have gotten a recent revival from GPUs. Especially because of the CUDA [9] and OpenCL [11] frameworks that enable programmers to run general purpose applications on GPUs and not just graphic related computations. Vector architectures require massive parallelism in problems where a problem is divided into a large amount of independent fine-grained tasks. Such fine-grained tasks are not impossible to handle for scientific workflows, but since scientific workflows are the tool of the scientific user low-level design choices should be avoided.

Today, scientific workflows are not executed on vector processors, but vector processors can be used indirectly through libraries or specialised implementations.

3.4 Heterogenous Computing

The MIMD and SIMD are two very different architectures, both built for parallel computing. Executing applications split on multiple architectures is a complex task that requires multiple binaries for different hardware. The advantage of running such a setup is that some types of computational problems benefit from SIMD architectures while others benefit from the more flexible MIMD architecture. Often applications running on GPUs require a part of the application to run on the main processor, in order to handle the basic control flow of the application. This is due to the GPU being highly optimised towards fine-grained parallelism with a minimum of diverging branches. Implementing for SIMD architectures is generally more challenging than implementing for MIMD architectures. In MIMD architectures there is a lot of help through cache pre-fetching and control flow logic to reduce the amounts of cache-misses and flushing of pipe-lines.

As time is a critical factor in implementing executable scientific workflows it would make sense to only execute tasks on specialised hardware, if it is possible to achieve a large performance improvement. Such a choice is made possible by using a framework that allow the use of heterogenous architectures.

The IBM CELL-BE [104] is a heterogenous architecture as it consists of two different architectures on the same silicon die. Equivalent to the GPU, the IBM CELL-BE is a challenge to program correctly. The IBM CELL-BE was designed with high throughput workloads in mind [77]. It includes a single PowerPC Processor core, its L2 cache as well as a set of high throughput cores. These cores each contain a local memory store that is incoherent with the rest of the memory system. The local store has a guaranteed latency for data delivery, allowing for a simpler execution pipeline than a system with a coherent cache hierarchy. It requires the user to manually manage the data contents through software-programmed DMA operations.

The production of the IBM Cell has been discontinued, but new heterogenous architectures are continually being developed. Both the AMD Fusion [1] and the Nvidia Tegra [10] are chips with multiple architectures on a single silicon die.

By orchestrating applications as executable scientific workflows tasks can be split onto multiple architectures if needed. Components of a graphical user interface may also be executed as tasks in an executable scientific workflow, thus control panels could be executed on mobile units (tablets, smart phones).

3.5 Grid Computing

Grid computing [57] started as a means of sharing supercomputers or clusters between research facilities. Such resources are expensive to have running, when they are unused for periods of time. Through collaboration and the sharing of resources it is possible to achieve a higher utilisation.

The resources in a Grid may be of multiple hardware architectures, thus when requesting resources it is necessary to request the type of the resources needed. It is common for grid resources to differ and that the grid middleware allow the user to select grid resources based on the requirements for a specific application.

The grid middleware is the control software that handles resources, job queues, job scheduling and users. Submitting jobs to a grid for execution has a high latency and should not be done for sequential tasks, but only for tasks where the problem can be split into multiple jobs for concurrent execution.

Grid resources are used in scientific workflow systems, where tasks (jobs) are submitted to a grid environment for execution. Only the computational-intensive tasks are sent to a grid, as the scheduling of grid jobs can have a high latency and often there is no upper limit to the latency.

Chapter 4

Communicating Sequential Processes

The Communicating Sequential Processes algebra, CSP [45, 46], was introduced more than 25 years ago and while it was highly popular and thoroughly investigated in its first years, interest dropped off in the late 1980 because the algebra appeared to be a solution in search of a problem, namely modelling massively concurrent processes and providing tools to solve many of the common problems associated with writing parallel and concurrent applications.

CSP provides many attractive features with respect to the next generation processors; it is a formal algebra with automated tools to help prove correctness, it works with entirely isolated process spaces, thus the inherent coherence problem is eliminated by design, and it lends itself to being modelled through both programming languages and graphical design tools.

4.1 The CSP Algebra

Hoare's CSP book [46] describes Communicating Sequential Processes in detail including the algebraic laws. Processes are defined as patterns that define all possible behaviors of events. These events are atomic and synchronous between the individual processes. Events often behave as channels communicating data objects synchronously between processes. The patterns are described in an explicit syntax. Table 4.1 displays a selected subset of the patterns that are especially relevant for the content presented in this thesis.

The process $a \rightarrow P$ denotes that the event a must happen for process P to be executed. For an event a to happen means that the environment must communicate a. The process $P = (a \rightarrow P)$ is a process that continues to be willing to communicate a to the enviroment. The processes $P\ ;\ Q$ and $P \parallel Q$ denote the sequential and parallel composition of process P and Q. Process $P \mid\mid\mid Q$ denotes that the processes P and Q do not communicate on any shared events and thus are not dependent on each other.

Table 4.1: A subset of the algebraic syntax for processes in CSP

$a \rightarrow P$	event a then process P
$P \; ; \; Q$	process P followed by process Q
$P \parallel Q$	process P in parallel with process Q
$P \mid\mid\mid Q$	process P interleaving process Q
$P \sqcap Q$	process P or process Q (non-deterministic)
$P \square Q$	process P choice process Q
$(a \rightarrow P \mid b \rightarrow Q)$	a then process P choice b then process Q (provided $a \neq b$)
$SKIP$	successful termination process
$STOP$	deadlock process
$b!e$	on channel b output value of e
$b?x$	on channel b input to x

Process $P \sqcap Q$ denotes the internal choice between processes P and Q where one will execute, but the choice is nondeterministic to the environment. Process $P \square Q$ denotes the external choice where both processes are available to communicate to the environment, but only one must be chosen and the choice is made by the environment. The $SKIP$ process is the successful termination at which the execution ends and the $STOP$ process denotes a deadlock.

Using the syntax from table 4.1 it is possible to specify concurrent behaviour. The functionality described by the syntax in table 4.1 is the minimum set of functionality for the implementations with support for CSP presented in Section 4.4.

The CSP algebra is supported by the automated model checker FDR [82, 64]. The FDR has been used extensively in industrial applications [62]. The combination of the mathematics of refinement and the existance of a model checker is crucial for the CSP algebra as a tool for specifying concurrent models.

4.1.1 Concurrent Processes

In CSP a process is an independent unit of behaviour. It executes in a system where the other processes are called the *environment* of the process. A process can be run sequentially, concurrently or a combination thereof to the environment (Table 4.2).

Processes participate in events. Events are the basic CSP construct for synchronisation and communication. When two processes want to engage in the same event concurrently they are allowed to proceed to their continuations, provided that both processes agree to continue. If one process wants to engage in an event where the environment is unable to engage in the same event, the process will be unable to move to a new state and remains unchanged. In case all processes want to engage in unmatched events the system has dead-locked.

Table 4.2: Process $P1$ concurrently with $P2$ and $P3$. $P2$ and $P3$ are executed in sequence.

$$\boxed{Q = P1 \parallel (P2\ ;\ P3)}$$

4.1.2 Nondeterminism

Nondeterminism occurs in a situation where the exact same actions from the environment may lead to different behaviour for the same set of processes. In CSP there are two sources of nondeterminism: The explicit choice $P \;\square\; Q$ and the implicit choice $P \;\sqcap\; Q$. The implicit choice is equivalent to asynchronous concealed events where the environment must be able to engage concurrently in all concealed events to avoid a possible dead-lock. The explicit choice or general choice is the choice which is made by the environment. This nondeterminism can be controlled by a single construct, the external choice. The external choice $P \mid Q$ allows a single process to engage in one event among a set of events. If the environment wants to engage in multiple events the choice is nondeterministic. If the environment wants to engage in a single event, the choice is deterministic.

4.1.3 Communication

The only means of communication between processes is through channels (Table 4.3). A channel is a directed connection between processes along which messages may be sent. Two processes engaging in an event is similar to two processes engaging in a directed communication across a channel. A communication is a special case of event-based synchronisation. In the CSP applications relevant for this thesis all events in a CSP network are communications. None of the CSP libraries in Section 4.4 implement a means of engaging in an event that is not a communication.

Table 4.3: Process $P1$ concurrently sending a message on channel b to $P2$

$$\boxed{\begin{array}{l} P1 = b!e \rightarrow SKIP \\ P2 = b?x \rightarrow SKIP \\ Q = P1 \parallel P2 \end{array}}$$

4.2 π-calculus

The π-calculus [69, 86] is a separate work from CSP and was developed on the calculus of concurrent systems [68]. The calculus of concurrent systems was being developed in parallel with CSP during the 1980's. The process-oriented approach has today been influenced by the π-calculus together with CSP. This thesis primarily focuses on CSP but uses a few features from the π-calculus as an addition to the CSP algebra.

The π-calculus is a calculus for mobile processes where the process network is allowed to change dynamically by interchanging channel ends between processes. Mobile channel ends have the benefit of simplifying process networks, as the different process patterns specifically designed for mobile channel ends (patterns for mobility) have shown in [84].

4.3 CSP in Scientific Workflows

Only few [106, 55] have previously looked at CSP and thought that this might be a good specification for describing scientific workflows. In Section 6.1 I present a framework that uses ideas from the CSP algebra and the scientific workflow systems (in Section 2.2), combined in a framework that allows CSP-based applications to be designed in a visual tool, and executed in a variety of ways (depending on the hardware available). I stipulate that CSP is ideal for reasoning about the dataflow of scientific applications, particularly when the motivation is concurrent execution. The compositional structure of a CSP network enables application developers to reuse networks of components as top-level components themselves.

The work by Wong [105, 106] has demonstrated an equivalence between workflow processes and the CSP algebra by describing CSP models of van der Aalst et al.'s [96] workflow patterns. It has been demonstrated by Wong that refinement is useful in the development of workflow processes because it allows for formal comparisons between workflows. The results by Wong are also true for scientific workflows, though the often used workflow patterns may be different than the ones presented by van der Aalst et al.

As object-oriented programming has design patterns, it makes sense that process-oriented programming would have process patterns. Process patterns for the CSP algebra (or process-oriented languages) have only recently been presented by Sampson [84]. The patterns presented by Sampson are basic, flexible and diverse. They describe often-occurring scenarios in concurrent applications. When modeling scientific workflows in CSP it is beneficial to utilise the process patterns by Sampson, as this will almost always result in a reduced complexity for a workflow.

4.4 Programming with CSP

This section discusses the programming languages that have native support for CSP constructs, that provide CSP constructs through embedded domain-specific languages (EDSLs) and the languages with libraries that add CSP constructs.

4.4.1 Native CSP

The occam programming language [66], which was developed in the early 1980's, was the first programming language to support CSP. occam was also the only supported language available for the INMOS Transputer; a microprocessor with hardware support for concurrency. A lot of work has later been put into occam to make it a modern programming language, now called occam-π. The latest implementation for occam-π is the Tock [85] compiler, which is written in Haskell and probably going to replace occ21 in KRoC [6]. Tock interprets occam-π code into portable C code that uses the CCSP [71] library for execution. The CCSP library is extremely fast. CCSP implements processes as user-level threads and uses a very robust and optimised scheduler that can handle millions of processes. The utilisation of multiple cores is handled automatically by the scheduler in CCSP (see [81]). occam applications can also run on clusters using the P0ny network environment [87]. The P0ny network environment allows for channels to be defined as network-enabled.

The occam-π compilers perform a rigorous checking at compile-time e.g. static type checking, detecting possible dead-locks and more. There are no other languages with support for CSP that perform as extensive checking at compile time.

The Go programming language [16] by Google is a new language and was first published as a Google project in 2007. The Go language has native support for CSP as the only way to do concurrent programming. Processes are lightweight co-routines and a single application can easily use 10.000 co-routines. Similar to occam-π, Google Go can utilise multiple cores on multi-core systems. The co-routines in Go are called go-routines and are executed through an asynchronous call. It is not yet possible to wait for a go-routine to finish, except through explicitly communicating with it. Channels are by default synchronous, but can be defined with a bounded buffer. The external choice is provided through a `select` call that can randomly select between ready input and output ends of a channel. Since the `select` call does not provide a prioritised choice it is not possible to guarantee a fair choice by ordering the channel ends. Currently the channels are not yet network-enabled, thus node-to-node communication must be handled through other means of communication. The slogan of Go is ”Do not communicate by sharing memory; instead, share memory by communicating.”

4.4.2 EDSL-based CSP

The Java environment provides the Java Virtual Machine (JVM) which executes Java byte code. Alternative compilers for new and old languages have been created which enable other languages than Java to run on the JVM. Scala [74] fuses object-oriented and functional programming in a statically typed programming language, provides lightweight processes and compiles to Java byte code. The CSP implementation; Communicating Scala Objects (CSO) [92, 63] has been implemented for Scala on top of lightweight processes, making it possible to reduce communication overhead and increase the amount of processes in a single application.

Communicating Haskell Processes (CHP) [28] is a CSP implementation for Haskell. It is built on top of Haskell monads which are special units of execution. The Haskell scheduler from the Glasgow Haskell Compiler (GHC) is very fast and is able to utilise multi-core systems. All of the known CSP constructs are implemented in CHP. CHP has strongly typed channels and even allows for direct model-checking of CHP applications.

4.4.3 Library-based CSP

The support for CSP can be added to programming languages through libraries. Depending on the programming language, the CSP constructs can be made to appear more or less as built-in constructs. It is usually desired to make the CSP constructs appear as natural elements of the programming language, since CSP is supposed to facilitate the possibility of process-oriented programming in a non process-oriented programming language. The external choice construct is particularly difficult to implement as a basic statement similar to `if` or `switch` statements. Thus the external choice is for all the library-based CSP implementations almost always followed by an `if` or `switch` statement selecting a branch depending on the result from the external choice. The CSP implementation for a programming language is built on the concurrent features available in the language, such as threads, shared memory and locks.

CTJ [43] and JCSP [101] are CSP implementations for the Java programming language. Both implementations have processes and channels implemented as objects and use the standard threading library in Java to facilitate the concurrent properties for processes. CTJ is primarily for real-time systems, while JCSP is a more general CSP library. The channels in JCSP are typed and separated into one-to-one, one-to-any, any-to-one and any-to-any channels. JCSP also has network-enabled channels, making it possible to create distributed JCSP applications. External choice is in JCSP limited to one-to-one channels for output channel ends.

C++CSP2 [27] is a CSP implementation for C++ which is able to both utilise multiple cores and use user-level threads for fast context switching. User-level threads are more efficient and provide greater flexibility than kernel threads. It is possible to optimise the scheduling of user-level threads to fit with the internal priority in the application, since the scheduler is in user-level code and the operating system is not involved. In C++CSP2 it is necessary to specify whether processes should be run as user-level threads or kernel-level threads.

Python-CSP [72] is a CSP library for Python presented in 2009. The synchronisation design is similar to JCSP and implements CSP processes using OS processes or threads. There is currently no network support for Python-CSP.

The PyCSP [25, 40, 98, 42] library also implements the CSP constructs for Python. PyCSP is the main work of this thesis and thoroughly discussed in chapter 6.

Chapter 5

Python in Science

The Python programming language [83] and the CPython interpreter were first released in 1991 and have continually been updated while popularity has increased. There are numerous alternative interpreters available for Python today which all have their advantages and disadvantages that are discussed in Section 5.1.

Most relevant for this thesis is the wide usage of Python within the scientific field. This usage has had the positive effect that many scientific libraries (modules) [7] have been added to Python. More and better libraries have caused a feedback loop of more scientific users, which again motivates more scientific libraries. Besides the large amount of scientific libraries Python already comes with a standard library [18] which covers everything from asynchronous processing to zip files. The following is a short list of interesting and relevant scientific libraries (modules). There are definitely many more.

NumPy [75] - Provides both basic and advanced array operations. This library is required by most other scientific libraries, as it enables Python to perform fast vector operations on arrays. The array object is particularly well-suited for interoperability with C / C++ libraries.

Matplotlib [50] - A plotting library for 2D and 3D plotting. It provides a set of functions familiar to MATLAB users and uses the NumPy array as the primary datatype.

DistNumPy [60] - Executes Python / NumPy code on parallel architectures by splitting the vector operations and distributing the workload to multiple hosts. Code must be written to use NumPy array operations as all other operations execute sequentially.

SciPy [53] - Is a set of scientific tools for Python. It includes NumPy and implements matrix operations for NumPy matrices such as linear algebra and fourier transforms together with many other scientific tools. It focuses on the interoperability between the provided set of tools with the overall goal to simplify scientific computing in Python. The list of features is comprehensive.

PyCUDA and PyOpenCL [58] - They are libraries for handling GPGPU (General Purpose GPU) programming through Python. This approach is especially powerful, as the binary GPU code is run-time generated. Python is a dynamic interpreted language, thus function arguments are not known until run-time. Providing run-time code generation enables the library to make run-time optimisations based on the actual function arguments.

The approach to writing the compute-intensive parts in C or another compilable language such as Fortran is described in [61] and believed to be within the capabilities of non computer scientists. Programming skills for sequential C and Fortran code have been taught to many non computer scientists. Some scientific problems will have libraries that can perform the computational tasks, such as the scientific libraries presented above, thus eliminating or reducing the need for a compilable language.

5.1 Python Interpreters

The standard CPython interpreter is the official interpreter from the Python community. The CPython interpreter has a Global Interpreter Lock (GIL) that ensures exclusive access to Python objects, when running multiple threads in one interpreter instance. Because of this GIL, it is difficult to achieve any speedup in Python from running multiple threads, unless the actual computation is performed in external modules (libraries) that release the GIL. Instead of releasing and acquiring the GIL in external modules it is possible to use multiple processes that run separate CPython interpreters with separate GILs. In Python 2.6 the multiprocessing module [14] can handle processes, which enables us to make a comparison between threads and processes. The comparison in Table 5.1 shows the result of computing π (Monte Carlo method) in parallel using threads and processes (Listing 5.1).

Table 5.1: Comparison of threads and multiprocessing on a dual core system with Python 2.6.2

Workers	1	2	3	4	10
Threads	0.98s	1.52s	1.56s	1.55s	1.57s
Processes	1.01s	0.57s	0.54s	0.54s	0.56s

Listing 5.1: Calculation of π (Monte Carlo method) in parallel using threads and processes

```
cnt = 1000000 / workers
def compute(id, cnt):
  pi = 4.0*reduce(
    lambda x,y: x+(random()**2+random()**2<1.0),
    xrange(cnt)
  ) / cnt
  subpi[id] = pi

# Threads
import threading
subpi = [0 for i in range(workers)]
threads = [ threading.Thread(target=compute, args=(cnt,))
  for i in range(workers) ]
map(threading.Thread.start, threads)
map(threading.Thread.join, threads)
print "Threading result:", sum(subpi)/len(subpi)

# Processes
import multiprocessing
subpi = multiprocessing.Array('d', [0.0 for i in range(workers)])
processes = [multiprocessing.Process(target=compute, args=(cnt,))
  for i in range(workers) ]
map(multiprocessing.Process.start, processes)
map(multiprocessing.Process.join, processes)
print "Multiprocessing result:", sum(subpi)/len(subpi)
```

The GIL is to blame for the poor performance for threads illustrated in Table 5.1. It is possible to obtain good performance for threads, but through the use of external, non-Python, modules which release the GIL. The developers of CPython have tried to replace the GIL with a more fine-grained locking scheme, but so far they have been unsuccessful [17] as it reduced the performance of unthreaded Python code too much. Another project, the unladen swallow project [19], has also researched on how to remove the GIL. They believe that CPython should drop reference counting and move to a pure garbage collection system. This is a major change to CPython and has not yet been tried. In addition, the unladen swallow project has meanwhile been discontinued.

In parallel with the evolution of the CPython interpreter, many other interpreters have surfaced. Two especially relevant in relation to the GIL are Jython [5] and IronPython [4]. Jython is a Python interpreter that translates Python code to Java byte code for execution on the JVM (Java virtual machine) runtime system. It is very fast and it uses the JVM garbage collector instead of reference counting, thus eliminating the GIL. Similar to Jython, IronPython translate Python code to MSIL (Microsoft Intermediate Language) byte code for execution on the .NET CLR (Common Language Runtime). IronPython also avoids the GIL through the use of a garbage collector and provides an effective JIT compilation in the runtime system.

Cython [2] is not an interpreter but a compiler, which can generate C-extensions (modules written in C) for Python from annotated Python code. Functions that are compiled to C are annotated with C types. Using these tools, it is possible to speed up the execution of Python code.

Unfortunately none of the scientific libraries presented in the previous section are available for other interpreters than the CPython interpreter. The reason being that the libraries are mostly written in C or C++ and interface with CPython using the CPython API (defined in header files). Stackless Python [94] is the only exception, as this is a modified CPython interpreter with native support for micro-threads (non-preemptive user-level threads).

The micro-threads in Stackless Python is an extension of a co-routine implementation named greenlets. The greenlets [93] have been made available to CPython through a greenlet module, thus allowing fast co-routines in the standard CPython interpreter. Greenlets are also available in the PyPy interpreter, where functions such as Just-in-Time compilation and many other interesting features are being worked on. The PyPy [80] interpreter is written in the Python language and is compatible with existing Python code. It is not yet stable, but shows promising results and has recently gained support for serialising executing co-routines in one interpreter and then later resuming execution in another interpreter instance.

5.2 Parallel Programming in Python

Parallel programming in Python is possible through the use of many different libraries. There are fork-based systems similar to Python's built-in map function. Classic systems such as the built-in threading or multi-processing library provide basic support with synchronisation structures such as locks, monitors and queues. Common for those libraries is the use of shared data structures which may hide dependencies between processes. Such dependencies easily cause data-hazards, while use of locks can result in dead-locks or race-conditions. Shared memory, locks and monitors are difficult constructs to get right in parallel programming.

Most high-level approaches to parallel programming have been implemented for Python. There is task based systems, such as Parallel Python [95] administering a pool of workers (or servers). The Pyro [13] library for handling remote procedure calls and remote Python objects. The ZeroMQ [20] library that provides an alternative socket implementation, targeted towards distributed computing and clusters. Data-parallel systems that parallelise vector or matrix operations, such as DistNumPy [60]. Message-passing libraries such as MPI for Python [34] which enable fast and advanced communication between processes.

Chapter 6

PyCSP

The work presented in this chapter focuses on the development of PyCSP; a CSP library for Python. PyCSP was first presented in 2007 [25] and implemented processes as threads, motivated by an application domain with scientific users and for hands-on teaching of distributed and parallel computing in computer science. It was quickly adopted by many students at the participating institutions. The synchronisation mechanisms used for the first PyCSP was based on JCSP [102].

This chapter will present different usages of CSP in Python as well as a completely new PyCSP that removes many limiting factors compared to the original version from 2007.

The graphical approach to CSP applications has existed since the introduction of the INMOS Transputer and occam. Many alternatives on how to explore or edit process networks have been tested and used. Most of these are discussed by Jacobsen (Chapter ”The Process Explorer” in [52]) and include occam based environments and non-occam based environments as well as the author's environment : POPExplorer (The Process Oriented Programming Explorer). Section 6.1 presents a graphical approach to CSP that are targeted towards scientific workflow modeling in Python. The content of Section 6.1 was originally presented in my Master's thesis [38]. It has later been published in [41] which is included in Appendix A.3.

6.1 A graphical approach to Python CSP applications

Many scientists (chemists, physicists, etc.) are not experienced programmers but are able to do scientific computing by programming sequential applications. So far they have been relying on the hardware manufacturers to produce hardware which improved the performance of their applications – allowing for solving more sophisticated problems that require large computations. Scientists are now forced to develop concurrent applications, in order to take advantage of parallel hardware. The amount of difficulty involved in creating concurrent applications depends on the programming language and methodology. Traditional concurrent programming, with threads and locks, makes it difficult to program even simple applications – adding more parallelism to an already threaded program tends to result in problems, not solutions. As a direct result, concurrent programming is seen as challenging, and is generally avoided by the majority of scientists.

The goal of PyCSP is that scientists should develop concurrent programs using a CSP-based approach, where applications are built as layered networks of communicating processes. Such an approach is reliable; no unexpected surprises; scalable, to different numbers of processes and processors; and compositional, enabling processes to be ”glued” together to build increasingly complex functionality.

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

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

When doing scientific work, which often relies on particular math libraries for performing computations, the functions provided are not necessarily all implemented in the same language. By using tools such as SWIG [15] and F2PY [76] these issues are addressed, making it possible to use code from C, C++ and Fortran in a single scientific application.

CSPBuilder is a framework, written in Python, that assists scientists in creating concurrent applications based on a CSP design. CSPBuilder uses a graphical user interface similar to other scientific workflow systems (Section 2.2) already available.

6.1.1 CSPBuilder

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

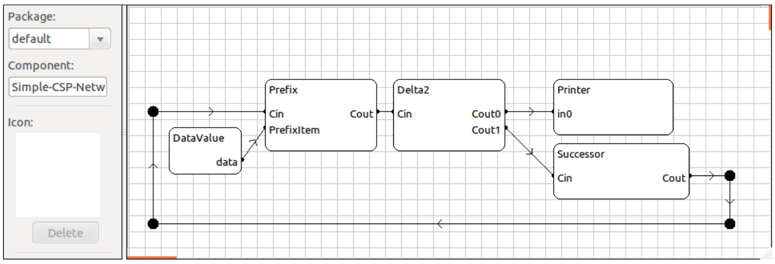


Figure 6.1: A CSPBuilder application that generates incrementing natural numbers

A number of connected processes is defined as a process network (Figure 6.1). The network (in Figure 6.1) can also be saved as a component for reuse in another application, thus enabling compositional process networks.

Connecting processes using a one-to-one channel is simple, since it is represented by a single line going from one process to another. Representing the any-to-any, one-to-any and any-to-one channel types is not as trivial. Connection points are introduced to address this issue. A connection point can have any number of inbound and outbound connections to processes or other connection points, enabling visualisation of all channel types and the ”bending” of channels.

Further implementation details of the CSPBuilder framework is in Appendix A.3.

CSPBuilder provides a simple and intuitive means for designing concurrent applications. The graphical tool produces CSP network descriptions that are interpreted into a PyCSP network and supports transparent integration of C, C++ and Fortran functions. Experiments were performed where a CSPBuilder application was distributed onto networks of workstations and cluster systems. Because PyCSP at that time did not support networked channels, extensive modifications to the basic channel code in PyCSP was required to make the experiments possible.

The CSPBuilder framework has exposed a need for a more flexible channel type in PyCSP. It was found that dealing with multiple channel types in a dynamic language was not ideal and that components could not be re-used easily, since only the one-to-one and any-to-one supported the alt construct. A single any-to-any channel type is semantically equivalent to a one-to-one, one-to-any and any-to-one depending on the number of readers and writers. Thus, by providing an any-to-any channel type with support for the alt construct processes can be re-used without being limited by the CSP library.

6.2 A New Implementation of PyCSP

The primary user group for the PyCSP library is scientists with a background in something other than computer science, though the first adopters have been students at the

University of Copenhagen. The reported experience from the students was regarded as valuable information, since the students were given scientific computing assignments. The most frequent comment we received from the students was disappointment that true parallelism could not be obtained using pure Python code. This is because of the Global Interpreter Lock in the standard Python interpreter (see Section 5.1). To address this, a new PyCSP implementation was added with support for operating system processes in addition to threads. Strictly speaking, this could be done with no changes to the PyCSP API and a new process-based implementation could transparently replace the previous. However, a number of other comments we received also addressed the syntax and semantics of PyCSP and we thus decided to revisit the design. The work on the PyCSP implementations has been published in [98, 40] and is included in Appendix A.4 and A.5.

The new API is implemented in four versions: Threads, processes, greenlets and networked. All four versions are packed in a single module, to motivate the developer to switch between them as needed. A common API is used for all implementations making it trivial to switch between them, as shown in Table 6.1. When switching to another implementation the PyCSP application may execute very differently as processes may be scheduled in another order. Hidden latencies may also become more apparent when all other processes are waiting to be scheduled.

Table 6.1: Switching between implementations of the PyCSP API

pycsp.threads	**pycsp.processes**

`import pycsp.threads as p`
 	
`@pycsp.process`
`def P(msg):`
`    print msg`
 	
`pycsp.parallel(P("Hello"))`
	`import pycsp.processes as p`
	
	`@pycsp.process`
	`def P(msg):`
	`    print msg`
	
	`pycsp.parallel(P("Hello"))`

The implementations are:

pycsp.threads - A CSP process is implemented as an OS thread. The internal synchronisation is handled by thread-locking mechanisms. This is the default implementation. Because of the Python Global Interpeter Lock, this is best suited for applications that spend most of their time in external routines that release the GIL.

pycsp.processes - A CSP process is implemented as an OS process. The internal synchronisation is more complex than pycsp.threads and is built on top of the multiprocessing module available in Python 2.6. This implementation is not affected by the Global Interpreter Lock, but has some limitations on a Windows OS and generally has a larger communication overhead than the threaded version.

pycsp.greenlets - This implementation uses co-routines instead of threads. Greenlets [93] is a simple co-routine implementation available as a Python module. It provides the possibility of creating 100.000 CSP processes in a single CSP network. This version is optimal for single-core architectures since it provides the fastest communication, but with no parallelism.

pycsp.net - A simple network-enabled implementation based on pycsp.threads. All synchronisation is handled in a single process. This provides the same functionality as pycsp.threads, but adding a larger cost and a bottleneck by introducing a server process. It uses Pyro [13] for communication.

pycsp.dist (experimental) - An implementation which automatically distributes processes to random hosts provided in a host list. Every host must be accessible through SSH with key-based authentication. The individual channels communicate using the **pycsp.net** implementation.

pycsp.grid (experimental) - Provides a new `@grid\_process` decorator that submits a process as a Grid job for execution at a remote resource. The resource is found through a set of parameters given to the `@grid\_process` decorator specifying the requirements for a grid job. The channel communication is setup using **pycsp.net**. Additionally Pyro [13] has been replaced with XML-RPC in an alternative version, since many Grid resources are behind firewalls that only allow outgoing HTTP traffic on port 80.

pycsp.dist and **pycsp.grid** are not available in the official version of PyCSP but in experimental branches.

It is not possible to mix the process construct from one implementation with the channel construct of another implementation. This issue is approached in Section 6.5. The model of synchronisation used as the basis for the implementations is described and model-checked successfully in Section 6.2.5.

6.2.1 Processes

Just as in the original PyCSP processes are wrapped in a process decorator, i.e. they are not merely implementations of a Process class as in JCSP [100] or C++CSP [27]. The advantages of this approach are that processes will be easily recognisable in the source code, and that it gives great flexibility for the PyCSP runtime environment to handle processes in different ways. The constructor used is `@process`, and a hello world example could look like the following example:

```
@pycsp.process
def hello_world(msg):
    print "Hello world, this is my message " + msg
```

Usually one or more channel ends will be part of the parameters for a process. Defining a process as above will not instantiate or execute any code: it is simply defined as a process to be used in a network at a later time.

6.2.2 Process sets

Once a process is defined, a set of processes may be instantiated and executed using the Parallel or Sequence constructs similar to the old version. However, in order to accommodate variable size networks a process set may now include lists of processes as well as individual processes.

```
pycsp.Parallel(
    source(),
    [worker() for i in range(10)],
    worker() * 10, # syntactic sugar
    sink()
)
```

In the above example source, worker and sink have all been defined as processes and the parallel construct will run one source, many workers and one sink process in parallel and return once all processes have terminated. Naturally the example makes little sense without the use of channels for communication; these will be introduced below. Apart from the support for mixing scalars and vectors of processes, the Parallel and Sequence constructs work as in the previous version and should be intuitive to anybody with any CSP experience.

6.2.3 Channels

PyCSP originally based much of its design on JCSP, continuing the use of specialised channel types: one-to-one, one-to-any, any-to-one and any-to-any. The type names designate how many writer and reader processes were allowed to be attached to the respective channel ends.

The main reason for the specialised channel types was that the implementation of the alt construct, which allowed external choice, was based on the JCSP version and placed strict limitations on the use of channels: only one process could safely use an alt construct with a given channel end. To safeguard against misuse, only the reading end of channel types that were restricted to one reader could be used as guards in an external choice. Limitations such as these can be cumbersome to work around when designing your CSP application and even more so for newcomers to PyCSP.

6.2.3.1 New Channel type

There is only one channel type in the new PyCSP, as PyCSP is targeted towards scientific uses. The channel is similar to the previous any-to-any channel, but with the difference that both input and output channel ends support external choice. The use of external choice is described in Section 6.2.4.

```
chanA = pycsp.Channel('A')
chanB = pycsp.Channel('B', buffer=5)
chanList = pycsp.Channel('B') * 10
```

Retrieving channel ends for use in processes has also changed in PyCSP. Previously, a programmer would grab a channel end by calling the `read()` or `write()` method of the channel. This has been replaced with the `channel.reader()` and `channel.writer()` functions which also have a role in channel poisoning described below.

6.2.3.2 Channel poison

The concept of poisoning channels with the purpose of shutting down an application was introduced in C++CSP[29] and later investigated in some detail by Bernhard Sputh[90]. A channel is poisoned and all subsequent reads or writes on this channel will throw an exception. This exception can be caught and used as a shut-down procedure or just to shut down that single channel. In the following example we create two processes, `source` and `sink`, and a channel to connect them. The `source` process finally poisons the channel to terminate the network, which will happen since the `sink` process does not catch the exception.

```
@pycsp.process
def source(chan_out):
    for i in range(10):
        chan_out("Hello world")
    pycsp.poison(chan_out)

@pycsp.process
def sink(chan_in):
    while True:
        print chan_in()

chan = pycsp.Channel()
pycsp.Parallel(source(chan.writer()), sink(chan.reader()))
```

Since all channels now support multiple readers and writers it is easy to add more readers and writers:

```
pycsp.Parallel(source(chan.writer()), sink(chan.reader()),
               source(chan.writer()), sink(chan.reader()),
               source(chan.writer()), sink(chan.reader()),
               source(chan.writer()), sink(chan.reader()),
               source(chan.writer()), sink(chan.reader()))
```

or

```
 pycsp.Parallel([source(chan.writer()) for i in range(5)],
          [sink(chan.reader())   for i in range(5)])
```

Both versions produce five source and five sink processes, however the created network will not do what the user may intuitively think it does. One of the sources are bound to finish first and it will then poison the channel, which will terminate the network before all the expected messages have been printed. The problem is extremely common in producer-consumer class applications and users end up with complex solutions for terminating the network.

To address this we introduce a poison mechanism based on reference counting. Creating channel ends and retires from them update a counter of how many readers or writers we have on a channel, and the leave method may perform automatic poisoning when no readers or no writers are left.

The reader() and writer() methods automatically join the respective ends of a channel, returning a unique reference to that channel end. A new function, retire(), is used to leave a channel end. All subsequent requests to this channel end reference will raise an exception. When all readers or writers have retired a channel, the other end of the channel is also retired. This is similar to how poison is propagated in the previous versions of PyCSP, but with one important difference: with a poisoned channel any reference to that channel will trigger a ChannelPoisonException which is caught in the Process class that wraps all PyCSP processes. The exception handler then poisons all the other channels that were passed to the process upon initialisation. With a retired channel, the ChannelRetireException is thrown and the other channel ends are retired rather than poisoned. Implementation-wise the two are identical apart from the name of the exception that is raised.

The following code demonstrates how the retire expression can be used instead of the poison expression. The network will now be poisoned by the last source process to finish, rather than the first. This feature hugely simplifies many networks.

```
@pycsp.process
def source(chan_out):
    for i in range(10):
        chan_out("Hello world")
    pycsp.retire(chan_out)
```

6.2.4 External choice

One of the criticisms that the original PyCSP attracted was the way that external choice was implemented, which had more in common with UNIX socket programming using select than the more compact occam ALT operation. After executing an external choice (Alternative) you are required to read from the selected channel. Failing to do so would break the rules for the choice construct in CSP. Thus we decided to simplify the usage of alt by combining select with a custom-defined action on the guard, similar to the occam ALT. Based on this, a new choice is introduced named AltSelect.

AltSelect has changed significantly from Alternative, partly to make it more like occam, partly to support output guards.

A guard set is now represented as a prioritised list of guards where a guard is a Guard object and can be initialised with an attached action. An action is a function of type choice and is executed if the guard is selected. If the guard is an input guard then the choice function will always be handed the parameter `channel_input`, which is the message that was read from the channel. AltSelect will also always return the tuple (guard, msg). The returned guard is the guard that was chosen by AltSelect. If the guard was an input guard, msg is the message that was read from the chosen channel, otherwise msg has the value None. If a guard action was defined then it is executed before AltSelect returns.

Listing 6.1: AltSelect

```
@pycsp.choice
def read_action(info, channel_input):
    print info + str(channel_input)

@pycsp.choice
def write_action(val):
    print "Wrote " + str(val)

@pycsp.process
def par_in_out(cin1, cin2, cout3, cnt):
  for i in range(cnt):
    pycsp.AltSelect(
      pycsp.InputGuard(cin1,
                 read_action(info="received on cin1")),
      pycsp.InputGuard(cin2,
                 read_action(info="received on cin2")),
      pycsp.OutputGuard(cout3, i, write_action(i))
    )
```

Note that AltSelect always performs the guard that was chosen, i.e. channel input or output is executed within the AltSelect, so an AltSelect execution with no declared choice, or a choice where the results are simply ignored, still performs the guarded input or output. An example of AltSelect usage is shown in Listing 6.1.

An action might alternatively be passed as a string. This string is then evaluated with a copy of the current namespace. All mutable types can be updated from the evaluation of this string. In Python, the `list`, `dict` and `set` types are built-in mutable types.

```
@pycsp.process
def counter(cin0, cin1):
    try:
        cnt = [0,0] # use mutable type
        while True:
          pycsp.AltSelect(
            pycsp.InputGuard(cin0, action="cnt[0] += 1"),
            pycsp.InputGuard(cin1, action="cnt[1] += 1")
          )
    except ChannelPoisonException:
        print 'Counted:', cnt
```

AltSelect performs a prioritised select from the guards in the guard list. This is especially useful when using timeout or skip guards. A guard set having a skip guard as the first item always commits to the skip guard, thus skip is usually used as the last item in a guard set. A timeout guard will try to commit when the defined seconds have passed. An example usage of the PyCSP timeout guard is shown in Listing 6.2.

Listing 6.2: AltSelect with timeout

```
(guard_selected, msg) = pycsp.AltSelect(
  pycsp.InputGuard(cin),
  pycsp.TimeoutGuard(seconds=1)
)

if isinstance(guard_selected, pycsp.TimeoutGuard):
  print timeout!
```

PyCSP provides four built-in guard types to use with external choice. AltSelect allows for any combination of the available guard types in a guard list. The channel input, timeout and skip guard are well known to the CSP community:

`InputGuard(chan_input_end, action=[optional])`
- channel end input

`OutputGuard(chan_output_end, value=<msg>, action=[optional])`
- channel end output

`TimeoutGuard(seconds=<sec>, action=[optional])`
- when expired, it will commit

`SkipGuard(action=[optional])`
- at first chance, it will commit

The output guard is new in PyCSP, although thoroughly discussed throughout the years and previously seen in Communicating Java Threads[44]. It is well understood by most programmers that use process algebra that output guards are not needed from a CSP point of view, and one may with relative ease construct equivalences for any type of output guard using only input guards. However, output guards are convenient for the user of PyCSP and equivalences are hard to construct for users that are not professional programmers, thus we provide the output guard as a primitive in PyCSP. All guard types supported can be interrupted by channel poisoning or retiring. PyCSP channels may be guarded in both ends, i.e. an output guard can be matched by an input guard.

A common method for avoiding starvation when using external choice is to reorder the guards in relation to the history of previous choices. A FairSelect construct is provided that performs a fair choice based on history. The interface of FairSelect is identical to AltSelect.

6.2.5 Channel Synchronisation

The synchronisation mechanism used for JCSP and the original PyCSP implementation is different from the one used in the new PyCSP [98, 40]. One particularly interesting difference is that the JCSP model by Welch et al. [102, 103, 101] handles read and write operations differently than alt operations. Alt operations are handled similarly to the three-way handshake protocol to negotiate whether a guard from an alt can be committed. If a guard is partially selected, the process activating the alt operation must be asked to reply with an acknowledgement, to ensure that the guard is still active and can be selected. The addition of the alting barrier [99] has created new possibilities for JCSP including the support for output guards on symmetric one-to-one channels.

In the new PyCSP model read, write and alt operations are handled similarly by a channel. The channel synchronisation performs the selection by entering a critical region that includes all processes depending on a channel. Every process that wants to read or write to a channel must submit a request. This request is put on a queue and redirects to a locking object owned by the process. This means that the time-complexity of a selection has the worst case time-complexity of $O(n * k)$, where n is the amount of unavailable input guards and k is the amount of unavailable output guards.

A simple way of describing the difference between the two models is that in JCSP the critical region is in the channel resource which makes the selection, whereas in the new PyCSP model the critical region includes the dynamic set of processes which depend on the channel resource. The first model may perform an invalid selection, that must be rolled back. The second model will never have to roll back.

In Figure 6.2 I show an example of how the matching of channel operations comes about in the new PyCSP model. Four processes are shown communicating on two channels using the presented design for negotiating read, write and external choice. Three requests have been posted to channel A and two requests to channel B. During an external choice, a request is posted on multiple channels. Process 2 has posted its request to multiple channels and has been matched. Process 1 is waiting for a successful match. Process 3 has been matched and is going to remove its request. Process 4 is waiting for a successful match. In the future, process 1 and process 4 are going to be matched. The matching is initiated by both, but exactly one process marks the match as successful.

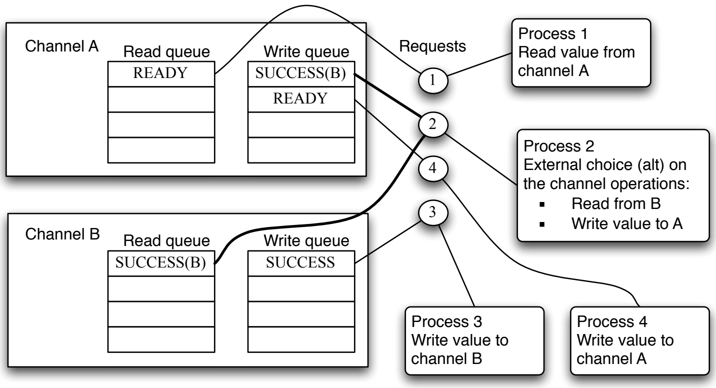


Figure 6.2: Example of four processes matching channel operations on two channels

The matching of communication requests on a channel use a method similar to the two-phase locking protocol where the locks of both involved processes are acquired before the system state is changed. To handle specific cases where multiple processes have posted multiple read and write requests, a global ordering of the locks (Roscoe's deadlock rule 7 in [82]) must be used to make sure they are always acquired in the same order. In this local thread system the locks are ordered based on their memory address. This is both quick and ensures that the ordering never changes during execution. An alternative index for a distributed system could be a combination of the node address and the memory address.

The algorithm in Listing 6.4 is executed for every possible pair of read and write requests posted on channels in Listing 6.3. The first phase acquires locks and the second phase releases locks. Between the two phases updates can be made. Eventually when a matching is successful three things are updated: the message is transferred from the writer to the reader, the active state is changed and the possibly waiting process is notified.

Listing 6.3: Pseudo code for the channel synchronisation method. `op` is a read, write or alt operation. `request` is either a read or write request.

```
request = new_request()
for every channel involved in op:
    # post request to channel
    if op.type = read:
        channel.read_queue.add(request)
    else:
        channel.write_queue.add(request)
    if channel.match() then break

if not done:
    waitFor match by another posted request

for every channel involved in op:
    channel.remove(op, request)
```

Listing 6.4: Pseudo code for the channel match algorithm

```
for read in channel.read_queue:
    for write in channel.write_queue:
        if active(read) and active(write):
            # Two-phase locking
            if (memory_addr(read) < memory_addr(write)):
                lock(read), lock(write)
            else:
                lock(write), lock(read)

            if active(read) and active(write):
                #Execute communication
                transfer(write, read)
                change_active_state(write, read)
                notify_possibly_waiting_process(write, read)

            if (memory_addr(read) < memory_addr(write)):
                release(write), release(read)
            else:
                release(read), release(write)

            if success then break loops
```

A model of this channel synchronisation method and an automatic exhaustive verification of the model is presented in [42] and included in Appendix A.8. The automatic exhaustive verification is performed using the SPIN model checker on the process configurations shown in Figure 6.3. All six process configurations have been verified to be free of deadlocks, livelocks, race conditions, unspecified receptions, unexecutable code and user-specified assertions.

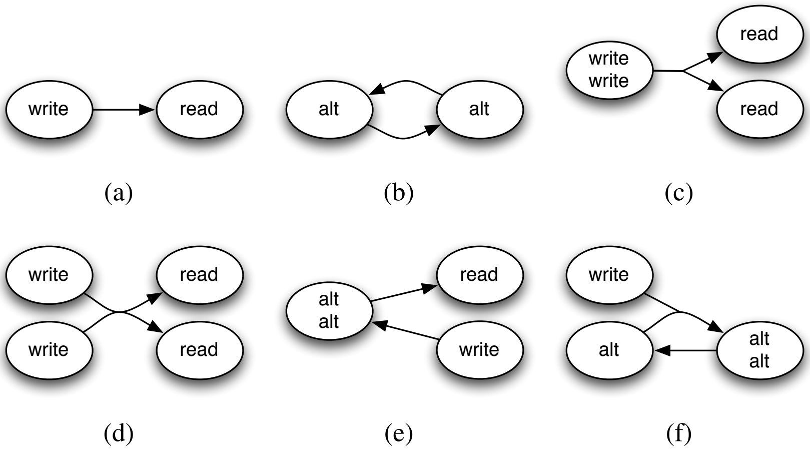


Figure 6.3: Process configurations used for verification

6.2.6 IO and Co-routines

A limitation with co-routines is that everything runs in a single thread, which means that a blocking call will block all other co-routines as well. This is especially a problem with IO operations, since the blocking action might happen in a system call, which is not detectable from the Python environment.

The `@io` decorator (Listing 6.5) attempts to solve this by wrapping a function into a *run* method on an `Io` thread object. This `Io` thread object is created on-the-fly and yields execution to the scheduler after starting the thread. When the thread's run method finishes, the return value is saved and the calling co-routine is rescheduled.

Listing 6.5: Example of IO wrapper

```
@io
def wait(seconds):
  time.sleep(seconds)
```

The idea of delegating a blocking system call to a separate thread was presented by Barnes [22] for the Kent Retargetable occam-π Compiler. occam-π implements a set of channels `keyboard` and `screen` that can be used to communicate to processes reserved for these IO operations. This approach could also be an option for PyCSP, but it would break when using third-party Python modules. Another alternative would be to overwrite internal IO functions with a set of functions that yield co-routine execution while waiting for IO.

Both alternatives have been deemed too comprehensive and not feasible for Python, thus it is required that IO calls are wrapped using the **`@io`** decorator when running with the **`pycsp.greenlets`** implementation.

6.2.7 Visualisation of PyCSP traces

The traces provided by the trace module in PyCSP are not traces as defined by Hoare [46]. It is a log of events that can be used to replay the process events. A trace of events that may help finding performance bottlenecks in communication, load balancing and contention. From this trace of events it is possible to show a visual representation of active processes and communication. Figure 6.4 shows a screenshot from a playback of a PyCSP application. The trace module only records the logic order of events and not the exact order, thus two concurrent events independent of each other may come in any order.

Listing 6.6: Example of how to enable PyCSP tracing

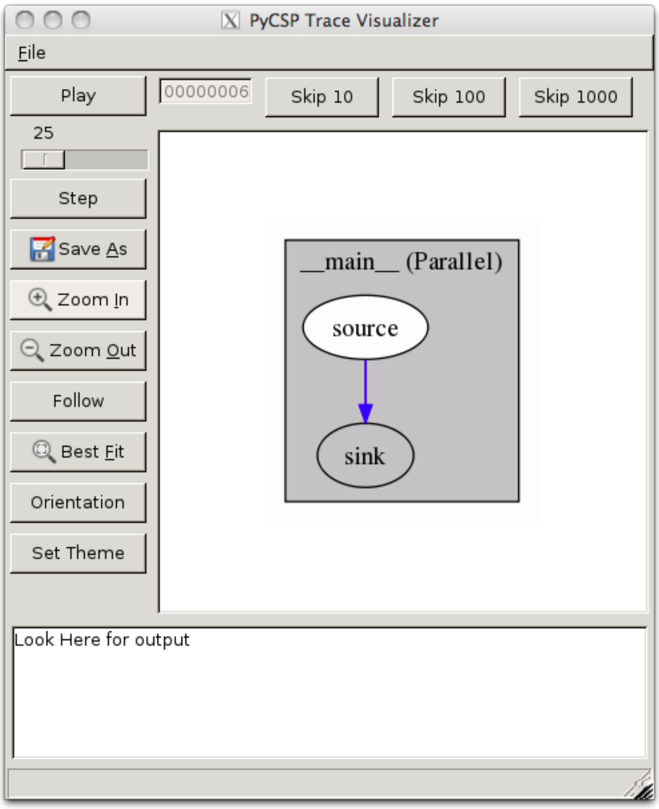
```
import pycsp.threads as pycsp
import pycsp.common.trace as pycsp

pycsp.TraceInit("output.trace")

@pycsp.process
def source(chan_out):
    chan_out("Hello world!")
    pycsp.retire(chan_out)

@pycsp.process
def sink(chan_in):
    while True:
        chan_in()

chan = pycsp.Channel()
pycsp.Parallel( source(-chan), sink(+chan) )

pycsp.TraceQuit()
```

The code in Listing 6.6 outputs the trace file in Listing 6.7. Every line is formatted in JSON [79] syntax. The trace output can be opened and played back using the PlayTrace tool shown in Figure 6.4.

Listing 6.7: The trace output from executing the example in Listing 6.6

```
{’chan_name’: ’UNIQUEID1’, ’type’: ’Channel’}
{’chan_name’: ’UNIQUEID1’, ’type’: ’ChannelEndWrite’}
{’chan_name’: ’UNIQUEID1’, ’type’: ’ChannelEndRead’}
{’processes’: [
  {’func_name’: ’source’, ’process_id’: ’UNIQUEID2’},
  {’func_name’: ’sink’, ’process_id’: ’UNIQUEID3’}
], ’process_id’: ’__main__’, ’type’: ’BlockOnParallel’}
{’func_name’: ’source’,
  ’process_id’: ’UNIQUEID2’, ’type’: ’StartProcess’}
{’func_name’: ’sink’,
  'process_id': 'UNIQUEID3', 'type': 'StartProcess'}
{'process_id': 'UNIQUEID2',
  'chan_name': 'UNIQUEID1', 'type': 'BlockOnWrite', 'id': 0}
{'process_id': 'UNIQUEID3',
  'chan_name': 'UNIQUEID1', 'type': 'BlockOnRead', 'id': 0}
{'process_id': 'UNIQUEID3',
  'chan_name': 'UNIQUEID1', 'type': 'DoneRead', 'id': 0}
{'process_id': 'UNIQUEID2',
  'chan_name': 'UNIQUEID1', 'type': 'DoneWrite', 'id': 0}
{'process_id': 'UNIQUEID3',
  'chan_name': 'UNIQUEID1', 'type': 'BlockOnRead', 'id': 1}
{'process_id': 'UNIQUEID2',
  'chan_name': 'UNIQUEID1', 'type': 'Retire', 'id': 1}
{'func_name': 'source',
  'process_id': 'UNIQUEID2', 'type': 'QuitProcess'}
{'func_name': 'sink',
  'process_id': 'UNIQUEID3', 'type': 'QuitProcess'}
{'process_id': 'UNIQUEID3',
  'chan_name': 'UNIQUEID1', 'type': 'Retire', 'id': 1}
{'processes': [
  {'func_name': 'source', 'process_id': 'UNIQUEID2'},
  {'func_name': 'sink', 'process_id': 'UNIQUEID3'}
], 'process_id': '__main__', 'type': 'DoneParallel'}
{'type': 'TraceQuit'}
```

In order to allow the tracing of the different PyCSP implementations it was necessary to be able to collect the logged events from threads, processes and remote hosts. The task of collecting events has been implemented using PyCSP, thus when tracing a PyCSP application another ”hidden” PyCSP network is created. Figure 6.5 shows the CSP network including the ”hidden” channels and processes. Here, the sink and source processes are reporting events using the ”hidden” CSP channel connected to the Convert2Str process. These events are then streamed to a file. The hidden PyCSP network uses the same process implementation as the main application, thus if a networked PyCSP application is being traced all events will be collected at the main node and streamed to a single file.



Look Here for output

Figure 6.4: The tool for visualising PyCSP traces plays back the trace output shown in Listing 6.7

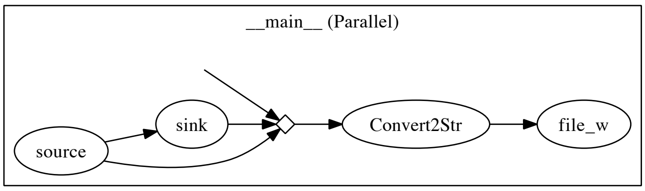


Figure 6.5: The CSP graph for the tracing of the PyCSP application in Listing 6.6

6.3 Micro Benchmarks

The results of these micro benchmarks provides a detailed view of how PyCSP behaves when stressed. The benchmarks are designed with the purpose of measuring the channel communication time including the necessary time required to context switch. Extra unnecessary context switches may be added by the operating system which is related to the PyCSP implementation used.

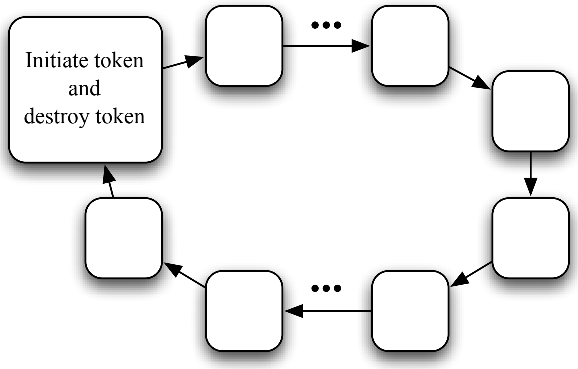


Figure 6.6: Ring of variable size

Using the ring design in Figure 6.6 we run a benchmark that sends a token around a ring of increasing size. The ring benchmark was inspired by a similar micro benchmark in [81]. N elements are connected in a ring and every element passes a token from the previous element to the next. This challenges the PyCSP implementation's ability to handle an increasing amount of processes and channels. The time measurements do not include startup and shutdown time and each measured run is divided by the size of the ring in order to compute an average channel communication time.

The test system has been tweaked to allow a larger number of threads and processes than the default. The results for our test system (in Figure 6.7) show that we can reach 512, 16384 and 262144 CSP processes depending on the PyCSP implementation used. It is obvious that `pycsp.processes` should only be used for applications with few CSP processes because of the exponential increase in latency. As expected, `pycsp.greenlets` is able to handle a large number of CSP processes with only a small decrease in performance.

Investigating the performance in a different perspective we use four rings of static size N and then send 1 to N-1 tokens to cycle concurrently. In the previous benchmark there was only one active communication at any given time, which is a rare situation for an actual application. With this benchmark we see `pycsp.processes` performs much better, since it can now utilise more cores. Based on the results in Figure 6.8

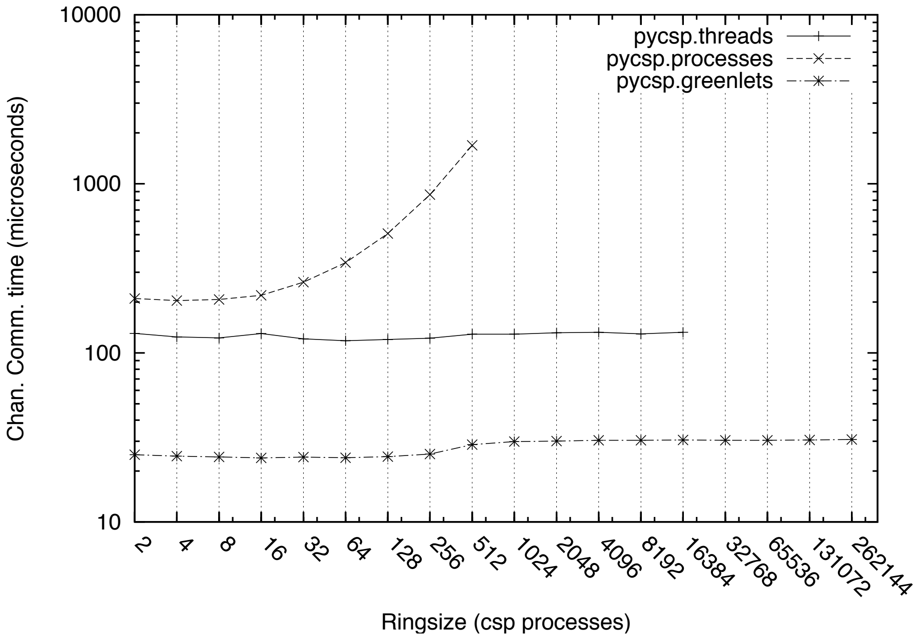


Figure 6.7: Micro benchmark measuring the channel communication time including the overhead of context switching for an increasing number of CSP processes

we can conclude that `pycsp.processes` has a higher throughput of channel communications than `pycsp.threads` when enough concurrent communications can utilise several cores.

Looking at the results for the four rings of size N in Figure6.8, an interesting pattern is observed whenever the number of concurrent tokens comes close to N. For N-1 concurrent tokens the performance of `pycsp.threads` is almost equal to the performance of one concurrent token. This behaviour is explained by the blocking nature of CSP, because when all processes but one has a token, then only this one is able to receive. This behaviour mimics the behaviour of the test with one token and explains why the results in Figure 6.8 are mirrored around the center.

From these micro benchmarks we can see that `pycsp.threads` performs consistently in both benchmarks. `pycsp.processes` does poorly in Figure 6.7 where the cost of adding more processes is high, but performs better in Figure 6.8 where a number of concurrent tokens are added. Finally `pycsp.greenlets` has proven able to do fast switching and handle many processes, regardless of the amount of concurrent tokens.

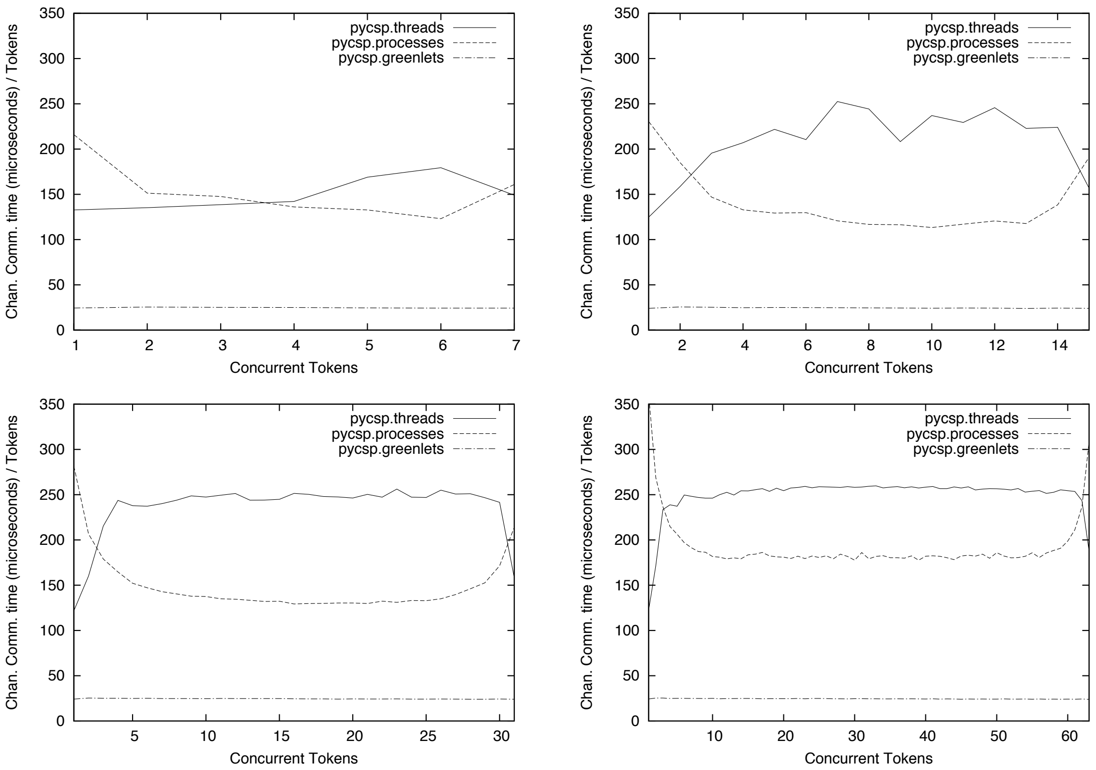


Figure 6.8: Micro benchmarks measuring the average channel communication time including the overhead of context switching for an increasing number of concurrent tokens in four rings of size 8, 16, 32 and 64

6.4 Rapid Development of Scalable Scientific Software

To demonstrate the flexibility and diversity of target architectures that PyCSP supports, this section describes three different applications using PyCSP to enable concurrent execution. The chosen problems are: Stochastic minimum search[48], k-nearest-neighbour search[35] and McStas[54]. They show three different usages of PyCSP; a master-worker design, a ring design and a network that combines master-worker with a set of utility processes. The work in this section has been published in [39] and the paper is included in Appendix A.7.

All PyCSP solutions to the mentioned problems are benchmarked and compared with an optimal solution. For the stochastic minimum search and *k*NN this means that we compare the speedup with a sequential solution written in C, thus the speedup from the PyCSP solutions will include the overhead of PyCSP, Python and the ctypes Python module for switching between Python and C.

McStas is an entirely different challenge and comes with a Perl wrapper for a large code-base that contains C code and supports MPI. For McStas we replace the Perl wrapper with our own PyCSP wrapper and use channel communications instead of MPI. The PyCSP solution for McStas is compared with the packaged solution using MPI for inter-process communication.

The applications are benchmarked on three different systems: A multi-core system with two Intel Xeon E5310 (8 cores total), a cluster system with eight Intel Core 2 Quad Q9400 (32 cores total) interconnected with one gigabit ethernet and an Intel Core 2 Duo 2.4 Ghz (2 cores total) with workers running in a grid system on various hardware. The stochastic minimum search and *k*NN are benchmarked on the eight-core system and the cluster system. McStas is benchmarked on the eight-core system and on the dual-core system with workers running in a grid system.

Every benchmark was executed three times and the average execution time was used for computing the speedup plots. The sequential C reference benchmark used for computing the speedup plots was executed on all architectures, making the speedup plots comparable.

6.4.1 Stochastic Minimum Search

The PyCSP network for the stochastic minimum search[48] takes a function as input and uses a Monte Carlo approach to produce a suggestion for a global minimum value of that function within a user-specified window. The Monte Carlo algorithm is run in parallel in a set of processes. Each process runs independently of the others and tests internally for a new minimum local to this process. Whenever a new local minimum is found it is sent to a master. The master feeds a filter that decides when a result is a new best global minimum and decides when to terminate the remaining network. A subset of the code required to create the network in Figure 6.9 is shown in Listing 6.8

to demonstrate what lines are necessary to create a parallel stochastic minimum search.

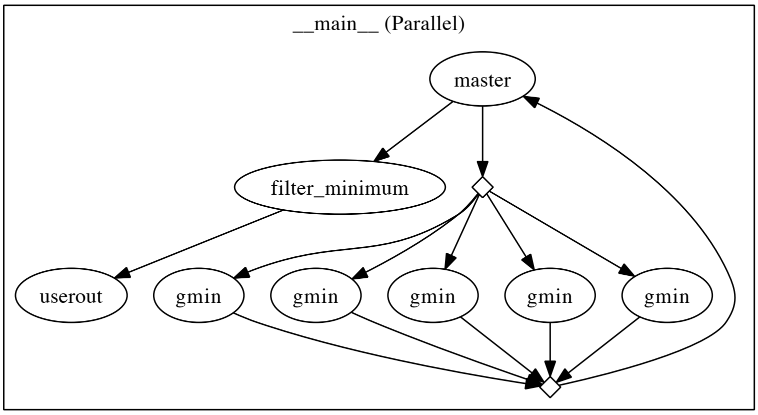


Figure 6.9: An extracted snapshot of the stochastic minimum search CSP network with five processes executing the Monte Carlo algorithm

Listing 6.8: Source code for the stochastic minimum search application where the `userout` and `filter` processes have been omitted. Figure 6.9 shows a visualised trace of this application.

```
@process
def gmin(chin,chout, loops):
  fname = chin()
  res = min( [compute_in_C() for j in xrange(loops)] )
  chout(res)
  retire(chin, chout)

@process
def master(filter, workers_o,
           workers_i, n_workers, f):
  for i in range(n_workers): workers_o(f)
  while True: filter(workers_i())

to_worker=Channel();
from_worker=Channel()

Parallel(
  master(<filter channel>,
    to_worker.writer(), from_worker.reader(),
    nprocs, <target function> ),
  gmin(to_worker.reader(), from_worker.writer(), loops) * nprocs
)
```

The results from running the application on a multi-core and a cluster system are presented in Figure 6.10. The extra overhead introduced by Python and PyCSP is the reason why we only get a speedup of 0.6 for the multi-core execution with 1 worker process. This overhead is caused by the application that continuously sends new local results, which in turn add channel communication and thus require the processes to switch between Python and C. The cluster execution adds an extra overhead for network communication, thus the speedup is 0.5 with 1 worker process. For 8 worker processes the speedup compared to a sequential C solution is 5.0 for an 8-core host. On an 8-node cluster with 16 workers we get a speedup of 9.6. The speedup improves as searches run for longer periods, since the amount of channel communication drops, as the occurrence of new minimums decreases.

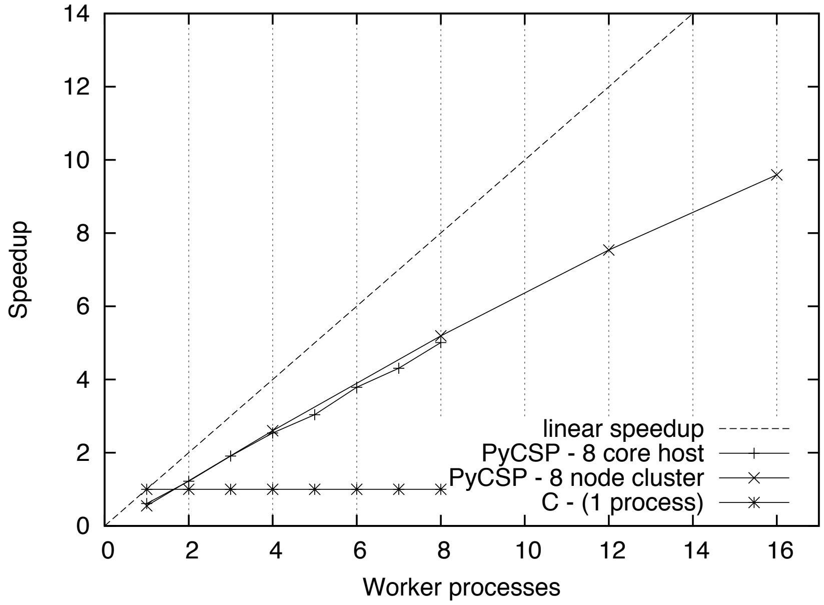


Figure 6.10: Speedup of stochastic minimum search. The sequential C implementation executed 500.000 approximations in 38.4 seconds (avg. of three) on the 8 core host and 5.000.000 approximations in 221.2 seconds (avg. of three) on the 8 node cluster.

6.4.2 k Nearest Neighbour Search

The *k* Nearest Neighbour search problem[35] consists of finding the *k* nearest neighbours of each target point in a set of *N* targets with *D* dimensions. This is used in machine learning for finding the *k* nearest neighbours to a sample among a large set of positive and negative targets. If the *k* nearest neighbours to a sample are primarily positive, then the algorithm would give the result that a sample is positive together with a likelihood value. Any measurement of the distance between each target can be used, but for this benchmark we have used the Euclidian distance in the space with D dimensions. To compute the distances the brute force approach is used which has the complexity of $O(N^2D)$ but is easy to parallelise. To compute in parallel the targets are split into T sets and divided among T worker processes connected in a ring. The CSP network in Figure 6.11 shows the workers connected in a ring, where every worker is given a local set which it will pass around the ring. In each pass the worker executes a *k*NN algorithm on the local set and the received set, until finally all results are collected and joined to the end result. Listing 6.9 shows the code for creating the ring network and how to compute the result for each worker. The ring approach is too fine-grained for a grid architecture, but well suited for a closely connected parallel architecture such as cluster computers.

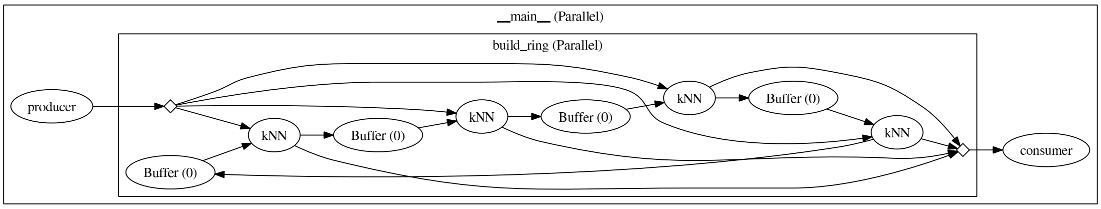


Figure 6.11: *k* Nearest Neighbour search with four worker (`kNN`) processes connected in a ring

In Figure 6.12 we have plotted the speedup for computing the 5 nearest neighbours in 72 dimensions in a set of 10.000 targets (8 core host) and a set of 60.000 targets (8 node cluster). For the 8 core host we get a speedup of 9.0 with 8 workers, which is caused by the better cache usage when the local set-size for each worker gets smaller. The poorer performance for the 8 node cluster can be explained by the necessary transfers of the actual arrays, and the associated serialisation, but a reasonable speedup of 12.4 is still achieved for 16 workers.

Listing 6.9: Source code for the *k* Nearest Neighbour search application where the producer and consumer processes have been omitted. Figure 6.11 shows a visualised trace of this application.

```
@process
def build_ring(proc, N, B, pargs):
  channels=Channel(buffer=B) * N
  processes=[]
  for i in range(N):
    processes.append(proc(*pargs,
      ring_in=channels[i-1].reader(), ring_out=channels[i].writer(),
      ring_size=N, ring_id=i))
  Parallel(processes)
```

```
@process
def kNN(job_ch, result_ch, D, k,
        ring_in=None, ring_out=None,
        ring_size=0, ring_id=0):

  # channel -> channel end
  job=job_ch.reader()
  result=result_ch.writer()

  work=data=job()

  best = numpy.zeros((len(data), k))
  for i in range(ring_size):
    # Invoke a C impl.
    kNN_in_C(data, work, best, D, k)
    ring_out(work)
    work=ring_in()
  result(best)
  retire(result)

job_ch=Channel(); result_ch=Channel()
Parallel(
  producer(<N*D input array>, 4, job_ch.writer()),
  build_ring(kNN, size=4, buffer=1, (job_ch, result_ch, D, k)),
  consumer(result_ch.reader()))
```

6.4.3 Neutron Scattering Simulation

Neutron-based imaging is a powerful tool in several sciences, including solid state physics and biology, where neutron imaging is used to produce high resolution non-intrusive images of samples. However, neutron-based imaging is not as simple as using an optical microscope or an x-ray imaging, and before a neutron image is produced the imaging process must first be simulated to tune several parameters. This simulation is hugely time-consuming and the quality of the simulation is often dictated by the available time for simulation. The de-facto standard tool for such simulations is McStas[54] performing Monte Carlo simulations of neutron instruments. McStas comes with a complicated Perl script that enables parallel execution with MPI. We replace this Perl script with PyCSP to use a process-oriented model that executes on any number of grid resources through many levels of parallism, and finally we merge the results and save them for the user. The directed graph (Figure 6.13) is generated from a trace of an actual execution. This is made possible by the compositional nature of CSP which is easily traced and visualised using PyCSP.

The first task is to load the description of the experiment in a domain specific lan-

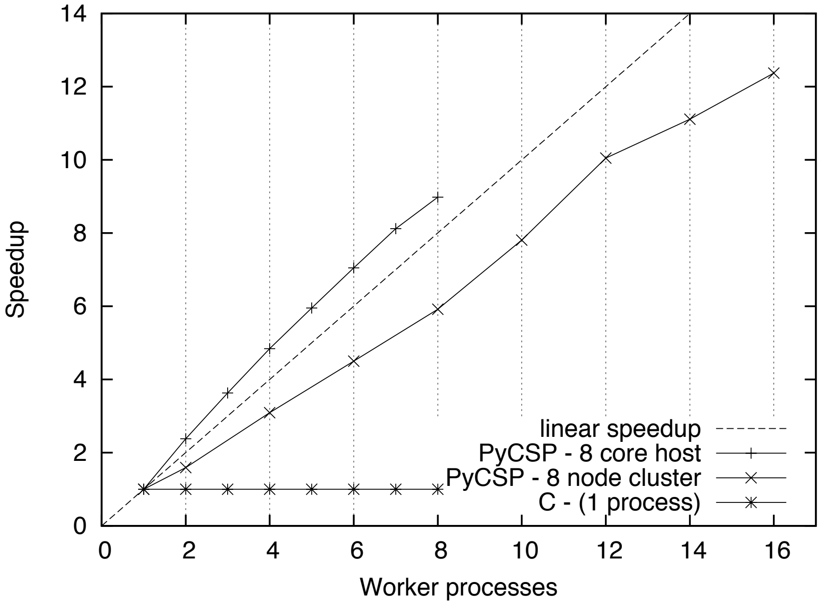


Figure 6.12: Speedup of *k* Nearest Neighbour search (*k*NN). The sequential C implementation computed the result for 10.000 bodies in 23.6 seconds (avg. of three) on the 8 core host and for 60.000 bodies in 442.9 seconds (avg. of three) on the 8 node cluster.

guage. This language is then compiled into a C source file which in turn is compiled into an executable. This two-phase compilation is quite demanding and requires a non-trivial installation of the McStas package.

To improve the productivity of the user we have enabled the possibility to move the compilation to a service resource by putting it into a separate process. This allows the user to configure the simulation without having to install McStas on the local computer. The resulting executable is passed on to the worker (`simulate`) processes that run the simulation numerous times over a set of parameters. The final merged result is sent back to the user for presentation.

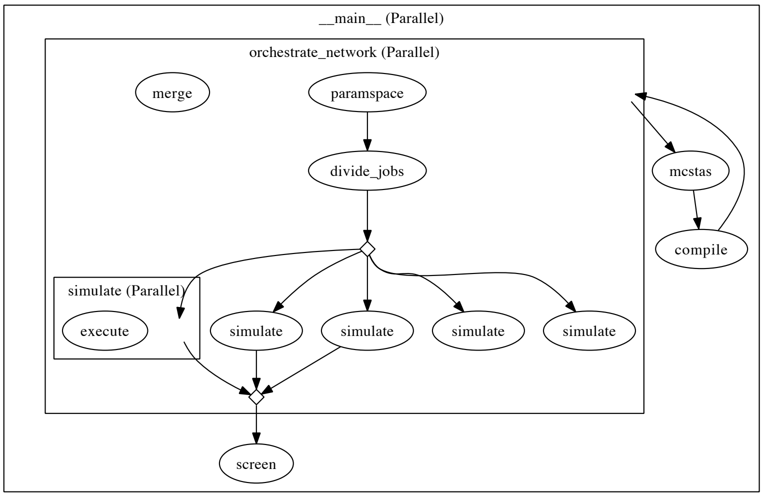


Figure 6.13: Neutron scattering simulation with five worker (simulate) processes. A dynamic orchestration of workload is in place to be able to adapt to uneven resources.

In Figure 6.14 we compare the MPI-enabled Perl wrapper with PyCSP. The PyCSP speedup suffers slightly from the extra overhead of handling jobs. For the PyCSP executions the work was divided into 50 jobs.

The benefit of having a dynamic orchestration of workload becomes apparent when the executing resources differ. This is the case in grid computing, and by changing the configuration of the worker processes as described in Listing 6.10 we can utilise many more resources.

When executing a PyCSP network with processes submitted to a grid system the workload must be split dynamically to cope with inactive processes, since it is unknown when a worker process might be moved from 'queued' to 'executing' status. One execution might experience that only half of the workers are executing the entire simulation while another execution will have all the workers in 'executing' state almost immediately.

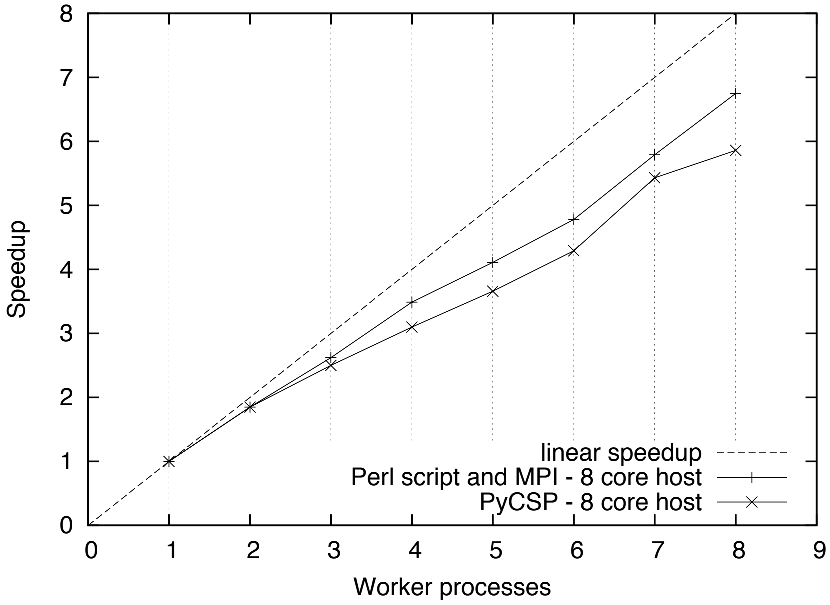


Figure 6.14: Speedup of neutron scattering simulation (McStas). The sequential execution simulated 50.000.000 rays in 61.2 seconds (avg. of three) on the 8 core host.

Listing 6.10: Worker (simulate) process. The @grid_process decorator configures the process to behave as a grid process.

```
@grid_process(vgrid='DCSC', cputime=600)
def simulate(job_in, result_out, exec_file):
  while True:
    ncount, params = job_in()
    cmd=tuple(['./' + exec_file,
               '--ncount=' + str(ncount),
              ] + params )
    Parallel(
      execute(cmd, retire_on_eof=False)
    )
    output=open('mcstas.sim').readlines()
    result_out(output)
```

We have performed two large neutron simulations where one was performed on a dual core laptop and the other was run from the same laptop but with up to 64 worker (simulate) processes running in a grid system. From the execution times in Table 6.2 we demonstrate that the simulation finished in under 7 minutes instead of 1.5 hours.

The results produced by the three applications are not expected to scale linearly, but even a decent speedup is a good result, since the cost for producing the PyCSP application is limited and the scientist is still involved directly in the programming. It is

Table 6.2: Execution times for running McStas using PyCSP and grid computing. These numbers are indicative for grid executions but will vary greatly due to large variations in resources, size of job queue and grid overhead.

System	Simulated Rays	Time (seconds)
dual core laptop with 2 workers	$2.5 * 10^9$	5928
grid with up to 64 workers	$2.5 * 10^9$	411
grid with 1 worker (expected overhead)	1	43

expected that all three applications scale to at least 64 workers for larger problem sizes.

The three applications presented in this section have demonstrated that PyCSP can be used to model a scientific workflow, can use compilable languages for computational-intensive parts and that, given enough concurrent processes, PyCSP is able to execute such workflows on parallel architectures with an acceptable speedup.

6.5 The Dynamic Channel

Section 6.4 presented PyCSP as a means of creating scalable scientific software. The scientific users of the PyCSP library should not have to think about whether they might be sending a channel-end to a process that might be running in a remote location. Or how they avoid using external choice on channels, that do not support external choice. One of the powerful characteristics of CSP is that every process is isolated, which means that we can move it anywhere and as long as the channels still work, the process will execute correctly. The network-enabled PyCSP implementation is a prototype and uses a single channel server to handle all channel traffic. The single channel server runs the thread implementation of PyCSP internally, which creates a temporary thread for every request. The server is a serious bottle-neck for the channel communication and has limited the type of parallel applications implemented using PyCSP.

The interface of the dynamic channel resembles a single channel type. The idea is that when the channel is first created it may be an any-to-any channel specialised for co-routines. The channel is then upgraded on request, depending on whether it participates in an alt (external choice) and on the number of channel-ends connected. The next synchronisation level for the channel may be an optimised network-enabled one-to-one channel with no support for alt. Every upgrade stalls the communication on the channel momentarily while all active read or write requests are transformed to a higher synchronisation level. The upgrades continue until the lowest common denominator (a network-enabled any-to-any channel with alt support) is reached. The network-enabled any-to-any channel uses the distributed synchronisation model (Section 6.5.1) that does not suffer from the bottle-necks of the former implementation.

The SPIN model checker [47] has been used to check the correctness of the distributed channel model and the transition model. Both models are described in detail in [42] (Appendix A.8). In 1986 Vardi and Wolper [97] published the foundation for SPIN; an automata-theoretic approach to automatic program verification. SPIN can verify a model for correctness by generating a C program that performs an exhaustive verification of the system state space. During simulation and verification SPIN checks for the absence of deadlocks, livelocks, race conditions, unspecified receptions and unexecutable code. The model checker can also be used to show the correctness of system invariants, find non-progress execution cycles and linear temporal constraints, though these features have not been used here.

6.5.1 Distributed Channel Synchronisation

Communication between distributed processes is particularly challenging when implementing any-to-any channels with support for both input and output guards. To ensure the robustness and scalability of the implementation the distributed channel synchronisation is based on the local channel synchronisation (Section 6.2.5).

The local channel synchronisation has a process waiting until a match has been made. The matching protocol performs a continuous testing for all pairs, thus the state of a waiting process is constantly being tried through shared memory. This method is not possible in a distributed model with no shared memory, instead an extra process (a process state server) is created to function as a remote lock, protecting updates of the posted channel requests. Similar to the local channel synchronisation, it is necessary to ensure the critical region that includes all processes depending on a selection for a channel and retrieve the current process state from the two processes being tried. When a match is found both processes are notified and their process states are updated through the process state server. The state of the channel is also moved to a separate process, to allow channels to post read and write request directly to a channel home. The location of a channel must be known or found through a discovery service.

The CSP network (in Figure 6.15), consisting of five processes connected by four channels, can now be distributed to multiple hosts by using the distributed channel synchronisation model (see [42] in Appendix A.8). Figure 6.16 shows the internal process dependencies when the CSP network (in Figure 6.15) is distributed on two hosts. The thick gray lines in Figure 6.16 show the active dependencies in a communication between processes P2 and P3. In any communication only the affected channel home server and the process state servers are activated, thus there are no bottle-necks for concurrent communication between independent processes. A SPIN model of the dis-

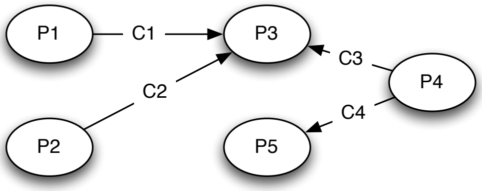


Figure 6.15: Sample CSP network

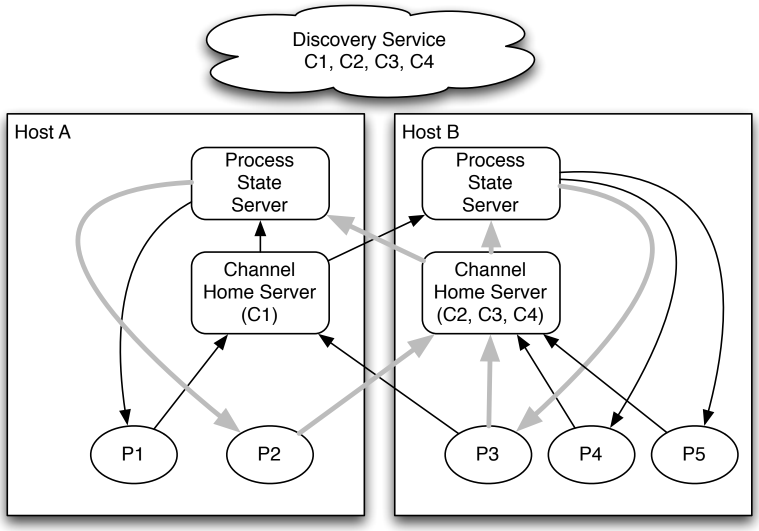


Figure 6.16: Process dependency graph of the CSP network (Figure 6.15) distributed on two hosts. The thick gray arrows show the active dependencies in a communication on C2 between P2 and P3.

tributed channel synchronisation is presented and model-checked successfully in Appendix A.8.

6.5.2 Dynamic Synchronisation Layer

The dynamic synchronisation layer allows channels to change the synchronisation mechanism on-the-fly. This means that a local channel can be upgraded to become a distributed channel. Activation of the upgrade may be caused by a remote process requesting to connect to the local channel. The model is presented and model-checked successfully in Appendix A.8.

A feature of the dynamic synchronisation mechanism is that specialised channels, such as a low-latency one-to-one channel, can be used, resulting in improved communication time and lower latency. The specialised channels may not support constructs such as the alt operation, but if an alt operation occurs the channel can be upgraded. The upgrade procedure adds an overhead, but since channels are often used more than once this is an acceptable overhead.

Given the process layout in Figure 6.16 the use of a distributed channel is unnecessary and slow for the communication between processes P3, P4 and P5. If the scientific user explicitly used a local channel for that link it would not be possible to use the alt operation across multiple implementations. Processes P3, P4 and P5 must always run in the same interpreter instance and termination signals must be handled explicitly. Using the dynamic synchronisation layer it is possible to change the process dependency graph from Figure 6.16 to Figure 6.17.

The switching of synchronisation level works by notifying all processes which have posted a communication request to the channel. The channel level is changed before notifying processes. When a process either tries to enter wait or is awoken by the notification it checks that the level of the posted request still matches the level of the channel. If these levels do not match the transition is activated. During a transition the process state is temporarily changed to **SYNC**, such that the request is not unintentionally matched by another process.

The models presented in Appendix A.8 can be used separately for new projects or can be combined to the following: a CSP library for a high-level programming language where channel ends are mobile and can be sent to remote locations. The channel is automatically upgraded, which means that the communicating processes can exist as co-routines, threads and nodes. Specialised channel implementations can be used without the awareness of the communicating processes.

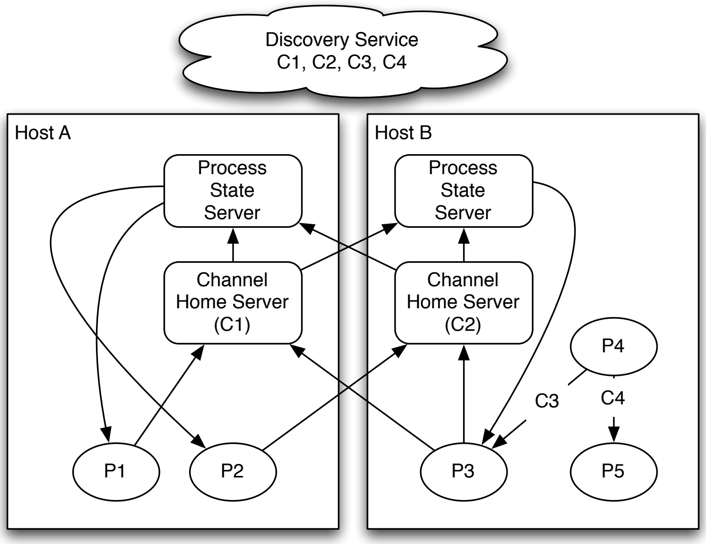


Figure 6.17: Process dependency graph of the CSP network (Figure 6.15) distributed on two hosts. The dynamic channel synchronisation has upgraded the synchronisation mechanism for the channels C1 and C2, while C3 and C4 can continue to communicate through shared memory.

Chapter 7

Future Work

This thesis presents PyCSP as a way for scientific users to create concurrent Python applications, while maintaining a scientific workflow organisation. This Future Work section is focused around issues that, if solved, would improve PyCSP for the scientific user.

Models for channel synchronisation have been model-checked successfully in [42] using the SPIN Model Checker [47]. This has shown that there are no dead-locks and no race-conditions in the models, but it has not shown that the models were equivalent to the CSP channels defined by Hoare [45]. Thus the equivalence between the dynamic channel presented in this thesis and CSP channels, as defined in the CSP algebra, needs to be verified, as it has been verified for JCSP [102].

7.1 The Dynamic PyCSP Library

The PyCSP synchronisation mechanisms presented in ([42] and Appendix A.8) are the building blocks for a new PyCSP library, with a channel implementation that can start out as a local channel and evolve into a distributed channel spanning multiple nodes. This channel implementation will support mobility of channel ends, scheduling of lightweight processes, distributed processes and a lookup service for locating channel homes. In this section it is discussed how these and other interesting functionalities can be achieved. Mobility of channel ends is simple when all channels can be upgraded to a distributed channel synchronisation mechanism. The only requirement is a lookup service for locating a channel home. Initially this lookup service may be a simple name server, providing the location of a channel home based on the channel id or name. If a channel home has moved the name server must be updated and the old location must be able to inform a channel request that the channel has moved.

I suggest selecting co-routines as the primary process type. Co-routines makes it possible to allow for very large process networks and low overhead for communication between co-routines sharing a scheduler. Using PyCSP it must be possible to benefit from the concurrent properties by allowing parallel execution. Thus it is necessary to distribute co-routines to a process type that may execute multiple co-routines in parallel. The options for Python are OS threads or OS processes, but since parallel execution of threads is limited by the Global Interpreter Lock in Python, OS processes are the natural choice. Co-routines distributed to multiple threads must also be able to communicate on channels. Because of the transition layer, the co-routine channel can dynamically be upgraded to a network-enabled channel. Figure 7.1 shows a process dependency graph of the CSP network in Figure 6.15, where the co-routines have been distributed to multiple OS processes, having one scheduler per OS process.

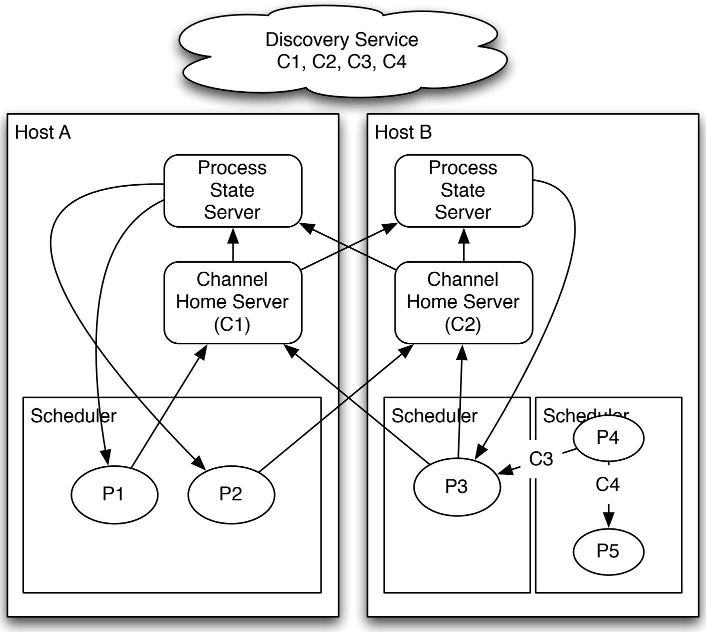


Figure 7.1: Process dependency graph of the CSP network (in Figure 6.15) distributed on two hosts. The processes $P1, \ldots, P5$ are co-routines and split between OS processes to allow parallel execution. A scheduler is handling the scheduling of co-routines inside an OS process.

C++CSP2 [27] is another CSP library allowing a mixture of user-level and kernel-level threads. C++CSP2 uses the kernel-level threads as the default process model and has user-level threads as an option provided through the API. There are several reasons for this choice: To allow the parallel execution of multiple processes by default. To avoid novice users accidentally blocking the user-level scheduler through a blocking IO call. The OS scheduler is capable of balancing the load of kernel-level threads onto the cores of a multi-core processor.

For PyCSP co-routines (greenlets) should be the primary process implementation. Since all processes are co-routines any calls to blocking IO would have to be wrapped in the IO decorator (Section 6.2.6). Standard operating systems are not made to handle large amounts of OS processes, thus PyCSP would quickly be bounded by the OS if the default process implementation was OS processes. To allow parallel execution co-routines must be distributed to multiple OS processes (see Section 7.1.1).

7.1.1 Distributing Co-routines

This section exclusively concerns the challenge of how to balance the load of processes (co-routines) between OS processes with the purpose of gaining from running on a multi-core system or distributed system. The challenge is the same for OS processes running on different hosts as the channel communication between OS processes uses the distributed synchronisation mechanism (Section 6.5.1).

Each process is a black box, thus the only perfect solution is to distribute exactly one CSP process per OS process. This requires as many OS processes as CSP processes and forces all channels to be network-enabled. Finding the optimal distribution is in the class of optimisation problems, but to benefit from optimisation algorithms weights must be applied to processes and channels. Weights can be approximated by running a profiling execution on a reduced set of input data, but such an approach is very impractical and the weights may be very different from the weights based on real data. Another method is to compute the weights during execution and continually rebalance the distribution of CSP processes, but this requires mobile CSP processes (see Section 7.1.1.2). A simpler approach is to record the utilisation for each of the available resources and place new CSP processes on the least busy resources. This would also allow for multiple users of the same cluster system to automatically share the resources. The resources would be determined by a host list, e.g. containing the nodes in a cluster system or eight entries of the local system.

As we have no knowledge of the weights for processes and channels the suggested solution is to annotate the Parallel, Spawn and Sequence construct with hints as to how the CSP processes should be distributed. Such as:

```
Parallel(processes..., hint=[local | strided | blocked | auto])
Spawn(processes..., hint=[local | strided | blocked | auto])
Sequence(processes..., hint=local)
```

The hints would have the following behaviour: `local` – Run CSP processes in current OS process. `strided` – Distribute CSP processes in a round-robin fashion to the resources. `blocked` – Distribute CSP processes in a blocked fashion, by cutting up the list into chunks and place each chunk on a resource. `auto` – This would randomly distribute the CSP processes to a set defined by the least busy resources. The `auto` hint would be default for the Parallel and Spawn constructs, but does not provide any guarantee of an optimal distribution.

Scientific applications need to access data files, interface with users and output results. All of these actions are tied in to a location, thus the constructs must also be able to specify a host location and thus override any hints.

```
Parallel(processes..., host=[hostname/ip])
Spawn(processes..., host=[hostname/ip])
Sequence(processes..., host=[hostname/ip])
```

To assist the scientific user the PyCSP trace module (Section 6.2.7) should support tracing the location and the CPU time for processes and display it in the PlayTrace application (Figure 6.4). Users may then use this information to update Parallel and Spawn constructs with specific hints if the current distribution of processes is unwanted.

7.1.1.1 CSP Topologies

The process patterns described in [84] solves many basic concurrency issues. For parallel computing where the desire is to spread the workload on multiple processes there are a set of patterns that are particularly useful. These are line, ring, tree, star, mesh and torus topologies and are commonly known in scientific computing. Such topologies can be created using processes and channels in PyCSP, but by providing a special set of topology constructs the use of such constructs provides additional information on how processes should be distributed. Additionally, it simplifies the creation of topologies for the user.

The kNN application code (in Listing 6.9) uses a ring topology for computation. This ring topology is created using the process in Listing 7.1

Listing 7.1: Process constructing a ring topology

```
@process
def build_ring(proc, N, B, pargs):
  channels=Channel(buffer=B) * N
  processes=[]
  for i in range(N):
    processes.append(proc(*pargs,
      ring_in=channels[i-1].reader(),
      ring_out=channels[i].writer(),
      ring_size=N, ring_id=i))
  Parallel(processes)
```

The `@ring` decorator (Listing 7.2) is an example of how a special ring construct may simplify creating advanced process patterns. The `@ring` adds hints to the process distribution to facilitate a higher utilisation of available computing resources.

Listing 7.2: Ring decorator constructing a ring topology

```
@ring(N, B)
def proc(pargs, ring_in,  ring_out, ring_size, ring_id)
  # ring node code
```

7.1.1.2 Mobile Processes

Mobile processes is the corner stone of the π-calculus [70] and enabling mobile processes for PyCSP would broaden the possibilities. occam-π is currently the only CSP implementation supporting mobile processes [21].

This is the situation for the future PyCSP library: Every PyCSP process is a greenlet [93] (co-routine). The greenlets may be distributed to multiple OS processes or hosts. Distributing the load of greenlets evenly on computing resources requires hints. If a greenlet could be mobile it would be possible to redistribute greenlets in case the load was unevenly distributed.

The Python Interpreter PyPy [80] (Section 5.1) provides a greenlet implementation which allows running greenlets to be paused and serialised [12]. A serialised greenlet instance is represented by a string that can be sent to another intrepreter instance where it is unserialised and resumed. The interpreter is not yet stable, but it should be stable enough to do experiments with mobile processes for PyCSP. Mobile processes introduce a few problems, such as the active requests on channels that would also have to be suspended during a process move. However, this problem can be easily avoided by limiting the move of processes to the moment before a channel operation or after a channel operation. As all channels are able to dynamically upgrade to network-enabled channels and can be located through a lookup service it should not be necessary to reinitialise a channel connection until a new channel operation is initiated. The move of a mobile process can be activated through the explicit channel communication of a running process or it may be a joint decision between a set of schedulers.

7.1.2 Process Wrappers

The neutron scattering simulation presented in Section 6.4.3 used a `@grid_process` decorator, which wrapped a CSP process into a Grid job for execution in a Grid environment. Similarly, it is possible to create other wrappers. It would be interesting to experiment with a `@ssh_process` decorator and decorators for other Python interpreters, such as `@jython_process`, `@ironpython_process` and `@cython_process`. Code wrapped in a `@jython_process` and running in the Jython interpreter would be able to access Java libraries.

7.1.2.1 Emulating Greenlets

The greenlets library is only available for the primary Python intrepreter CPython and for the PyPy interpreter. To support other Python interpreters a fallback mechanism should load a greenlet emulator when the greenlets module is unavailable. Greenlets can be emulated using operating system threads and conditions. The overhead would be considerable, but for the purposes in Section 7.1.2 the overhead would still be acceptable.

7.2 Latency-hiding in a One-to-One Channel

In this section I suggest a specialised channel type: The network-enabled buffered one-to-one channel. The distributed channel synchronisation in Section 6.5.1 handles one-to-one communication, but performs poorly when communicating messages compared to a standard point to point socket connection. The buffered channel suggested should be semantically equivalent to a chain of CSP processes, except that it must be possible to disable and enable buffering on-the-fly. The reason is that the dynamic synchronisation layer in Section 6.5.2 provides the means of changing the channel synchronisation mechanisms during execution. However to support this the one-to-one buffered channel must be able to export the buffer content to a more relaxed channel synchronisation (e.g. the distributed channel synchronisation).

The latency-hiding of this one-to-one channel is obtained through the use of acknowledgement tokens that are given to the sender. This enables the sender to accept a message instantly by decreasing the tokens for every accepted message. Messages are transferred asynchronously to the receiver, and for every message which the receiver delivers a new token is sent to the sender asynchronously. If the sender runs dry of acknowledgement tokens it must block until new tokens have arrived. For the latency-hiding to work the channel must be enabled with a buffer large enough to hide the latency of the network layer.

Such a channel would allow scientific users to hide the latency of the network, even though such results are normally reserved for experienced programmers using double or triple-buffering techniques.

7.3 Forced Process Isolation

A common obstacle for non-trained programmers when implementing parallel applications is to avoid the use of global variables. I suggest adding a forced process isolation for PyCSP, where access to any global variable is equal to accessing a non-existing variable.

The standard library in the Python programming language provides the inspect module, which can extract source and byte code from the Python interpreter instance. This would allow a PyCSP process to extract the source code of the process body and reevaluate it using a different set of globals.

As some variables in Python are mutable, they should also be protected such that the sender process of a mutable variable do not access the variable after it has been sent on a channel. Here I suggest making a deep copy of a variable communicated on a channel.

7.4 Secure Channels

Many of the CSP libraries available do not include security measures to protect against intruders on a channel. PyCSP could easily be used on unprotected public networks, thus security measures should be developed for PyCSP. It should be investigated how exposed or unprotected channel communication could be moved to SSL (Secure Socket Layer).

7.5 Routing for Channel Communication

In distributing CSP networks it could be interesting to allow connecting hosts by a PyCSP channel, even though the hosts are not able to connect directly to each other. This case is especially relevant for mobile channel ends, as a channel end might, in rare cases, be sent to a CSP process which is not able to connect directly to a channel home.

7.6 Stability in case of Host Failure

The behaviour of a channel communication failure or a faulted process must be defined, such that failures can be handled. Rare errors are allowed to break execution while other errors, such as a host being disconnected should only cause problems if other processes are directly dependent on data from this host. The current PyCSP model breaks when any host included in a CSP network fails.

7.7 CSP for Popular Scientific Workflow Systems

The scientific workflow systems (Section 2.2) are mature, stable and have a large userbase, but more work is necessary to improve on their ability to execute scientific workflows on parallel architectures. This thesis have shown that CSP can be used for executing scientific workflows. Knime [88] and Taverna[49] are both very popular scientific workflow systems written in Java, it should be investigated whether it is possible to create a CSP network using JCSP [101] where each of the tasks of the scientific workflow is a single JCSP process.

Chapter 8

Conclusion

The compositional nature of CSP enables any researcher to take any process of a CSP network and optimise it in isolation, since all processes are isolated and have no side effects. This brings huge benefits on cost and time since dedicated programmers can be hired to optimise the bottle-necks in a workflow of any size. The scientific user still has control of the rest of the application since it remains unchanged from the original design. This results in an improved situation where the scientific user is still in control and able to directly make changes to a high performance application, even if it is running on multiple architectures and clusters at remote locations.

The close mapping between the graphical representation of CSP programs and the PyCSP source code makes it easy to compare design documents and implementations, helping scientific users manage the complexity that is often introduced by parallel architectures. The neutron scattering simulation example showed how PyCSP can be used to structure the execution of binaries into a concurrent Python application, which can be traced and the process network visualised for a better understanding.

Experiments have been made on different process implementations for the CSP process, resulting in a PyCSP where any application written in Python and using PyCSP can change the concurrent execution model from threads to co-routines or processes only by changing which PyCSP module is imported. Depending on a user's domain and application a user can choose to circumvent the CPython Global Interpreter Lock by using processes. Alternatively, a user may want to speed up the communication time by a factor of ten by using co-routines. Then again if the application is changed further and the user wants to return to using threads, this is a simple task that does not require the user to transfer code changes to an older revision.

The PyCSP with multiple process implementations had the issues that different CSP processes could not communicate on CSP channels between implementations and that the networked channel implementation did not scale at all. If PyCSP is to be used on a larger scale, these issues must be solved. Thus this thesis presents the building blocks for a dynamic CSP channel capable of transforming the internal synchronisation mechanisms during execution. The change in synchronisation mechanism is a basic part of the channel and can come about at any time. Python is a dynamic language and with the dynamic channel it is possible to create dynamic CSP networks where the location of processes is irrelevant, as any process is allowed to connect to any channel identified by id or name.

The SPIN model checker has been used to perform an automatic verification of three models separately, that together make the new dynamic channel. During verification it was checked that the communicated messages were transferred correctly using assertions. All models were found to verify with no errors for a variety of configurations with CSP networks. The full model of the dynamic channel has not been verified since the large state-space makes it unsuited for exhaustive verification using a model checker.

The results from model-checking additionally showed that the synchronisation mechanism in the current PyCSP [40, 98] was model-checked successfully.

The external choice in PyCSP now supports output guards in addition to input guards. This works with multiple readers and writers on a channel. The use of output guards is a heavily debated issue in CSP as they are clearly not needed nor trivial to implement. However, it is evident that the users of PyCSP find output guards a very convenient feature and considerable work has been put into supporting output guards in the external choice implementation in PyCSP.

In order to reduce the risk of race-conditions when using poison to terminate a CSP network this version of PyCSP introduces the concept of retirement from a channel. When all processes on one end of a channel retire their channel ends, the channel becomes retired. The effect is that the propagation of the retire signal is activated upon the termination of the last process at a given channel end rather than the first as with the poison operation.

Overall, the changes to PyCSP are well integrated and I believe that using PyCSP is now easier for the non-trained user than with the previous version. The flexibility of PyCSP has been demonstrated through executions in four different environments; single-core, multi-core, cluster and grid. The dynamic channel for PyCSP is a candidate to handling heterogeneous architectures effectively in a single application, since PyCSP is portable and will run on most architectures and operating systems.

Bibliography

[1] AMD Fusion APU - Accelerated Processing Unit. http://www.amd.com/fusion/. Viewed online July 2011.

[2] Cython: C-Extensions for Python. http://cython.org/. Viewed online August 2011.

[3] Intel Parallel Studio. http://www.intel.com/go/parallel. Viewed online July 2011.

[4] IronPython: Python for the .NET Framework. http://ironpython.net. Viewed online August 2011.

[5] Jython: Python for the Java Platform. http://www.jython.org. Viewed online August 2011.

[6] Kent retargetable occam compiler (kroc) / occam-π. http://www.cs.kent.ac.uk/projects/ofa/kroc/.

[7] Large Scientific Software Collection for Python. http://www.scipy.org/Topical_Software. Viewed online August 2011.

[8] MS Parallel Computing. http://msdn.microsoft.com/en-us/concurrency. Viewed online July 2011.

[9] NVIDIA CUDA Compute Unified Device Architecture. http://www.nvidia.com/cuda. Viewed online July 2011.

[10] NVIDIA TEGRA. http://www.nvidia.com/tegra. Viewed online July 2011.

[11] OpenCL - Open Computing Language. http://www.khronos.org/opencl/. Viewed online July 2011.

[12] PyPy: Coroutine pickling and unpickling. http://doc.pypy.org/en/latest/stackless.html. Viewed online August 2011.

[13] Pyro: Python remote objects. http://pyro.sourceforge.net/. http://pyro.sourceforge.net/.

[14] Python Multiprocessing Module - effectively side-stepping the Global Interpreter Lock. http://docs.python.org/library/multiprocessing.html. Viewed online August 2011.

[15] SWIG: Simplified Wrapper and Interface Generator. http://www.swig.org. Viewed online July 2011.

[16] The Go Programming Language. http://www.golang.org. Viewed online August 2011.

[17] The Python FAQ: Can't we get rid of the Global Interpreter Lock. http://docs.python.org/faq/library. Viewed online August 2011.

[18] The Python Standard Library. http://docs.python.org/library/. Viewed online July 2011.

[19] The Unladen Swallow Project - A faster implementation of Python. http://code.google.com/p/unladen-swallow/wiki/ProjectPlan. Viewed online August 2011.

[20] ZeroMQ: The Intelligent Transport Layer for Python. http://www.zeromq.org/. Viewed online August 2011.

[21] Frederick R. M. Barnes and Peter H. Welch. Communicating Mobile Processes. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 201–218, sep 2004.

[22] Frederick R.M. Barnes. Blocking system calls in KRoC/Linux. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering Series*, pages 155–178. Computing Laboratory, University of Kent, IOS Press, sep 2000.

[23] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. Knime: The konstanz information miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*, pages 319–326. Springer, 2008.

[24] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. Knime - the konstanz information miner: version 2.0 and beyond. *SIGKDD Explor. Newsl.*, 11:26–31, nov 2009.

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

[26] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[27] Neil C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–206, jul 2007.

[28] Neil C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 67–83, sep 2008.

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

[30] Olivier Capp, Simon J. Godsill, and Eric Moulines. An overview of existing methods and recent advances in sequential Monte Carlo. *Proceedings of the IEEE*, 95(5):899–924, 2007.

[31] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[32] V. Curcin and M. Ghanem. Scientific workflow systems - can one size fit all? In *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pages 1 –9, dec. 2008.

[33] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[34] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. Mpi for python: Performance improvements and mpi-2 extensions. *J. Parallel Distrib. Comput.*, 68:655–662, may 2008.

[35] B. V. Dasarathy. *Nearest neighbor (NN) norms: NN pattern classification techniques*. IEEE Computer Society Press, 1990.

[36] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528 – 540, 2009.

[37] Michael J. Flynn. Very high-speed computing systems. *Proceedings of The IEEE*, 54:1901–1909, 1966.

[38] Rune Møllegaard Friborg. Distributed Scientific Computing in Python. *Master Thesis at University of Copenhagen*, page 124, feb 2008.

[39] Rune Møllegaard Friborg and Brian Vinter. Rapid development of scalable scientific software using a process oriented approach. *Journal of Computational Science*, In Press, Corrected Proof:1–10, 2011.

[40] Rune Møllegard Friborg, John Markus Bjørndalen, and Brian Vinter. Three Unique Implementations of Processes for PyCSP. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, G. S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 277–292, nov 2009.

[41] Rune Møllegard Friborg and Brian Vinter. CSPBuilder - CSP based Scientific Workflow Modeling. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 347–363, sep 2008.

[42] Rune Møllegard Friborg and Brian Vinter. Verification of a Dynamic Channel Model using the SPIN Model-Checker. In Peter H. Welch, Adam T. Sampson, Jan Baekgaard Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 35–54, jun 2011.

[43] Gerald H. Hilderink, Andry W. P. Bakkers, and Jan F. Broenink. A distributed real-time java system based on csp. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '00, pages 400–, Washington, DC, USA, 2000. IEEE Computer Society.

[44] Gerald H. Hilderink, Jan F. Broenink, Wiek Vervoort, and Andrè W. P. Bakkers. Communicating Java Threads. In Andrè W. P. Bakkers, editor, *Proceedings of WoTUG-20: Parallel Programming and Java*, pages 48–76, mar 1997.

[45] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, pages 666–676, aug 1978.

[46] C.A.R. Hoare. Communicating Sequential Processes. *Prentice-Hall*, 1985.

[47] Gerard J. Holzman. The Model Checker Spin. *IEEE Trans. on Software Engineering*, pages 279–295, may 1997.

[48] Holger Hoos and Thomas Sttzle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[49] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Res*, 34(Web Server issue), jul 2006.

[50] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science and Engineering*, 9:90–95, 2007.

[51] National Instruments. Labview. `http://www.ni.com/labview/`.

[52] Christian L. Jacobsen. *A portable runtime for concurrency research and application*. PhD thesis, University of Kent at Canterbury, dec 2006.

[53] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

[54] A. Tennant K. Lefmann, K. Nielsena and B. Lake. Mcstas 1.1: a tool for building neutron monte carlo simulations. *Physica B: Condensed Matter*, pages 152–153, mar 2000.

[55] P Kacsuk, G Dózsa, J Kovács, R Lovas, N Podhorszki, Z Balaton, and G Gombás. P-grade: a grid programming environment. *Journal of Grid Computing*, 1(2):171–197, 2003.

[56] P Kacsuk, T Kiss, and G Sipos. Solving the grid interoperability problem by p-grade portal at workflow level. *Future Generation Computer Systems*, 24(7):744–751, 2008.

[57] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[58] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *ArXiv e-prints*, March 2011.

[59] Hajo N. Krabbenhöft, Steffen Möller, and Daniel Bayer. Integrating ARC Grid Middleware with Taverna Workflows. *Bioinformatics*, March 2008.

[60] Mads R. B. Kristensen and Brian Vinter. Numerical Python for Scalable Architectures. In *Fourth Conference on Partitioned Global Address Space Programming Model, PGAS'10*. ACM, 2010.

[61] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer, 2008.

[62] Jonathan Lawrence. Practical application of csp and fdr to software design. In Ali Abdallah, Cliff Jones, and Jeff Sanders, editors, *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 717–721. Springer Berlin / Heidelberg, 2005.

[63] Gavin Lowe. Implementing Generalised Alt. In Peter H. Welch, Adam T. Sampson, Jan Baekgaard Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 1–34, jun 2011.

[64] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 Manual, 1997.

[65] Bertram Ludscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[66] David May. Occam. *SIGPLAN Not.*, 18:69–79, apr 1983.

[67] Hans W. Meuer. The TOP500 Project: Looking Back Over 15 Years of Supercomputing Experience. *Informatik-Spektrum*, 31(3):203–222, June 2008.

[68] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[69] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[70] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.

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

[72] Sarah Mount, Mohammad Hammoudeh, Sam Wilson, and Robert Newman. CSP as a Domain-Specific Language Embedded in Python and Jython. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, G. S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 293–309, nov 2009.

[73] Fiona G. G. Nielsen, Kasper Galschiøt Markus, Rune Møllegaard Friborg, Lene Monrad Favrholdt, Hendrik G. Stunnenberg, and Martijn Huynen. CATCH-profiles: Clustering and Alignment Tool for ChIP profiles, 2011. Submitted to PLoS ONE.

[74] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[75] Travis E. Oliphant. *Guide to NumPy*. 2006.

[76] Pearu Peterson. F2py: a tool for connecting fortran and python programs. *Int. J. Comput. Sci. Eng.*, 4:296–305, nov 2009.

[77] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. pages 184–592 Vol. 1, 2005.

[78] Ronald Poppe. Vision-based human motion analysis: An overview. *Computer Vision and Image Understanding*, 108(1-2):4–18, 2007.

[79] Armin Rigo. JSON (JavaScript Object Notation). `http://www.json.org/`. Viewed online August 2011.

[80] Armin Rigo. PyPy: An alternative Python Interpreter implemented in Python. `http://pypy.org/`. Viewed online August 2011.

[81] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and

Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, jun 2009.

[82] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[83] Guido Van Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[84] Adam T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, October 2010.

[85] Adam T. Sampson and Neil C. Brown. Tock (translator from occam to C from Kent). `http://projects.cs.kent.ac.uk/projects/tock/`.

[86] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[87] Mario Schweigler and Adam T. Sampson. pony - The occam-pi Network Environment. In Peter H. Welch, Jon Kerridge, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2006*, pages 77–108, sep 2006.

[88] C Sieb, T Meinl, and M Berthold. Parallel and distributed data pipelining with knime. *Mediterranean Journal of Computers and Networks*, jan 2007.

[89] Munindar P. Singh and Mladen A. Vouk. Scientific Workflows: Scientific Computing Meets Transactional Workflows, 1996.

[90] Bernhard H.C. Sputh and Alastair R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. In Jan F. Broenink, Herman Roebbers, Johan P. E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures 2005*, pages 71–107, sep 2005.

[91] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[92] Bernard Sufrin. Communicating Scala Objects. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 35–54, sep 2008.

[93] Christian Tismer. Greenlet: Lightweight in-process concurrent programming for Python. `http://pypi.python.org/pypi/greenlet`. Viewed online August 2011.

[94] Christian Tismer. Continuations and Stackless Python. Technical report, 2000.

[95] Vanovschi V. Parallel Python Software. `http://www.parallelpython.com`.

[96] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14:5–51, jul 2003.

[97] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. *Proc. First IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986.

[98] Brian Vinter, John Markus Bjørndalen, and Rune Møllegard Friborg. PyCSP Revisited. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, G. S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 263–276, nov 2009.

[99] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Alting barriers: synchronisation with choice in java using jcsp. *Concurrency and Computation: Practice and Experience*, 22(8):1049–1062, 2010.

[100] Peter H. Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, jul 2007.

[101] Peter H. Welch, Neil C. Brown, James Moores, Kevin Chalmers, and Bernhard H.C. Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 349–369, jul 2007.

[102] Peter H. Welch and Jeremy M. R. Martin. Formal Analysis of Concurrent Java Systems. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 275–301, sep 2000.

[103] P.H. Welch and J.M.R. Martin. A CSP model for Java multithreading. In *Software Engineering for Parallel and Distributed Systems, 2000. Proceedings. International Symposium on*, pages 114–122, jun 2000.

[104] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[105] 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.

[106] 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.

Appendix A

Publications

All published papers have been peer-reviewed.

A.1 GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking

Paper presented at Computer Vision GPU (CVGPU 2010), ECCV 2010 Workshop, Hersonissos, Heraklion, Crete, Greece, September 11, 2010
Rune Møllegaard Friborg, Søren Hauberg, Kenny Erleben: GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking

GPU Accelerated Likelihoods for Stereo-Based Articulated Tracking

Rune Møllegaard Friborg, Søren Hauberg, and Kenny Erleben

{runef, hauberg, kenny}@diku.dk,
The eScience Centre, Dept. of Computer Science, University of Copenhagen

Abstract. For many years articulated tracking has been an active research topic in the computer vision community. While working solutions have been suggested, computational time is still problematic. We present a GPU implementation of a ray-casting based likelihood model that is orders of magnitude faster than a traditional CPU implementation. We explain the non-intuitive steps required to attain an optimized GPU implementation, where the dominant part is to hide the memory latency effectively. Benchmarks show that computations which previously required several minutes, are now performed in few seconds.

Keywords CUDA · GPU Computing · Articulated Tracking · Particle Filtering

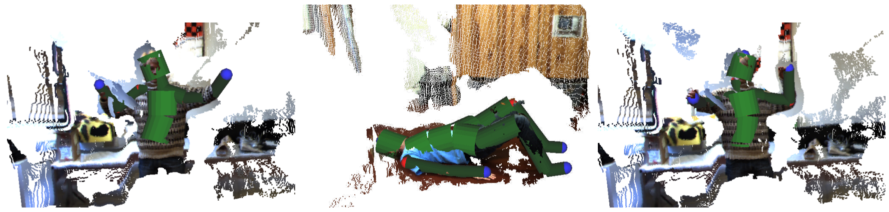


Fig. 1. The type of articulated tracking for which we achieve a speed up factor of up to 600 when using a GPU optimization. The images show stereo points with a super imposed illustration of the skin model.

1 The Computational Problem of Articulated Tracking

Three dimensional articulated human motion tracking is the process of estimating the configuration of body parts over time from sensor input [1]. One approach to this estimation is to use motion capture equipment where e.g. electromagnetic markers are attached to the body and then tracked in three dimensions. While this approach gives accurate results, it is intrusive and cannot be used outside laboratory settings. Alternatively, computer vision systems can be used for non-intrusive analysis such as the one shown in Figure 1. One standard approach is to use a particle filter [2] for finding a sequence of poses that match the observed data well. From a practical point of view this means making many random guesses of the current pose and comparing these to the observed data. In terms of performance, the critical part is comparing each guess to the data. In this paper, we present a GPU-based solution to this problem and show a substantial increase in performance compared to a CPU-based implementation. Such performance increases are essential in allowing us to build proper generative likelihood models, that otherwise would be impractical.

Before dwelling into the details of this work, we briefly describe in Section 2 the general particle filter based framework for articulated tracking that forms the foundation for this work. Next we consider related work in Section 3 and in Section 4 we describe the likelihood model for our work. We focus on using the GPU in Section 5 and results can be found in Section 6 before we conclude in Section 7.

2 Particle Filtering for Articulated Tracking

The objective of articulated human tracking is to estimate the position and orientation of each limb in the human body. This, as such, requires a representation of the human body. The most common choice [1] is the *kinematic skeleton* which is a collection of rigid bones organised in a tree structure (see Fig. 2(a)). Each bone can be rotated at the point of connection between the bone and its parent. We will refer to such a connection point as a *joint*.

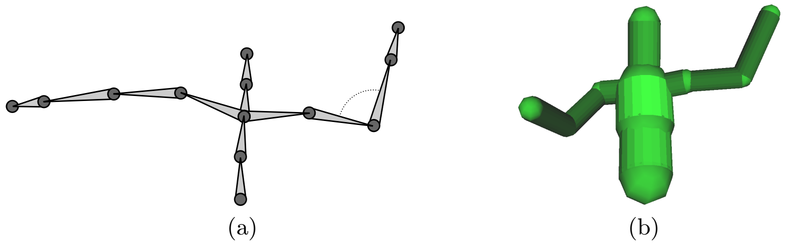


Fig. 2. (a) A rendering of the kinematic skeleton. Each bone position is computed by a rotation and a translation relative to its parent. The joints are drawn as circles. (b) A rendering of the skin model.

We model the bones as having known constant length (i.e. rigid), so the direction of each bone constitute the only degrees of freedom in the kinematic skeleton. The direction in each joint can be parametrised with a vector of angles, noticing that different joints may have different number of degrees of freedom. We may collect all joint angle vectors into one large vector $\boldsymbol{\theta}_t$ representing all joint angles in the model. The objective of the tracking system then becomes to estimate this vector at each time step.

At the heart of our articulated tracker is the well-known particle filter [2], which we will briefly describe here. The particle filter is, in general, concerned with estimating an unobserved state of a system from observations. In terms of articulated tracking it is concerned with estimating the pose $\boldsymbol{\theta}_t$ at each frame in a video sequence. In terms of statistics, we seek $p(\boldsymbol{\theta}_t|\mathcal{X}_{1:t})$, where the subscript denotes time and $\mathcal{X}_{1:t} = \{\mathcal{X}_1, \dots, \mathcal{X}_t\}$ denotes all observations seen at time t. This distribution is crudely represented as a set of samples that are propagated through time by sampling from $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1})$. Each sample $\boldsymbol{\theta}_t^{(j)}$ is assigned a weight according to its likelihood $p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)})$. Thus, at each time step t we compute

```
for $j = 1$ to $J$ do
  Sample $\boldsymbol{\theta}_t^{(j)}$ from $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}^{(j)})$ ;
  $w_j \leftarrow p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)})$ ;
end for
```

Usually it is computationally cheap to sample from $p(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}^{(j)})$, whereas it is expensive to evaluate the likelihood $p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)})$. It is worth noting that the loop can be executed in parallel as each sample is treated completely independent.

Once we have drawn new samples and assigned them weights, we can estimate the current pose as the mean value of $p(\boldsymbol{\theta}_t|\mathcal{X}_{1:t})$. This can be approximated as

$$\bar{\boldsymbol{\theta}}_t \approx \sum_{j=1}^{J} \frac{w_j}{\sum_{l=1}^{J} w_l} \boldsymbol{\theta}_t^{(j)} \quad . \tag{1}$$

3 Related Work on Computational Tracking

Most work in the articulated tracking literature falls in two categories. Either the focus is on improving the image likelihoods or on improving the predictions. Due to space constraints, we forgo a review of various predictive models as this paper is focused on computational efficient likelihoods. For an overview of predictive models, see the review paper by Poppe [1].

Most publications on likelihood models for articulated tracking are concerned with finding descriptive image features. Sminchisescu and Triggs [3] showed successful tracking using a combination of edge strength and horizontal flow in a monocular setup. This approach is, however, bound to have difficulties due to only having one viewpoint. One solution is to use multiple calibrated cameras as, amongst others, was done by Deutscher et. al. [4] who used a combination of edge strength and background subtraction. Due to the difficulties of calibration, such approaches are, however, hard to use in non-laboratory settings. A possible compromise is to use a pre-calibrated stereo camera as was done by Hauberg et. al. [5]. Their solution did, however, not cope with limbs occluding each other.

While much work has gone into developing functional likelihood models, not much has been published on efficient implementations on GPU hardware. Exceptions include the work of Bandouch et. al. [6] that use a simple colour based appearance model in a multiple camera setup. By representing pixel colours as bitmasks they are able to make likelihood evaluations using only bitwise operations that can be efficiently implemented on the GPU. Cabido et. al. [7] use a combination of background subtraction along with binary template matching for a planar low-dimensional articulated model. They rephrase the entire optimisation as an application of textures on the GPU and as such get very high frame rates.

4 Our Likelihood Model

In this section we define the likelihood model $p(\mathcal{X}_t|\boldsymbol{\theta}_t)$ used in this paper. We use an off-the-shelf consumer stereo camera[1], which provides us with a set of points in 3D at each time step. We, thus, have $\mathcal{X}_t = \{\boldsymbol{x}_{1,t}, \ldots, \boldsymbol{x}_{I,t}\}$, where I denotes the number of points and each $\boldsymbol{x}_{i,t} \in \mathbb{R}^3$.

We will assume that each point generated by the stereo camera is independent and is normally distributed around the skin of the pose. Thus, we have

$$p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)}) \propto \prod_{i=1}^{I} \exp\left(-\frac{d_i^2(\boldsymbol{\theta}_t^{(j)})}{2\sigma^2}\right) \quad , \tag{2}$$

where $d_i^2(\boldsymbol{\theta}_t^{(j)})$ denotes the square Euclidean distance between the i^{th} stereo point and the skin of the pose parametrised by $\boldsymbol{\theta}_t^{(j)}$. For numerical stability [2] we implement the particle filter on a logarithmic scale and as such only need to compute

$$\log p(\mathcal{X}_t|\boldsymbol{\theta}_t^{(j)}) = -\frac{1}{2\sigma^2}\sum_{i=1}^{I} d_i^2(\boldsymbol{\theta}_t^{(j)}) + \text{constant} \quad , \tag{3}$$

where the constant term can be ignored. For this definition to be complete, we need a definition of the skin model and a suitable metric.

For the skin of the j^{th} sample we will use a collection of capsules $\mathcal{C}^j = \{c_1^j, \ldots, c_K^j\}$. Specifically, we assign a capsule to each bone in the kinematic skeleton, such that the capsule is aligned with the bone. The radius of the capsule depends on the bone. We then define the skin of the skeleton as the union of these capsules. This gives us skins such as the one in Fig. 2(b). This model is very similar to the common model (see e.g. [8, 9]) where a cylinder is assigned to each bone. Here, we use capsules for mathematical convenience.

To compute the distance between a point and the skin, we compute the distance from the point to each capsule and pick the smallest, i.e.

$$d_i^2(\boldsymbol{\theta}_t^{(j)}) = \min_k d^2(\boldsymbol{x}_{i,t}, c_k^j) \quad , \tag{4}$$

where $d^2(\boldsymbol{x}_{i,t}, c_k^j)$ denotes the square distance from the i^{th} stereo point to the k^{th} capsule of the j^{th} sample. We will define this distance in terms of ray casting in

[1] `http://www.ptgrey.com/products/bumblebee2/`

the following. To avoid notational clutter, we will omit the time subscript from our notation in the rest of the paper.

Let the capsule c_k^j be defined by the two bone end points $\boldsymbol{a} \in \mathbb{R}^3$ and $\boldsymbol{b} \in \mathbb{R}^3$ and the radius $r \in \mathbb{R}_+$. Consider the stereo point $\boldsymbol{x}_i$. This is a point seen by the camera. Thus, $\boldsymbol{x}_i$ must lie on a ray starting at the camera origin $\boldsymbol{p} \in \mathbb{R}^3$ and casting in the direction of $\boldsymbol{v} = \frac{\boldsymbol{x}_i - \boldsymbol{p}}{\|\boldsymbol{x}_i - \boldsymbol{p}\|}$. We can therefore think of $\boldsymbol{x}_i$ as a function of the ray length parameter Δ. That is, we have the ray definition

$$\boldsymbol{x}_i(\Delta) = \boldsymbol{p} + \boldsymbol{v}\Delta \qquad \forall \Delta \geq 0 \ . \tag{5}$$

From this definition we may define a measure indicating how well a given stereo point $\boldsymbol{x}_i$ fits with a given capsule. Let Δ be the ray length of the stereo point and let $\Delta_{\min}$ be the shortest ray length corresponding to an intersection point between the ray and the capsules then intuitively a distance measure may be taken as $|\Delta - \Delta_{\min}|$. This corresponds to rendering a depth map of the capsules, and computing the absolute difference between this and the depth map from the stereo camera.

Since stereo data contains outliers, both from other objects appearing in the scene and from false matches, we need a robust metric. Here we simply truncate the distance if it exceeds a given threshold

$$\mathbf{d}(\boldsymbol{x}_i, c_k^j) = \begin{cases} |\Delta_{\min} - \Delta| & \text{if } \Delta_{\min} \text{ exists and } |\Delta_{\min} - \Delta| \leq \tau . \\ \tau & \text{otherwise.} \end{cases} \tag{6}$$

For this metric to be computable, we need to be able to determine if a given ray intersects the capsules and if so compute the distance $\Delta_{\min}$. The details of ray capsule intersection can be found in Appendix A. It is worth noting that the basic model works for all skin models, though ray casting details will have to be adapted.

5 Optimizing for the GPU

The algorithm presented in this paper achieves a major speedup when implemented on the GPU. However, it requires careful planning in designing for the massive parallelism in the GPU architecture. The first problem to be addressed is how to block data and computations most efficiently with respect to performance. The task is to minimize data communication and maximize the amount of computations done by one block of threads. Our targeted GPU architectures are the CUDA enabled Nvidia GPUs with compute capability from 1.1 to 1.3.

For our current applications we typically use in the order of $I \approx 50000$ stereo points, $K \approx 40$ capsules and $J \approx 2000$ samples. One simple approach would be to create a 3D float array of dimension $J \times K \times I$ where entry (j, k, i) would hold the value of $\mathbf{d}(\boldsymbol{x}_i, c_k^j)$. This would result in a naive data parallel computation where each thread would compute a single distance measurement. However, such an array would require $2000 \times 40 \times 50000 \times 4$ bytes ≈ 16 Gigabytes of memory.

This clearly exceeds the maximum available device memory, so some tiling must be applied to our problem.

Thus, we create a grid of thread blocks in such a way that each thread block corresponds to one sample and one tile of stereo points and we launch a measure kernel on this grid. During execution the measure kernel will loop over samples in consecutive launches to avoid kernel time-outs. Additionally, support for multiple GPU devices is performed by dividing the samples into one chunk for each GPU. If multiple GPUs are available the same number of CPU worker threads is created and then given a GPU to control. The overhead of launching CPU threads is small and the effect will only be visible for small problem sizes which are not the target for this paper. This orchestration results in the grid setup illustrated in Figure 3. Using this approach we will have an intermediate 2D result array $\mathcal{A}$ consisting of $J \times$ POINTS_TILES computed measurements, where POINT_TILES is set to $\frac{I}{POINTS\_PER\_BLOCK}$. The number of threads in each block is identical to POINTS_PER_BLOCK, thus this value is tuned to achieve the best occupancy for a given GPU.

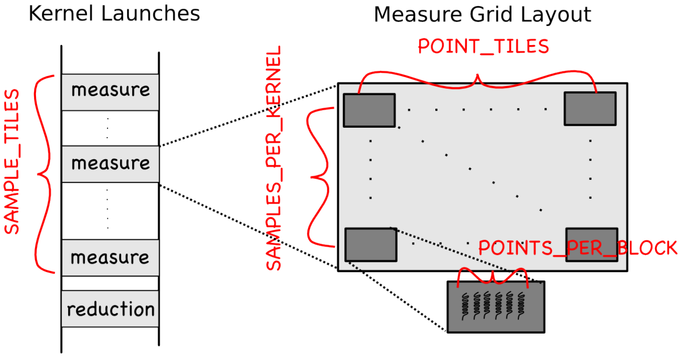


Fig. 3. Illustration of the grid layout and kernel launches for a single GPU. A sequence of measure kernel launches is executed: one for each tile of samples where SAMPLE_TILES = SAMPLES_PER_GPU/SAMPLES_PER_KERNEL. Only a single reduction kernel is launched prior to returning to the CPU thread handling the GPU.

Subsequently we will launch a partial sum reduction kernel. We execute the reduction kernel on a grid where each thread block corresponds with one sample. The kernel performs partial sum reduction on the result array $\mathcal{A}$ to produce the final measurement set $\mathcal{M}$. The j^{th} component in set $\mathcal{M}$ holds the final measurement value of the j^{th} sample. Observe that partial sum reduction is a well studied problem on the GPU and we will therefore not treat it further in this paper. The NVIDIA CUDA SDK version 3.0 contains a sample with code [10] and the next release of CUDPP will also contain sum reduction [11].

To perform the entire computation on the GPU we need to transfer the stereo points $\mathcal{X}$ and the capsules $\{\mathcal{C}^j\}_{j=1}^{J}$ to the GPU device and then read back the set $\mathcal{M}$ from the GPU device. We also need to setup the intermediate storage $\mathcal{A}$. Since each capsule takes 7 floats to store and each stereo point 3 floats the total memory requirements on device memory is for our typical use: $7JK + 3I + J$ POINT_TILES $+ I \approx 3$ Megabytes. This is far from our upper bound on global memory of 256 Megabytes and means that we can keep all points, capsules and measurements in device memory during execution.

The problem that we have specified is memory bound, since it traverses the set of capsules $\{\mathcal{C}^j\}_{j=1}^{J}$ for every stereo point in $\mathcal{X}$ while the computation does not outweigh the latency of the memory. It is essential that we hide this memory latency. The GPU is perfect for doing exactly this, if enough thread blocks are active and the memory operations are handled with care. For optimal performance it will be necessary to keep data aligned in host memory and ensure coalesced access to host memory by using the 16 Kb shared memory available in each SM (streaming multiprocessor)[2]. Seven threads are used to fetch the data of one capsule (7 floats). In Figure 4 and Listing 1.1 we show how the stereo point data, consisting of the coordinates x, y, and z for a single point, are handled in a similar manner, where every set of three threads is working together to fetch one stereo point (3 floats).

```
/* blockDim.x = POINTS_PER_BLOCK */
__shared__ float Xds[3][blockDim.x];
size_t i = threadIdx.x;
size_t total = blockDim.x*3u;
size_t offset = blockDim.x*blockIdx.y*3u;
for (size_t ii = i; ii < total; ii += blockDim.x)
  Xds[ii%3][ii/3] = Xd[offset+ii];
```

Listing 1.1. All threads in a warp of 32 threads will request data from aligned neighboring addresses in device memory, Xd. This results in two coalesced memory requests of maximum size (64 bytes). The data is then copied to shared memory and organized as illustrated in Figure 4, to avoid bank conflicts.

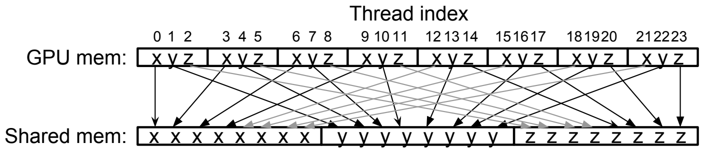

Thread index
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14
15 16 17
18 19 20
21 22 23
GPU mem:
x y z x y z x y z x y z x y z x y z x y z x y z
Shared mem:
x x x x x x x x y y y y y y y y z z z z z z z z

Fig. 4. Ensuring coalesced memory transfers when transferring stereo points from GPU device memory to shared memory. Data is fetched in blocks of 16 and thus aligned in host memory.

[2] The Nvidia Fermi architecture has 64 Kb of cache / shared memory reserved for each SM.

The reason for orchestrating the arrays coordinate-wise in shared memory is to avoid bank conflicts [12]. The GPU is a SIMT (single instruction, multiple thread) architecture and executes in an SM one instruction for a warp of 32 threads. When the 32 threads access a shared memory address, it is crucial that they balance the requests onto all 16 banks. Since the shared memory is organized in a round-robin fashion to the 16 banks, we can make sure that we access neighbouring addresses.

When the GPU executes branch instructions all threads in a warp (32 threads) follow the same branch. This means that if some threads in a warp follow one branch and others follow another branch, all threads must visit both branches and the instruction count goes up. With this in mind we have worked to minimize the number of divergent branches, and where we knew there would be divergent branches, conditional expressions were preferred instead, since both expressions would be evaluated anyway.

The resulting measure kernel uses 24 registers, which means that we can run up to 320 threads on devices with compute capability 1.1 or 1.2 (8192 registers) and 640 threads on devices with compute capability 1.3 (16384 registers). 24 registers is not low enough to completely hide the memory latency, but to go lower would require to split the measure kernel into multiple kernels which could each use less registers. This task would require a huge temporary data set in device memory and thus we concluded that 24 registers is the best we can do. The block size used for the benchmarks in Section 6 is chosen so that the maximum number of active blocks is 8 and can go to either 320 or 640 active threads.

6 Two Orders of Magnitude Speedup

To benchmark the implementation, it was run on the three systems listed in Table 1. For every benchmark, a sequential CPU implementation was also executed and the result values compared for correctness. We varied the number of stereo points and the number of samples to see how well the solution scales for up to 43000 stereo points and 3500 samples. The number of capsules was constant at 48. The current GPU implementation is only limited by the maximum grid sizes and the shared memory, thus it actually supports up to 4.194.240 stereo points and 65535 samples of 64 capsules, which can all fit inside 256Mb device memory.

Table 1. Benchmark systems

System 1	System 2	System 3
Intel Core 2 Quad @ 2.4Ghz	Intel Core 2 Duo @ 2.33Ghz	Intel Core 2 Duo @ 2.4Ghz
4Gb DDR2 800Mhz	4Gb DDR2 800Mhz	2Gb DDR2 667Mhz
Nvidia C1060 Tesla 4Gb	2 * Nvidia 9800GX2 1Gb	Nvidia 8600M GT 256Mb
Compute cap. 1.3	Compute cap. 1.1	Compute cap. 1.1
240 cores @ 1.30Ghz	512 cores @ 1.50Ghz	32 cores @ 0.94Ghz

When comparing the performance of the two 9800GX2 with the C1060, notice that one 9800GX2 actually consists of 2 GPUs with hardware similar to a 8800GTX. This means that we are comparing a system with a total of 4 GPUs with a system with 1 GPU, which gives a disadvantage to the system with 4 GPUs, since the benchmark results include the overhead of handling 4 threads. The plots in Figure 5 clearly shows that the GPU implementation scales linearly with an increasing number of stereo points or samples for both systems. The effect of handling the extra threads can be seen for the smaller problems and we expect that the C1060 will be fastest for small problems. For the largest problem the two 9800GX2 are 2.1 times faster than the C1060, but theoretically two 9800GX2 can actually execute 2.46 times more FLOPS than one C1060. The two 9800GX2 are also more capable at hiding the memory latency, since they can have 4 times 320 active threads, while the C1060 is limited to 512 active threads for our implementation. System 3 was not included in these plots, since the benchmark results was around 20 times slower.

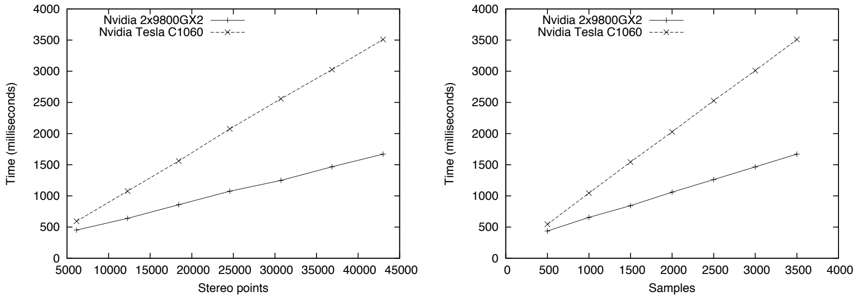


Fig. 5. Plots showing linear scaling for increasing number of stereo points or samples. The number of capsules is kept constant at 48.

The speedup plot in Figure 6 is created using the CPU implementation in Listing 1.2 as the reference. We have used the same input data set for the CPU and the GPU benchmarks. The measurement function used in the CPU implementation (Listing 1.2) is identical to the measurement function used in the GPU implementation (Listing 1.3), but the invocation of the measurement function in listing 1.2 is purely sequential and thus only utilize one core. Since the problem is memory bound, the one thread will have to wait on memory. We expect that an optimized CPU implementation could execute twice as fast, compared to the reference CPU implementation. On the GPU the memory latency has been successfully hidden, which becomes apparent when looking at the speedup numbers in Figure 6.

```
for(size_t j = 0; j < J; ++j)
{
  M[j] = 0.0f;
  for(size_t i = 0; i < I; ++i)
  {
    size_t const ii = i*3u;
    float3 const x_i = make_float3(X[ii],X[ii+1],X[ii+2] );
    float value = MAX_DISTANCE;
    for(size_t k = 0u; k < K; ++k)
    {
      size_t const kk = (j*K + k)*7u;
      float3 const a = make_float3(C[kk],  C[kk+1], C[kk+2] );
      float3 const b = make_float3(C[kk+4],C[kk+5], C[kk+6] );
      float  const r = C[kk+3];
      value = min( measurement(  x_i, r, a, b ), value );
    }
    M[j] += value;
  }
}
```

Listing 1.2. The CPU implementation used for benchmarking. This code is executed in a single thread for the CPU.

```
/* Extracted from the body of the measurement_kernel */
float3 const x_i=make_float3(Xds[0][i], Xds[1][i], Xds[2][i]);
float value = Ads[i];
for(size_t k = 0u; k<K; ++k)
{
  float3 const a = make_float3(Cds[0][k],Cds[1][k],Cds[2][k]);
  float3 const b = make_float3(Cds[4][k],Cds[5][k],Cds[6][k]);
  float  const r = Cds[3][ k ];
  value = min( value,  measurement( x_i, r, a, b ) );
}
Ads[i] = value;
```

Listing 1.3. The GPU implementation, which computes results identical (apart from rounding differences) to the CPU implementation in Listing 1.2. This code is executed in $J * I$ threads for the GPU.

The 8600M GT achieves a stable speedup of ≈ 20, while the others increase in speedup until reaching their maximum stage. The increase in speedup is explained by the overhead of running many kernels. For these benchmarks a kernel was called for every 8 samples, thus the overhead of calling a kernel takes up a larger proportion when the problem size is small and the GPUs are fast.

The fact that we see a correlation in Figure 6 between the speedup of the GPUs and with the GPU hardware specifications, means that we can conclude that the GPU implementation has succeeded to utilize the GPUs efficiently.

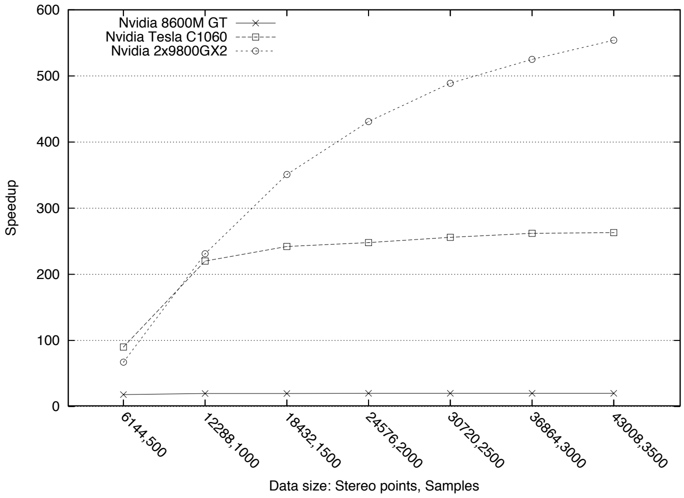


Fig. 6. The speedup achieved when computing a data set of the specified size on a GPU vs. the CPU. The number of capsules is kept constant at 48.

7 Conclusions and Future Work

In this work we have presented a tiling approach that results in a very efficient GPU acceleration of the measurement process for articulated tracking with a particle filter. The main causes to our two orders of magnitude speedup factor lies in careful hiding memory latencies from device memory and avoiding memory bank conflicts in the shared memory. We not only gain from the raw processing power of the GPU, but also from its alternative memory layout.

Our future work involves benchmarking on small scale GPU clusters as this may further interactive markerless computer vision based articulated tracking. Besides this, the sampling process of the particle filter is currently implemented in a naive consumer-producer scheme using a single CPU thread for each sample. This appears to be the next performance bottleneck that we will investigate.

References

1. Poppe, R.: Vision-based human motion analysis: An overview. Computer Vision and Image Understanding **108** (2007) 4–18
2. Cappé, O., Godsill, S.J., Moulines, E.: An overview of existing methods and recent advances in sequential Monte Carlo. Proceedings of the IEEE **95** (2007) 899–924
3. Sminchisescu, C., Triggs, B.: Kinematic Jump Processes for Monocular 3D Human Tracking. In: In IEEE International Conference on Computer Vision and Pattern Recognition. (2003) 69–76
4. Deutscher, J., Blake, A., Reid, I.: Articulated body motion capture by annealed particle filtering. In: cvpr, Published by the IEEE Computer Society (2000) 2126

5. Hauberg, S., Sommer, S., Pedersen, K.S.: Gaussian-like spatial priors for articulated tracking. In: Proceedings of ECCV'10. Lecture Notes in Computer Science, Springer (2010)
6. Bandouch, J., Beetz, M.: Tracking Humans Interacting with the Environment Using Efficient Hierarchical Sampling and Layered Observation Models, IEEE Int. Workshop on Human-Computer Interaction (HCI) (2009)
7. Cabido, R., Concha, D., Pantrigo, J.J., Montemayor, A.S.: High Speed Articulated Object Tracking Using GPUs: A Particle Filter Approach. In: 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, IEEE (2009) 757–762
8. Rohr, K.: Towards model-based recognition of human movements in image sequences. CVGIP-Image Understanding **59** (1994) 94–115
9. Sidenbladh, H., Black, M.J., Fleet, D.J.: Stochastic tracking of 3d human figures using 2d image motion. In: Proceedings of ECCV'00. Volume II of Lecture Notes in Computer Science 1843., Springer (2000) 702–718
10. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for gpu computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, Aire-la-Ville, Switzerland, Eurographics Association (2007) 97–106
11. CUDPP: Cuda data parallel primitives library. Accessed Online April (2010) `http://code.google.com/p/cudpp/`.
12. NVIDIA Corporation: NVIDIA CUDA Best Practices Guide. (2010) version 3.0.

A Computing the Ray–Capsule Intersection Point

To find the intersection between the ray and the capsule we first consider the situation with an infinitely long capsule. Here we can find the point of intersection by first finding the point $\boldsymbol{y}_i$ on the line through $\boldsymbol{a}$ and $\boldsymbol{b}$ that is closest to the ray $\boldsymbol{x}_i(\Delta)$. By orthogonal projection we find this as

$$\boldsymbol{y}_i = \boldsymbol{a} + \left((\boldsymbol{x}_i(\Delta) - \boldsymbol{a})^T \boldsymbol{c} \right) \boldsymbol{c} \tag{7}$$

where we have defined $\boldsymbol{c} = \frac{\boldsymbol{b}-\boldsymbol{a}}{\|\boldsymbol{b}-\boldsymbol{a}\|}$.

At the point of intersection between the ray and the infinite capsule we must have

$$\| \boldsymbol{x}_i(\Delta) - \boldsymbol{y}_i \|^2 = r^2 \ . \tag{8}$$

Inserting the ray definition from Eq. 5 gives us

$$r^2 = \| \boldsymbol{p} + \boldsymbol{v}\Delta - \boldsymbol{y}_i \|^2 = \| \boldsymbol{v}_\perp \Delta + \boldsymbol{p}_\perp \|^2 \ , \tag{9}$$

where $\mathbf{P} = \left(\mathbf{I} - \boldsymbol{c}\boldsymbol{c}^T\right)$ and $\boldsymbol{v}_\perp = \mathbf{P}\boldsymbol{v}$ and $\boldsymbol{p}_\perp = \mathbf{P}\left(\boldsymbol{p} - \boldsymbol{a}\right)$. This is readily identified as a second order polynomial in Δ

$$P_c(\Delta) = \boldsymbol{v}_\perp^T \boldsymbol{v}_\perp \Delta^2 + 2\boldsymbol{v}_\perp^T \boldsymbol{p}_\perp \Delta + \boldsymbol{p}_\perp^T \boldsymbol{p}_\perp - r^2 = 0 \ . \tag{10}$$

If no roots to this polynomial exist then the ray does not intersect the infinite long capsule. Otherwise we solve for the minimum positive root Δ_{cap} which will give us the intersection point on the infinite long capsule.

In practice, the skeleton model does not have infinite long limbs and as such we do not have infinite long capsules. The above approach thus needs to be modified to cope with finite capsules. In the case where $0 \leq \boldsymbol{c}^T(\boldsymbol{y}-\boldsymbol{a}) \leq 1$ the above analysis still holds. In all other cases we only need to see if the ray intersects with the spheres of radius r centred in $\boldsymbol{a}$ and $\boldsymbol{b}$. If the ray intersects the sphere centred in $\boldsymbol{a}$, we must have

$$\| \boldsymbol{x}_i(\mathit{\Delta}) - \boldsymbol{a} \|^2 = r^2 \ . \tag{11}$$

Once again, this gives as a second order polynomial

$$P_a(\mathit{\Delta}) = \boldsymbol{v}^T\boldsymbol{v}\mathit{\Delta}^2 + 2\boldsymbol{v}^T(\boldsymbol{p}-\boldsymbol{a})\mathit{\Delta} + (\boldsymbol{p}-\boldsymbol{a})^T(\boldsymbol{p}-\boldsymbol{a}) - r^2 = 0 \ . \tag{12}$$

If this polynomial has no roots then the ray does not intersect the sphere centred in $\boldsymbol{a}$. If it does have roots, we find the intersection from the smallest positive root. A similar treatment can be given to the sphere centred in $\boldsymbol{b}$.

Thus, the ray intersection algorithm will solve three second order polynomials $P_a(\mathit{\Delta})$, $P_b(\mathit{\Delta})$, and $P_c(\mathit{\Delta})$ and use some **if**-statements that will determine select the proper smallest positive root as the ray intersection length.

A.2 CATCHprofiles: Clustering and Alignment Tool for ChIP profiles

Submitted to PLoS ONE

Fiona G. G. Nielsen, Kasper Galschiøt Markus, Rune Møllegaard Friborg, Lene Monrad Favrholdt, Hendrik G. Stunnenberg, Martijn Huynen: CATCHprofiles: Clustering and Alignment Tool for ChIP profiles

CATCHprofiles: Clustering and Alignment Tool for ChIP profiles

Fiona G. G. Nielsen, (a,c)
Kasper Galschiøt Markus, (b)
Rune Møllegaard Friborg (d)
Lene Monrad Favrholdt, (b)
Hendrik G. Stunnenberg *, (c)
Martijn Huynen *, (a,c)

Affiliations
(a) Centre for Molecular and Biomolecular Informatics, Radboud University Nijmegen Medical Centre, Geert Grooteplein 26-28, 6525 GA Nijmegen, The Netherlands
(b) Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
(c) Molecular Biology, Nijmegen Centre for Molecular Life Sciences, Faculty of Science, Geert Groteplein 28, 6525 GA Nijmegen, The Netherlands
(d) eScience Centre, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen, Denmark
* corresponding authors

Abstract

Chromatin Immuno Precipitation (ChIP) profiling detects *in vivo* protein-DNA binding, and has revealed a large combinatorial complexity in the binding of chromatin associated proteins and their post-translational modifications. To fully explore the spatial and combinatorial patterns in ChIP-profiling data and detect potentially meaningful patterns, the binding patterns must be aligned and clustered, which is an algorithmically and computationally challenging task. We have developed CATCHprofiles, a novel tool for exhaustive pattern detection in ChIP profiling data. CATCHprofiles is built upon a computationally efficient implementation for the exhaustive alignment and hierarchical clustering of ChIP profiling data. The tool features a graphical interface for examination and browsing of the clustering results. CATCHprofiles requires no prior knowledge about functional sites, detects known binding patterns "ab initio", and enables the detection of new patterns from ChIP data at a high resolution, exemplified by the detection of asymmetric histone and histone modification patterns around H2A.Z-enriched sites. CATCHprofiles' capability for exhaustive analysis combined with its ease-of-use makes it an invaluable tool for explorative research based on ChIP profiling data.
CATCHprofiles and the CATCH algorithm run on all platforms and is available for free through the CATCH website: http://catch.cmbi.ru.nl/
User support is available by subscribing to the mailing list catch-users@bioinformatics.org

Author Summary

Tools for the alignment and clustering of biological data are invaluable in biological research. In recent years, a new type of data has become available: Chromatin Immunoprecipitation (ChIP) profiling data. This data contains the distribution of proteins, like histones or transcription factors, along the DNA. We have developed CATCHprofiles, a software tool for aligning and clustering such data, allowing the detection of new, potentially biologically relevant patterns in the binding of proteins on the DNA. With the aid of the CATCH clustering and alignment algorithm and the graphical user interface in CATCHprofiles, it is possible to detect and distinguish all the different patterns present in a ChIP dataset. We exemplify the use of our algorithm and tool by detection of new patterns in published data sets.

Introduction

Chromatin Immuno Precipitation (ChIP) profiling techniques detect *in vivo* protein-DNA binding. The DNA bound by the protein of interest is co-immunoprecipitated using protein-specific antibodies (ChIP), and mapped to the genome either using a DNA microarray chip (ChIP-on-chip) or by sequencing (ChIP-seq), for a review see Collas *et al*.
ChIP profiling has been used not only to detect *in vivo* transcription factor binding sites[1-5] but also to map the epigenetic profile of the chromatin, e.g. histone occupancy and histone modifications[6-9]. ChIP profiling has revealed a high complexity of binding patterns, both for transcription factor binding sites and for epigenetic markers. The DNA-binding proteins show temporal variation in binding[9,7,10], as well as a combinatorial variation over different binding sites in the genome[11]. The various combinations of histone modifications are thought to instruct the cellular machinery[12] while the combinatorial presence of transcription factors could provide a mechanism to exert complex gene regulation[13].

The initial analysis of ChIP-profiling data is primarily concerned with detecting the binding sites in the genome and correlating regions that have specific combinations of chromatin modifications with other observables like gene expression. Such an exploration of the biological relevance of the spatial and temporal combinations of DNA-binding proteins and their modifications requires the clustering of similar ChIP profile regions. One approach is to discretize the data to a simple presence/absence call of each ChIP signal per region, and then classify regions by their binary presence/absence combinations[14,15]. However, this approach does not exploit the rich information of the individual signal shapes and relative positions within the regions. Another approach is to compile sets of genomic regions with similar annotated functions and determine their average ChIP signal pattern. This approach is easy to apply but does not allow the exploration of new patterns in unannotated regions. In general, a major challenge in the clustering of ChIP patterns is to compare and cluster binding profiles to enable further analysis of the identified clusters without *a priori* binning genomic locations of known functions such as transcription start sites, or reducing the complexity of the data by not including the relative positions and shapes of the ChIP signals. Not only does this call for an unsupervised clustering method that can manage high-resolution ChIP profiling data, it also requires the method to account for the unknown relative positioning of novel patterns, necessitating the alignment of the ChIP profile regions. Furthermore, it requires a flexible organization and graphical presentation of the results to allow browsing and selecting the results for further analysis.

To meet this challenge we have developed the CATCH (Clustering and AlignmenT of ChIp profiles) algorithm and implemented it in the tool CATCHprofiles. The CATCH algorithm is designed to handle ChIP profiling data and accounts for variable positioning of significant patterns within profile regions by incorporating alignment in the profile comparison. CATCHprofiles supports the analysis workflow by an interactive graphical visualization of data and results.

Two other analysis tools are currently available that include aligning of ChIP profile regions. The first one, ChromaSIG[16], implements a heuristic clustering and alignment based on Gibbs sampling[17]. The second, ArchAlign[18], performs exhaustive alignment using a similar approach to the CATCH algorithm, but does not perform clustering. The non-exhaustive and probabilistic search of ChromaSIG has an advantage in speed, but also the disadvantage of varying, non-deterministic results. Also, the heuristic approach to alignment and clustering cannot guarantee sensitivity, and some patterns may go undetected. CATCHprofiles and ArchAlign circumvent this by performing an exhaustive comparison of all pairwise profile windows in the dataset. However, since ArchAlign does not perform clustering, but reports the average aligned pattern of a set of preselected profiles, it cannot be used for discovery of more than one pattern in the given data. Our CATCHprofiles tool presents advantages over both ChromaSIG and ArchAlign, since we include both hierarchical clustering and exhaustive alignment in a deterministic algorithm. Furthermore, the Java tool CATCHprofiles has an interactive graphical user interface to browse and export results and the CATCH core algorithm is implemented for parallel execution on multi-core machines.

CATCHprofiles can be used to detect ChIP profile patterns in an unbiased approach, i.e. not based on functional annotation, as well as to extract new biological information from the alignment of individual patterns. We demonstrate the power of CATCHprofiles by genome-wide clustering of H2A.Z-enriched sites in a ChIP-seq dataset, revealing the H2A.Z context to contain various patterns of CTCF, RNA Polymerase II (PolII) and histone modifications. We also show how the orientation of the individual ChIP patterns correlates with the orientation of genomic elements, namely how the relative orientations of the H2A.Z and CTCF peak patterns are correlated with the orientation of the CTCF binding motif.

Results

The CATCH algorithm

We designed and implemented the CATCH algorithm to perform simultaneous alignment and clustering of ChIP profile patterns. To run the CATCH algorithm, the user must provide one or more ChIP profiling data sets along with the genomic regions to analyse, e.g. peak regions of interest. In the following, we use the shorthand 'profiles' refer to genomic regions of the ChIP profiling data, unless stated otherwise. Our implementation represents the profiles internally as multi-dimensional vectors of equidistant floating point values along their specified regions of the genome.

The CATCH algorithm uses a hierarchical clustering approach combined with pairwise alignment: it keeps a pool of profiles from which it iteratively aligns all pairs and chooses the most similar pair. Initially, this pool is the set of all profiles in the data set. Each time the most similar profile pair (P_1, P_2) is chosen, P_1 and P_2 are merged to obtain P', the average profile of their alignment, and P_1 and P_2 are replaced by P' in the profile pool. P' is then aligned to all the remaining profiles in the pool to determine their pairwise similarity. The sequence of merging events determines the topology of the tree. Conceptually this type of clustering is an unweighted pair-group centroid clustering[19]. As default similarity measure for comparing the profiles we use the sum of squared distances and every profile pair is compared in both forward and reverse (mirrored) direction. CATCH represents the profiles internally by a series of signals for fixed equidistant positions within the profile window, estimating missing values by linear interpolation of neighbouring signals, thereby allowing comparison of profiles with varying resolution. The CATCH algorithm and the options for the similarity measure and normalization are described in detail in Supplementary Methods: CATCH algorithm.

Visualization and graphical interface

CATCHprofiles is a stand-alone tool for ChIP profiling clustering analysis and visualization. The tool implements the CATCH algorithm, as described above, for the alignment and clustering of ChIP profiles. It takes as input selected areas from the ChIP profiling data, e.g. areas obtained from peak calling, or areas selected from annotation, such as promoter regions. Through the graphical user interface, the user can selectively load one or more ChIP profiling data sets, along with a bed format file defining the positions of the profiles to be analysed within the selected profiling data. When the data has been loaded into CATCHprofiles, the selected profiles are presented to the user in the Graph view with each included ChIP experiment plotted in a different colour for easy distinction (Supp.Figure 4). After alignment and clustering, the result is visualized in two different types of displays, the Cluster view (Supp.Figure 5) to explore the tree obtained by the clustering, and the Branch view to visualize and compare profile patterns at selected branches of the tree. The graphical interface allows the user to examine and select distinctive ChIP-profile patterns and the corresponding branches of the tree for further analysis. At any level in the tree the average profile patterns and the genomic positions of the profiles can be exported as plain text while clusters can be marked and saved for later browsing in CATCHprofiles.

Computational efficiency

The exhaustive all-against-all comparison and alignment in the CATCH algorithm comes at a cost in computation time. Since the similarity score is calculated per track in the pairwise comparisons, adding more ChIP experiments (signal tracks) to the profiles adds linearly to the computation time. Adding more profiles, however, causes a quadratic increase in pair-wise profile comparisons and computation time. We have therefore implemented the CATCH clustering algorithm in C, optimizing for both memory efficiency and computation speed. Furthermore, we have enabled parallel computation of the comparison scores, so the computation time scales inversely with the number of available processors (see Supplementary Methods: Parallel Implementation).

Clustering of PolII sites and alignment of promoters

We demonstrate the capability of CATCH for unbiased discovery by clustering regions of PolII binding in the ChIP-seq dataset of PolII, H2A.Z and a selection of histone modifications from Wang *et al.*[15] . In these data CATCHprofiles detects a cluster of 2093 profiles with a high signal for H3K4me3 and for almost all the histone acetylation marks under study, a profile pattern that has been reported for actively transcribed promoters[15] (Figure 1). We validated the positions of the profiles in the cluster to be enriched in promoters by comparing to annotation. Indeed, 81% of the profiles are within 1kb of annotated Ensembl TSS. From the remaining 19% more than half (253/389) were within 1kb of TSS predicted by Aceview[20] based on transcription data (Supplementary material: cluster12750.xls).

We used the same dataset to study how the alignment changes the average profile of the promoters. We selected the active promoters (TSS) from ENCODE regions and used CATCHprofiles to align the H3K4me3 signals in the promoter regions. When disregarding the direction of transcription, the average TSS has a peak of H3K4me3 on both sides of the centre (Figure 2 A). However, the average profile patterns change when allowing both alignment and mirroring (Figure 2 B, C), revealing that the individual profile patterns are actually asymmetric around the TSS (Figure 2 D).

Clustering of H2A.Z profiles

To demonstrate the power of CATCH for the discovery of new, potentially biologically relevant patterns in ChIP-seq data we analysed the chromatin modification patterns accompanying H2A.Z. H2A.Z is a histone variant that is found throughout the genome. In both yeast and human, H2A.Z occupies two consecutive nucleosomes around the nucleosome-free region at transcriptionally active promoters[21], but little is known about binding patterns at other H2A.Z sites and their functional relevance.

We applied the CATCH algorithm and the CATCHprofiles tool to analyse the patterns around H2A.Z enriched sites using a genome-wide ChIP-seq dataset from human CD4+ cells of histone modifications, RNApolII and CTCF[15]. The dendrogram of the total 37456 ChIP-seq profile regions contained seven major clusters (Figure 3). Each of the clusters presented a unique combination and shape of binding patterns around the H2A.Z signal. The average profiles of the clusters were viewed and exported from the CATCHprofiles tool.

One cluster pattern (cluster 35517) consisted of H2A.Z binding sites with no apparent PolII, CTCF or histone mark. Another cluster (cluster 37112) has an H2A.Z peak co-located with peaks for H3K4me and H3K9me. Two of the clusters (cluster 37163 and 36420) have patterns closely resembling the known pattern of active promoters[6,15], the main difference between them is that cluster 36420 has a CTCF peak immediately adjacent to the PolII peak while cluster 37163 has no CTCF. And finally, three clusters (cluster 36426, 36884 and 36899) have novel and asymmetric patterns with a CTCF peak flanking the H2A.Z and around them varying degrees of histone methylation (Figure 3 and Supp.Figure 6).

For each cluster, we extracted and compared the genomic context of the regions in the cluster with the whole-genome distribution of H2A.Z sites to asses which cluster pattern was over-represented in genomic regions located at 5' end of genes, 3' end of genes, in introns, in exons and gene distant regions (see Methods).

Gratifying, the two clusters that contain patterns resembling active promoters (cluster 37163 and 36420) contained regions close to annotated promoter regions (83% and 80% were within 5kb of annotated TSS, respectively).

CTCF/H2A.Z asymmetric patterns

Of particular interest are the three clusters in which the CTCF protein co-occurs with H2A.Z. Each of these three clusters is significantly over-represented in 3' regions of genes as compared to the complete set of H2A.Z sites (Supp.Figure 7). These clusters show a pattern of H2A.Z located asymmetrically near the CTCF binding sites. Instead of an H2A.Z double peak as is seen in the promoter pattern, H2A.Z is present only on one side of the CTCF and thus incorporated in only one of the two neighbouring nucleosomes.

CTCF (CCCTC-binding factor) is a zinc finger protein that has been reported to be critical in regulation of gene expression[22]. The distinct positioning relative to the H2A.Z site uncovered by CATCHprofiles suggests a (possibly indirect) physical link between the CTCF binding site and the adjacent H2A.Z nucleosome. To corroborate the asymmetry of the CTCF/H2A.Z patterns we performed a CTCF motif detection for each profile region and correlated the motif orientation with the orientation of the profile in the CATCH alignment. The orientation of the CTCF/H2A.Z pattern has a highly significant correlation with the orientation of the CTCF motif for each of the clusters that feature the CTCF/H2A.Z peak pattern: cluster 36884 (0.33, $P < e-32$), cluster 36426 (0.39, $P < e-19$), cluster 36420 (0.29, $P < e-5$) while there was no correlation in the remaining clusters (Supp.Table 3).

The CTCF binding affinity to the CTCF motif was investigated by Renda *et al[23]* who showed that of the eleven zinc fingers in the protein, only four are required for strong binding, and these zinc fingers (numbered ZF4 to ZF7) have a specific orientation with respect to the sequence motif. The correlation of the asymmetric CTCF/H2A.Z pattern with the CTCF binding motif indicates that the H2A.Z nucleosome is most likely to be found 3' of the CTCF motif that corresponds to the ZF4 side of the bound CTCF protein (Figure 4).

Two earlier studies on CTCF and nucleosome positioning that did not apply alignment did not report any asymmetric patterns, but instead showed that H2A.Z is highly enriched in nucleosomes flanking the CTCF binding sites[24], and that H2A.Z has one major enrichment peak at the centre of intergenic CTCF-sites[25]. In their recent paper, Lai and Buck[18] did report an asymmetry in the nucleosome pattern as well as in the H2A.Z pattern when they aligned the signal in both forward and reverse direction around a preselected set of 1000 CTCF binding sites. However, in their study Lai and Buck did not find a correlation between the pattern orientation and the orientation of the underlying CTCF motif that links the asymmetry of the pattern to the orientation of the CTCF protein.

Discussion

The analysis of ChIP profiling data aims to discover the functional relevance of DNA-binding proteins. A prerequisite for such discovery is to be able to either detect patterns in sites of known functionality, or the opposite, to interrogate and annotate the function of sites with specific patterns. Both of these approaches require a method for clustering the ChIP profile patterns, and for this purpose we developed CATCHprofiles - a ChIP profile clustering and alignment algorithm integrated in a Java tool to visualize and browse the results.

We designed the CATCH algorithm specifically to handle the structure of ChIP profiling data, including taking advantage of the genome-wide coverage for unbiased discovery: Firstly, CATCH performs an exhaustive comparison and clustering based solely on the signal patterns in the profiles, thus eliminating the need to incorporate pre-existing knowledge, like the presence of Transcription Start Sites, into the search for patterns. Secondly, because the CATCH clustering includes alignment of the profiles, we do not need e.g. annotated Transcription Start Sites (TSS) to align the promoters, and we can actually improve the resolution of annotation-based profiles. When comparing, for known promoters, the average profile based on a TSS alignment with one based on a Chip-profile based alignment using CATCH, the resolution of the average profile improved markedly after CATCH alignment (Figure 2). Thirdly, because CATCH automatically mirrors profiles in the alignment procedure we can detect asymmetric patterns even if we have no prior knowledge about their direction, as shown for both promoters (Figure 2) and H2A.Z patterns (Figure 3). Fourthly, since the ChIP signal can vary between experiments depending on e.g. the difference in affinity of the various antibodies, CATCH incorporates options for normalizing the signal between the experiments included in the clustering to prevent the dominance of e.g. a single high signal track (Supp.Figure 3). Finally, Chip profiling data can have various resolutions and coverage and the internal interpolation in CATCHprofiles allows seamless combination of data of various resolution and coverage.

Next to discovering and characterizing individual binding patterns, CATCH may also be applied to compare binding patterns between cell types. Or, within one cell type, to compare temporal variation in binding patterns by combining the ChIP experiments from different time points. It should thereby be noted that the CATCH algorithm is not limited to ChIP profiling data, but can just as easily be applied to e.g. DNA methylation or DamID[26] profiles. In fact, CATCHprofiles is not dependent on the platform used to produce the data, and the pattern analysis can be applied for any genomic data where shape and the relative genomic location of the signals adds to the biological interpretation of the result.

The challenge for many high-throughput analysis techniques is the handling and visualization of the high-dimensional data. Often a viable solution is abstraction, as when plotting in the space of principal components when using principal component analysis[27] for clustering. But in the cases where representative and intuitive visualization is feasible, the tools that provide a graphical visualization often achieve the highest resonance in the scientific community, as was the case with the alignment program ClustalW which has had a full graphical interface since 1997[28]. CATCHprofiles provides the ChIP profiling community with an efficient implementation of an exhaustive alignment and clustering algorithm alongside an easy-to-use interactive graphical display of the results.

CATCHprofiles - with example datasets and installation instructions - is available for download from http://catch.cmbi.ru.nl

Methods

Implementation

The CATCHprofiles tool is implemented using a combination of two programming languages; Java and C. The graphical user interface is implemented in Java, while the CATCH clustering and alignment algorithm is implemented in C as the CATCHprofiles clustering engine. To accommodate the computational load of large-scale analysis we have optimized the CATCHprofiles clustering engine for parallel efficiency and achieved a close to linear scaling with the number of cores (Supp.Figure 9). The speed-up plot was produced from benchmarks on an 8-core system. Based on the algorithm design and the parallel implementation, the running time of the CATCHprofiles clustering engine scales quadratically with the number of profiles and linearly with the number of signal tracks. A more detailed description of the parallel implementation is available in Supplementary Methods: Parallel implementation.

H2A.Z enriched sites

The binding sites were defined by peak calling on the H2A.Z ChIP-seq data from Wang et al[15] using the peak calling program MACS with default settings resulting in a total of 37456 sites. We then defined the profiles for the analysis as the 5000bp windows around the H2A.Z sites and we selected 11 ChIP-seq tracks of histone modifications (H3K18ac, H3K9ac, H4K5ac, H4K8ac,H2BK5me1, H3K27me1,H3K4me1, H3K4me2, H3K4me3, H3K9me1, H4K20me1) together with H2A.Z, CTCF and PolII as input to the CATCH algorithm. The computation was executed in parallel on a 64-core machine. Determination of genomic context and the comparison of genomic distributions were done using the online tool PinkThing based on Ensembl NCBI 36 gene annotation (http://pinkthing.cmbi.ru.nl).

Acknowledgements

The authors would like to thank all the students who have contributed to the development and improvement of the CATCHprofiles tool, including Moniek Riemersma, Maarten Kooyman and the GiPCATCH team members: Daan Pijper, Julius Muecke, Berry Lijklema, Richard Willems and Mark Zandstra. The authors would also like to thank Hendrik Marks for useful discussion and feedback on the manuscript. Special thanks to SARA for the use of the Huygens supercomputer and to the Life Sciences Department of Barcelona Supercomputing Center, Barcelona.

Author contributions

F. Nielsen: developed the idea of hierarchical clustering with alignment in the CATCH algorithm, developed the analysis workflow and user interface in collaboration with the GiPCATCH students, analysed the ChIP datasets, tested the algorithm, implemented user interface features, fixed bugs in the implementation and wrote the paper. Kasper Markus: Implemented the first JAVA version of the CATCH algorithm and user interface. Rune Friborg: Implemented C version of the CATCH algorithm, optimized it for memory-efficient parallel execution and wrote the methods on efficient implementation. Lene Favrholdt: developed the idea of hierarchical clustering with alignment in the CATCH algorithm and supervised Kasper Markus. Henk Stunnenberg: supervised F. Nielsen and edited the manuscript. Martijn Huynen: supervised F. Nielsen and edited the manuscript.

Figures

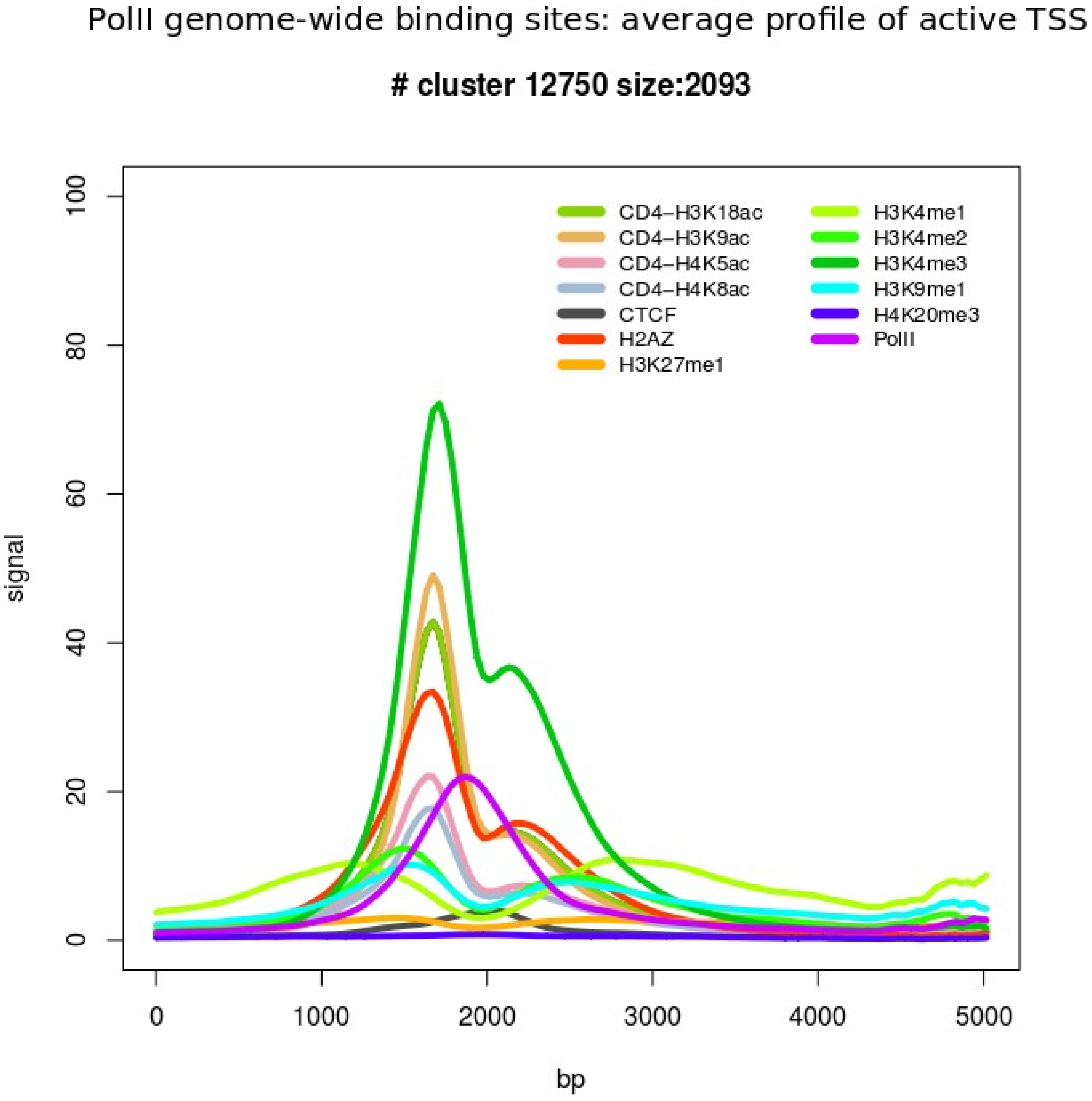


Figure 1: Example profile of PolII cluster with marks of active transcription. The average profile pattern of cluster 12750 (containing 2093 profiles) from the CATCH clustering of PolII binding sites. The profile pattern has a high signal for both H3K4me3 and all the histone acetylation marks, which are known to correlate with active transcription. 81% of the profiles are within 1kb of annotated Ensembl TSS, and of the remaining 389 regions, 253 were within 1kb of Aceview predicted TSS.

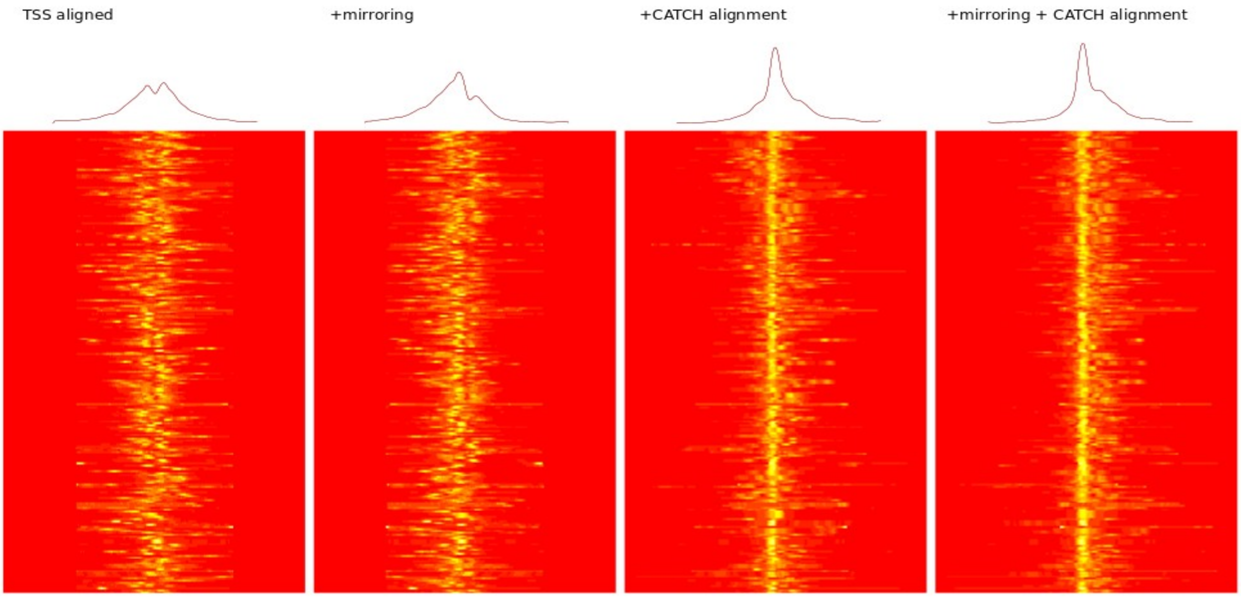


Figure 2: The effect of CATCH alignment on H3K4me3 profile on a subset of ENCODE TSS. A set of 241 promoter regions with high H3K4me3 was selected from the CATCH analysis of ENCODE TSS. The H3K4me3 signal is shown (a) aligned by the genomic position of the TSS disregarding the direction of the TSS (b) aligned by TSS and allowing mirroring (c) by CATCH alignment without mirroring (d) by CATCH alignment and mirroring. The alignment becomes better and the average signal more localized when using both mirroring and CATCH alignment.

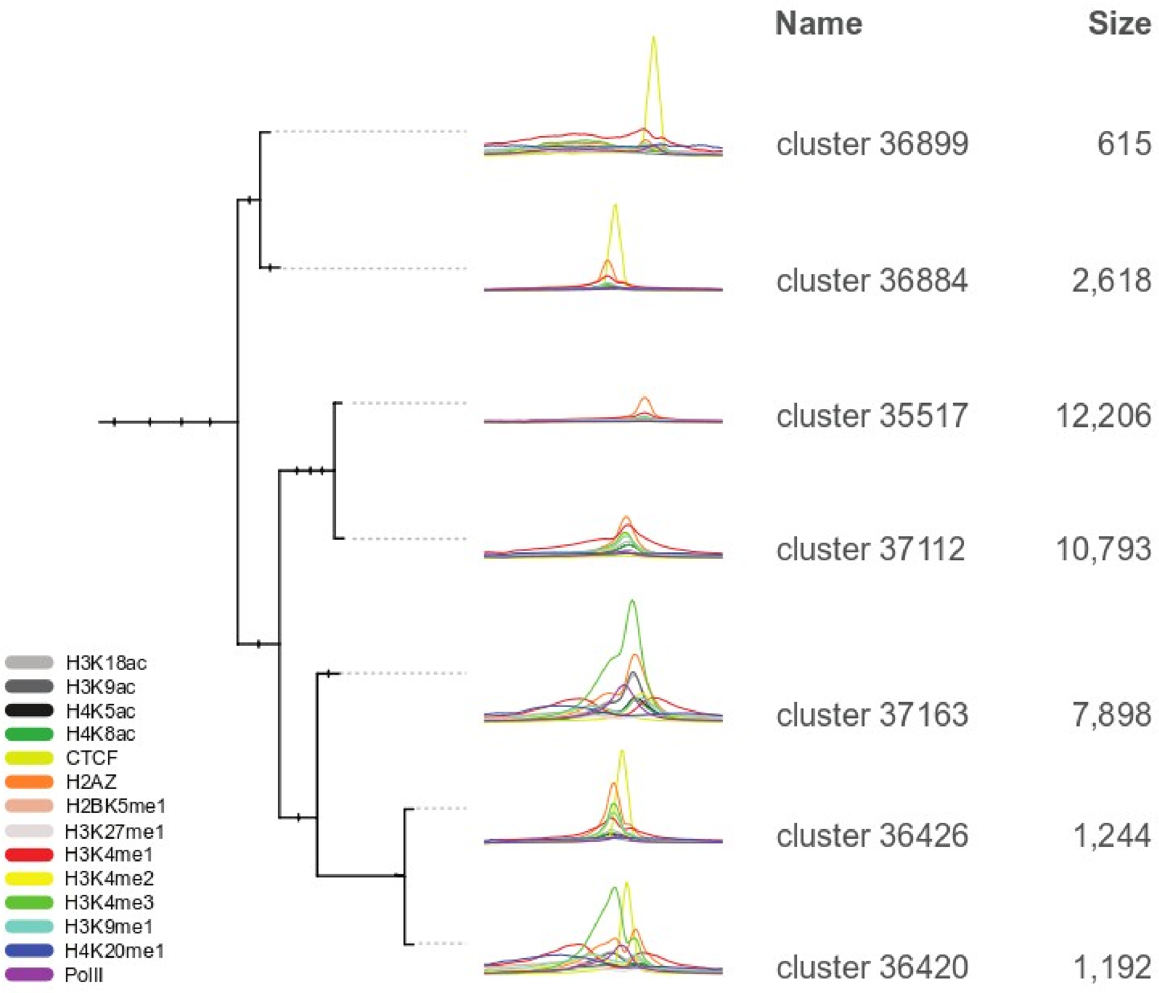


Figure 3: Dendrogram with overview of H2A.Z clusters. The tree of the 37456 H2A.Z profiles has been collapsed to show only the relation and patterns of the seven main clusters. Cluster profile patterns are shown in detail in Supp.Figure 6.

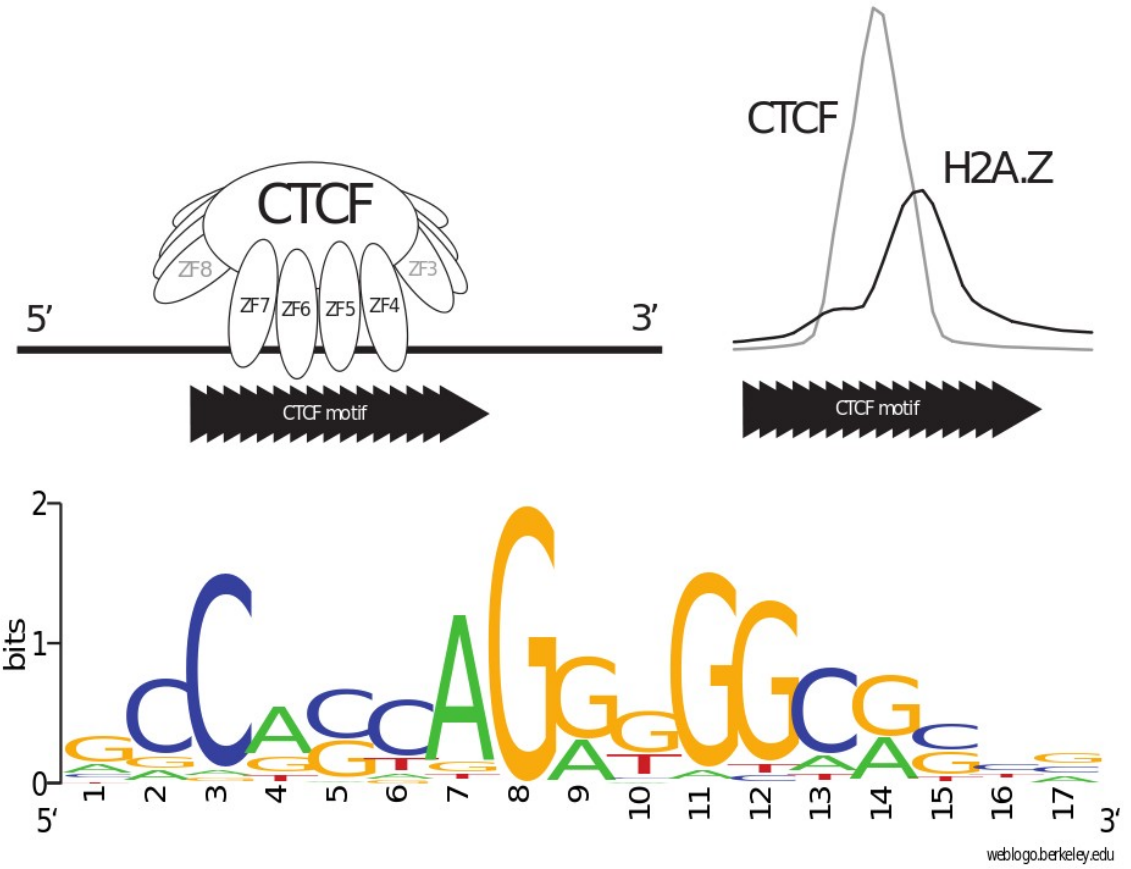


Figure 4: The orientation of the CTCF/H2A.Z pattern is correlated with the orientation of the CTCF binding motif. (a) Of the eleven zincfingers in CTCF, only four are required for strong binding. The orientation of the binding with respect to the CTCF motif was determined by Renda et al[23]. (b) The dominant orientation of the CTCF/H2A.Z pattern with respect to the orientation of the underlying CTCF motif. (c) The CTCF motif as derived from motif detection in genome-wide CTCF peaks in the ChIP-seq dataset of Barski et al[6].

Supplementary Methods

The CATCH algorithm

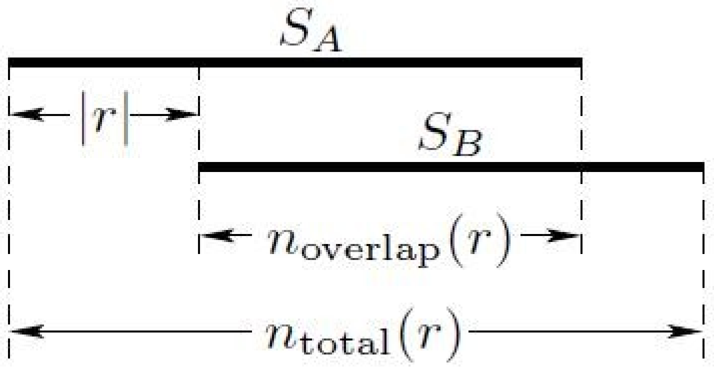
The CATCH algorithm uses a standard hierarchical clustering approach: it keeps a pool of profiles from which it iteratively chooses the most similar pair, according to the chosen similarity measure. Initially, this pool is the set of input profiles. Each time a most similar pair (P_1, P_2) is chosen, P_1 and P_2 are merged to a representative profile P', and P_1 and P_2 are replaced by P' in the profile pool. The sequence of merging determines the topology of the dendrogram (Supp.Figure 2). When running CATCHprofiles, it is possible to choose between different similarity scores, normalization methods, etc. These options are described in the following sections and summarized in Supp.Table 1.

Profile similarity measures

The choice of the alignment scoring measure will determine which type of signal similarity is given the highest score and thus also determine the sequence of clustering of the profiles. CATCH implements three different scoring schemes, described in detail in the sections below:

- *Cross Correlation* (CC), which is used extensively in signal analysis, see e.g. Smith[29]. It is equivalent to the vector dot product, which is also sometimes called the inner product.
- *Pearson's Correlation Coefficient* (PC), which measures the linear correlation between two signals. It can be seen as a normalized version of CC.
- *Sum of Squared Differences* (SSD).

The three scoring measures are all described in Gelder et al[30].
For two profiles containing multiple tracks of signal sequences, the scores are calculated per track, and the alignment score for the complete profile is computed as the average of the alignment score per track.



Supp.Figure 1: Illustration of the shift r and the numbers $n_{\text{overlap}}(r)$ and $n_{\text{total}}(r)$

The alignment of two signal sequences $S_A[0..n_A-1]$ and $S_B[0..n_B-1]$ is characterised by an integer r. If r is positive, S_B is shifted r positions to the left, relative to S_A. If r is negative, S_B is shifted $-r$ positions to the right. The latter situation is illustrated in Supp.Figure 1.

For a given score, CC, PC, or SSD, the *alignment score* $\mathrm{Score}(S_A, S_B, r)$ is calculated for each shift $r \in R = [-r_{\max} .. r_{\max}]$, where $r_{\max} = p \cdot \min(n_A, n_B)$ and $p \in [0;1]$ is a parameter that can be adjusted. The *similarity score* of S_A and S_B is the alignment score of the best shift. Thus, if high values correspond to a good score, like PC and CC, the similarity score is calculated as

$$\mathrm{Score}(S_A, S_B) = \max_{r \in R} \left(\mathrm{Score}(S_A, S_B, r) \right)$$

and if low values correspond to a good score, like SSD, the similarity score is calculated as

$$\mathrm{Score}(S_A, S_B) = \min_{r \in R} \left(\mathrm{Score}(S_A, S_B, r) \right)$$

The following notation is used in the descriptions below (See Supp.Figure 1 for more intuitive definitions).
For each shift r, let $n_{\text{overlap}}(r)$ denote the number of positions in the alignment where both sequences are defined. Then,

$$n_{\text{overlap}}(r) = \begin{Bmatrix} \min\{n_A - |r|, n_B\}, & \text{if } r \leq 0 \\ \min\{n_A, n_B - r\}, & \text{otherwise} \end{Bmatrix}$$

Similarly, let $n_{\text{total}}(r)$ denote the total number of positions in the alignment, i.e.

$$n_{\text{total}}(r) = \begin{Bmatrix} \max\{n_A, n_B + |r|\}, & \text{if } r \leq 0 \\ \max\{n_A + r, n_B\}, & \text{otherwise} \end{Bmatrix}$$

Finally, let

$$I_{\text{overlap}}(r) = \begin{Bmatrix} \left[|r| .. n_{\text{overlap}}(r) + |r| - 1 \right], & \text{if } r \leq 0 \\ \left[0 .. n_{\text{overlap}}(r) - 1 \right], & \text{otherwise} \end{Bmatrix}$$

and

$$I_{\text{total}}(r) = \begin{Bmatrix} \left[0 .. n_{\text{total}}(r) - 1 \right], & \text{if } r \leq 0 \\ \left[-r .. n_{\text{total}}(r) - r - 1 \right], & \text{otherwise} \end{Bmatrix}$$

Cross Correlation

The cross correlation alignment score is defined for sequences S_A and S_B as the sum of the signal products at all positions in the alignment where both sequences are defined:

$$CC(S_A, S_B, r) = \sum_{i \in I_{overlap}(r)} S_A[i] \cdot S_B[i + r]$$

Large CC values correspond to good alignments, and low values correspond to bad alignments. The intuition is: If two similar signals are aligned correctly, high signal values are aligned with high signal values, and thus, the high values are amplified as much as possible. An advantage of the cross correlation score is that it is very noise insensitive.

Pearson's Correlation Coefficient

Pearson's correlation coefficient can be seen as a variant of the cross correlation coefficient, where the signals are first normalized by subtracting the mean and dividing by the standard deviation times the square root of the length of the signal. It measures linear correlation and is defined as

$$PC(S_A,S_B,r)=\sum_{i\in I_{overlap}(r)}\frac{S_A[i]-E_r[S_A]}{\sigma_r[S_A]}\cdot\frac{S_B[i+r]-E_r[S_B]}{\sigma_r[S_B]}$$

where

$$E_r[S_A]=\frac{1}{n_{overlap}(r)}\sum_{i\in I_{overlap}(r)}S_A[i]$$

is the average value of S_A in the overlap with S_B and

$$\sigma_r[S_A]=\sqrt{\sum_{i\in I_{overlap}(r)}(S_A[i]-E_r[S_A])^2}$$

is $\sqrt{n_{\text{overlap}}(r)}$ times the standard deviation of S_A in the overlap with S_B . Similarly,

$$E_r[S_B]=\frac{1}{n_{overlap}(r)}\sum_{i\in I_{overlap}(r)}S_B[i+r]$$

is the average value of S_B in the overlap with S_A and

$$\sigma_r[S_B]=\sqrt{\frac{1}{n_{overlap}(r)}\sum_{i\in I_{overlap}(r)}(S_B[i+r]-E_r[S_B])^2}$$

is $\sqrt{n_{\text{overlap}}(r)}$ times the standard deviation of S_B in the overlap with S_A .
The possible values of Pearson's Correlation Coefficient lie between -1 and 1 . A value of 1 means that the two signals are perfectly linearly correlated, a value of 0 means that there is no linear relationship between the signals, and a value of -1 means that they are anticorrelated.

Sum of Squared Differences

When using the sum of squared differences as the scoring measure, one could decide to consider only positions where both sequences are defined, as with the CC score:

$$\sum_{i\in I_{overlap}(r)}\left(S_A[i]-S_B[i+r]\right)^2$$

However, this would generally favour alignments with very large shifts, since they would result in very few terms in the sum.
One solution to this could be to weight the sum by the ratio of the total number of positions in the alignment to the number of overlap positions:

$$\text{SSD}_{\text{overlap}}(S_A,S_B,r)=\frac{n_{total}(r)}{n_{overlap}(r)}\sum_{i\in I_{overlap}(r)}\left(S_A[i]-S_B[i+r]\right)^2$$

This is called weighted SSD.

Another solution could be to consider the full alignment and represent missing signal values by 0:

$$SSD_{total}(S_A, S_B, r) = \sum_{i \in I_{total}(r)} \left(S_A[i] - S_B[i+r]\right)^2$$

where $S_A[i]=0$, for $i<0$ or $i \geq n_A$, and $S_B[i]=0$, for $i<0$ or $i \geq n_B$. Thus, shifting high signal values of one signal past one end of the other signal is penalized significantly more than having small signal values sticking out. CATCH implements both options.

Signal normalization

Depending on the choice of similarity measure, high signal values may be either favoured (CC) or disfavoured (SSD) in the calculation of the similarity score. By normalizing the profiles before calculating the alignment and similarity scores, the dependence on signal strength can be removed or weakened. It may be desirable to let the normalization factor depend on both signals. Thus, when comparing two signals S_A and S_B , each signal value in S_A is divided by a normalization factor $\mathrm{Norm}(S_A, S_B)$ depending on S_A and possibly on S_B . Similarly, each signal value in S_B is divided by a normalization factor $\mathrm{Norm}(S_B, S_A)$ depending on S_B and possibly on S_A .

With Pearson's Correlation Coefficient, additional normalization is not necessary, since the normalization is “built-in” in the measure. The built-in normalization ensures that the results lie between -1 and 1 and thus, in a sense, it improves the interpretation of the results. On the other hand, we do not have the flexibility of choosing between different normalizations. Thus, e.g. two signals that are identical except for a scaling factor will score the same as two signals that are exactly identical.

Sum of Values

Normalizing by the sum of values ensures that only the shape of the signal is evaluated in the similarity measure. However, this also entails that comparing two signal shapes S_A and S_B that are the same except for a scaling factor, will have the same similarity as comparing S_A with itself.

$$Norm_{SV}(S_A, S_B) = \sum_{i=0}^{n_A - 1} S_A[i]$$

$$Norm_{SV}(S_B, S_A) = \sum_{i=0}^{n_B - 1} S_B[i]$$

Largest Maximum Value

If it is desirable to take the scaling into account, so two identical signals will score higher than the same two signals differing by a scaling factor, it is an option to normalize by the maximum value of either sequence. Thus,

$$\mathrm{Norm}_{\mathrm{MV}}(S_A, S_B) = \mathrm{Norm}_{\mathrm{MV}}(S_B, S_A) = \max(m_A, m_B)$$

$$m_A = \max_{0 \leq i \leq n_A - 1} \left(S_A[i]\right),\ m_B = \max_{0 \leq i \leq n_B - 1} (S_B[i])$$

Largest Average Value

As a second way of taking the scaling into account, CATCH also implements normalization by the maximum average signal value of S_A and S_B .

$$\mathrm{Norm}_{\mathrm{AV}}(S_A, S_B) = \mathrm{Norm}_{\mathrm{AV}}(S_B, S_A) = \max\left(a_A, a_B\right)$$

$$a_A = \frac{\sum_{i=0}^{n_A-1} S_A[i]}{n_A} \qquad a_B = \frac{\sum_{i=0}^{n_B-1} S_B[i]}{n_B}$$

Representative profile of a cluster

Every time a pair of profiles is selected as the pair with the highest similarity score, the two profiles are merged to a new representative profile for the cluster. Each position in the representative profile is the (weighted) average of the corresponding positions in the two merged profiles. The merging of profiles has to take into account that the aligned pair may have positions where only one of the two profiles have defined values, possibly due to the shift of the alignment. The merging may also account for the number of original profiles represented by each of the two profiles.

Weighted merging

When merging two profiles, the number of original profiles represented by each of the two profiles may differ.

If weighted merging is chosen, the new representative profile will be a weighted average of the two merged profiles, where the weights are defined per position by the total number of original profiles in each of the two merged profiles.

Pruning the alignment

To determine the merged profile even when there are positions with undefined values for one of the two profiles in the alignment, there are two main options:

1. Cut away the positions with undefined values, so the merged profile will contain only positions with defined values for both profiles
2. Use the value of the profile that does have a value defined at that position

CATCH will prune merged profiles from left and right until reaching a position where the weight is above a given threshold defined as a percentage of the maximum weight of the alignment. The threshold can be set as a parameter (Supp.Table 1). The second case above corresponds to a threshold of 0.

Options

A number of options and parameters can be set for adjusting the alignment and clustering. Available algorithm options are listed in Supp.Table 1.

Parameter	Default	Options
Weighted merge	*	Yes
		No
Similarity Score		CC
	*	SSD
		weighted SSD
		Pearson
Normalization		None
		Sum of both
	*	Largest maximum value
		Largest average value
Maximum pruning	1/15	(percentage)
Minimum overlap	1/5	(percentage)

Supp.Table 1: CATCH algorithm options.

Parallel implementation

When activating a clustering of N selected profiles from the CATCHprofiles java application, the entire job description of data and user-selected parameters are compiled in JSON format and used as input to the CATCH engine written in C. The complete clustering result is then loaded into the java application for visualization.

The CATCHprofiles clustering algorithm has four main components: the initial comparison and similarity score computation for all profile pairs, the selection of the highest scoring profile pair, the merging of the selected pair into a representative profile, and the updating of the similarity score table (Supp.Figure 8: The clustering algorithm flow diagram). In Supp.Table 2 we show the time spent in each of these four main components for three different problem sizes. Increasing the number of tracks, requires more time for similarity score computation while reducing the percentage of time spent selecting the highest scoring profile pair. Increasing the number of profiles adds to the percentage of time spent selecting the highest scoring profile pair. Since both variations are equally important we must optimize for both cases.
We have written a threaded (parallel) implementation of both the pairwise score computations and the selection of the highest scoring profile pair. Since all the parts were contained in loops and we wanted the threaded code to be platform independent, the OpenMP (Open Multi-Processing) API was used to implement the threading. It handles creating and terminating threads, synchronization between threads, shared and local variables and dividing the workload between the created threads.

Profiling benchmark data set	Initial similarity score computation	Selecting the highest scoring profile pair	Updating similarity scores after merging	Merging profiles	Other
1480 profiles, 1 track	40.71% (27.6s)	4.51% (3.1s)	53.94% (36.5s)	0.75% (0.5s)	0.09% (0.1s)
1480 profiles, 2 tracks	42.19% (53.8s)	2.37% (3.0s)	55.15% (70.3s)	0.23% (0.3s)	0.06% (0.1s)
2960 profiles, 1 track	43.80% (110.0s)	9.11% (22.9s)	46.85% (117.7s)	0.19% (0.5s)	0.06% (0.1s)

Supp.Table 2: Time spent in the different parts of the CATCH algorithm as measured on three benchmark data sets.

The running time of the CATCHprofiles clustering engine depends on the size of the input data, the user-specified parameters (see Supp.Table 2) and the available hardware. We benchmark the CATCHprofiles clustering engine using a test data set of 1480 profiles, 2960 profiles and 5920 profiles, all with 8 tracks of 52 data points per profile. The test data sets of 2960 and 5920 profiles were generated by replicating the data set of 1480 profiles. The size of the data set varies with the number of profiles and tracks. The benchmarks are performed by executing the clustering engine in isolation, thus communication with the Java application is not included in the measurements.

The CATCH executable outputs the processing time spent clustering profiles. For specifying the amount of threads, the environment variable OMP_NUM_THREADS is set to the desired threadcount. If OMP_NUM_THREADS is not set, then OpenMP automatically uses a threadcount that matches the available cores on a system. The benchmarks are executed on a computer with 8 cores: two Intel Xeon E5310 Quad Core processors and 8 GB RAM running Ubuntu 9.04. The user-specified parameters equal to the clustered results demonstrated in this paper are:

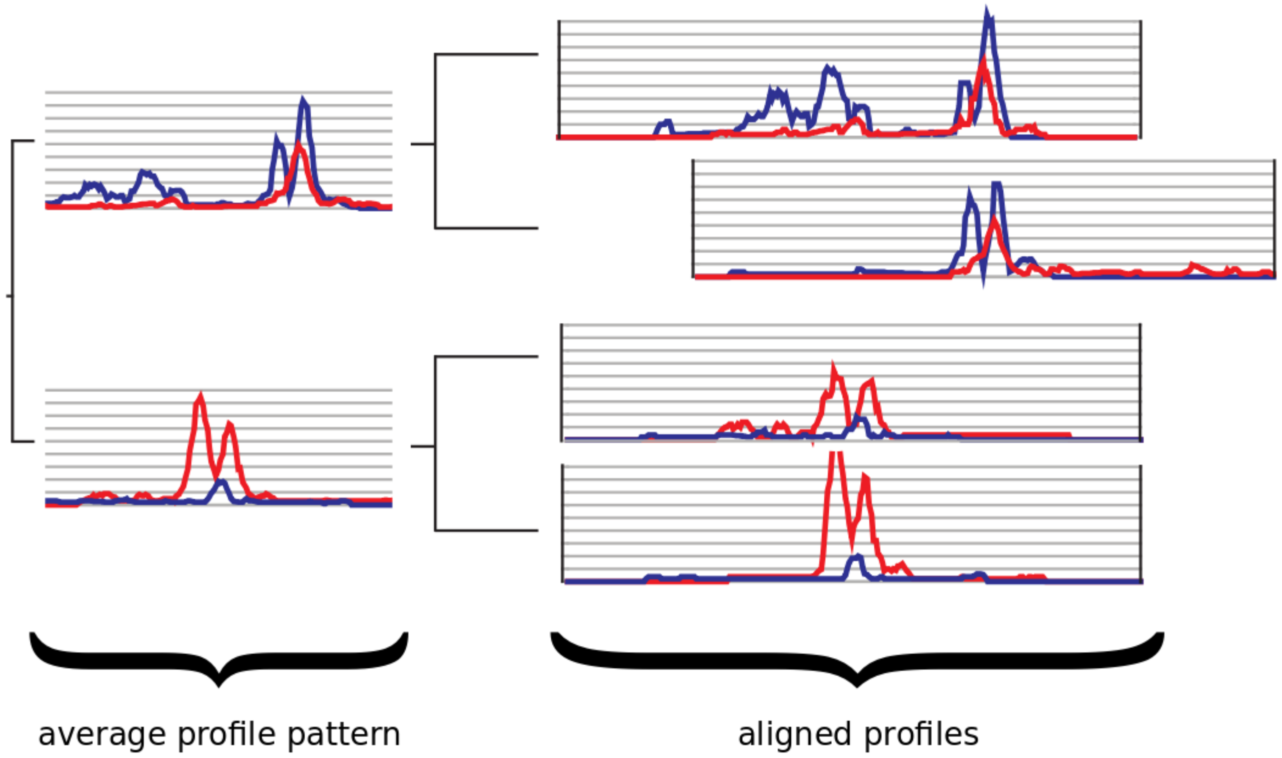
Weighted merge: Yes
Score method: Sum of squared differences
Normalization: Largest average value
Minimum pruning: 6,7%
Minimum overlap: 50%

We measured the total computation time for parallel execution on multiple cores and calculated the speedup gained by comparing to execution on a single core. The speedup plot (Supp.Figure 9) shows a close to linear speedup for up to 8 cores, when executing our benchmarks. The benchmark data set consisted of 5920 profiles, 8 tracks and a track length of 52. On the 8-core machine the CATCH algorithm required approximately 57 minutes to finish, illustrating the algorithm capacity for clustering larger data sets within a reasonable running time.

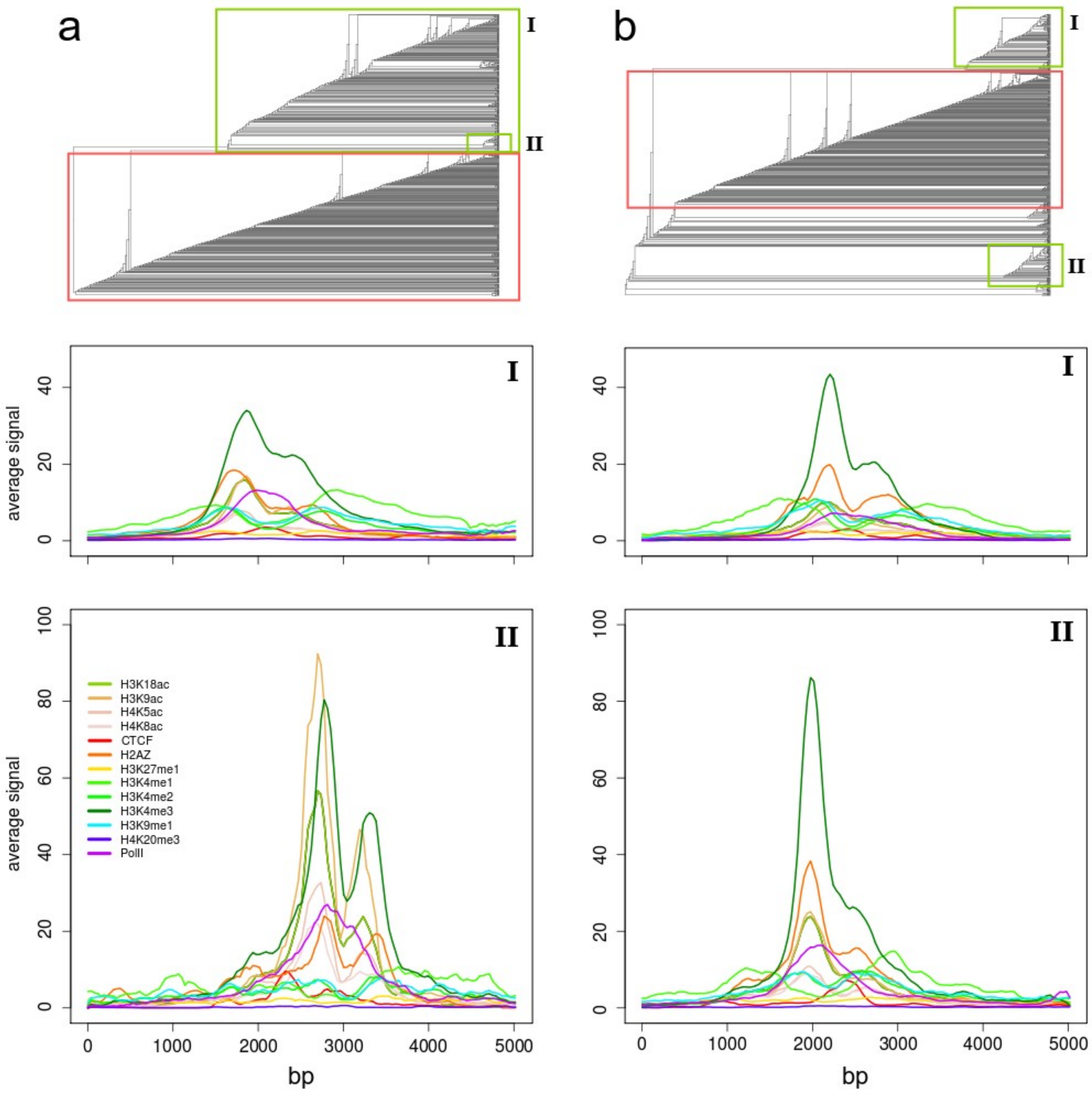
The time spent on alignment of the profiles can be reduced by increasing the percentage minimum overlap of the aligned profiles. Setting the minimum overlap to 100%, requires the shortest of two profiles to be aligned 100%, and for profiles of equal length this corresponds to disallowing alignment. Decreasing the minimum overlap increases the computation time for the similarity score.

The added cost of performing alignment of the profiles is less when the execution is running on 8 cores vs. 1 core. Running time as a function of increasing the minimum overlap is shown for 1, 4 and 8 threads in Supp.Figure 10.

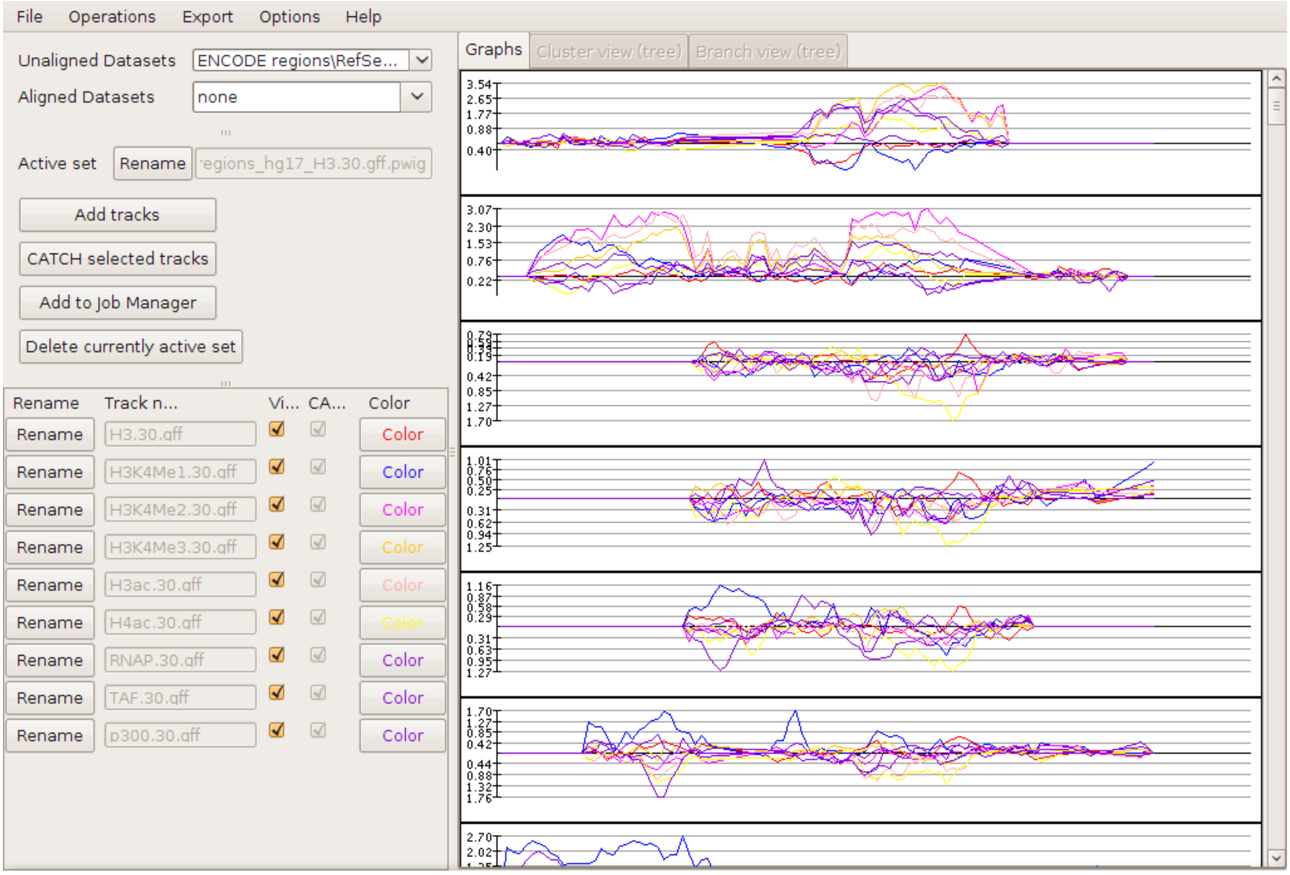
Supplementary Figures



Supp.Figure 2: Conceptual illustration of the CATCH clustering algorithm. Example of clustering four profiles with two tracks of ChIP data, plotted in red and blue respectively. All pairs of profiles are aligned to find the alignment of highest similarity. In each iteration, the profile pair of highest similarity is clustered and their cluster is represented by their average aligned profile. The hierarchical clustering continues until all profiles and clusters are included in the dendrogram.



Supp.Figure 3: Normalization affects the clustering and the resolution of the patterns. (a) with normalization of the signal strength the profiles cluster by the intensity and shape of all tracks equally, resulting in a clear split between patterns of active and inactive promoters as highlighted in the dendrogram with green and red respectively. The inactive promoters pattern is low signal for all the tracks shown. Within the cluster I of active promoters subclusters arise with variations of the active promoter pattern, e.g. cluster II. (b) Without the use of normalization, the intensity of the signals dominates the clustering. Most of the inactive promoter patterns of low signal intensity are still clustered together, highlighted in red. However, the biggest cluster with a pattern resembling the active promoter pattern is cluster I, and it is clustered separately from e.g. cluster II which differs mainly in signal intensity. Clustering using normalization is the recommended and default option for clustering in CATCH to avoid the dominance of high signal tracks in the clustering.



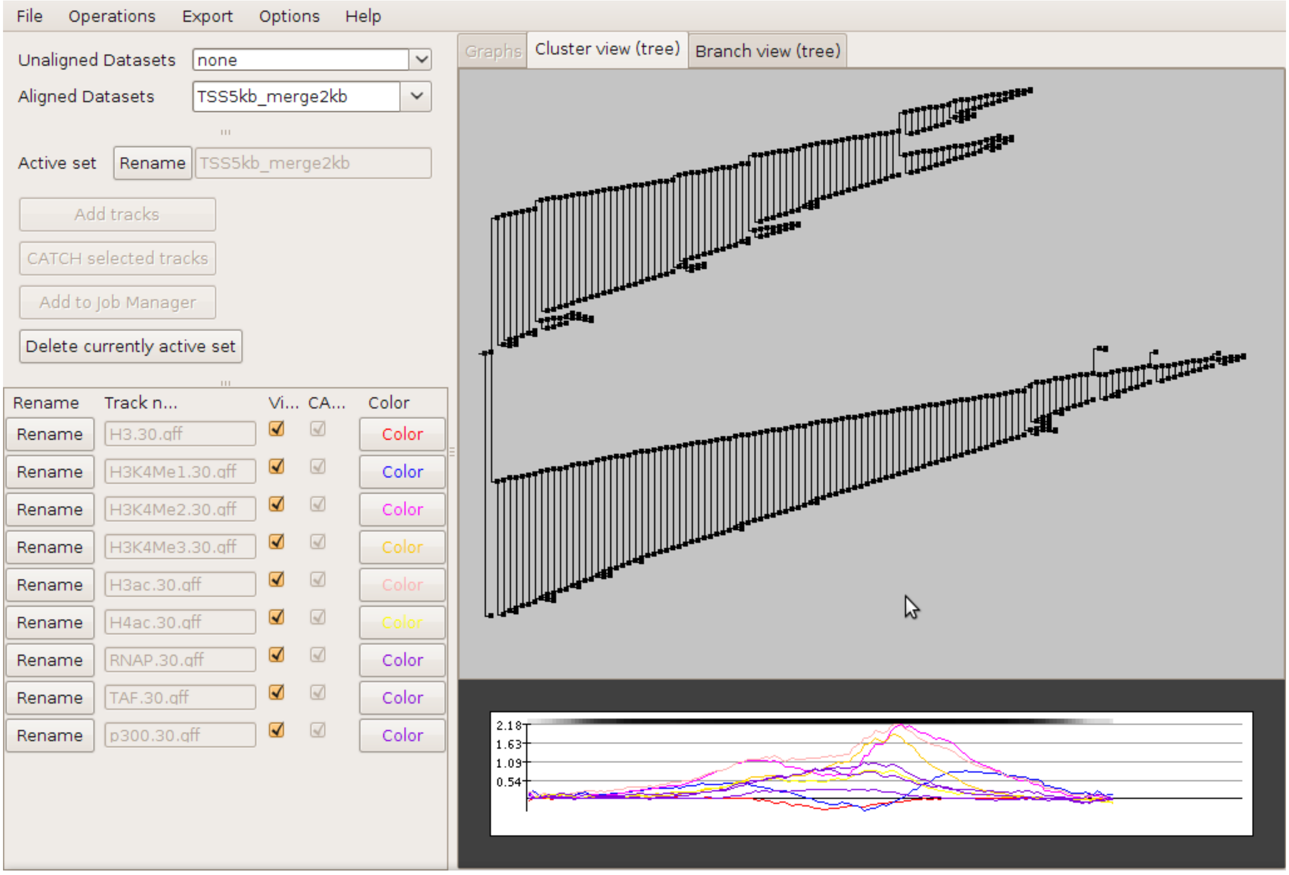
Unaligned Datasets: ENCODE regions\RefSe...

Aligned Datasets: none

Active set: Rename ·egions_hg17_H3.30.gff.pwig

Rename	Track n...	Vi...	CA...	Color
Rename	H3.30.gff	☑	☑	Color
Rename	H3K4Me1.30.gff	☑	☑	Color
Rename	H3K4Me2.30.gff	☑	☑	Color
Rename	H3K4Me3.30.gff	☑	☑	Color
Rename	H3ac.30.gff	☑	☑	Color
Rename	H4ac.30.gff	☑	☑	Color
Rename	RNAP.30.gff	☑	☑	Color
Rename	TAF.30.gff	☑	☑	Color
Rename	p300.30.gff	☑	☑	Color

Supp.Figure 4: CATCH Graph view. After loading a data set of ChIP profiles, the Graph view shows plots of all profile regions. On the left the track names and colours can be adjusted for easy distinction.



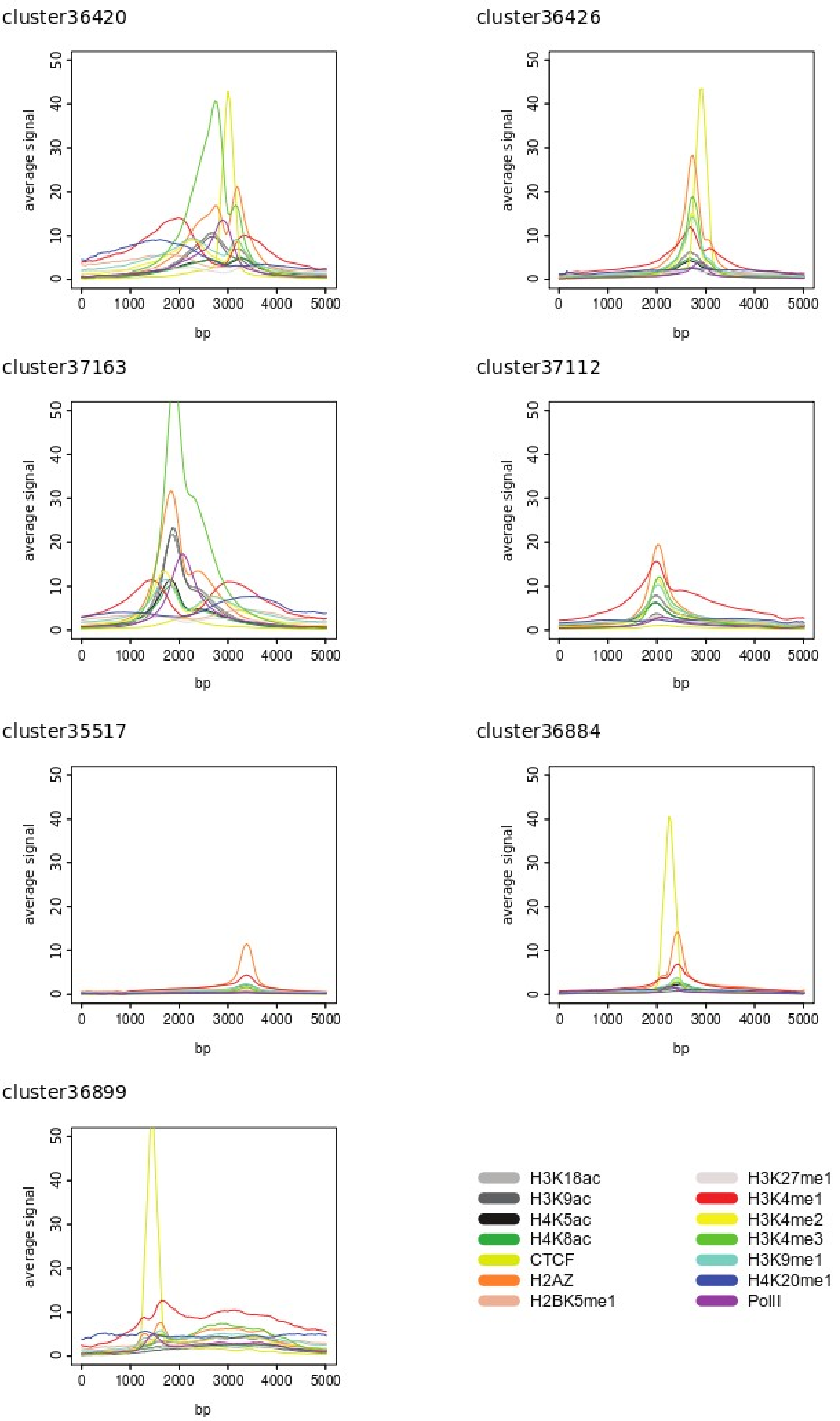
Unaligned Datasets: none

Aligned Datasets: TSS5kb_merge2kb

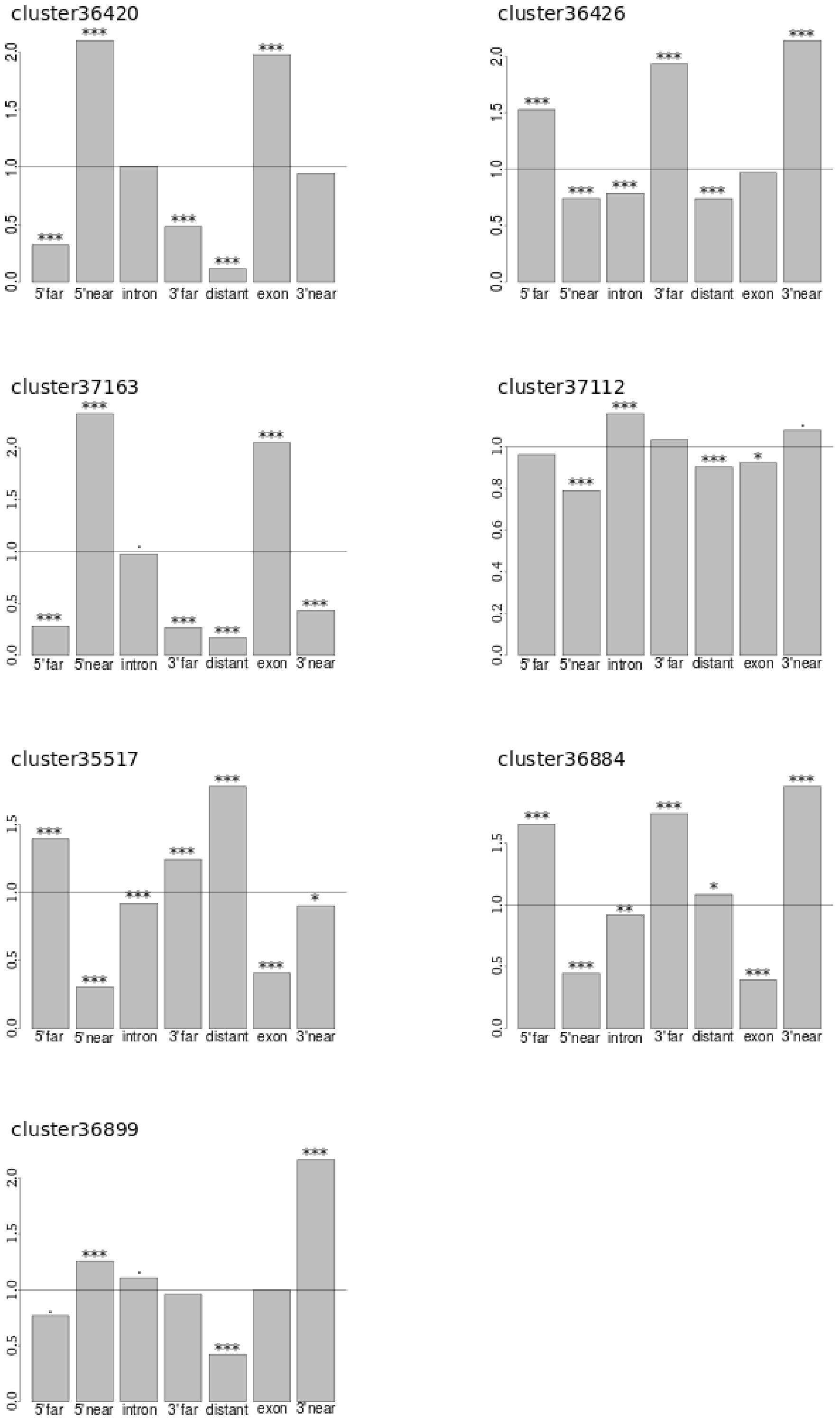
Active set: TSS5kb_merge2kb

Rename	Track n...	Vi...	CA...	Color
Rename	H3.30.gff	☑	☑	Color
Rename	H3K4Me1.30.gff	☑	☑	Color
Rename	H3K4Me2.30.gff	☑	☑	Color
Rename	H3K4Me3.30.gff	☑	☑	Color
Rename	H3ac.30.gff	☑	☑	Color
Rename	H4ac.30.gff	☑	☑	Color
Rename	RNAP.30.gff	☑	☑	Color
Rename	TAF.30.gff	☑	☑	Color
Rename	p300.30.gff	☑	☑	Color

Supp.Figure 5: Screenshot CATCH cluster view. The result of the CATCH clustering algorithm is shown on the right as a dendrogram. The tree can be interactively browsed to examine the average profile patterns at any level in the tree. Individual profiles and subclusters can be exported by right-clicking on the cluster node in the tree. Below the tree, the average profile is shown for the currently selected cluster.



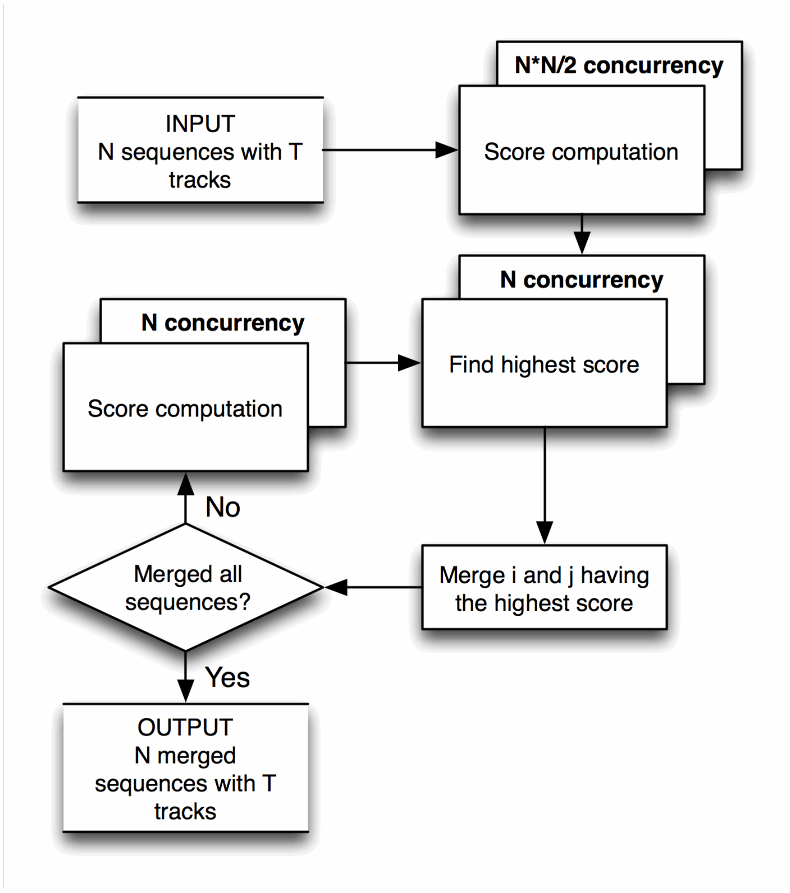
Supp.Figure 6: Detailed view of the H2A.Z genome-wide cluster patterns. Each pattern represents the average profile pattern for the profiles in the cluster. The patterns of clusters 36420 and 37163 contain high signals for PolII, methylation and acetylation marks correlating with active transcription. Four clusters (36420, 36426, 36884 and 36899) have a CTCF peak close to the H2A.Z. The genomic distributions corresponding to these clusters are shown in Supp.Figure 7.



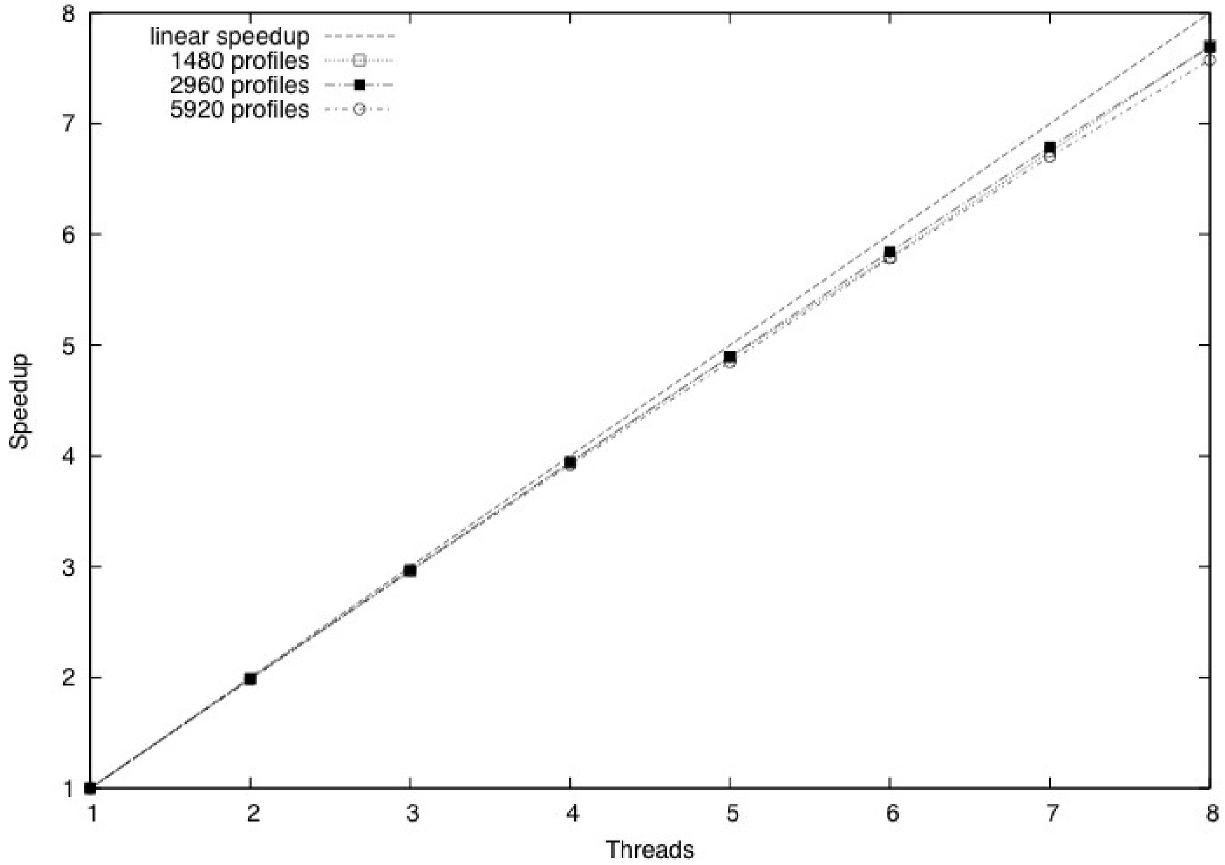
Supp.Figure 7: Genomic distributions of the seven clusters of H2A.Z binding sites. Each plot shows the distribution of the categories: exon, intron, 5'near, 5'far, 3'near, 3'far and distant. The limit for 'near' regions is 5kb, the limit for 'far' regions is 25kb. The categories are shown as numbers relative to the H2A.Z genomic distribution with p-values indicating significant differences per category. The clusters with CTCF, but no acetylation marks, e.g. clusters 36426, 36884 and 36899, are all significantly enriched in the 3' regions of genes.

Cluster name	Brief description	Cluster size	CTCF Motif correlation
36899	Low H2A.Z + CTCF + H3K4me1	615	0.040
36884	H2A.Z + CTCF	2,618	0.326
35517	H2A.Z alone	12,206	0.098
37112	H2A.Z + met	10,793	0.075
37163	H2A.Z + Promoter	7,898	0.019
36426	H2A.Z + CTCF + met	1,244	0.390
36420	H2A.Z + Promoter + CTCF	1,192	0.285

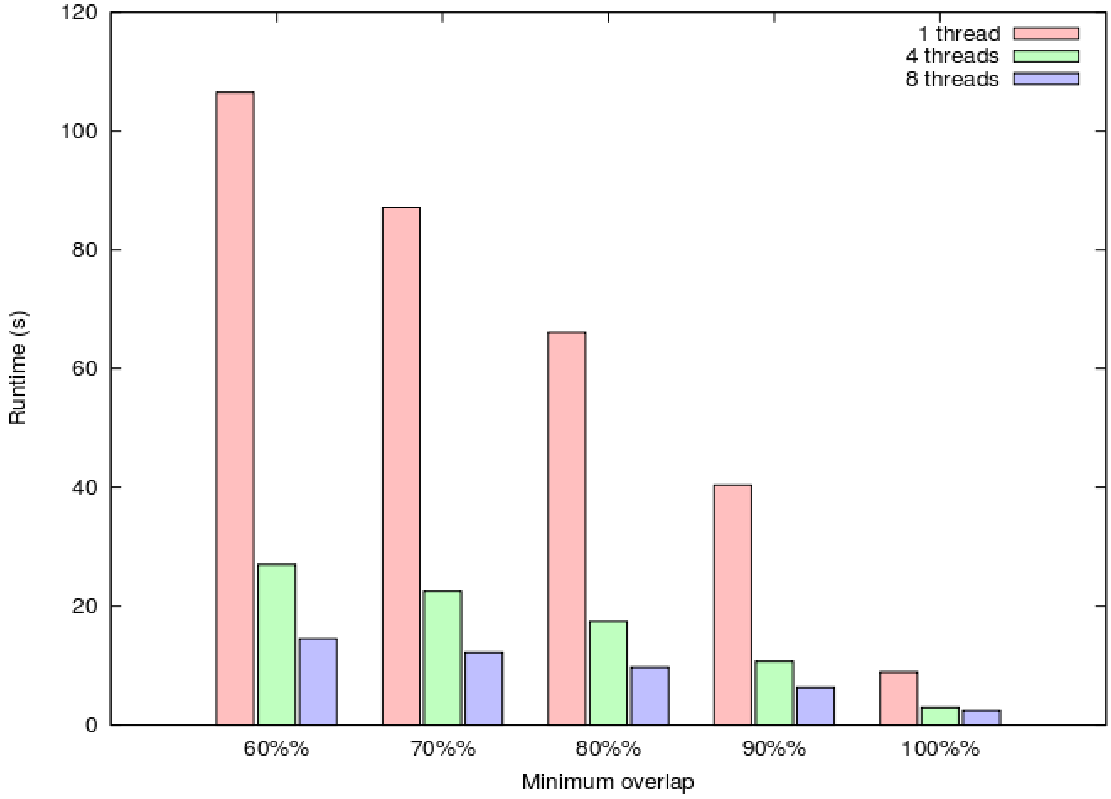
Supp.Table 3: Correlation of pattern orientation with orientation of CTCF motif for each of the H2A.Z clusters. Only the CTCF containing patterns with a clear H2A.Z peak show correlation with the orientation of the CTCF motif. Promoter: Marks of active promoters including PolII, histone acetylation and histone methylation marks. Met: Histone methylation.



Supp.Figure 8: ***CATCH algorithm flow diagram indicating concurrent computation****. Score computation: the initial comparison and similarity score computation for all profile pairs. Find highest score: the selection of the highest scoring profile pair. Merge i and j having the highest score: the merging of the selected pair into a representative profile. Dependencies are visualized by arrows and parallel parts marked with the order of concurrency available.*



Supp.Figure 9: ***Speedup plot of the relative performance increase in the CATCHprofiles clustering engine.*** *The parallel implementation of the CATCH clustering engine results in a near-linear speedup of computation time with increased number of threads. The y-axis shows the speedup, and the x-axis the number of threads used. The profiles contain 8 tracks and the alignment was set to use a minimum overlap of 50%, the other parameters were set to default as listed in Supp.Table 1.*



Supp.Figure 10: ***Running time dependence on alignment.*** *Running time of clustering 1480 profiles with 8 tracks, when the minimum overlap is varied. Results are shown for executions with 1, 4 and 8 threads.*

References

1. Boyer LA, Plath K, Zeitlinger J, Brambrink T, Medeiros LA, et al. (2006) Polycomb complexes repress developmental regulators in murine embryonic stem cells. Nature 441: 349-53. doi:10.1038/nature04733

2. Welboren W-J, Driel MA van, Janssen-Megens EM, Heeringen SJ van, Sweep FC, et al. (2009) ChIP-Seq of ERalpha and RNA polymerase II defines genes differentially responding to ligands. The EMBO journal 28: 1418-28. doi:10.1038/emboj.2009.88

3. Lee TI, Jenner RG, Boyer LA, Guenther MG, Levine SS, et al. (2006) Control of developmental regulators by Polycomb in human embryonic stem cells. Cell 125: 301-13. doi:10.1016/j.cell.2006.02.043

4. Hatzis P, Flier LG van der, Driel MA van, Guryev V, Nielsen F, et al. (2008) Genome-wide pattern of TCF7L2/TCF4 chromatin occupancy in colorectal cancer cells. Molecular and cellular biology 28: 2732-44. doi:10.1128/MCB.02175-07

5. Nielsen R, Pedersen TA, Hagenbeek D, Moulos P, Siersbaek R, et al. (2008) Genome-wide profiling of PPARgamma:RXR and RNA polymerase II occupancy reveals temporal activation of distinct metabolic pathways and changes in RXR dimer composition during adipogenesis. Genes & development 22: 2953-67. doi:10.1101/gad.501108

6. Barski A, Cuddapah S, Cui K, Roh T-Y, Schones DE, et al. (2007) High-resolution profiling of histone methylations in the human genome. Cell 129: 823-37. doi:10.1016/j.cell.2007.05.009

7. Mikkelsen TS, Ku M, Jaffe DB, Issac B, Lieberman E, et al. (2007) Genome-wide maps of chromatin state in pluripotent and lineage-committed cells. Nature 448: 553-60. doi:10.1038/nature06008

8. Guenther MG, Levine SS, Boyer LA, Jaenisch R, Young RA (2007) A chromatin landmark and transcription initiation at most promoters in human cells. Cell 130: 77-88. doi:10.1016/j.cell.2007.05.042

9. Bernstein BE, Mikkelsen TS, Xie X, Kamal M, Huebert DJ, et al. (2006) A bivalent chromatin structure marks key developmental genes in embryonic stem cells. Cell 125: 315-26. doi:10.1016/j.cell.2006.02.041

10. Reid G, Gallais R, Métivier R (2009) Marking time: the dynamic role of chromatin and covalent modification in transcription. The international journal of biochemistry & cell biology 41: 155-63. doi:10.1016/j.biocel.2008.08.028

11. Heintzman ND, Stuart RK, Hon G, Fu Y, Ching CW, et al. (2007) Distinct and predictive chromatin signatures of transcriptional promoters and enhancers in the human genome. Nature genetics 39: 311-8. doi:10.1038/ng1966

12. Berger SL (2007) The complex language of chromatin regulation during transcription. Nature 447: 407-12. doi:10.1038/nature05915

13. Barrera LO, Ren B (2006) The transcriptional regulatory code of eukaryotic cells – insights from genome-wide analysis of chromatin organization and transcription factor binding. Current Opinion in Cell Biology 18: 7. doi:doi:10.1016/j.ceb.2006.04.002

14. Fischer JJ, Toedling J, Krueger T, Schueler M, Huber W, et al. (2008) Combinatorial effects of four histone modifications in transcription and differentiation. Genomics 91: 41-51. doi:10.1016/j.ygeno.2007.08.010

15. Wang Z, Zang C, Rosenfeld JA, Schones DE, Barski A, et al. (2008) Combinatorial patterns of histone acetylations and methylations in the human genome. Nature genetics 40: 897-903.

doi:10.1038/ng.154

16. Hon G, Ren B, Wang W (2008) ChromaSig: a probabilistic approach to finding common chromatin signatures in the human genome. PLoS computational biology 4: e1000201. doi:10.1371/journal.pcbi.1000201

17. Bishop CM (2007) Pattern Recognition and Machine Learning (Information Science and Statistics). Springer. 738 p.

18. Lai W, Buck MJ (2010) ArchAlign: Coordinate-free chromatin alignment reveals novel architectures. Genome biology 11: R126. doi:10.1186/gb-2010-11-12-r126

19. Sneath PHA, Sokal RR (1973) Numerical Taxonomy: The Principles and Practice of Numerical Classification. W.H.Freeman & Co Ltd. p.

20. Thierry-Mieg D, Thierry-Mieg J (2006) AceView: a comprehensive cDNA-supported gene and transcripts annotation. Genome biology 7 Suppl 1: S12.1-14. doi:10.1186/gb-2006-7-s1-s12

21. Tolstorukov MY, Kharchenko PV, Goldman JA, Kingston RE, Park PJ (2009) Comparative analysis of H2A.Z nucleosome organization in the human and yeast genomes. Genome research 19: 967-77. doi:10.1101/gr.084830.108

22. Phillips JE, Corces VG (2009) CTCF: master weaver of the genome. Cell 137: 1194-211. doi:10.1016/j.cell.2009.06.001

23. Renda M, Baglivo I, Burgess-Beusse B, Esposito S, Fattorusso R, et al. (2007) Critical DNA binding interactions of the insulator protein CTCF: a small number of zinc fingers mediate strong binding, and a single finger-DNA interaction controls binding at imprinted loci. The Journal of biological chemistry 282: 33336-45. doi:10.1074/jbc.M706213200

24. Fu Y, Sinha M, Peterson CL, Weng Z (2008) The insulator binding protein CTCF positions 20 nucleosomes around its binding sites across the human genome. PLoS genetics 4: e1000138. doi:10.1371/journal.pgen.1000138

25. Jin C, Zang C, Wei G, Cui K, Peng W, et al. (2009) H3.3/H2A.Z double variant-containing nucleosomes mark “nucleosome-free regions” of active promoters and other regulatory regions. Nature genetics 41: 941-5. doi:10.1038/ng.409

26. Steensel B van, Henikoff S (2000) Identification of in vivo DNA targets of chromatin proteins using tethered dam methyltransferase. Nature biotechnology 18: 424-8. doi:10.1038/74487

27. Pearson K (1901) On lines and planes of closest fit to systems of points in space. Philosophical Magazine 2: 559-572.

28. Larkin MA, Blackshields G, Brown NP, Chenna R, McGettigan PA, et al. (2007) Clustal W and Clustal X version 2.0. Bioinformatics (Oxford, England) 23: 2947-8. doi:10.1093/bioinformatics/btm404

29. Smith SW (1997) The Scientist & Engineer′s Guide to Digital Signal Processing. California Technical Pub. p.

30. Gelder R de, Wehrens R, Hageman JA (2001) A generalized expression for the similarity of spectra: application to powder diffraction pattern classification. Journal of Computational Chemistry 22: 273-289. doi:10.1002/1096-987X(200102)22:3<273::AID-JCC1001>3.0.CO;2-0

A.3 CSPBuilder - CSP based Scientific Workflow Modeling

Communicating Process Architectures 2008, WoTUG-31, Proceedings of the 31st WoTUG Technical Meeting, University of York, Yorkshire, September 7-10, 2008
ISBN: 978-1-58603-907-3, Concurrent Systems Engineering 66, IOS Press, pp. 347 – 369

Rune Møllegaard Friborg, Brian Vinter: CSPBuilder - CSP based Scientific Workflow Modeling

Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

CSPBuilder – CSP based Scientific Workflow Modelling

Rune Møllegård FRIBORG and Brian VINTER

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

{runef , vinter} @diku.dk

Abstract. This paper introduces a framework for building CSP based applications, targeted for clusters and next generation CPU designs. CPUs are produced with several cores today and every future CPU generation will feature increasingly more cores, resulting in a requirement for concurrency that has not previously been called for. The framework is CSP presented as a scientific workflow model, specialized for scientific computing applications. The purpose of the framework is to enable scientists to exploit large parallel computation resources, which has previously been hard due of the difficulty of concurrent programming using threads and locks.

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

Introduction

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

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

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

1. Background

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

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

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

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

1.1. Motivation

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

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

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

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

Recent reports of using the GPU[1] and CELL-BE for scientific computing, have reported performance increases of up to 100-fold for some scientific algorithms. However, the diffi-

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

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

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

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

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

1.2. PyCSP

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

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

1.3. Scientific Workflow Modelling and CSP

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

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

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

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

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

1.4. Summary of Contributions

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

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

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

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

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

2. The Visual Tool

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

In CSPBuilder every application starts with a blank canvas, where processes and channels can be inserted. Processes appear as named boxes, with their external connections labelled. Channels are shown as lines connecting the processes. To simplify things, any in-

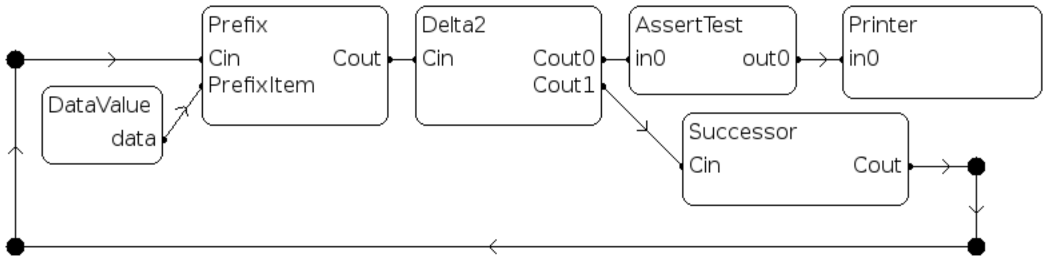


Figure 1. A CSPBuilder application that generates incrementing natural numbers.

bound or outbound connection will only accept one channel going in or out, depending on the connection type.

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

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

2.1. Component System

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

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

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

2.1.1. Scalable Organisation

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

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

2.1.2. Components

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

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

```python
from common import *

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

Listing 1. Example CSP process implementation – the IDProcess

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

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

2.1.3. Component Library

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

2.1.4. A Wizard for Building Components

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

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

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

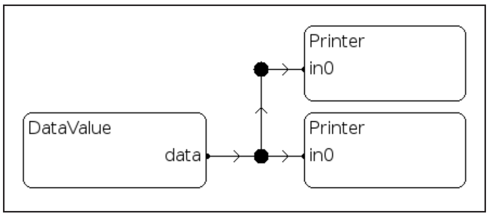


Figure 2. One2AnyChannel formed by connecting three processes to a single connection point, single outputting process, multiple inputters.

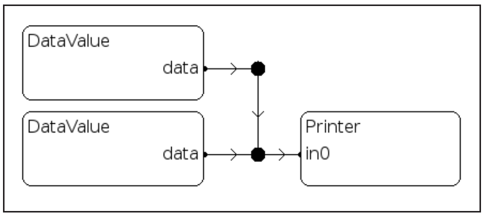


Figure 3. Any2OneChannel formed by connecting three processes to a single connection point, single inputting process, multiple outputters.

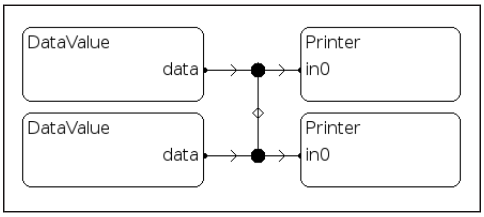


Figure 4. Any2AnyChannel formed by connecting four processes to a single connection point, multiple inputting and outputting processes.

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

2.2. Channels and Connection Points

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

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

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

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

2.3. Configuring a Component

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

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

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

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

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

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

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

2.4. Process Replication

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

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

default_data = None

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

    saveDir = os.getcwd()

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

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

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

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

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

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

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

3. Concurrent Execution

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

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

3.1. Building and Executing a Process Network

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

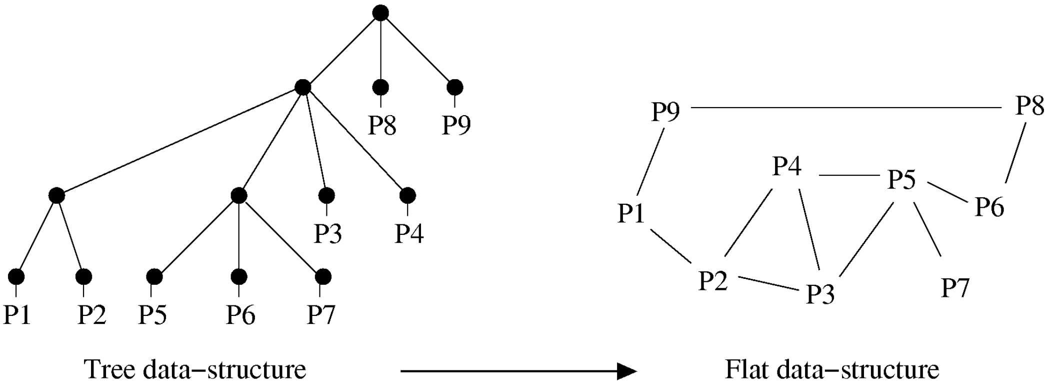


Figure 5. Data structures. In the left figure the tree data-structure is illustrated, which represents the structure of the CSP network when the `.csp` files are parsed. The black dots are a process structure and the lines represent any number of connection structures. This data-structure is converted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

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

3.1.1. Multiplying Processes

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

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

3.1.2. Channel Poisoning

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

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

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

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

3.1.3. Importing External Code

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

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

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

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

3.1.4. Releasing the GIL

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

To be able to utilize the processors in an SMP system we will release the GIL while doing computations outside the domain of Python. In section 3.1.3 it was explained how external code can be imported into Python. When calling into Fortran code using F2PY the

GIL is released automatically and acquired again when returning to Python. With C and C++ the situation is different, because here it is possible to access Python objects by using the API declared in `python.h`. It is the responsibility of the component developer to not access Python objects while the GIL is released. Releasing and acquiring is done with the following macros defined in `python.h`:

```
// Release GIL
Py_BEGIN_ALLOW_THREADS

// Acquire GIL
Py_END_ALLOW_THREADS
```

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

3.2. Performance Evaluation

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

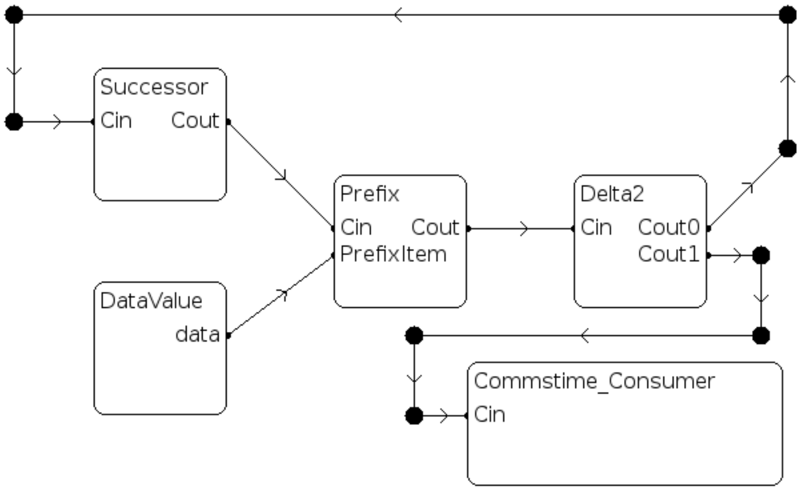


Figure 6. Commstime. A CSPBuilder application that resembles the Commstime performance test.

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

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

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

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

4. Experiments

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

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

4.1. Prime Factorisation

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

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

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

4.1.1. Implementation Details

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

[2] Trial division: `http://mathworld.wolfram.com/TrialDivision.html`

[3] Direct search factorisation: `http://mathworld.wolfram.com/DirectSearchFactorization.html`

[4] PLEAC: `http://pleac.sourceforge.net/pleac_python/numbers.html`

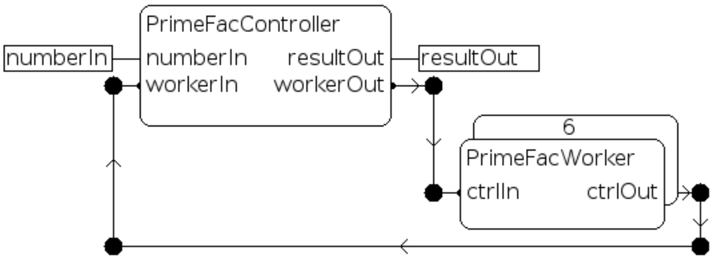


Figure 7. PrimeFac Component, consisting of a controller and a worker multiplied 6 times.

On initialisation, the worker process sends an empty result to the controller, to indicate that it is ready for more work. The controller loops until all primes have been found, sending jobs to and collecting results from workers. If a non-empty result is received, the controller 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 $[2 \ldots d]$ are the divisors tested. All the prime factorisations of n can be found in $[2 \ldots \sqrt{n}]$.

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

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

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

4.1.2. Performance Evaluation

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

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

In the performance test we compare the two implementations, one with the worker written in Python and one with the worker as an external component written in C which also releases the GIL. In the C implementation we use the large number library *LibTomMath*. This large number implementation is actually slower than the large number implementation

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

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

in Python, shown in the tests where the “Python only” version outperforms the “Python and C” version for the case with only one worker. We base this conclusion on the fact that the sequential test for “Python and C” finishes in 1547 minutes, while the “Python only” version finishes in 1005 minutes. Both implementations spend all of the execution time in the worker loop with very little communication between processes.

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

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

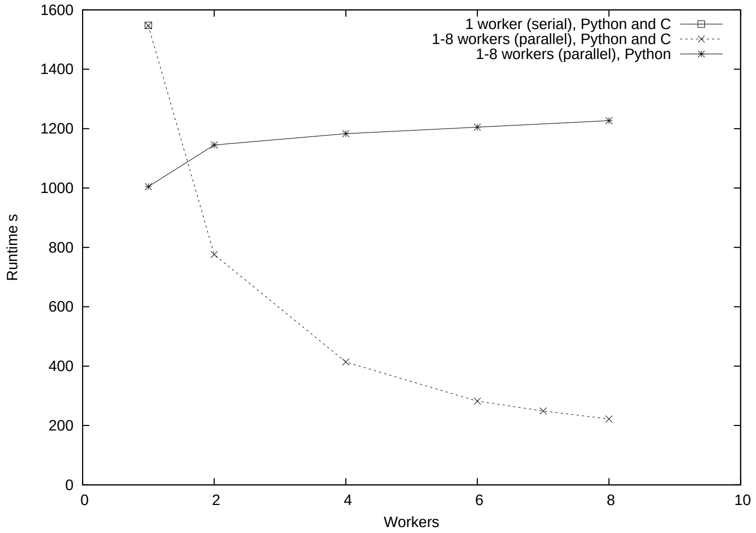


Figure 8. Prime factorisation of the Mersenne number $2^{222} - 1$.

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

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

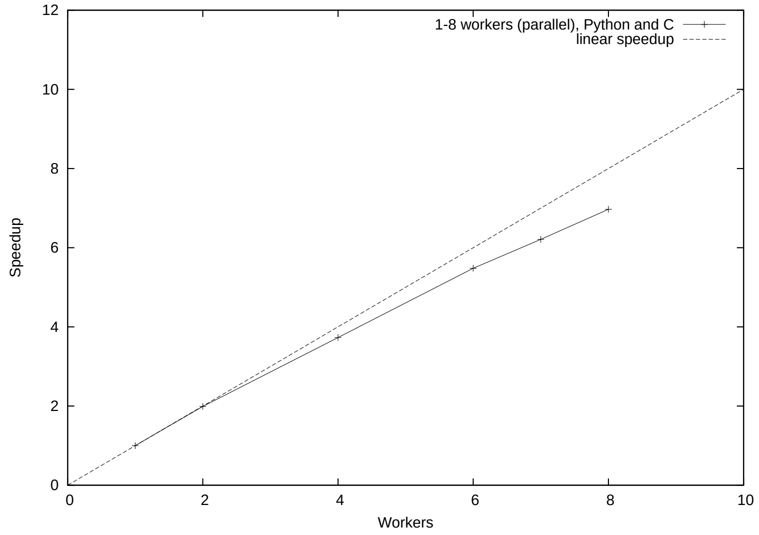


Figure 9. Speedup of prime factorisation of the Mersenne number $2^{222} - 1$.

5. Related Work

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

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

6. Conclusions and Future Work

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

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

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

[7] The Kepler Project: `http://www.kepler-project.org/`

[8] Knime: `http://www.knime.org/`

[9] LabVIEW: `http://www.ni.com/labview/`

[10] FlowDesigner: `http://flowdesigner.sourceforge.net/`

[11] Taverna: `http://taverna.sourceforge.net/`

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.4 PyCSP Revisited

Communicating Process Architectures 2009, WoTUG-32, Proceedings of the 32nd WoTUG Technical Meeting, Technische Universiteit Eindhoven, The Netherlands, November 1-4, 2009
ISBN: 978-1-60750-065-0, Concurrent Systems Engineering 67, IOS Press, pp. 263 – 276

Brian Vinter, John Markus Bjørndalen, Rune Møllegaard Friborg: PyCSP Revisited

Communicating Process Architectures 2009
P.H. Welch et al. (Eds.)
IOS Press, 2009

PyCSP Revisited

Brian VINTER [a,1], John Markus BJØRNDALEN [b] and Rune Møllegaard FRIBORG [a]
[a] *Department of Computer Science, University of Copenhagen*
[b] *Department of Computer Science, University of Tromsø*

Abstract. PyCSP was introduced two years ago and has since been used by a number of programmers, especially students. The original motivation behind PyCSP was a conviction that both Python and CSP are tools that are especially well suited for programmers and scientists in other fields than computer science. Working under this premise the original PyCSP was very similar to JCSP and the motivation was simply to provide CSP to the Python community in the JCSP tradition. After two years we have concluded that PyCSP is indeed a usable tool for the target users; however many of them have raised some of the same issues with PyCSP as with JCSP. The many channel types, lack of output guards and external choice wrapped in the select-then-execute mechanism were frequent complaints. In this work we revisit PyCSP and address the issues that have been raised. The result is a much simpler PyCSP with only one channel type, support for output guards, and external choice that is closer to that of occam than JCSP.

Keywords. Python, CSP, PyCSP, Alternation, Concurrency

Introduction

When PyCSP was introduced in 2007 [1] it was a CSP [2] library in the JCSP [3,4,5] tradition and primarily targeted pedagogical purposes. After having worked with PyCSP for another two years the authors decided to evaluate the experiences made and decide on the future of PyCSP. The outcome of the evaluation had three potential conclusions:

1. PyCSP was a nice exercise but of little or no practical use and the project should be stopped
2. PyCSP is a success as it is and no further work is needed, thus the research should be stopped
3. PyCSP has shown potential but needs more work and/or alternative approaches

Having included PyCSP in the Extreme Multiprogramming class at the University of Copenhagen three years in a row with a combined number of students in excess of 200, we did have a sizable set of inputs on PyCSP. On the upside the students claimed to like PyCSP for a number of reasons:

- It is Python and thus perceived to be easier to work with than most other languages[2]
- The fact that PyCSP channels are type indifferent[3] is convenient when changing the functionality in an application

On the downside, a number of students also had reservations with PyCSP:

[1] Corresponding Author: *Brian Vinter, Department of Computer Science, University of Copenhagen, DK-2100 Copenhagen, Denmark*. Tel.: +45 3532 1421; Fax: +45 3521 1401; E-mail: `vinter@diku.dk`.

[2] This may only be true in the context of this class where the focus is on scientific applications.

[3] The type indifference is easy because Python is dynamically typed.

- The many channel types make code less intuitive and any-to-any was the de-facto choice, though it does not support external choice
- No real parallelism unless functionality is written in C

Going through the final reports for the last exam, we discovered that more than 80% of the students had chosen PyCSP for their solution, second was JCSP, then followed C++CSP. A single report used occam. While some of the success of PyCSP is bound to be due to veneration for a locally developed system, there is little doubt that the students do like PyCSP: Java is the usual language of choice in other classes. It was especially interesting that students with a non-CS background, such as math, physics, nano-science and biology, all chose PyCSP, which indicates that our original intention of making a system for multi-core programming for scientists is within reach.

We decided that option 3, “PyCSP has shown potential but needs more work and/or alternative approaches”, was the conclusion of our evaluation and went on to address the input we have gotten from the many users.

The most frequent comment we received was disappointment that true parallelism could not be obtained using pure Python code. This is because Python uses a global interpreter lock, GIL, which means that threads in Python are useful only if a thread calls outside Python or to handle asynchronous events. To address this, the new implementation supports operating system processes in addition to threads. Strictly speaking this could be done with no changes to PyCSP and a new process-based implementation could transparently replace the old one. However, a number of other comments we received addressed the syntax and semantics of PyCSP and we thus decided to revisit the design. The work on the new implementations is presented in another paper [6].

It was also decided to make changes to the PyCSP API. Originally, PyCSP had been inspired by the other CSP libraries – most importantly JCSP – but it was evident that many students found the compact expressions in occam, especially the representation of external choice, attractive. While students easily understood why external choice on output channels is not needed in CSP, they still, rightly, claimed that they would be convenient. Finally, termination through poisoning was easily understood but also claimed to be inconvenient. Thus we decided to change PyCSP in four major ways:

1. There should be only one channel type, any-to-any, and it must support external choice
2. The channels should support both input and output guards for external choice
3. PyCSP should provide a mechanism for joining and leaving a channel with support for automatic poisoning of a network
4. The expressive power in Python should be used to make PyCSP look more like occam where possible

In the following we describe the new PyCSP library based on the above four design criteria. The result is a PyCSP implementation that follows the decisions and, in our own opinion, makes PyCSP programs even more readable and maintainable.

1. The New PyCSP

1.1. Processes

Just as in the original PyCSP, processes are wrapped in a process decorator, i.e. they are not merely implementations of a Process class as in JCSP or C++CSP. The advantage of this approach is partly that processes will be easily recognizable in the source code, and that it gives great flexibility for the PyCSP runtime environment to handle processes in different ways.

The constructor used is `@process`, and a hello world example could look like the following example:

```
@process
def hello_world(msg):
    print "Hello world, this is my message " + msg
```

Usually one or more channel ends will be part of the parameters for a process. Defining a process as above will not instantiate or execute any code: it is simply defined as a process to be used in a network at a later time.

1.2. Process Sets

Once a process is defined, a set of processes may be instantiated and executed using the Parallel or Sequential constructs similar to the old version. However, in order to accommodate variable size networks a process set may now include lists of processes as well as individual processes.

```
Parallel(
    source(),
    [worker() for i in range(10)],
    sink()
)
```

In the above example source, worker and sink have all been defined as processes and the parallel construct will run one source, ten workers and one sink process in parallel and return once all processes have terminated. Naturally the example makes little sense without the use of channels for communication; these will be introduced below. Apart from the support for mixing scalars and vectors of processes, the Parallel and Sequential constructs work as in the previous version and should be intuitive to anybody with any CSP experience.

1.3. Channels

PyCSP originally based much of its design on JCSP, continuing the use of specialized channel types: One2One, One2Any, Any2One and Any2Any. The type names designate how many writer and reader processes were allowed to be attached to the respective channel ends.

The main reason for the specialized channel types was that the implementation of the Alternative construct, which allowed external choice, was based on the JCSP version and placed strict limitations on the use of channels: only one process could safely use an Alternative construct with a given channel end. To safeguard against misuse, only the reading end of channel types that were restricted to one reader could be used as guards in an external choice. Limitations such as these can be cumbersome to work around when designing your CSP application and even more so for newcomers to PyCSP.

1.3.1. New Channel Type

There is only one channel type in the new PyCSP. The channel is similar to the previous Any2Any channel, but with the difference that both input and output channel ends support external choice. The use of external choice is described in section 1.4.

Retrieving channel ends for use in processes has also changed in PyCSP. Previously, a programmer would grab a channel end by calling the `read()` or `write()` method of the channel. This has been replaced with the `channel.reader()` and `channel.writer()` functions which also have a role in channel poisoning described below. As an experiment a shorthand for `channel.reader()` and `channel.writer()` is introduced as `-channel` and `+channel`; whether this more compact notation introduce more confusion than it is worth is left to future observations.

1.3.2. Channel Poison

The concept of poisoning channels with the purpose of shutting down an application was introduced in C++CSP [7] and later investigated in some detail by Bernhard Sputh [8]. A channel is poisoned and all subsequent reads or writes on this channel will throw an exception. This exception can be caught and used as a shut-down procedure or just to shut down that single channel. In the following example we create two processes, `source` and `sink`, and a channel to connect them. The `source` process finally poisons the channel to terminate the network, which will happen since the `sink` process does not catch the exception.

```
@process
def source(chan_out):
    for i in range(10):
        chan_out("Hello world")
    poison(chan_out)

@process
def sink(chan_in):
    while True:
        print chan_in()

chan = Channel()
Parallel(source(chan.writer()), sink(chan.reader()))
```

Since all channels now support multiple readers and writers it is easy to add more readers and writers:

```
Parallel(source(chan.writer()), sink(chan.reader()),
         source(chan.writer()), sink(chan.reader()),
         source(chan.writer()), sink(chan.reader()),
         source(chan.writer()), sink(chan.reader()),
         source(chan.writer()), sink(chan.reader()))
```

or

```
Parallel([source(chan.writer()) for i in range(5)],
         [sink(chan.reader())   for i in range(5)])
```

Both versions produce five source and five sink processes, however the created network will not do what the user may intuitively think it does. One of the sources is bound to finish first and it will then poison the channel, which will terminate the network before all the expected messages have been printed. The problem is extremely common in producer-consumer class applications, and users end up with complex solutions for terminating the network.

To address this we introduce a poison mechanism similar to reference counting. Creating channel ends and retiring from them updates a counter of how many readers or writers we have on a channel, and the leave method may perform automatic poisoning when no readers or no writers are left.

The `reader()` and `writer()` methods automatically join the respective ends of a channel, returning a unique reference to that channel end. A new function, `retire()`, is used to leave a channel end. All subsequent requests to this channel end reference will raise an exception. When all readers or writers have retired a channel, the other end of the channel is also retired. This is similar to how poison is propagated in the previous versions of PyCSP, but with one important difference: with a poisoned channel any reference to that channel will trigger a ChannelPoisonException which is caught in the Process class that wraps all PyCSP processes. The exception handler then poisons all the other channels that were passed to the process upon initialization. With a retired channel, the ChannelRetireException is thrown and the other channel ends are retired rather than poisoned, implementation wise the two are identical apart from the name of the exception that is raised. This can remove some potential race conditions when terminating networks, as seen in the Monte Carlo Pi example in section 3 and in the example below.

The following code demonstrates how the retire expression can be used instead of the poison expression. The network will now be poisoned by the last source process to finish, rather than the first. This feature hugely simplifies many networks.

```
@process
def source(chan_out):
    for i in range(10):
        chan_out("Hello world")
    retire(chan_out)
```

1.4. External Choice

One of the criticisms that the original PyCSP attracted was the way that external choice was implemented, which had more in common with UNIX socket programming using select than the more compact occam ALT operation. After executing an external choice (Alternative) you are required to read from the selected channel. Failing to do so would break the rules for the choice construct in CSP. Thus we decided to simplify the usage of Alternative by combining select with a custom-defined action on the guard, similar to the occam ALT. Based on this, we introduce a new choice named Alternation.

Alternation has changed significantly from Alternative, partly to make it more like occam, partly to support output guards. A guard set is now represented as a list of Python dictionaries where the keys can be channels from which to read, or two-tuples where the first entry is a channel and the second the value that should be written to that channel. The value of each dictionary entry is a function of type choice which may be executed if the guard becomes true. If the guard is an input guard then the choice function will always have the parameter `__channel_input` available which is the value that was read from the channel. Alternation also supports other guard types, inheriting from a common Guard class. Alternation has two calls:

- Execute – which waits for a guard to complete and then executes the associated choice function, similar to the occam ALT instruction
- Select – which returns a two-tuple: the guard that was chosen by Alternation and, if the guard was an input-guard, the message that was read. This is equivalent to the original Alternative

Note that the execute call in alternation always performs the guard that was chosen, i.e. channel input or output is executed within the alternation, so even the empty choice or a choice function where the results are simply ignored still performs the guarded input or output.

The code that is executed within a guard may be specified in two ways, either as a function that is defined using a choice decorator, similar to processes, or as a string containing code to be executed. The latter is easy to use but becomes quite slow since runtime compilation is required. A choice function is defined as follows:

```
@choice
def action(__channel_input=None):
    print __channel_input
```

It is not possible to change the name of `__channel_input` in a choice function since it is passed as a keyword argument when it is the result of a selected input guard. Once the choice is defined, a process may perform an alternation on a set of channel ends. In the following example the same guarded code, action, is called independently of which channel becomes ready. This is an option but naturally not a requirement.

```
@process
def par_reader(cin1, cin2, cin3, cin4):
    Alternation([
        { cin1:action() },
        { cin2:action() },
        { cin3:action() },
        { cin4:action() }
    ]).execute()
```

An action might alternatively be passed as a string. This string is then evaluated with a copy of the current namespace. All mutable types can be updated from the evaluation of this string. In Python, the `list`, `dict` and `set` types are built-in mutable types.

```
@process
def counter(cin0, cin1):
    try:
        cnt = [0, 0] # use mutable type
        while True:
            Alternation([
                { cin0: 'cnt[0] += 1' },
                { cin1: 'cnt[1] += 1' }
            ]).execute()
    except ChannelPoisonException:
        print 'Counted:', cnt
```

Guards are prioritized in the order they occur in the guard list, while guards in a dictionary are unordered. This gives us the option to model both an ordinary external choice and a prioritized external choice. It is important to note that priority only makes real sense in the scenario where more than one guard is ready when the alternation is entered, which guard is woken first if no guards are immediately available may be down to race-condition or the priority order of another guard statement.

Ordinary external choice is obtained by a list with just one dictionary holding guards. The entries in the dictionary are then treated in an non-prioritized way:

```
[{
    cin1:action(),
    cin2:action(),
    cin3:action()
}]
```

On the other hand, prioritized external choice is obtained by providing a list of dictionaries with guards. These are then prioritized in the way the dictionaries appear in the list.

```
[
    { cin1:action() },
    { cin2:action() },
    { cin3:action() }
]
```

It is fully possible to mix the two models, i.e. a prioritized list with dictionaries of non-prioritized guards. This option should only be used for special purposes.

PyCSP provides four built-in guard types to use with external choice. The first three of them are well known to the CSP community:

- Channel input
- Timeout - A counter relative to current time, when it expires, it will become true and allow the alternation to complete
- Skip - Always true, and often used to define a default alternative
- Channel output

The fourth is new in PyCSP, although thoroughly discussed throughout the years and previously seen in Communicating Java Threads [9]. It is well understood by most programmers that use process algebra that output guards are not needed from a CSP point of view, and one may with relative ease construct equivalences for any type of output guard using only input guards. However, output guards are convenient for the user of PyCSP and equivalences are hard to construct for users that are not professional programmers, thus we provide the output guard as a primitive in PyCSP. All guard types supported can be interrupted by channel poisoning or retiring. PyCSP channels may be guarded in both ends, i.e. an output guard can be matched by an input guard.

The following code example shows how non-blocking writes and input with timeouts can be modelled using the new Alternation construct:

```
# Non-blocking write
Alternation([
    { (cout, datablock): None },  #Try to write to a channel
    { Skip(): "print 'skipped!'" }  #Skip the alternation
]).execute()

# Input with timeout
Alternation([
    { cin: "print __channel_input"},
    { Timeout(seconds=1): "print 'timeout!'" }
]).execute()
```

2. Implementation

This section introduces only highlights of the implementation. An in depth description of the implementation details of PyCSP may be found in [6].

The only implementation detail that is non-trivial is the support for output guards and channels with multiple processes at either end. The implementation is quite complex and uses more than a hundred lines of Python code. The overall design is based on each alternation being represented by a request structure, called a handle in the pseudocode below, that includes a lock which ensures mutual exclusion. When a new alternation is activated it will traverse the guards in the choice list by priority, and for each guard it will look for a waiting handle that matches the handle for the alternation, i.e. a read matches a write and vice versa. If no match is found, the handle is added to the set of waiting handles for that channel. Please note that the pseudocode is heavily simplified and the actual implementation relies on global ordering of events to avoid livelocks; for details refer to [6].

```
handle = new_request_handle()
for guard in choice:
    lock(guard.channel)
    if handle match registered_handle in guard.channel:
        perform communication
```

```
            make_active(handle, registered_handle)
        else:
            guard_channel.registered_handle.add(handle)
        unlock(guard.channel)
    waitfor active(handle)
```

This is the procedure for every channel communication possible. Whenever a match is tried, two locks are required: one owned by the reading end and one owned by the writing end. In the case of alternation, this lock is shared between all guards to ensure the integrity of the alternation. A diamond design where every process alternates on an input and an output end could look similar to the example code below.

```
@process
def P(id, c1, c2):
    while True:
        Alternation([{(c1, True): None, c2: None}]).select()

c = [Channel(str(i)) for i in range(4)]

Parallel(
    P(1,c[0].writer(), c[1].reader()),
    P(2,c[1].writer(), c[2].reader()),
    P(3,c[2].writer(), c[3].reader()),
    P(4,c[3].writer(), c[0].reader())
)
```

Without acquiring the locks in perfect order, this code eventually results in a deadlock. Two processes have both acquired one of the alternation owned locks and are waiting to acquire the other in opposite order. To acquire the locks in order, we always acquire the lock with the lowest memory address first. This ensures the same lock order for all processes.

We are synchronizing input and output guards without the *Oracle* process used in JCSP [10]. The *Oracle* process was introduced in JCSP to handle external choice on barriers and output guards. The new PyCSP views all communication requests as an offer and all offers are protected by individual locks. One lock per offer eliminates the need for an *Oracle* process, because it is guaranteed that an offer is only matched while it is active. The purpose of the *Oracle* process is to ensure that an offer is still active, when it is matched.

3. Examples

Some of the changes in PyCSP refer to either performance and implementation, or pure syntactical presentation of the concepts. In the following, we show two examples that motivate the three semantic changes that have been introduced: retire as an alternative to channel poisoning, output-guards, and support for alternation with channels that have multiple readers and/or writers. The purpose of the examples is to demonstrate why PyCSP becomes easier to use after the introduced changes.

3.1. Monte Carlo Pi

In the original PyCSP we did not have the retire feature, which meant that most producer-consumer programs tended to look like the example in figure 1.

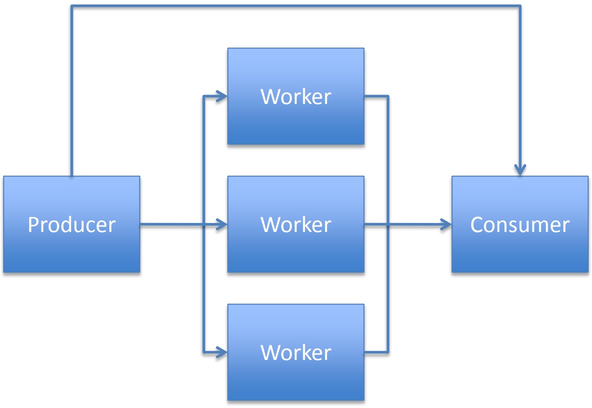


Figure 1. Producer-consumer program forwarding termination criteria between producer and consumer.

```
from pycsp import *
from random import random

@process
def producer(term_out, job_out, bagsize, bags):
    term_out(bags)
    for i in range(bags): job_out(bagsize)
    poison(job_out)

@process
def worker(job_in, result_out):
    try:
        while True:
            cnt = job_in()             #Get task
            sum = reduce(lambda x, y: x+(random()**2+random()**2<1.0),
                         range(cnt))
            result_out((4.0*sum)/cnt)  #Forward result
    except ChannelPoisonException:
        pass                           #When done, _don’t_ forward poison
@process
def consumer(term_in, result_in):
    cnt = term_in()          #Get number of results
    sum = 0
    for i in range(cnt):
        sum += result_in()   #Get result
    print sum/cnt            #We are done - print result

jobs = Channel()
results = Channel()
term = Channel()

Parallel(producer(term.writer(), jobs.writer(), 1000, 10000),
         [worker(jobs.reader(), results.writer()) for i in range(10)],
         consumer(term.reader(), results.reader()))
```

Listing 1: Implementation of producer-consumer program forwarding with an explicit termination channel between producer and consumer.

A simple Monte Carlo simulation of the design in figure 1 is implemented in listing 1. This approach is simple to implement but suffers from two complexities: first of all, the termination criterion (number of bags) must be sent from the producer to the consumer, bypassing the workers. Secondly, the workers must explicitly avoid forwarding the channel poison in the network as the consumer will otherwise die before all results are received and processed. The complexity of the solution grows even further if the setup has multiple producers or consumers.

With the new retire operation, termination of the network can be handled in a more straightforward way. When the producer retires the `job_out` channel end, the channel is not poisoned, but the retire operation is forwarded to the other end of the channel in a similar way to poisoning. The main difference is that when the channel is retired and `job_in()` terminates the worker process, the workers channels are retired rather than poisoned. This will delay termination propagation along those channels until all workers have retired, and the consumer will not be prematurely terminated.

Figure 2 shows the new network, which no longer needs to forward a termination criterion between the producer and consumer process. Multiple producers can also be plugged into the network without changing more than the parallel construct. Note that the consumer in listing 2 catches a ChannelRetireException, which allows it to terminate cleanly and print out the results before terminating.

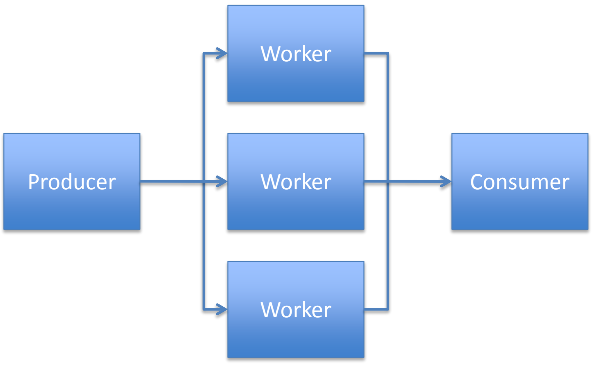


Figure 2. Producer-consumer program using retire, avoiding termination criteria forwarding between producer and consumer.

```
from pycsp import *
from random import random

@process
def producer(job_out, bagsize, bags):
    for i in range(bags): job_out(bagsize)
    retire(job_out)

@process
def worker(job_in, result_out):
    while True:
        cnt = job_in()          #Get task
        sum = reduce(lambda x, y: x+(random()**2+random()**2<1.0),
                     range(cnt))
        result_out((cnt, sum))  #Forward result

@process
def consumer(result_in):
    cnt, sum = 0, 0
    try:
        while True:
            c, s = result_in() #Get result
            cnt, sum = cnt+c, sum+s
    except ChannelRetireException:
        print 4.0*sum/cnt      #We are done - print result
```

```
jobs = Channel()
results = Channel()

Parallel(producer(jobs.writer(), 1000, 10000),
         [worker(jobs.reader(), results.writer()) for i in range(10)],
         consumer(results.reader()))
```

Listing 2: Producer-consumer using retire, avoiding termination criteria forwarding between producer and consumer

3.2. Branch-and-bound

Branch-and-bound algorithms in CSP are easy [11] but associated with some complex decisions with regards to how and when to update the bound variable. The challenge in the bound update is to balance the communication and work: if the bound variable is updated too rarely the parallel work will perform more work than necessary. If the bound is updated too often, it will result in too frequent communication. Basically, three approaches exist:

1. Update the bound only when a job is finished. Submitting a bound equals requesting a job
2. Update the bound as soon as you find it. A special bound value identifies a job request
3. Update the bound as soon as you find it. Jobs are requested independently

The first approach is simple and a common choice; however, the infrequent update of the bound results in slower overall execution. The second approach is complex and requires parsing of the input to determine if an incoming message requires an outgoing job. The third is easy but requires output guards.

To keep this section from growing too large we only present the code required for receiving bound variables and passing on jobs.

Solution 1 is the trivial case, where the master sends a new job back when receiving a result from a worker. We don't need an alternation in this case since all channels are any-to-any. When there are no more jobs to be executed, the master retires the job channel which will terminate workers trying to read from it. The master continues to receive results until all workers have retired which will retire the result channel. This will in turn throw a ChannelRetireException in the master, which can be caught to print out the final result.

```
bound = 10e10
while jobs:
    next = jobs.pop()
    bid = results_in()
    bound = best(bid, bound) #Best is an optimization specific function
    jobs_out((next, bound))

#Without retire the code becomes even more complex
retire(jobs_out)

try:
    while True:
        bound = best(results_in(), bound)
except ChannelRetireException:
    print bound
```

Listing 3: Solution 1

Solution 2 allows workers to submit new bound variables before the job is finished by sending an *update* message over the request channel. This allows the bound variable to be updated faster and thus potentially reduces the total work that must be done. The solution is almost identical to solution 1, except that messages from workers are parsed to know if the message is an update. Updates should not trigger a blocking write of a new job to the jobs channel. Termination is identical to solution 1.

```
bound = 10e10
while jobs:
    next = jobs.pop()
    request, bid = results_in()
    if request == ’Update’:      #Update means don’t send new job
        bound = best(bid, bound) #Best is an optimization specific function
    else:
        jobs_out((next, bound))

#Without retire the code becomes even more complex
retire(jobs_out)

try:
    while True:
        bound = best(results_in()[1], bound)
except ChannelRetireException:
    print bound
```

Listing 4: Solution 2

Solution 3 uses output guards to eliminate parsing of the incoming messages. Instead, the alternation accepts either an incoming result or an outgoing job to a worker. Once there are no more jobs, the solution terminates like the other solutions.

It should be noted that this design, which is as simple as solution 1 and as efficient as solution 2, also provides simpler initialization of the workers since they are not required to submit a bogus result to trigger the delivery of the first job.

```
#A Python limitation requires a mutable type here
my_locals = {
   ’bound’: 10e10,
   ’next’ : jobs.pop() #We require at least two jobs to start with!!!
}
while jobs:
    Alternation([{
        results_in:
          "my_locals[’bound’] = best(__channel_input, my_locals[’bound’])",
        (jobs_out, (next, my_locals[’bound’])) :
          "my_locals[’next’] = jobs.pop()"
    }]).execute()

#Without retire the code becomes even more complex
retire(jobs_out)

try:
    while True:
        my_locals[’bound’] = best(results_in(), my_locals[’bound’])
except ChannelRetireException:
    print bound
```

Listing 5: Solution 3

If one wishes the workers to update their knowledge of the bounds, this may easily be done in solution 3 by adding a dedicated channel for propagating the bound and letting the workers do a blocking input from that channel as frequently as desired. The server then adds another output guard that at any time is ready to write the best known bound.

4. Future Work

The new version of PyCSP provides a very convenient means of writing concurrent applications for non-computer scientists, allowing them to use CSP for parallel and concurrent programming. Initial response on the new version has been quite positive and we thus plan to continue the work.

An extension to alternation in JCSP is the concept of fairSelect, which can be used to avoid starvation. This would be of interest also to PyCSP users, probably as default with the truly random alternation becoming an option for special purposes.

Network construction is still fairly complex in PyCSP, and the only improvement that the new version offers is the option of mixing single processes and lists of processes in one Parallel constructor. We are working on a library of network constructors that will allow users to easily specify networks of processes in rings, meshes, fully interconnected and other common process-oriented design patterns.

The previous version of PyCSP was extended with a module that allowed processes to be executed on Grid when channel communication can be represented as a synchronous event, i.e. input;execute;output, the Grid enabled processes cannot support all channel communications, i.e. alternation or patterns as input;execute;input;execute;output cannot be used but classic, client-server patterns fits will with Grid execution. This feature is desired for very demanding jobs and would be relevant to reintroduce in the new version of PyCSP.

While the type indifference of channels in PyCSP is highly praised by students there are scenarios where type matching is equally attractive. Future plans include adding support for type-checking channels.

Conclusions

The original PyCSP borrowed heavily from JCSP to get semantics and functionality correct while still attempting to make the solution native to Python. It was quite well received, especially amongst students and scientists who often find Python a productive programming environment. After exposing more than 200 students to PyCSP, we did however receive some negative feedback. One of the central complaints was about the many channel types and especially the hardship of changing between them in an existing application. Another frequent complaint was the lack of support for output guards and channels with multiple readers and/or writers in alternation. In addition to the feedback from the users, the authors identified two shortcomings in the original version of PyCSP: first, students frequently demonstrated race-conditions when terminating a network by use of poisoning, and second, it is desired to make PyCSP look more like occam.

The complaints and identified shortcomings resulted in an evaluation that confirmed the need for the following major changes to PyCSP. All channels are now any-to-any which greatly simplifies design changes since a user may add more readers or writers to a channel that previously had only one. Since external choice is central to CSP, these any-to-any channels are naturally supported in the alternation implementation of external choice.

PyCSP external choice now supports output guards in addition to input guards. This works with multiple readers and writers on a channel. The use of output guards is a heavily debated issue in CSP as they are clearly not needed nor trivial to implement. However, it is evident that the users of PyCSP find output guards a very convenient feature and considerable work has been put into supporting output guards in the alternation implementation in PyCSP.

External choice has also been modified to more closely mimic occam so that a guard and the associated code can be expressed in one statement. This brings PyCSP much closer to conventional CSP than the previous model where a ready guard was first identified and then read from.

In order to reduce the risk of race-conditions when using poison to terminate a CSP network, this version of PyCSP introduces the concept of retirement from a channel. When all processes on one end of a channel retire their channel ends, the channel becomes retired. The effect is that the propagation of the retire signal is activated upon the termination of the last process at a given channel end rather than the first as with the poison operation.

Overall, the changes to PyCSP are well integrated and we believe that using PyCSP is now easier for the unsophisticated users than with the previous version. The newest version may be found as PyCSP under Google-code [12].

References

[1] John Markus Bjørndalen, Brian Vinter, and Otto Anshus. PyCSP - Communicating Sequential Processes for Python. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

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

[3] Communicating Sequential Processes for Java. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[4] Jim Moores. Native JCSP: the CSP-for-Java library with a Low-Overhead CPS Kernel. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering*, pages 263–273. WoTUG, IOS Press (Amsterdam), September 2000.

[5] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[6] Rune Møllegaard Friborg, John Markus Bjørndalen, and Brian Vinter. Three Unique Implementations of Processes for PyCSP. In Peter H. Welch, editor, *Communicating Process Architectures 2009*, Amsterdam, The Netherlands. WoTUG, IOS Press.

[7] Neil C.C. Brown and Peter H. Welch. An introduction to the Kent C++CSP Library. *Communicating Process Architectures 2003*, September 2003.

[8] Bernhard H.C. Sputh and Alastair R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. *Communicating Process Architectures 2005*, September 2005.

[9] Gerald H. Hilderink, Jan F. Broenink, Wiek Vervoort, and Andrè W. P. Bakkers. Communicating Java Threads. In Andrè W. P. Bakkers, editor, *Proceedings of WoTUG-20: Parallel Programming and Java*, pages 48–76, mar 1997.

[10] Peter H. Welch, Neil C.C. Brown, Jim Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In Steve Schneider, Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 349–370, Amsterdam, The Netherlands, July 2007. WoTUG, IOS.

[11] Peter H. Welch and Brian Vinter. Cluster Computing and JCSP Networking. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 203–222, sep 2002.

[12] PyCSP distribution. `http://code.google.com/p/pycsp`.

A.5 Three Unique Implementations of Processes for PyCSP

Communicating Process Architectures 2009, WoTUG-32, Proceedings of the 32nd WoTUG Technical Meeting, Technische Universiteit Eindhoven, The Netherlands, November 1-4, 2009
ISBN: 978-1-60750-065-0, Concurrent Systems Engineering 67, IOS Press, pp. 277 – 292

Rune Møllegaard Friborg, John Markus Bjørndalen, Brian Vinter: Three Unique Implementations of Processes for PyCSP

Communicating Process Architectures 2009
P.H. Welch et al. (Eds.)
IOS Press, 2009

Three Unique Implementations of Processes for PyCSP

Rune Møllegaard FRIBORG [a,1], John Markus BJØRNDALEN [b] and Brian VINTER [a]
[a] *Department of Computer Science, University of Copenhagen*
[b] *Department of Computer Science, University of Tromsø*

Abstract. In this work we motivate and describe three unique implementations of processes for PyCSP: process, thread and greenlet based. The overall purpose is to demonstrate the feasibility of Communicating Sequential Processes as a framework for different application types and target platforms. The result is a set of three implementations of PyCSP with identical interfaces to the point where a PyCSP developer need only change which implementation is imported to switch to any of the other implementations. The three implementations have different strengths; processes favors parallel processing, threading portability and greenlets favor many processes with frequent communication. The paper includes examples of applications in all three categories.

Keywords. Python, CSP, PyCSP, Concurrency, Threads, Processes, Co-routines

Introduction

The original PyCSP [1] implemented processes as threads, motivated by an application domain with scientific users and the assumption that these applications would spend most of their time in external C calls. While the original PyCSP was well received, users often aired two common complaints. First and foremost programmers were disappointed that pure Python applications would not show actual parallelism on shared memory machines, most frequently multi-core machines, because of Python's Global Interpreter Lock. The second common disappointment was the limited number of threads supported, typically an operating system limitation in the number of threads per process, and the overhead of switching between the threads.

In this paper we present a new version of PyCSP that addresses these issues using three different implementations of its concurrency primitives.

PyCSP

The PyCSP library presented in this paper is based on the version of PyCSP presented in [2] which we believe reduces the complexity for the programmer significantly. It is a new implementation of CSP constructs in Python, that replaces the original PyCSP implementation from [1]. This new PyCSP uses threads like the original PyCSP, but introduces four major changes and uses a better and simpler approach to handle the internal synchronization of channel communications. The four major changes are: simplification to one channel type, input and output guards, automatic poisoning of CSP networks and making the produced Python code look more like occam where possible.

[1] Corresponding Author: *Rune Møllegaard Friborg, Department of Computer Science, University of Copenhagen, DK-2100 Copenhagen, Denmark.* Tel.: +45 3532 1421; Fax: +45 3521 1401; E-mail: `runef@diku.dk`.

When we refer to the threads implementation of PyCSP, we are referring to the new PyCSP presented in [2] and referenced in this paper as `pycsp.threads`. This is used as our base to implement the alternatives to threading presented in this paper.

1. Motivation

We have looked at three underlying mechanisms for managing tasks and concurrency: co-routines, threads and processes. Each provide different levels of parallelism that come with increasing overhead. All of them are available in different forms, and in this paper we define them as follows:

> **Co-routines** provide concurrency similar to user-level threads and are scheduled and executed by a user-level runtime system. One of the main advantages is very low overhead.
> **Threads** are kernel-level threads scheduled by the operating system, has a separate execution stack, but share a global address space.
> **Processes** are operating system processes and data can only be shared through explicit system calls.

When programming a concurrent application, it is necessary to choose one or several of the above. If the choice turns out to be wrong, then the application needs to be rewritten. A rewrite is not a simple task, since the mechanisms are very different by design.

Using Python and PyCSP, we want to simplify moving between the three implementations. The intended users are scientists that are able to program in Python and who want to create concurrent applications that can utilize several cores. Python is a popular programming language among scientists because of a simple and readable syntax and the many scientific modules available. It is also easy to extend with code written in C or Fortran and does not require explicit compilation.

1.1. Release of GIL to Utilize Multi-Core Systems

Normally PyCSP is limited to execution on a single core. This is a limitation within the CPython[1] interpreter and is caused by the Global Interpreter Lock (GIL) that ensures exclusive access to Python objects. It is very difficult to achieve any speedup in Python from running multiple threads unless the actual computation is performed in external modules that release the GIL. Instead of releasing and acquiring the GIL in external modules it is possible to use multiple processes that run separate CPython interpreters with separate GILs. In Python 2.6 we can use the new multiprocessing module [3] to handle processes, enabling us to compare threads to processes. The comparison in Table 1 shows the result of computing Monte Carlo pi in parallel using threads and processes.

Table 1. Comparison of threads and multiprocessing on a dual core system with Python 2.6.2

Workers	1	2	3	4	10
Threads	0.98s	1.52s	1.56s	1.55s	1.57s
Processes	1.01s	0.57s	0.54s	0.54s	0.56s

The GIL is to blame for the poor performance for threads illustrated in Table 1. It is possible to obtain good performance for threads, but to do this you must compute in an external module and manually release the GIL. The unladen-swallow project [4] aims to remove the Global Interpreter Lock entirely from CPython.

[1] CPython is the official Python interpreter.

1.2. Maximum Threads Available

On a standard configured operating system, the maximum number of threads in a single application is limited to around 1000. In PyCSP, every CSP process is implemented as a thread. Thus, there can be no more CSP processes than the maximum number of threads. We want to overcome this and give PyCSP the ability to handle CSP networks consisting of more than 100000 CSP processes, by using co-routines.

We thus decided to address these issues by providing two additional implementations, one that provides real parallelism for multi-core machines and one that does not expose the processes to the operating system. All versions should implement the exact same interface, and a programmer should need only to change the code that imports PyCSP to change between the three different versions. Having a common interface for three implementations of PyCSP has another purpose besides being a fast and effective method for changing the concurrent execution platform. It is also an easy method for students to learn what consequences it has to run a specific PyCSP application with co-routines, threads or processes. PyCSP is often chosen by students in the Extreme Multiprogramming Class, which is a popular course at the University of Copenhagen teaching Communicating Sequential Processes [5].

2. Three Implementations of PyCSP

The three implementations of concurrency in PyCSP – `pycsp.threads`, `pycsp.processes` and `pycsp.greenlets` – are packaged together in the `pycsp` module. Although packaged together these are completely separate implementations sharing a common API. It is possible to combine the implementations to produce a heterogeneous application with threads, processes and greenlets, but the support is limited since the choice (Alternation) construct does not work with channels from separate implementations and when communicating between implementations only channels from the processes implementation are supported. The primary purpose of packaging the three implementations in one module is to motivate the developer to switch between them as needed. A common API is used for all implementations making it trivial to switch between them, as shown in Listing 1. A summary of advantages and limitations for each of the implementations are listed at the end of this section.

Listing 1: Switching between implementations of the PyCSP API

```
# Use threads                     # Use processes
from pycsp.threads import *       from pycsp.processes import *
```

When switching to another implementation, the PyCSP application may execute very differently as processes may be scheduled in another order and less fair. Hidden latencies may also become more apparent when all other processes are waiting to be scheduled. In the following sections we present an overview of the implementations in order to understand how they affect the execution of a PyCSP application.

2.1. pycsp.threads

This implementation uses the standard threading module in Python, which implements kernel-level threads. All threads access the same memory space, thus when communicating data only the reference to the data is copied. If the data is a mutable Python type it can be updated from multiple threads in parallel, though it is not recommended to do so since it might cause unexpected data corruption and does not fit with the CSP programming model.

Details of `pycsp.threads` are presented in [2] and is a remake of the original PyCSP [1].

2.2. pycsp.greenlets

Greenlets are lightweight (user-level) threads, and all execute in the same thread. A simple scheduler has been created to handle new greenlets, dying greenlets and greenlets that are rescheduled after blocking on communication. The scheduler has a simple FIFO policy and will always try to choose the first greenlet among the greenlets ready to run.

The PyCSP API has been extended with an `@io` decorator that can wrap blocking IO operations and run the operations in a separate thread. In `pycsp.threads` and `pycsp.processes`, this decorator has no function while in `pycsp.greenlets` an `Io` object is created. It is necessary to introduce this construct because the greenlets are all running in one thread, and if one greenlet blocks without yielding control to the scheduler, all greenlets in this thread are blocked. For threads and processes, this is not a problem because the operating system can yield on IO and use time slices to interrupt execution, thus rescheduling new threads or processes. Greenlets are never forced to yield to another greenlet. Instead, they must yield execution control by themselves.

Invoking the `__call__` method on the `Io` object will create a separate thread running the wrapped function. After the separate thread has been started, the greenlet yields control to the scheduler in order to schedule a new greenlet. Listing 2 provides an example of how to use `@io`. Without `@io`, the greenlet would not yield, thus blocking all other greenlets ready to be scheduled. This would serialize the processes, and the total runtime of Listing 2 would be around 50 seconds instead of the expected 10 seconds.

Listing 2: Yielding on blocking IO operations

```
@io
def wait(seconds):
    time.sleep(seconds)

@process
def delay_output(msg, seconds):
    wait(seconds)
    print msg

Parallel(
    [delay_output('%d second delay' % i, i) for i in range(1, 11)]
    )
```

Communicating on channels from outside a PyCSP greenlet process is not supported, since the scheduler needs to work on a greenlet process to coordinate channel communication. This means that you can not communicate with the main greenlet at the top-level environment. Calls to `pycsp.greenlets` functions from a `@io` thread will fail for the same reason. Calls to the `pycsp.threads` or `pycsp.processes` implementations are recommended to be wrapped with the `@io` decorator, otherwise they could block the scheduler and cause a deadlock.

2.3. pycsp.processes

This implementation uses the multiprocessing module available in Python 2.6+. Processes started with the multiprocessing module are executed in separate instances of the Python interpreter. On systems supporting the UNIX system call `fork`, starting separate Python interpreters with a copy of all objects is trivial. On Microsoft Windows, this is much more challenging for the multiprocessing module, since no equivalent of `fork` is available. The multiprocessing module simulates the `fork` system call by starting a new Python interpreter, loading all necessary modules, serializing / unserializing objects and initiating the requested function. This is very slow compared to `fork`, but it still works in lack of a better alternative for Windows.

When an application is written in pure Python and PyCSP, it is now possible with `pycsp.processes` to utilize multi-core CPUs. For most cases all PyCSP applications will be able to run without any changes, but if the data communicated does not support serialization, the application will fail. An example of such data is an object containing pointers initialized by external modules, fortunately this type of data is not very common in Python applications.

`pycsp.processes` uses shared memory pointers internally and must allocate everything before any processes are forked. For this reason, it might in extreme cases be necessary to tweak a set of constants for `pycsp.processes`. To do this, a singleton Configuration class is instantiated as shown in the example (Listing 3). New constants must be set before any other use of `pycsp.processes`, since everything is allocated on first use.

Listing 3: Example of setting and getting a constant.

```
from pycsp.processes import *
Configuration().set(PROCESSES_SHARED_CONDITIONS, 50)
Configuration().get(PROCESSES_SHARED_CONDITIONS) # returns 50
```

Using this configuration class it is possible to change the size of shared memory and the amount of shared locks and conditions allocated on initialization. The allocated shared memory is used as buffers for channel communication, which means that the size of data communicated on channels at any given time can never exceed the size of the buffer. The default size of the shared memory buffer is set to 100MB, but can easily be increased by setting the constant `PROCESSES_ALLOC_MSG_BUFFER`.

2.4. Summary of Advantages and Limitations

The following is a summary of the advantages (+) and limitations (-) of the individual implementations before moving on to the *Implementation* and *Experiments* section:

Threads:

+ Only references to data are passed by channel communication.
+ Other Python modules usually only expect threads.
+ Compatible with all platforms supporting Python 2.6+.
- Limited by the Global Interpreter Lock (GIL), resulting in very poor performance for code not releasing the GIL.
- Limited in the maximum number of CSP processes.

Greenlets:

+ More optimal switching between CSP processes, since we can limit the context switches to the point where they are blocking. Performance does not decrease with more CSP processes competing for execution.
+ Very small footprint per CSP process, making it possible to run a large number of processes, only limited by the amount of memory available.
+ Fast channel communications ($\approx 20\mu s$).
- No utilization of more than one CPU core.
- Unfair execution, since execution control is only yielded when a CSP process blocks on a channel.
- Requires that the developer wraps blocking IO operations in an `@io` decorator to yield execution to another CSP process.

Processes:

- + Can utilize more cores, without requiring the developer to release the GIL.
- - Fewer processes possible than `pycsp.threads` and `pycsp.greenlets`.
- - Windows support is limited, because of lack of the `fork` system call.
- - All data communicated are serialized, which requires the data type be supported by the pickle module.
- + A positive side-effect of serializing data is that data is copied when communicated, rendering it impossible to edit the received data from the sending process.

3. Implementation

When processes communicate through external choice at both the reading and writing end a number of challenges must be addressed to avoid live-lock and dead-lock problems, this is well researched in [6,7,8]. The PyCSP solution introduces what we believe to be a new algorithm for this problem. The algorithm is very simple and quite fast in the common case.

Every channel has two queues associated with it, one for pending read-operations and one for pending write-operations. Every active choice (Alternation) is represented with a request structure, this request has a lock, to ensure mutual exclusion on changes to the request, an unique id, a status field, and the actual operation, i.e. read or write with associated data. When an Alternation is run a reference to the request structure is added to the queue it belongs to, i.e. input-requests (IR) and output-requests (OR), on every channel in the choice. Then all requests are tested against all potentially matching requests on all involved channels. When a match is found the state of the request structure is changed to `Done` to ensure that the request is matched only once. When the arbitration function comes across an inactive request structure it is evicted from the queue.

Listing 4: The double_lock operation in pseudo code

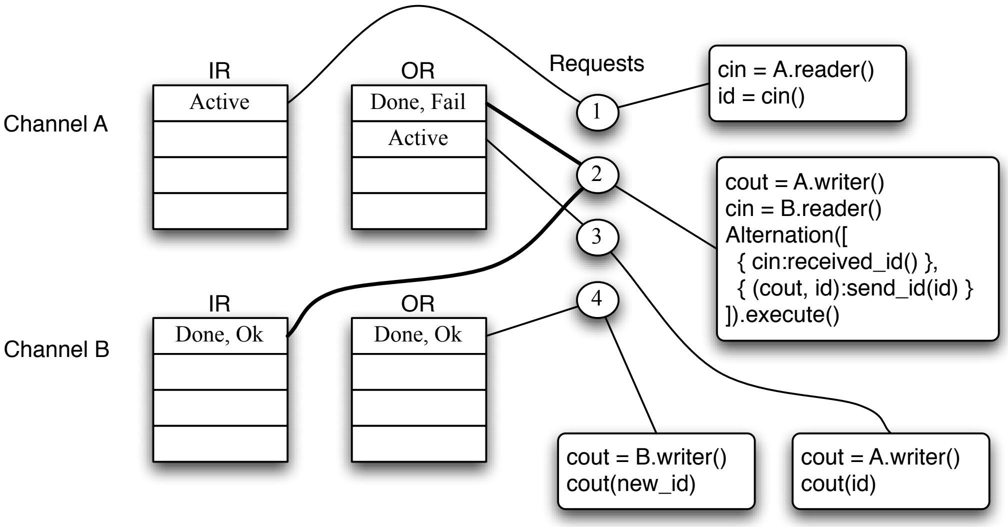
```
def double_lock(req_1, req_2):
    if req_1.id < req_2.id:
        lock(req_1.lock)
        lock(req_2.lock)
    else:
        lock(req_2.lock)
        lock(req_1.lock)
```

Live-lock is avoided by using blocking locks only, so if a legal match exists it will always be found the first time it is available. Deadlock is avoided by using the unique id of a request to sort the order in which locks are acquired, thus we have an operation, double_lock (Listing 4), that acquires two individual locks in order and returns once both locks are obtained. If two threads attempt to lock the same requests they will always do so in the same order and thus never deadlock.

Listing 5: The arbitration algorithm

```
for w in write_queue:
    for r in read_queue:
        double_lock(w, r)
        match(w, r)
        unlock(w, r)
```

The arbitration algorithm in Listing 5 then performs the protected matching by acquiring locks with the double_lock operation. For every Alternation, read or write action there is exactly one request and this request is always enqueued on the destination channel queues before the arbitration algorithm is run. It may seem unnecessarily expensive at first glance, but it is important to remember that if we do not enqueue the request before matching against potential matches, then there exists a scenario where a read-operation and a matching write-operation may be arbitrated in step-lock without detecting each other. An example of a correctly committed Alternation is shown in figure 1.



```
cin = A.reader()
id = cin()
```

```
cout = A.writer()
cin = B.reader()
Alternation([
  { cin:received_id() },
  { (cout, id):send_id(id) }
]).execute()
```

```
cout = B.writer()
cout(new_id)
```

```
cout = A.writer()
cout(id)
```

Figure 1. Snapshot of synchronization with two channels and four communicating processes. Channel B has found a match between two request structures; one in the input request queue (IR) and one in the output request queue (OR). Next, channel A will match the two active requests on channel A's request queues.

The presented algorithm for handling synchronization in PyCSP is relevant for `pycsp.threads` and `pycsp.processes`, while the `pycsp.greenlets` does not need this to ensure correctness. The algorithm is a main feature of the new PyCSP, if interested in other features of `pycsp.threads` then the description of these can be found in [2]. Next we will focus on the implementation details for `pycsp.processes` and `pycsp.greenlets`.

3.1. pycsp.greenlets

For co-routines, the greenlet module [9] was chosen because it is a very small module, easy to install, provides full control (no internal scheduler) and allows yielding from nested functions. Python's own generators which make it possible to create a co-routine-like API, do not allow yielding from nested functions, which would not allow us to yield when blocked on a channel communication. Another option was to use Stackless Python [10] for our implementation. Stackless Python was originally based on the greenlet design and has since then matured. It is slightly faster than the greenlet module and allows a larger number of allocated co-routines. However, having to install an extra Python interpreter to make the co-routine implementation run was found unacceptable, leaving the greenlet module as the only valid choice left.

A limitation with co-routines is that everything runs in a single thread, which means that a blocking call will block all other co-routines as well. This is especially a problem with IO operations, since the blocking action might happen in a system call, which we are not able to detect in the Python environment. The `@io` decorator attempts to solve this by wrapping a function into a *run* method on an `Io` thread object. This `Io` thread object is created on-the-fly and yields execution to the scheduler after starting the thread. When the thread's run method finishes, the return value is saved and the calling co-routine is moved onto the scheduler's next queue. Wrapping a function in an `@io` decorator introduces an overhead of starting and stopping a thread. We carried out a test, to see whether this overhead could be minimized by using a thread worker pool. The overhead was found to be similar to the time needed to start and stop a thread, thus the idea of a thread worker pool was abandoned. The idea of delegating a blocking system call to a separate thread was presented by Barnes [11] for the Kent Retargetable occam-π Compiler. occam-π implements a set of channels `keyboard` and `screen` that can be used to communicate to processes reserved for these IO operations. This could also be an option for PyCSP, but it was decided that the `@io` decorator would provide more flexibility for the programmer.

The channel communication overhead is much lower for greenlets than the other two implementations because we can avoid the conditions and locks when synchronizing.

To optimize for fast switching on channel communications, a central queue of blocked greenlets is not used when handling synchronizations. Whenever a greenlet blocks on channel communication, it saves a self-reference together with the channel communication request. Since channel communication requests are located in queues on channels these can be viewed as wait queues, from where a request is matched with an offer for communication. It is now the responsibility of another greenlet that matches this channel communication request to place the blocking greenlet on the scheduler's *next queue*. The scheduler uses a simple FIFO policy, thus choosing the first element of the *next queue* for execution. The *next queue* is usually short as most greenlets will be blocked on channel communication.

Listing 6: Blocking and scheduling a new greenlet CSP process

```
# Reschedule, without putting this process on either
# the next[] or a blocking[] list.
def wait(self):
    while self.state == ACTIVE:
        self.s.getNext().greenlet.switch()
```

When switching, we switch directly from CSP process to CSP process without spending any time on having to switch to a scheduler process. The code in Listing 6 is the wait method, which is executed when a CSP process blocks on channel communication. The method is responsible for scheduling the next CSP process. The `self.s` attribute is a reference to the scheduler, which is implemented as a singleton class. If the next and new queues are empty, then `getNext()` will return a reference to the scheduler greenlet which will then be activated. The scheduler greenlet will then investigate whether there are any current Timeout guards or `@io` threads active. In case all queues are empty it will terminate since everything must have been executed.

3.2. pycsp.processes

Using processes instead of threads requires that we run separate Python interpreters. For fast communication we can choose among several existing inter-process communication techniques, which includes message passing, synchronization and shared memory. Which techniques are available and how they are implemented differs between platforms. In order to have cross-platform support we construct the `pycsp.processes` implementation on top of the multiprocessing module available in Python 2.6. The multiprocessing module presents a uniform method of creating processes, shared values and shared locks. When Python objects are communicated through shared values, they are serialized using the pickle module [12]. Some Python objects cannot be serialized, shared values and locks are two examples of these. This requires us to initialize everything at startup, so that references can be passed to new processes as arguments. A singleton `ShmManager` class maintains all references to shared values and locks. This instance is automatically located in the memory address space of newly created processes.

Every channel instance requires a lock to protect critical regions, and every channel communication requires a condition linked to the channel request offered to processes to ensure that this request is updated in a critical region and can be signaled when updated. This usage of locks and conditions can be a problem when having many processes and channels. The total number of available locks and conditions in shared memory is much lower for the multiprocessing module than for the threading module. The solution was to let the ShmManager class maintain a small pool of shared conditions and locks. The size of the lock pool needs to be large enough to prevent a delay when entering a critical region. Likewise the size of the condition pool should be large enough to avoid waking up to many false processes, causing an overhead in context switches. 20 locks and 20 conditions seem to be enough for most situations possible with `pycsp.processes`, though a small performance increase is possible for the micro benchmark experiments by using more conditions.

Sending data around in a CSP network requires a method to actually transfer data from one process to another. Since all references to shared memory have to be initialized and allocated at startup a message buffer is allocated in shared memory. Unfortunately Python only supports allocating shared memory through the multiprocessing module, thus we will have to handle the memory management in PyCSP by calling get and set methods on objects allocated using the multiprocessing module. A large shared string buffer is allocated and partitioned into blocks of a static size. To handle the allocation of the required number of blocks for a channel communication and freeing them again afterwards, a dynamic memory allocator is implemented. The memory allocator uses a simple strategy that resembles the next-fit strategy:

init A list of free blocks is initialized with one entry that equals the entire message buffer.

alloc Any size is allocated by searching the list of free blocks for an entry that has enough space. The needed blocks are then cut from this entry and an index to the first block is returned.

free Allocated blocks are freed by appending an entry containing the index and size of the free blocks list.

Every new allocation will fragment the message buffer into smaller sections. If at some point we cannot find a partitioned area large enough, a step of combining free blocks is executed. This solution makes it possible to send both large messages and very small messages. If necessary, the buffer and block size can be tweaked using the `Configuration().set()` functionality.

We do not expect this dynamic memory allocator to affect the performance of parallelism in general, even though the allocation of a buffer is protected by a shared lock. The amortized cost of allocating buffers is low, since most allocations will be able to allocate blocks from the first entry in the list of free blocks and while the more rare and expensive action of reassembling blocks introduces a delay, it is a delay that will not affect the overall execution much. In the micro benchmarks (Section 5.1) and in the Mandelbrot experiment (Section 5.3) we successfully communicate small and larger data sizes.

4. Related Work

Communicating Sequential Processes (CSP) was defined by Hoare [5] in 1978, but it is during the last decade that we have seen numerous new libraries and compilers for CSP. Several implementations are optimized for multi-core CPUs that are becoming the de-facto standard when buying even small desktop computers. occam-π [13] and C++CSP2 [14] are two CSP implementations which stand out by being able to utilize multiple cores and use user-level threads for fast context switching. User-level threads are more efficient and provides greater flexibility than kernel-threads. They exists only to the user and can be made to use very little memory. It is possible to optimize the scheduling of threads to fit with the internal priority in the application, because the scheduler is in user code and the operating system is not involved. occam-π implements processes as user-level threads and uses a very robust and optimized scheduler that can handle millions of processes. The utilization of multiple cores is handled automatically by the scheduler and is described in detail in [15]. This is different from C++CSP2, where it is necessary to specify whether processes should be run as user-level threads or kernel-level threads.

Several libraries exist for Python that enable the Python programmer to manage tasks or threads, but they do not enable the programmer to easily change from threads to co-routines. Some of these libraries are Stackless Python [10], Fibra [16] and the multiprocessing module [3] and they provide an abstraction that uses the concept of processes and channels resembling a subset of the constructs available in the CSP algebra. Stackless Python is a branch of the standard CPython interpreter and provides very small and efficient co-routines (greenlets), bidirectional channels and a round-robin scheduler. Fibra is based on Python generators that are similar to co-routines, but it is impossible to hide the fact that a co-routine is a Python generator since the keyword `yield` is the only method to switch between generators. In Fibra, co-routines communicate through tubes by yielding values to a scheduler. The multiprocessing module in Python 2.6 provides a method of using operating system processes, shared memory and pipes for buffered communication. Operating system processes are heavy processes requiring a large amount of memory, but contrary to threads they are not affected by the Global Interpreter Lock (GIL).

However, no libraries exist for Python that provide the functionality of the choice construct that makes it possible to program with non-deterministic behaviour in the communication between processes.

5. Experiments

We have run three different experiments, to show the strengths and weaknesses of the PyCSP implementations. The first experiment consists of two micro benchmarks where one is showing how the implementations handle an increasing amount of processes until reaching the maximum possible amount. The other micro benchmark is showing how well an implementation copes with performing an increasing amount of concurrent communications in a network of static size. After the micro benchmarks, we generate primes using a simple PyCSP application as a case for when it is convenient to be able to switch from threads or processes to co-routines. Finally, a benchmark computing the Mandelbrot set is used to compare speedup on an 8-core system. The Mandelbrot set is computed twice using two different strategies and producing two very different speedup plots. One has the Global Interpreter Lock (GIL) released during computation by computing in an external module and one was computed using the numpy module [17]. All benchmarks are executed on a computer with 8 cores: two Intel Xeon E5310 Quad Core processors and 8 GB RAM running Ubuntu 9.04.

5.1. Micro Benchmarks

The results of these micro benchmarks provides a detailed view of how the implementations behave when they are stressed. The benchmarks are designed with the purpose of measuring the channel communication time including the necessary time required to context switch. Extra unnecessary context switches may be added by the operating system and is related to the PyCSP implementation used.

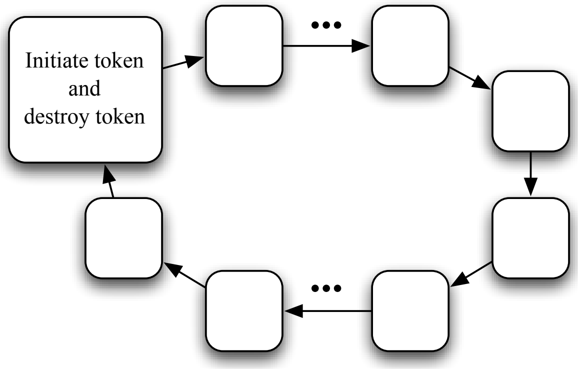


Figure 2. Ring of variable size

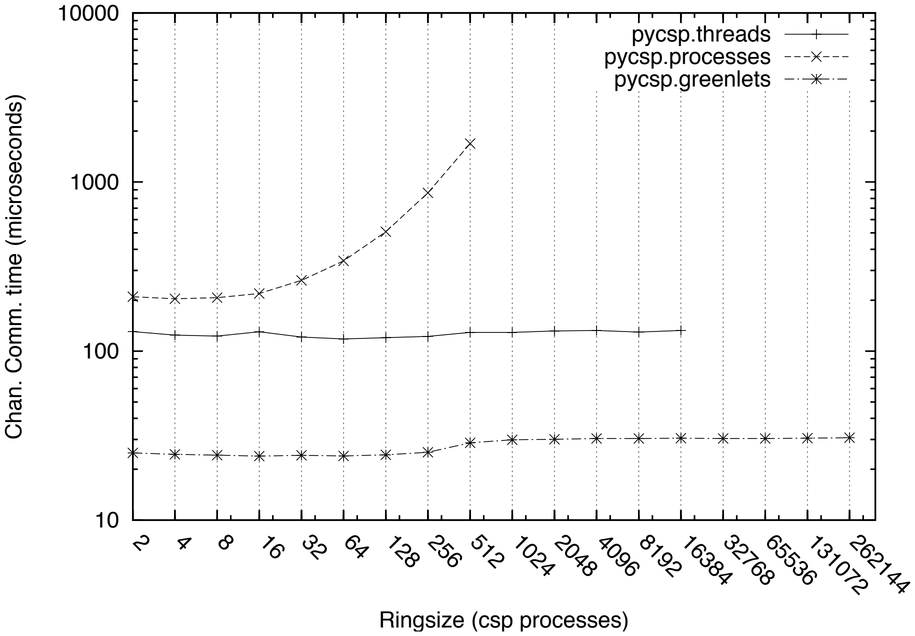


Figure 3. Micro benchmark measuring the channel communication time including the overhead of context switching for an increasing number of CSP processes.

Using the ring design in Figure 2, we run a benchmark that sends a token around a ring of increasing size. The ring benchmark was inspired from a similar micro benchmark in [15]. N elements are connected in a ring and every element passes a token from the previous element to the next. This challenges the PyCSP implementations ability to handle an increasing amount of processes and channels. The time measurements does not include startup and shutdown time and each measured run is divided by the size of the ring to compute an average channel communication time.

The test system has been tweaked to allow a larger number of threads and processes than the default. The results for our test system (in Figure 3) show that we can reach 512, 16384 and 262144 CSP processes depending on the PyCSP implementation used. It is obvious that `pycsp.processes` should only be used for applications with few CSP processes because of the exponential increase in latency, though it is possible to configure `pycsp.processes` using `Configuration().set(PROCESSES_SHARED_CONDITIONS, 50)` and achieve marginally better performance. As expected, `pycsp.greenlets` is able to handle a large number of CSP processes with only a small decrease in performance.

Investigating the performance in a different perspective, we use four rings of static size N and then send 1 to N-1 tokens to circle concurrently. In the previous benchmark there was only one communication at a time, which is a rare situation for an actual application. With this benchmark we see `pycsp.processes` performs much better, since it can now utilize more cores. Based on the results in Figure 4 we can conclude that `pycsp.processes` has a higher throughput of channel communications than `pycsp.threads` when enough concurrent communications can utilize several cores.

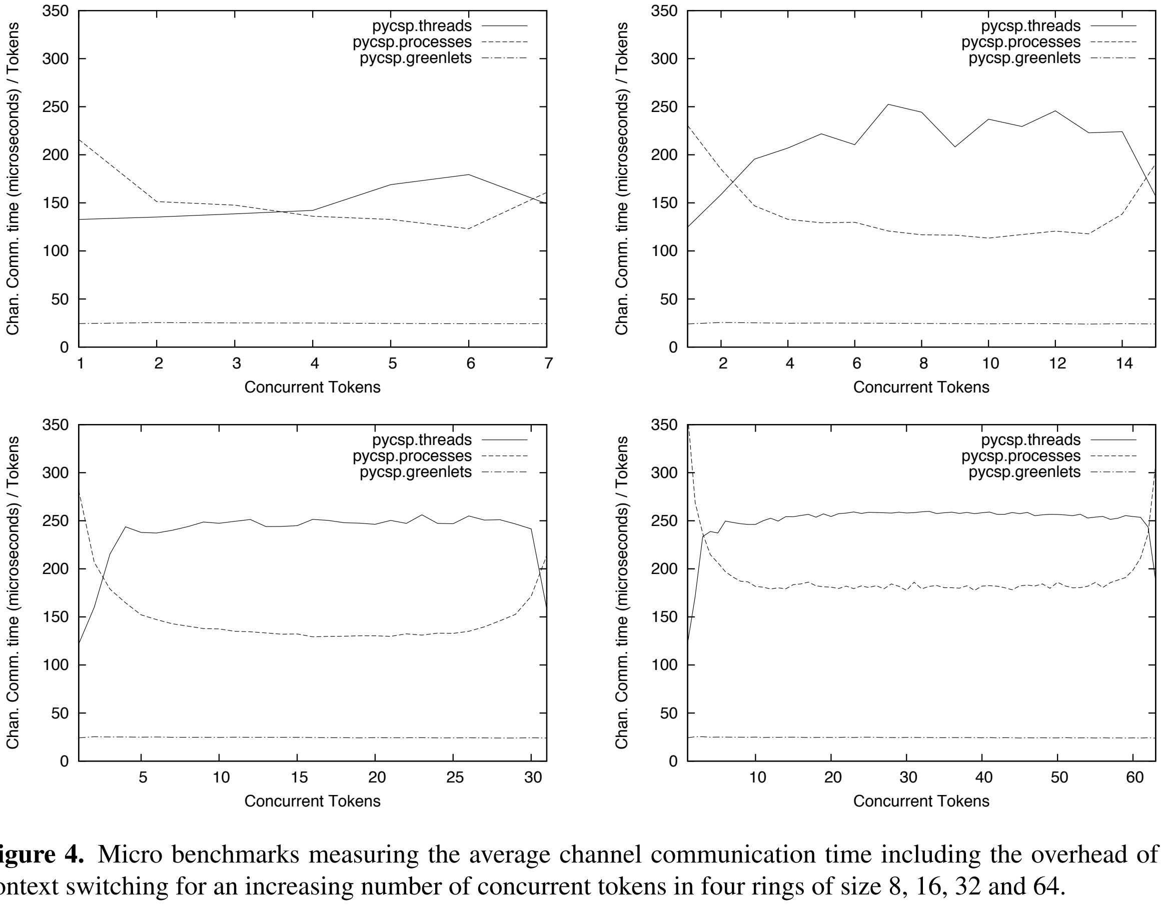


Figure 4. Micro benchmarks measuring the average channel communication time including the overhead of context switching for an increasing number of concurrent tokens in four rings of size 8, 16, 32 and 64.

Looking at the results for the four rings of size N in Figure 4, an interesting pattern is observed whenever the number of concurrent tokens comes close to N. For N-1 concurrent tokens the performance of `pycsp.threads` are almost equal to the performance of one concurrent token. The reason for this behaviour is explained by the blocking nature of CSP, because when all processes but one has a token, then only this one is able to receive. This behaviour mimics the behaviour of the test with one token and explains why the results in Figure 4 are mirrored around the center.

From these micro benchmarks we can see that, `pycsp.threads` performs consistently in both benchmarks. `pycsp.processes` does poorly in Figure 3 where the cost of adding more processes is high, but perform better in Figure 4 where a number of concurrent tokens are added. Finally `pycsp.greenlets` has proved able to do fast switching and many processes, regardless of the amount of concurrent tokens.

5.2. Primes

This is a simple and inefficient implementation of prime number generation found in [18]. The CSP design of the implementation is shown in Figure 5. It adds one CSP process for every computed prime, which sets a limit on how many primes can be calculated using this design. The maximum number of primes equals the maximum amount of CSP processes or channels possible. The latency involved in spawning new CSP processes and performing context switches varies when swapping between threads, processes and greenlets.

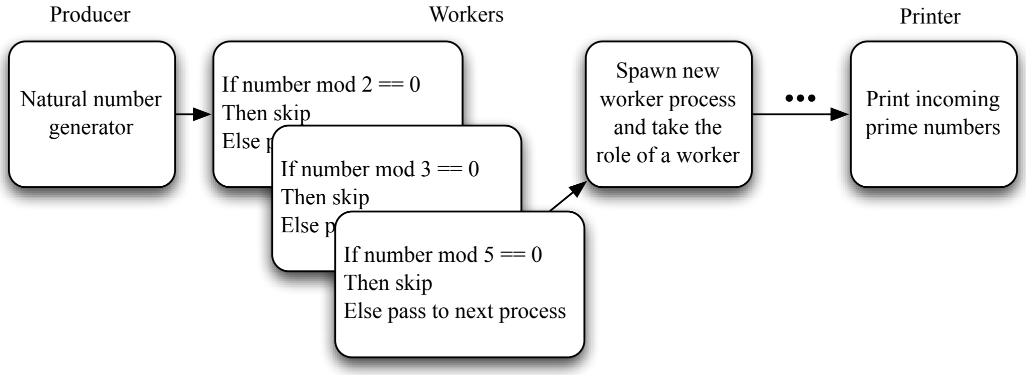


Figure 5. Primes CSP design

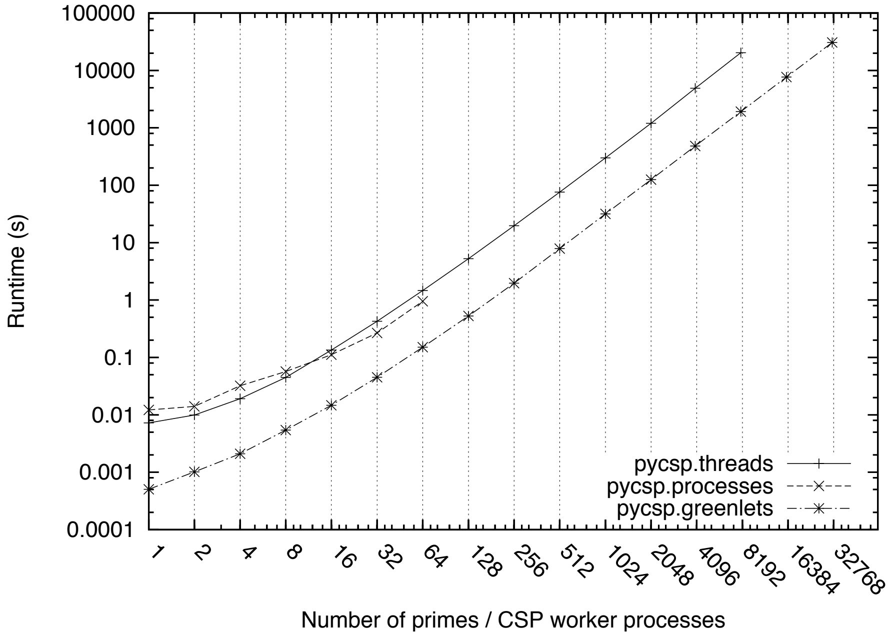


Figure 6. Results of primes experiment.

We run a benchmark computing primes, plotting the runtime results in Figure 6. The processes implementation failed with the message "maximum recursion depth exceeded" after creating 90 processes. This is a limitation in the Python multiprocessing module which is only apparent when spawning new processes from child processes.

This primes benchmark does not compare to a simple implementation in pure Python, which would be orders of magnitude faster than the implementation using PyCSP. This benchmark is meant as a method to compare one aspect of the PyCSP implementations and it proves why greenlets is an important player compared to threads and processes. Running for an entire day (86400s) would produce ≈ 16000 primes using the threads implementation and ≈ 60000 primes using the greenlets implementation. Also 16384 threads is close to an upper limit for threads, while greenlets has no real upper limit on the amount of greenlets.

5.3. Computing the Mandelbrot Set

This experiment is a producer-consumer-worker example that tests PyCSP's ability to utilize multiple cores. It produces the image in Figure 7 at a requested resolution. The image requires up to 5000 iterations for some pixels and is located in the Mandelbrot set at the coordinates:

xmin	=	-1.6744096758873175
xmax	=	-1.6744096714940624
ymin	=	0.000047164191972849 76
ymax	=	0.000047167062611931696

The simple CSP design in Figure 7 communicates jobs from the producer-consumer to the workers using the Alternation in Listing 7. Workers can request and submit jobs in any order they like.

Listing 7: Producer-Consumer: Delegating and receiving jobs

```
while len(jobs) > 0 or len(results) < jobcount:
    if len(jobs) > 0:
        Alternation([{
                workerIn: received_job,
                (workerOut, jobs[-1]): send_job
            }]).execute()
    else:
        received_job(workerIn())
```

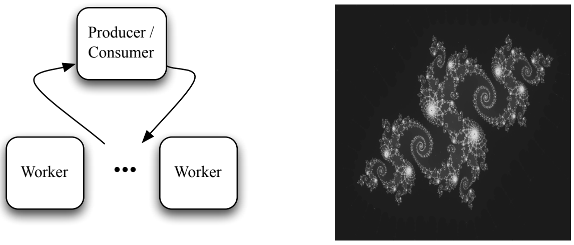


Figure 7. The Mandelbrot CSP design and the computed Mandelbrot set.

The experiment is divided into two different runs. They differ by using two different implementations of the worker process. One releases the GIL during computation by using the ctypes module [19] to call compiled code contained in an operating system specific dynamic library. Executing external code using ctypes is advanced, but does also provide a performance improvement over the other method which is using the numpy module [17] to manipulate and compute on matrices. The numpy module is a package used for scientific computing and provides a N-dimensional array object including tools to manipulate this array object. The numpy module also releases the GIL on every call, but this is much more fine-grained than the course-grained release and acquire used in the ctypes module, thus a larger overhead is expected for the numpy module.

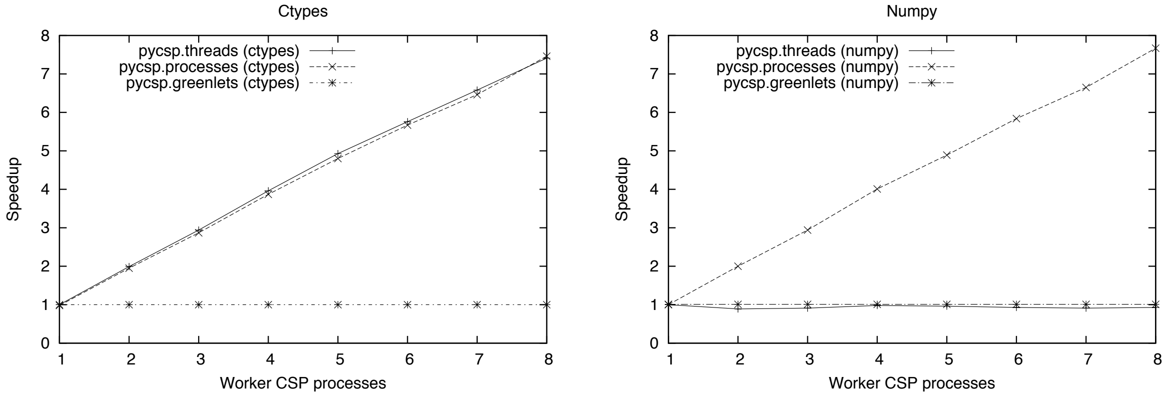


Figure 8. Speedup plots of computing the Mandelbrot set displayed in Figure 7. The resolution is $1000 * 1000$ and the work is divided in 100 jobs. The run time for the case with a single worker is used as the base for the speedup calculation and was 592.5 seconds for the numpy benchmark and 10.6 seconds for the ctypes benchmark.

The results in Figure 8 clearly shows that `pycsp.processes` is superior in this application by attaining a good speedup in both runs. It is interesting that `pycsp.processes` is able to compete with `pycsp.threads` when using the ctypes worker, since `pycsp.processes` for every communication includes an extra overhead of serializing data to a string format, allocating a message buffer, copying the string data to the message buffer, retrieving the string data from the message buffer, freeing the message buffer and finally unserializing the string data into a copy of the original data. As expected we have no multi-core speedup at all from using `pycsp.greenlets`. We could have wrapped the computation in the `@io` decorator and gained a speedup for the ctypes benchmark, but this is not the purpose of the `@io` decorator and would encourage wrong usage of the new PyCSP library.

Based on the experiments performed, the three implementations have different strengths; processes favors parallel processing, threading favors portability and applications that release the GIL and greenlets favor many processes and frequent communication.

6. Conclusions

With the PyCSP version presented in this paper, any application written in Python and using PyCSP can change the concurrent execution model from threads to co-routines or processes just by changing which module is imported. Depending on a user's domain and application a user can choose to circumvent the Global Interpreter Lock by using processes, provided that the application does not create more than the maximum allowed processes for the operating system. Alternatively, a user may want to speed up the communication time by a factor of ten by using greenlets. Then again if the application is changed further and the user suddenly wants to return to using threads, this is a simple task that does not require the user to transfer code changes to an older revision.

Using `pycsp.processes` it is now possible to utilize all cores of an 8-core system without requiring the computation to take place in an external module. This is important for programmers who want to utilize more cores when the performance of `pycsp.threads` is limited by the Global Interpreter Lock. Additionally, running more than 262144 processes in a single PyCSP application is made possible using `pycsp.greenlets`. This amount is smaller than what is possible with occam-π [13] or C++CSP2 [14], but it does open up to the possibility of developing more fine-grained CSP-designs using PyCSP.

PyCSP is available at Google-code using the project name `pycsp` [20].

6.1. Future Work

The obvious next step would be to create `pycsp.net`, a distributed version of PyCSP that connects processes by networked channels. `pycsp.net` would be required to be fully compatible with the current API, so that any PyCSP application can be transformed into a distributed application, just by changing the imported module. Channels could be given names so that they could be registered on a nameserver and identified from different hosts.

Using `pycsp.net` and running the Mandelbrot benchmark application from the *Experiments* section would allow us to utilize multiple machines. The producer-consumer would be started on one host, and starting additional worker processes on other hosts would be trivial, since they would request the correct channel reference from the nameserver by a known name and automatically start requesting jobs.

References

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

[2] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP Revisited. In Peter H. Welch, editor, *Communicating Process Architectures 2009*, Amsterdam, The Netherlands. WoTUG, IOS Press.

[3] Python multiprocessing module. `http://docs.python.org/library/multiprocessing.html`.

[4] unladen-swallow distribution. `http://code.google.com/p/unladen-swallow/`.

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

[6] Peter H. Welch, Neil C.C. Brown, Jim Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In Steve Schneider, Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 349–370, Amsterdam, The Netherlands, July 2007. WoTUG, IOS.

[7] Peter H. Welch. Tuna: Multiway synchronisation outputs. `http://www.cs.york.ac.uk/nature/tuna/outputs/mm-sync/`, 2006.

[8] Alastair R. Allen, Oliver Faust, and Bernhard Sputh. Transfer Request Broker: Resolving Input-Output Choice. In Frederick R.M. Barnes, Jan F. Broenink, Alistair A. McEwan, Adam Sampson, G. S. Stiles, and Peter H. Welch, editors, *Communicating Process Architectures 2008*, September 2008.

[9] greenlet distribution. `http://pypi.python.org/pypi/greenlet`.

[10] Christian Tismer. Continuations and stackless python. *Proceedings of the 8th International Python Conference*, Jan 2000.

[11] Frederick R.M. Barnes. Blocking system calls in KRoC/Linux. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering Series*, pages 155–178. Computing Laboratory, University of Kent, IOS Press, September 2000.

[12] Python pickle module. `http://docs.python.org/library/pickle.html`.

[13] occam-pi distribution. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.

[14] Neil C.C. Brown. C++CSP2: A many-to-many threading model for multicore architectures. *Communicating Process Architectures 2007: WoTUG-30*, page 23, Jan 2007.

[15] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009.

[16] Fibra distribution. `http://code.google.com/p/fibra/`.

[17] numpy distribution. `http://numpy.scipy.org/`.

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

[19] Python ctypes module. `http://docs.python.org/library/ctypes.html`.

[20] PyCSP distribution. `http://code.google.com/p/pycsp`.

A.6 PyCSP - controlled concurrency

IJIPM: International Journal of Information Processing and Management, Vol. 1, No. 2, pp. 40 – 49, 2010
DOI: 10.4156/ijipm.vol1.issue2.6

Rune Møllegaard Friborg, Brian Vinter, John Markus Bjørndalen: PyCSP - controlled concurrency

PyCSP - controlled concurrency

Rune Møllegaard Friborg
University of Copenhagen, Denmark
runef@diku.dk

Brian Vinter
University of Copenhagen, Denmark
vinter@diku.dk

John Markus Bjørndalen
University of Tromsø, Norway
jmb@cs.uit.no

doi: 10.4156/ijipm.vol1.issue2.6

Abstract

Producing readable and correct programs while at the same time taking advantage of multi-core architectures is a challenge. PyCSP is an implementation of Communicating Sequential Processes algebra (CSP) for the Python programming language, that take advantage of CSP's formal and verifiable approach to controlling concurrency and the readability of Python source code. We describe PyCSP, demonstrate it through examples and demonstrate how PyCSP compares to Pthreads in a master-worker benchmark.

Keywords: *Python, Concurrency, CSP*

1. Introduction

Maintaining scientific code is a well-known challenge; many applications are written by scientists without any formal computer science or software engineering qualifications, and are usually grown "organically" from a small kernel to hundreds of thousands of code-lines. These applications have traditionally targeted simple single core systems and have still grown to a complexity where the cost of maintaining the codes is prohibiting, and where the continued correctness of the code is often questionable. This problem is being addressed today by training scientists in some kind of structured program development. However, emerging architectures, which are massively parallel and often heterogeneous, may again raise the complexity of software development to a level where non computer scientists will not be able to produce reliable scientific software.

1.1. Motivation

PyCSP [1] is intended to help scientists develop correct, maintainable and portable code for emerging architectures. Python is highly suited for scientific applications. While it is interpreted and thus very slow, scientific libraries efficiently utilize the underlying hardware. CSP provides a formal and verifiable approach to controlling concurrency, fits directly into scientific workflows, and maps directly onto many graphical tools that present scientific workflows such as Taverna[2], Knime[3] and LabView[4].

CPUs are produced with multiple cores today and every announced future CPU generation[5] seems to feature an ever increasing number of cores. As single core performance increase very slowly, researchers are required to exploit this parallel hardware for increased performance. To this end a number of parallel libraries like BLAS and programming tools like Intel Parallel Studio[6] are appearing. Unfortunately parallel libraries are often not enough to achieve acceptable speed and even with advanced tools parallel programming remains a source of added complexity and new bugs in software development.

The intended users for PyCSP are not computer scientists, but scientists in general. General scientists can not be expected to learn CSP as formulated by Hoare [7], thus the approach to controlling concurrency in this paper is based on CSP, but does not require any knowledge of CSP. The key elements of controlling concurrency using PyCSP is presented in section 3.

1.2. Related Work

During the last decade we have seen numerous new libraries and compilers for CSP. Several implementations are optimized for multi-core CPUs that are becoming the de-facto standard when buying even small desktop computers. Occam-π [8], C++CSP [9] and JCSP [10] are three robust CSP implementations of CSP. C++CSP and JCSP are libraries for C++ and Java, while Occam-π uses CSP inherently in the programming language.

2. CSP

The Communicating Sequential Processes algebra, CSP [7], was introduced more than 25 years ago and while it was highly popular and thoroughly investigated in its first years, interest dropped off in the late 1980 because the algebra appeared to be a solution in search of a problem, namely modelling massively concurrent processes and providing tools to solve many of the common problems associated with writing parallel and concurrent applications.

CSP provides many attractive features with respect to the next generation processors; it is a formal algebra with automated tools to help prove correctness, it works with entirely isolated process spaces, thus the inherent coherence problem is eliminated by design, and it lends itself to being modelled through both programming languages and graphical design tools.

Another attractive feature of CSP, which has so far not been investigated, is the fact that it should lend itself towards modelling heterogeneous systems. This is important for the next generation processors since heterogeneity has already been introduced: the CELL-BE processor features two architectures on the core, while the Tesla processors require a classic processor for managing the overall system and the scalar portions of an application.

3. PyCSP

PyCSP provides an API that can be used to write concurrent applications based on CSP. PyCSP was introduced in 2007 [1] and revised in 2009 [11]. The API is implemented in four versions: Threads, processes, greenlets and networked. All four versions are packed in a single module, to motivate the developer to switch between them as needed. A common API is used for all implementations making it trivial to switch between them, as shown in table 1. When switching to another implementation, the PyCSP application may execute very differently as processes may be scheduled in another order and less fair. Hidden latencies may also become more apparent when all other processes are waiting to be scheduled. Having several implementations sharing one API was presented in [12].

The four implementations are:

- pycsp.threads - A CSP process is implemented as an OS thread. The internal synchronization is handled by thread-locking mechanisms. This is the default implementation. Because of the Python Global Interpeter Lock[1], this is best suited for applications that spend most of their time in external routines that release the GIL.

- pycsp.processes - A CSP process is implemented as an OS process. The internal synchronization is more complex than pycsp.threads and is built on top of the multiprocessing module available in Python 2.6. This implementation is not affected by the Global Interpreter Lock, but has some limitations on a Windows OS and generally has a larger communication overhead than the threaded version.

- pycsp.greenlets - This implementation uses co-routines instead of threads. Greenlets [13] is a simple co-routine implementation available as a Python module. It provides the possibility of creating 100.000 CSP processes in a single CSP network. This version is optimal for single-core architectures since it provides the fastest communication, but with no parallelism.

[1] Python uses a Global Interpreter Lock, the GIL, to protect the interpreter when multiple threads execute Python code. The GIL limits concurrency when executing Python code, but libraries commonly mitigate the problem by releasing the GIL when executing external code.

• pycsp.net - A proof-of-concept network enabled implementation based on pycsp.threads. All synchronization is handled in a single process. This provides the same functionality as pycsp.threads, but adding a larger cost and a bottleneck by introducing a server process. It uses Pyro [14] for communication.

Table 1. Switching between implementations of the PyCSP API

pycsp.threads	**pycsp.processes**
`import pycsp.threads as pycsp`	`import pycsp.processes as pycsp`
`@pycsp.process`	`@pycsp.process`
`def P(msg):`	`def P(msg):`
`    print msg`	`    print msg`
`pycsp.parallel(P("Hello World"))`	`pycsp.parallel(P("Hello World"))`

3.1. Processes

A process in PyCSP is an isolated unit of execution, not physically isolated as an operating system process, but by design should not share objects with other processes. A process is specified by using the @process decorator as depicted in table 2. Applying this decorator to the increment function, creates an increment factory, which produces increment process instances. Executing the process is covered in section 3.3.

3.2. Networks of Processes

The only allowed methods to communicate between processes are to either pass arguments when creating processes or by sending messages over channels. All channel communications are blocking operations and are guaranteed to be sent exactly once and received by exactly one process. A channel is created using the Channel class and can have any number of writing processes and any number of reading processes.

```
A = pycsp.Channel("A")
```

Processes are usually passed their input and output channels as parameters which can then be used to communicate with other processes. An example of this usage is shown in table 2.

Table 2. Example procress with IO

```
@pycsp.process
def increment(cin, cout,  inc_value=1):
   while  True:
      cout(cin() +  inc_value)
```

To communicate on a channel, an application is required to get a hold of an input or output end of the channel. A reference to a channel end is acquired by using the chan.reader() or chan.writer() channel methods, which will return a ChannelEnd object that can be used to read and write on the channel respectively. Acquiring a channel end object will also join the actual channel. This adds information to the channel, letting it know how many readers and writers that are connected to it. The number of readers and writers is used to automate poisoning explained in section 3.5. To create an increment instance P and provide it with channel ends we do the following:

```
P = increment(a.reader(), b.writer())
```

Communicating on a channel end is performed by invoking `cin()` or `cout(msg)` when the channel ends have been connected like this: `cin =  a.reader()` or `cout =  a.writer().`

Usually channel ends are provided as arguments to new processes as shown in the example of the increment instance P.

3.3. Concurrency

Creating a process will simply instantiate a copy of the process but not execute or start it in any way. A set of processes may be executed using one of three ways, Sequence, Parallel or Spawn. Sequence and Parallel are synchronous and will only terminate once all processes in their parameter list are terminated. Sequence executes the processes one at a time while Parallel executes all the processes concurrently. Spawn starts the set of processes in the background and then returns, one may view it as an asynchronous version of Parallel.

Table 3. Initiating processes for execution

```
a,b =  pycsp.Channel() * 2
pycsp.Parallel(
    counter(a.writer()), 10),
    increment(a.reader(), b.writer()),
    printer(b.reader())
)
```

The code in table 3 completes when the counter, increment and printer processes have completed. Section 3.5 explains how to provoke a termination of the `Parallel(processes)` execution.

3.4. Nondeterminism

When an input or output channel end is invoked, as explained in section 3.2, the executing process are committed to this channel until this communication has completed. This complicates writing processes that need to respond to one of several communication events that may come in any order.

Using the AltSelect construct, it is possible to commit to waiting for one of several conditions, called guards, to become true. Channel ends automatically have guards that become true when communication can be completed on that end. This allows a process to wait for one of several communication events to become ready, and automatically select and execute one of them (Figure 1).

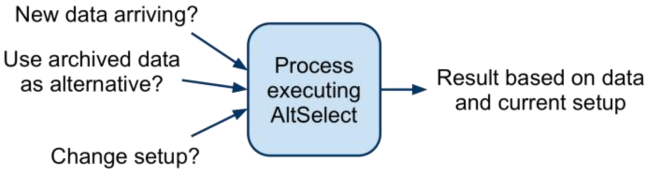

New data arriving?
Use archived data as alternative?
Change setup?
Process executing AltSelect
Result based on data and current setup

Figure 1. Selecting from mbultiple communication events

To do this, the programmer constructs a guard set, which is represented as a prioritized list of guards where a guard is a Guard object and can be initialized with an attached action. An action is a function of type choice and is executed if the guard is selected. If the guard is an input guard then the choice function will always be handed the parameter `channel_input` which is the message that was read from the channel. AltSelect will also always return the tuple (guard, msg). The returned guard is the guard that was chosen by AltSelect. If the guard was an input guard, msg is the message that was read from the chosen channel, otherwise msg has the value None. If a guard action was defined then it will be executed before AltSelect returns.

Note that AltSelect always performs the guard that was chosen, i.e. channel input or output is executed within the AltSelect, so an AltSelect execution with no declared choice, or a choice where the results are simply ignored, still performs the guarded input or output. An example of AltSelect usage is shown in table 4.

Table 4. AltSelect

```
@pycsp.choice
def read_action(info,  channel_input):
    print info,  channel_input

@pycsp.choice
def write_action():
    print  ’W’

@pycsp.process
def par_in_out(cin1,  cin2, cout3, cnt):
    for i in range(cnt):
        pycsp.AltSelect(
            pycsp.InputGuard(cin1, read_action(info="received  on  cin1")),
            pycsp.InputGuard(cin2, read_action(info="received  on  cin2")),
            pycsp.OutputGuard(cout3, i, write_action())
        )
```

The guard types included in the distribution are:

- `InputGuard` - channel end input
- `OutputGuard` - channel end output
- `TimeoutGuard(seconds)` - when expired, it will commit.
- `SkipGuard()` - at first change it will commit.

The order of guards in a guard set is important. A guard set having a skip guard as the first item will always commit to this skip guard, thus skip is usually used as the last item in a guard set. A timeout guard will try to commit when the defined seconds have passed. An example usage of timeout is shown in table 5.

Table 5. AltSelect with timeout

```
(guard_selected, msg) = pycsp.AltSelect(
    pycsp.InputGuard(cin),
    pycsp.TimeoutGuard(seconds=1)
)

if isinstance(guard_selected,  pycsp.TimeoutGuard):
    print ’timeout!’
```

3.5. Termination

A controlled shutdown of a CSP network (set of processes connected by channels) can be performed by using poisoning [15]. Poisoning of a network may happen in one of two ways, either as an explicit poison which will propagate the entire network instantly and cause a fast termination, or as an incremental retirement which allows all processes to finish their current work before termination. An explicit poison is performed using the `poison(channel/channelend)` call. The less intrusive poison can be performed by using the `retire(channelend)` call. Calling retire will cause a decrement of an internal counter inside a channel. When a `retire()` call causes a channel to have 0 readers or 0 writers left, the channel is permanently retired and any access will raise an exception as described below.

Upon the permanent retirement or poisoning of a channel, all processes that access the channel will raise a `ChannelRetireException()` or `ChannelPoisonException()` respectively. Any following reads or writes on same channel will also raise an exception. Whether this exception is caught inside the process or passed on is left to the programmer. The default behaviour is that the Process class will catch the exception and then, depending on whether it is poisoned or retired, the following occurs: all channels and channel ends in the argument list of the poisoned process are poisoned, thus propagating a poison signal through all known channels. All channel ends in the argument list of the retired process are retired, thus retiring the local channel ends from all known channels. In table 6 a `ChannelRetireException()` is caught inside a process.

Table 6. Controlling termination

```
@pycsp.process
def printer(cin):
    try:
        i = 0
        while True:
            print cin()
            i +=  1
    except pycsp.ChannelRetireException:
        print ’Printed’,  i, ’values’
```

To initiate termination, we poison the network nicely by calling retire in table 7. This propagates a retire signal when all output ends of a channel are retired. Instead of `retire()`, `poison()` could be used which poisons the channel instantly. Without the ability to poison or retire channels, a programmer would have to create the extra control flow necessary for controlling the shutdown of processes. Poisoning and retiring channels are compact and simple ways to shut down processes and saves the programmer from the work of creating the extra control flow necessary for a controlled shutdown of a group of processes.

Table 7. Initiating termination

```
@pycsp.process
def counter(cout,  N):
    for i in range(N):
        cout(i)
    pycsp.retire(cout)
```

4. Examples

The first example is an application that computes the Mandelbrot set. This example is also used in section 5 to compare PyCSP performance to Pthreads. It consists of a manager process and a number of worker processes. The design is modelled in figure 2. The manager divides the computation into jobs and loops on the AltSelect in table 8, until all jobs have been computed.

Table 8. Manager process: Deliver and receive loop

```
while jobs or len(results) < jobcount:
    if jobs:
        pycsp.AltSelect(

            # If selected, a job is read
            # from workerIn.
            pycsp.InputGuard(workerIn, action=received_job()),

            # If  selected, the job at the
            # end of the jobs list is
            # written to workerOut.
            pycsp.OutputGuard(workerOut, msg=jobs[-1], action=send_job())
        )
    else:
        # No more jobs left to send. Retrieve remaining results.
        # Invoke received_job action on input from worker.
        received_job()(workerIn())
```

This AltSelect handles all synchronization between the manager and the workers. For every loop, one of two things will occur. Either a new job is sent to a worker asking for a job, or a new job result is received from a worker finished computing. When all jobs are computed and received, they are glued together and the computation has ended.

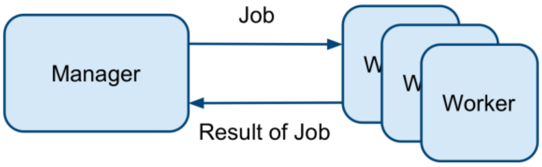


Figure 2. Mandelbrot PyCSP design

The second example is a simple webserver that runs a set of services. In figure 3 we demonstrate how easily this implementation can be mapped into a model consisting of connected processes. A service can register itself by sending an output channel end to the dispatcher. This is then entered into a service dictionary inside the dispatcher. When an incoming request is received it is sent to a matching service if one exists. A popular service might register several processes to handle a greater load. The dispatcher is able to listen for both new services registering and incoming requests by using AltSelect on input guards.

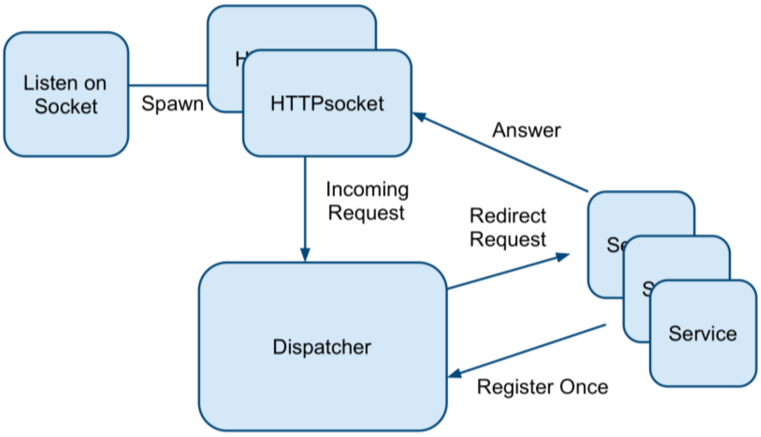


Figure 3. Webserver PyCSP design

An example of a hello world service is shown in table 9. It performs a self-registering step and then goes into a loop. In this loop requests are received and bundled together with an output channel end. When returning the result for a request, the result is sent on `cout`, which communicates directly to the HTTPSocket process handling the client connection.

Table 9. HelloWorld service

```
@pycsp.process
def HelloWorld(register):
    req_chan = pycsp.Channel()
    cin = req_chan.reader()
    register(('/hello.html', req_chan.writer()))
    while  True:
        (req_str, cout) = cin()
        cout("Hello  World")
```

The simple webserver has one bottleneck, the dispatcher. All other processes can be multiplied in numbers, to do loadbalancing. When a request is dispatched to a service, the dispatcher is free to handle other requests and is not required to wait for any services to finish.

The final example combines processes allocated for user input and output with performing a stochastic search for a local minimum in parallel. The PyCSP design is shown in figure 4 and maps directly to the actual code. A version of the code necessary to perform this search in parallel, excluding the actual worker, is shown in table 10. This simple network shows how a concurrent interactive application can be made using PyCSP. The UserIn process can easily parse inputs from the keyboard and perform actions depending on the input. In this example when the user enters an input, the network is terminated with a poison call.

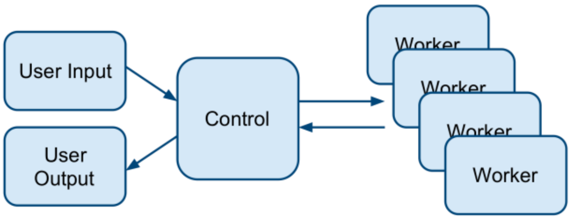


Figure 4. Stochastic Minimum Search PyCSP design

Table 10. Stochastic Minimum Search

```
@pycsp.process
def Control(kbd, scr,  workers_i):
    best =  9999 while  True:
    guard_selected, candidate = pycsp.AltSelect( pycsp.InputGuard(kbd),
                                                 pycsp.InputGuard(workers_i)
                                )
    if guard_selected == workers_i:
        if candidate < best:
            best = candidate
            scr(best)

@pycsp.process
def UserIn(kbd):
    raw_input("Terminate")
    poison(kbd)

@pycsp.process
def UserOut(scr):
    while  True:
    print scr()

pycsp.Parallel(
    UserIn(kbdChan.writer()),
    UserOut(scrChan.reader()),
    Control(kbdChan.reader(), scrChan.writer(),
            updateChan.writer(), workerChan.reader()),
    Worker(updateChan.reader(), workerChan.writer())  * 4
)
```

As long as the application is running the workers will search for a new local minimum, constantly updating the `Control` process with new candidates which are printed to the screen by `UserOut`. Upon termination the poison signal is propagated to all processes and the application quits.

5. Performance

To compare the overhead of using PyCSP with a C program using Pthreads, we run a benchmark that computes the Mandelbrot set. The benchmarks both execute the same C function for computing the points in the Mandelbrot set. The PyCSP version calls the C function using the standard Python ctypes library. Since the computation time for each region of the computed set is irregular, we use a bag-of-tasks scheme to provide automatic load-balancing. Each worker thread retrieves a task description from a task queue protected with a lock (C version) or a manger process using channels (PyCSP version), computes the subset requested in that task description, and stores the results before fetching a new task.

The benchmarks are executed on a computer with 8 cores: two Intel Xeon E5310 Quad Core processors and 8GB RAM running Ubuntu 9.04. We use PyCSP version 0.7.0 with Python 2.6.2.

The measured time for each run includes the startup and completion time for the worker threads or PyCSP processes, but not the startup time of the main program.

5.1. Results

Figure 5 shows the speedups of the PyCSP and Pthreads implementations of the benchmark when run using various problem sizes. The number of tasks is kept constant at 100, while the size of the total problem is varied from 10x10 points to 2560x2560 points.

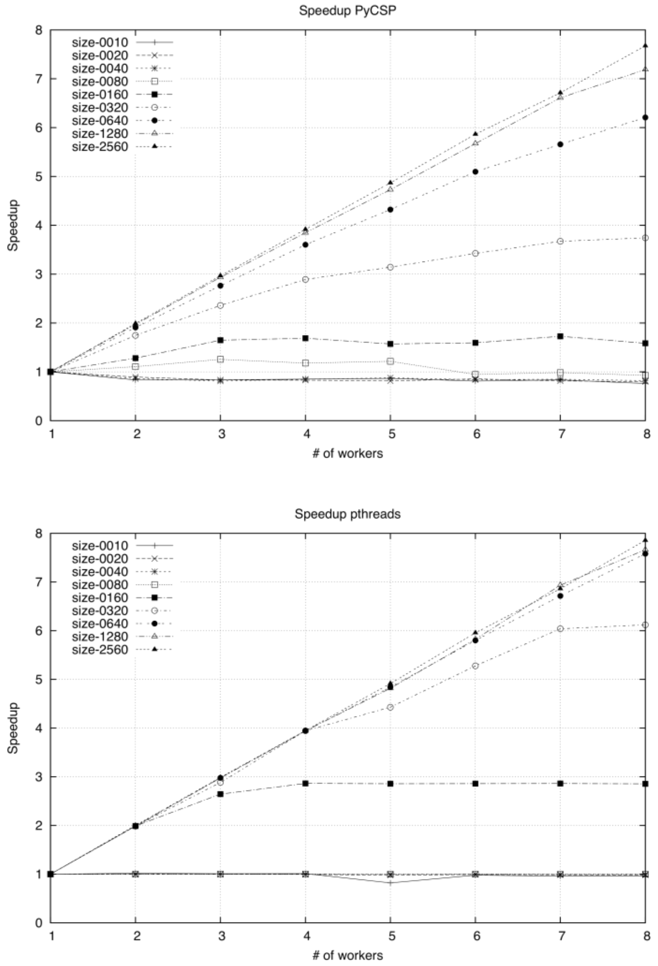


Figure 5. Speedup of PyCSP and Pthreads Mandelbrot computations

As expected, the Pthreads version approaches linear speedup earlier than the PyCSP version: the 640x640, 1280x1280 and 2560x2560 problems are close to linear, while in the PyCSP version only the two largest problems are close to linear. The execution time for the largest problem with a single worker is similar in both versions: 49.48 seconds for PyCSP and 49.36 seconds for Phtreads. For 8 workers, the numbers are 6.45 seconds for PyCSP and 6.28 seconds for Pthreads.

The main difference between the versions is the overhead of the interpreted Python language, and also that the PyCSP workers need to wake up and interact with a manager process, while in the Pthreads version, workers only need to grab and release a lock.

The results shows that for this benchmark, PyCSP and the choices made in the PyCSP solution add a small overhead compared to the Pthreads version, but that the overhead is not overly large: the smallest problem size that approaches linear speedup is 4 times larger for the PyCSP version than the Pthreads version.

6. Conclusions

We have described PyCSP, an implementation of Communicating Sequential Process algebra (CSP) for the Python programming language. PyCSP takes advantage of CSP’s formal and verifiable approach to controlling concurrency. The close mapping between the graphical representation of CSP programs and the PyCSP source code makes it easy to compare design documents and implementations, helping programmers manage the complexity that is often introduced when introducing parallel architectures.

In this paper we show that using PyCSP, we can get fairly close to the efficiency of a Pthreads implementation of a Mandelbrot benchmark. The smallest problem size that provides a near linear speedup with PyCSP using 8 CPU cores is four times the smallest problem size that provide a near linear speedup with the Pthreads version. This is close enough that we believe PyCSP to be usable for scientific computations.

7. References

[1] J. M. Bjørndalen, B. Vinter, and O. Anshus, “PyCSP - Communicating Sequential Processes for Python”, In Communicating Process Architectures 2007, pp. 229-248, 2007.

[2] Taverna Project, http://taverna.sourceforge.net/.

[3] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel, “Knime: The konstanz information miner”, In Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007), Springer, 2007.

[4] LabView. http://www.ni.com/labview/.

[5] B. Vinter, “Next generation processes”, In B. Topping and P. Ivanyi, editors, Parallel, Distributed and Grid Computing for Engineering, Computational Science, Engineering and Technology, pp. 21–33, Saxo-Coburg Publications, 2009.

[6] Intel parallel studio. http://software.intel.com/en-us/intel-parallel-studio-home/.

[7] C. Hoare, “Communicating sequential processes”, Communications of the ACM, 21(8):666-677, pp. 666–677, 1978.

[8] P.H.Welch and F. Barnes, “Communicating mobile processes - introducing occam-pi”, In Communicating Sequential Processes, Lecture Notes in Computer Science, pp. 175–210, Springer-Verlag, 2005.

[9] N. C. Brown, “C++CSP2: A Many-to-Many Threading”, In Communicating Process Architectures 2007, pp. 183–206, 2007.

[10] P. H. Welch, N. C. Brown, J. Moores, K. Chalmers, and B. Sputh, “Integrating and Extending JCSP”, In Communicating Process Architectures 2007, pages 349–369, 2007.

[11] B. Vinter, J. M. Bjørndalen, and R. M. Friborg, “PyCSP Revisited”, In Communicating Process Architectures 2009, pp. 263-276, 2009.

[12] R. M. Friborg, J. M. Bjørndalen, and B. Vinter, “Three Unique Implementations of Processes for PyCSP”, In Communicating Process Architectures 2009, pp. 277-292, 2009.

[13] Greenlet: Lightweight in-process concurrent programming, http://pypi.python.org/pypi/greenlet.

[14] Pyro: Python remote objects, http://pyro.sourceforge.net/.

[15] B. H. Sputh and A. R. Allen, “JCSP-Poison: Safe Termination of CSP Process Networks”, In Communicating Process Architectures 2005, pp. 71-107, 2005.

A.7 Rapid Development of Scalable Scientific Software Using a Process Oriented Approach

JOCS: Journal of Computational Science, In Press, Corrected Proof
DOI: 10.1016/j.jocs.2011.02.001

Rune Møllegaard Friborg and Brian Vinter: Rapid Development of Scalable Scientific Software Using a Process Oriented Approach

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Rapid development of scalable scientific software using a process oriented approach

Rune Møllegaard Friborg*, Brian Vinter

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

ARTICLE INFO

Article history:
Received 28 October 2010
Received in revised form 31 January 2011
Accepted 11 February 2011
Available online xxx

Keywords:
Communicating Sequential Processes
Many-core
Grid computing
Python

ABSTRACT

Scientific applications are often not written with multiprocessing, cluster computing or grid computing in mind. This paper suggests using Python and PyCSP to structure scientific software through Communicating Sequential Processes. Three scientific applications are used to demonstrate the features of PyCSP and how networks of processes may easily be mapped into a visual representation for better understanding of the process workflow. We show that for many sequential solutions, the difficulty in implementing a parallel application is removed. The use of standard multi-threading mechanisms such as locks, conditions and monitors is completely hidden in the PyCSP library. We show the three scientific applications: *k*NN, stochastic minimum search and McStas to scale well on multi-processing, cluster computing and grid computing platforms using PyCSP.

1. Introduction

This paper describes a Python library for building and executing scientific applications that utilize parallel architectures, from multi-core machines to full grid systems. The driving motivation is to give scientists that are not Computer Science majors the possibility to build correct, efficient, modular, reusable and scalable applications.

It is a well-known challenge to maintain scientific codes; many applications are written by scientists without any formal computer science or software engineering qualifications, and have usually grown "organically" from a small kernel to hundreds of thousands of code-lines. Such applications have traditionally targeted simple single-core systems and have still grown to a complexity where the cost of maintaining the codes is prohibiting, and where the continued correctness of the code is often questionable. This problem is being addressed today by training scientists in design patterns and good practice in program development. However, emerging architectures, which are massively parallel and often heterogeneous, may again raise the complexity of software development to a level where scientific users (non-computer scientists) are no longer able to produce reliable scientific software.

We use the term scientific users quite loosely to mean programmers that are scientists, but not computer scientists. Scientific users of computing is in itself a diverse group, one extreme is scientists that use existing applications, commercial or community codes, where they change configurations and input data, but in general do not change the code itself. The other extreme are scientists that primarily do program development, often the persons behind large community codes. The approach we promote here targets the set in between the two extremes, scientists that express the model they work on directly as a computer program, but where the programming is still a means to an end, and not the primary focus of the research. This kind of computational-scientist typically changes the code frequently, and the code is most often shared with a small number of co-researchers, typically within a research group. Thus we fundamentally target scientific users of computers that do their own programming and change their program frequently to match the development in their research. They are interested in better performance of their applications, but productivity and time to solution are the critical measures in their program development.

The classic approach to parallel programming involves threads, shared memory and locks, but this requires that the programmer identifies all critical regions and dependencies correctly without resulting in serialized executions, caused by large computational parts in critical regions. Scientific users usually avoid this kind of programming and use OpenMP[1] to add parallelization. OpenMP is mostly used for loop parallelization and requires identifying all critical regions within a loop. Another approach is to use parallelized libraries like BLAS,[2] but these

* Corresponding author. Tel.: +45 26297742.
E-mail address: runef@diku.dk (R.M. Friborg).

[1] OpenMP: http://openmp.org/.
[2] BLAS: http://www.netlib.org/blas/.

1877-7503/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.jocs.2011.02.001

are often not enough and are limited to shared memory systems.

Scientific users must be encouraged to write maintainable and well-structured code. Several tools like the Intel Parallel Studio[3] or Microsoft Parallel Computing[4] exist that aim to aid programmers in producing parallel programs, but these tools are not flexible enough for scientific applications and do not interface well with code produced during the past 30 years which is still in use. Most scientists have access to many different kinds of computing resources, thus the produced code must also be portable and must run on single-core, multi-core, cluster or grid systems. Graphical workflow systems as Knime [1,2] and Taverna [3,4] are helping scientists to structure code, but they lack sufficient support for parallel execution.

We present PyCSP as a Python module with a small API, that enables scientists to write concurrent applications based on a process algebra. The CSP (Communicating Sequential Processes) algebra allows the programmer to formally verify the correctness of an application, but even without formal verification CSP provides a set of design rules that, if followed, guarantees a correct parallel model. The applications built using Python and PyCSP are not intended to perform with perfect speedups, but any speedup is far better than what scientific users would see with sequential code.

This paper shows three scientific applications that use PyCSP and benchmarks them against a corresponding C implementation. The results show that the overhead introduced from a mixed Python-C-PyCSP is a small price to pay compared with the benefits from gaining a flexible, concurrent and compositional application. Notice that to reduce the overhead from Python and PyCSP we require a somewhat coarse-grained concurrency and must write compute intensive parts in a compilable language. Python is a dynamic language and for scientific applications it is gaining acceptance, since the ease of programming makes it ideal for high productivity while it is shown that high performance can be achieved.

2. Related work

We want to combine parallel programming with enabling the scientific users to program by stitching ready-made components together or using a high-level language for higher productivity.

The following graphical workflow tools enable scientists to construct advanced applications using Java: Knime [1,2] and Taverna [3,4]. They have a large amount of ready-made components and are able to utilize multi-core systems as well as interface with execution in a grid system. LabView[5] and Simulink[6] are two commercially available products which offer a similar level of functionality. For parallel computation, they all use a task-based approach with a single master handling the control-flow and dependencies between processes. A task-based system does not scale sufficiently with the number of processes and workers. Using a task-based system it is difficult to implement applications that are robust towards concurrent events and to handle such, special components must be implemented for each type of events.

CSP can handle concurrent events. A scalable implementation of Communicating Sequential Processes allows processes to execute and communicate in parallel, if no dependencies disallow it. Programming for non-deterministic behavior is simple with CSP, since it is inherent in the model. See Section 4.4 for an example of non-deterministic behavior with PyCSP. CSP provides a formal and verifiable approach to controlling concurrency, fits directly into scientific workflows, and maps into these graphical tools that present scientific workflows.

The P-GRADE [5] project prove that CSP style programming can be used for creating distributed scientific applications. The project combines low-level programming and a graphical workflow tool with a CSP-like communication model. Applications created with this tool are compiled to MPI or PVM and can deal with grid systems. A later project named P-GRADE portal [6] presents a platform for handling dependencies between grid jobs and enables a single application to use multiple grid middleware architectures. The P-GRADE portal has left the CSP style programming and is now using directed acyclic graphs for workflow programming. The first P-GRADE system had to be implemented in C or C++, which are high performance languages, but also difficult to learn. Python which is a high-level programming language with a dynamic type system has been designed to be easy to learn for newcomers and flexible enough for the expert. The large amount of scientific libraries available for Python also shows that Python is often used for producing scientific applications.

The following programming languages have support for CSP: Java (JCSP[7] and CTJ[8]), Haskel (CHP[9]), C++ (C++CSP2[10]), Python (PyCSP[11]), Google Go[12] and Occam-π[13]. Other programming languages with message-passing functionality similar to CSP, but not compatible with the formal CSP defined by Hoare [7] are: Erlang[14] and Microsoft Axum[15]. Scientific applications have been developed in all of these languages. The research group COSMOS is investigating using process-oriented programming for agent-based simulation [8]. They focus on using the concurrent properties from CSP to enhance the simulation, where we use CSP as a coarse-grained method to model and execute the scientific workflow.

Parallel programming in Python is possible through the use of many different libraries. There are fork-based systems similar to Python's built-in map function. Classic systems like the built-in threading or multi-processing library provide basic support with synchronization structures such as locks, monitors and queues. Common for those libraries are the use of shared data-structures, which may hide dependencies between processes. Such dependencies easily cause data-hazards, while use of locks can result in dead-locks or race-conditions. Shared memory, locks and monitors are difficult constructs to get right in parallel programming.

Most high-level approaches to parallel programming have been implemented for Python. There is task based systems administering a pool of workers. Data-parallel systems parallelize vector or matrix operations. Message-passing libraries similar to MPI enable fast and advanced communication between processes.

Creating an application based on a scientific workflow can be a reasonably complex project, if the parallel library does not support it. None of the above libraries are good at handling compositional structures, while such a thing is essential in a process-oriented approach with message-passing. A process-oriented approach for Python is Candygram[16] (implementation of Erlang concurrency primitives), but since Erlang uses asynchronous communication, it is not compatible with the CSP algebra. PyCSP is based on CSP and can thus benefit from the CSP algebra.

[3] Intel Parallel Studio: http://software.intel.com/en-us/intel-parallel-studio-home/.
[4] Microsoft Parallel: http://msdn.microsoft.com/en-us/concurrency/default.aspx.
[5] LabView: http://www.ni.com/labview/.
[6] Simulink: http://www.mathworks.com/products/simulink/.
[7] JCSP: http://www.cs.kent.ac.uk/projects/ofa/jcsp/.
[8] CTJ: http://www.ce.utwente.nl/javapp/.
[9] CHP: http://www.cs.kent.ac.uk/projects/ofa/chp/.
[10] C++CSP2: http://www.cs.kent.ac.uk/projects/ofa/c++csp/.
[11] PyCSP: http://code.google.com/p/pycsp/.
[12] Google Go: http://golang.org/.
[13] Occam-π: http://www.cs.kent.ac.uk/projects/ofa/kroc/.
[14] Erlang: http://www.erlang.org/.
[15] Microsoft Axum: http://msdn.microsoft.com/en-us/devlabs/dd795202.
[16] Candygram: http://candygram.sourceforge.net/.

Please cite this article in press as: R.M. Friborg, B. Vinter, Rapid development of scalable scientific software using a process oriented approach, J. Comput. Sci. (2011), doi:10.1016/j.jocs.2011.02.001

The Papy [9] library is specifically designed to handle a flow-based programming paradigm (limited to directed acyclic graphs) in Python and enables the construction and deployment of distributed workflows. CSP is not only limited to directed acyclic graphs.

3. Communicating Sequential Processes

The Communicating Sequential Processes algebra, CSP [7], was invented more than 25 years ago and while it was highly popular and thoroughly investigated in its first years, interest dropped in the late 1980s because the algebra appeared to be a solution in search of a problem, namely modelling massively concurrent processes and providing tools to solve many of the common problems associated with writing parallel and concurrent applications. In 1980 C.A.R. Hoare, the father of CSP, wrote the following:

> *"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies"*
>
> *The Emperor's Old Clothes – C.A.R. Hoare*

CSP provides many attractive features with respect to the next generation processors; it is a formal algebra with automated tools to help prove correctness, it works with entirely isolated process spaces, thus the inherent coherence problem is eliminated by design, it results in a design where dependencies are explicit and it lends itself to being modelled through both programming languages and graphical design tools.

Another attractive feature of CSP, which has so far not been investigated, is the fact that it should lend itself towards modelling heterogeneous systems. This is important for the next generation processors since heterogeneity has already been introduced: the coupling between the classic multi-core processor and co-processors in the form of GPUs, FPGAs or Cell-SPEs. Several manufactures are currently looking into producing chips with everything on a single silicon die.

4. PyCSP

PyCSP is a library to support CSP style programming in Python. It was first introduced in 2007 [10] and revisited in 2009 [11,12]. As in all process-oriented systems, the central abstractions offered by PyCSP are the process and the channel. In PyCSP, processes are available in three variants: as user-level threads, kernel-level threads or OS processes. A PyCSP program is composed of multiple processes communicating by sending messages over channels. All communication in PyCSP is synchronous, thus a channel operation is always blocking until a single matching pair consisting of a reader and writer is found. An example is shown in Listing 1. When executed, a trace of the communication can be recorded and later visualized as shown in Fig. 1.

```
from pycsp import *

@process
def source(chan_out):
    for i in range(10):
        chan_out("Hello (%d)\n" % (i))

    # Signals that the channel
    # has one less writer
    retire(chan_out)

@process
def sink(chan_in):
    # The loop will terminate on the
    # signal that announces that all
    # writers have retired
    while True:
        sys.stdout.write(chan_in())

chan = Channel()

# Run in parallel
Parallel(
    # Five source processes
    5 * source(chan.writer()),
    # Five sink processes
    5 * sink(chan.reader())
)
```

Listing 1. A simple PyCSP example demonstrating the concurrent nature in CSP upholding to unbounded non-determinism and protected against race conditions during termination.

4.1. Processes

A process in PyCSP is an isolated unit of execution, not physically isolated as an operating system process, but by design should

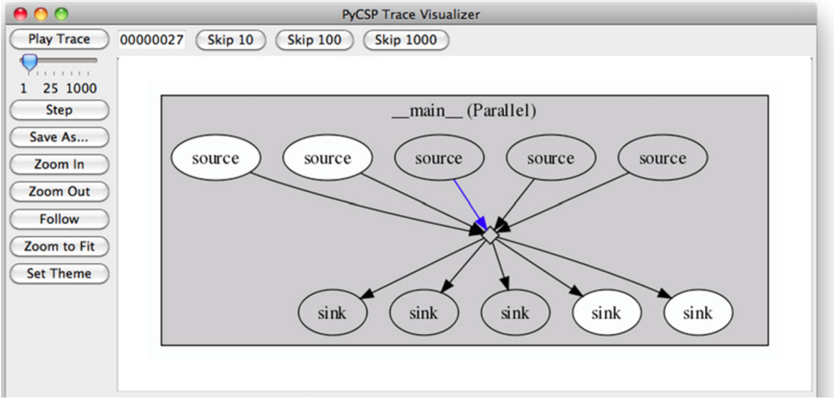


Fig. 1. Screenshot of the PyCSP trace visualizer performing a playback of a recorded trace. The processes with gray background are actively involved in communication.

Please cite this article in press as: R.M. Friborg, B. Vinter, Rapid development of scalable scientific software using a process oriented approach, J. Comput. Sci. (2011), doi:10.1016/j.jocs.2011.02.001

```
@process
def increment(cin, cout, inc_value=1):
    while True:
        cout(cin() + inc_value)
```

Listing 2. Example process with IO.

not share objects with other processes. A process is specified by using the `@process` decorator as depicted in Listing 2. Applying this decorator to the increment function, creates an increment factory, which produces increment process instances. Starting the process is covered in Section 4.3.

4.2. Networks of processes

The only allowed methods to communicate between processes are to either pass arguments when creating processes or by sending messages over channels. All channel communications are blocking operations and are guaranteed to be sent exactly once and received by exactly one process. A channel is created using the Channel class and can have any number of writing processes and any number of reading processes.

```
a = Channel('A')
```

Processes are passed their input and output channels as parameters which can then be used to communicate with other processes. An example of this usage is shown in Listing 2.

To communicate on a channel, an application is required to get a hold of an input or output end of the channel. A reference to a channel end is acquired by using the chan.reader() or chan.writer() channel methods, which will return a channel end object. Acquiring a channel end object will also join the actual channel. This adds information to the channel, registering how many readers and writers that are connected to it. The number of readers and writers is used to automate termination explained in Section 4.5. To create an increment instance *P* and provide it with channel ends we do the following:

```
P = increment(a.reader(), b.writer())
```

Communicating on a channel end is performed by invoking `cin()` or `cout(msg)`, where `cin = a.reader()` or `cout = a.writer()`. Usually channel ends are provided as arguments to new processes as shown in the example of the increment instance *P*.

4.3. Concurrency

Creating a process will simply instantiate a copy of the process but not execute or start it in any way. A set of processes may be executed using one of the three constructs, Sequence, Parallel or Spawn. Sequence and Parallel are synchronous and will only terminate once all processes in their parameter list are terminated. Sequence executes the processes one at a time while Parallel executes all the processes concurrently. Spawn starts the set of processes in the background and then returns, one may view it as an asynchronous version of Parallel.

The code in Listing 3 completes when the counter, increment and printer processes have completed. Section 4.5 explains how to initate a termination of the `Parallel(processes)` execution.

```
a,b = Channel('A'), Channel('B')
Parallel(
    counter(a.writer()), 10),
    increment(a.reader(), b.writer()),
    printer(b.reader())
)
```

Listing 3. Initiating processes for execution.

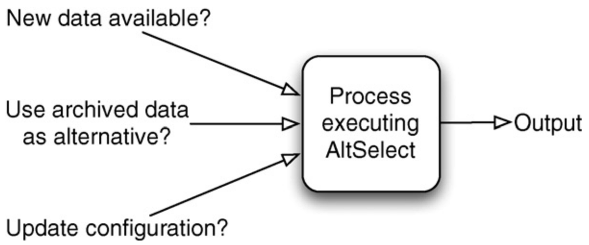

New data available?
Use archived data as alternative?
Update configuration?
Process executing AltSelect
Output

Fig. 2. Selecting from multiple concurrent events.

4.4. Nondeterminism

When an input or output channel end is invoked, the executing process is committed to the channel until the communication has completed. This complicates writing processes that need to respond to one of several communication events that may come in any order.

Using the AltSelect construct, it is possible to commit to waiting on multiple channel ends. This allows a process to wait for one of several communication events to become ready, and automatically select and execute one of them (Fig. 2). An example of AltSelect usage is shown in Listing 4.

4.5. Termination

A controlled shutdown of a CSP network (set of processes connected by channels) can be performed by using poisoning [13]. Poisoning of a network may happen in one of the two ways, either as an explicit poison which will propagate the entire network instantly and cause a fast termination, or as an incremental retirement which allows all processes to finish their current work before termination. An explicit poison is performed using the `poison(channel/channelend)` call. The less intrusive poison can be performed by using the `retire(channelend)` call. Calling retire will cause a decrement of an internal counter inside a channel. When a `retire()` call causes a channel to have 0 readers or 0 writers left, the channel is permanently retired and any access will raise an exception as described below.

Upon the permanent retirement or poisoning of a channel, all processes that access the channel will raise a `ChannelRe-`

```
@process
def par_in_out(cin1, cin2, cout3, cnt):
    for i in range(cnt):
        g, msg = AltSelect(
          InputGuard(cin1),
          InputGuard(cin2),
          OutputGuard(cout3, msg=i)
        )
        if g == cin1 or g == cin2:
          print 'Received', msg
```

Listing 4. AltSelect.

```
@process
def printer(cin):
    try:
        i = 0
        while True:
            print cin()
            i += 1
    except ChannelRetireException:
        print 'Printed', i, 'values'
```

Listing 5. Controlling termination.

tireException() or ChannelPoisonException() respectively. Whether this exception is caught inside the process or passed on is left to the programmer. The default behavior is that the Process class will catch the exception and then, depending on whether it is poisoned or retired, the following occurs: all channels and channel ends in the argument list of the poisoned process are poisoned, thus propagating a poison signal through all known channels. All channel ends in the argument list of the retired process are retired, thus propagating a retire signal through all known channel ends. In Listing 5 a ChannelRetireException() is caught inside a process.

Without the ability to poison or retire channels, a programmer would have to create the extra control flow necessary for controlling the shutdown of processes. Poisoning and retiring channels are compact and simple ways to shut down processes and save the programmer from the work of creating the extra control flow necessary for a controlled shutdown of a group of processes.

5. PyCSP in a scientific context

The purpose of a scientific application is usually to produce a result based on input data. The input data initiates a flow through the application which is viewed as sub-problems that result in sub-solutions until eventually a final result is returned. We use the term "scientific workflow" for the data-flow of computationally intensive scientific applications, normally run on shared-memory multi-processor hardware or in distributed network environments. An application with a scientific workflow might be anything from complex climate modelling to a simple n-body simulation. Generally, any application that performs a large number of computations in order to produce a result within a particular scientific field.

We find that CSP is ideal for reasoning about the concurrent aspects in scientific workflows. In particular this is true when the target environment is parallel architecture. The compositional structure of a CSP network enables application developers to reuse networks of components, i.e. libraries that are represented as top-level components themselves. Since CSP guarantees isolation we can also guarantee that library functions produce no side-effects within a process. This is important since it allows a programmer to produce a function for parallel execution which in itself includes parallel components. Such hierarchical parallelism is often not correct, or at least not recommended, in other library systems.

It is the compositional structure of CSP that makes it ideal for heterogeneous and emerging architectures. Scientific users can focus their performance improvements on processes, that may benefit from executing on specific architectures and leave everything else as is. This together with Python's broad support for different architectures provides a very flexible library, where processes can be run in a grid system or on dedicated clusters.

```
from pycsp.net import *
from pycsp.net.grid import *

# Declaring a grid process
# with the vgrid (VO) setting
@grid_process(vgrid='DIKU')
def source(chan_out):
  for i in range(10):
    chan_out("Hello (%d)\n" % (i))
  retire(chan_out)

GridInit()

# Start PyCSP channel server
server.start(host=public_ip)

chan = Channel()
Spawn(100 * source(chan.writer()))

# Read
cin = chan.reader()
try:
  while True:
    cin()
except ChannelRetireException:
  print "Received 1000 HelloWorld."
```

Listing 6. A PyCSP application communicating with 100 processes in a grid system.

5.1. Execute processes in a grid

The support for grid architectures is not fully transparent, rather a special grid process decorator must be used instead of the standard process decorator. To enable sending a PyCSP to a grid system for execution, we add this decorator. An example with grid processes is shown in Listing 6.

The supported grid system at this time is the Minimum intrusion Grid [14] (MiG), but support for other grid middleware systems can easily be added. The requirement for PyCSP to work on a grid middleware is a command-line interface for performing file operations on a grid user account, submitting jobs and querying of job states. Most grid middlewares offer such a command-line interface.

The PyCSP channel communication with remote processes is running on top of the Pyro[17] Python module, which uses sockets for communication. To enable a more broad support, we have implemented an alternative carrier for communication with grid in mind, to enable grid resources with restricted access to perform channel communications through HTTPS to a single hosted channel server via XML-RPC. This feature makes PyCSP feasible for most grid resources.

6. Scientific applications

To demonstrate the flexibility and diversity of target architectures that PyCSP support, this section describes three different applications using PyCSP to enable concurrent execution. The chosen problems for this section are: stochastic minimum search [15], *k*-nearest-neighbour search [16] and McStas [17]. They show three different usages of PyCSP, a master–worker design, a ring design

[17] Pyro: http://pyro.sourceforge.net/.

Please cite this article in press as: R.M. Friborg, B. Vinter, Rapid development of scalable scientific software using a process oriented approach, J. Comput. Sci. (2011), doi:10.1016/j.jocs.2011.02.001

and a network that combines master–worker with a set of utility processes.

All PyCSP solutions to the mentioned problems are benchmarked and compared with an optimal solution. For stochastic minimum search and *k*NN this means that we compare the speedup with a sequential solution written in C, thus the speedup from the PyCSP solutions will include the overhead of PyCSP, Python and the ctypes Python module for switching between Python and C. The approach to writing the compute intensive parts in C or another compilable language as Fortran is described in [18] and believed to be within the capabilities of non computer scientists. Programming skills for sequential C and Fortran code have been taught to many non computer scientists. Some scientific problems will have libraries that can perform the computational tasks, thus reducing the use of a compilable language.

McStas is an entirely different challenge and comes with a Perl wrapper for a large code-base that contains C code and supports MPI. For McStas we replace the Perl wrapper with our own PyCSP wrapper and use channel communications instead of MPI. The PyCSP solution for McStas is compared with the packaged solution using MPI for inter-process communication.

The applications are benchmarked on three different systems: a multi-core system with two Intel Xeon E5310 (8 cores total), a cluster system with eight Intel Core 2 Quad Q9400 (32 cores total) interconnected with one gigabit ethernet and an Intel Core 2 Duo 2.4 Ghz (2 cores total) with workers running in a grid system on various systems. Stochastic minimum search and *k*NN are benchmarked on the eight-core system and the cluster system. McStas is benchmarked on the eight-core system and on the dual-core system with workers running in a grid system.

Every benchmark was executed three times and then the average value was used for computing the speedup plots. The sequential C reference benchmark used for computing the speedup plots was executed on all architectures, making the speedup plots comparable.

6.1. Stochastic minimum search

The PyCSP network for stochastic minimum search [15] is given a function as input and uses a Monte Carlo approach to produce a suggestion for a global minimum value of that function within a user specified window. The Monte Carlo algorithm is run in parallel in a set of processes. Each process runs independently of the others and tests internally for a new minimum local to this process. Whenever a new local minimum is found, it is sent to a master. The master feeds a filter that decides when a result is a new best global minimum and decides when to terminate the remaining network. A subset of the code required to create the network in Fig. 3 is shown in Listing 7 to demonstrate what lines are necessary to create a parallel stochastic minimum search.

```
@process
def gmin(chin,chout, loops):
  fname = chin()
  res = min(
   [compute_in_C() for j in xrange(loops)]
  )
  chout(res)
  retire(chin, chout)

@process
def master(filter, workers_o,
           workers_i, n_workers, f):
  for i in range(n_workers): workers_o(f)
  while True: filter(workers_i())

to_worker=Channel();
from_worker=Channel()

Parallel(
  master(<filter channel>,
         to_worker.writer(),
         from_worker.reader(),
         nprocs, <target function> ),
  nprocs * gmin(to_worker.reader(),
                from_worker.writer(),
                loops))
```

Listing 7. Source code for the stochastic minimum search application where the userout and filter process have been omitted. Fig. 3 shows a visualized trace of this application.

The results from running the application on a multi-core and a cluster system are presented in Fig. 4. The extra overhead introduced by Python and PyCSP is the reason we only get a speedup of 0.6 for the multi-core execution with 1 worker process. This is caused by the application that continuously sends new local results, which add channel communication and thus requires the processes to switch between Python and C. The cluster execution adds an extra overhead for network communication, thus the speedup is 0.5 with 1 worker process. For 8 worker processes, the speedup compared to a sequential C solution is 5.0 for an 8-core host. On an 8-node cluster with 16 workers we get a speedup of 9.6. The speedup improves as searches run for longer periods, since the

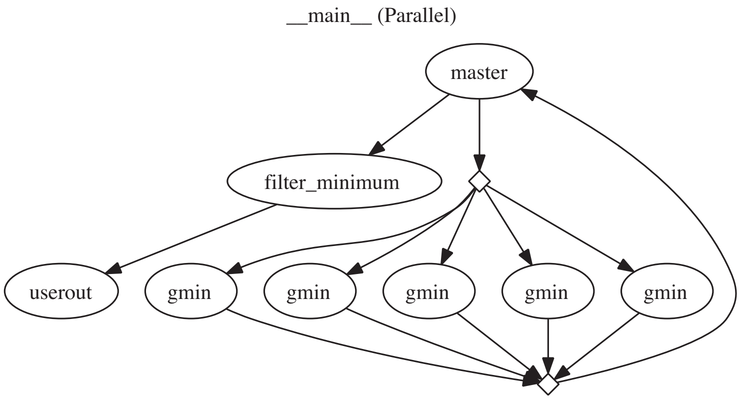


Fig. 3. An extracted snapshot of the stochastic minimum search CSP network with five processes executing the Monte Carlo algorithm.

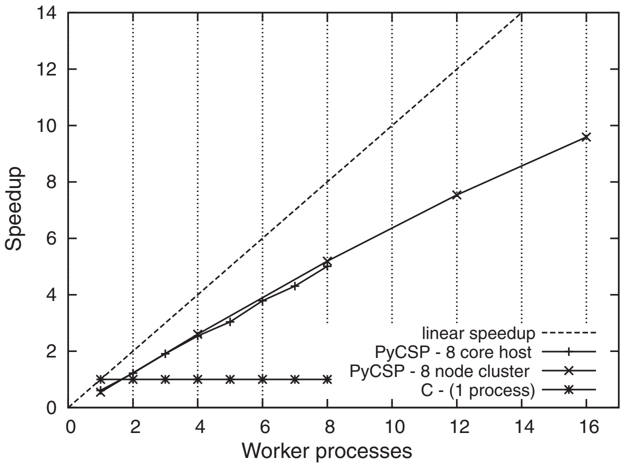


Fig. 4. Speedup of stochastic minimum search. The sequential C implementation executed 500,000 approximations in 38.4 s (avg. of three) on the 8 core host and 5,000,000 approximations in 221.2 s (avg. of three) on the 8 node cluster.

amount of channel communication goes down, as the occurrence of new minimums decrease.

6.2. *k* nearest neighbour search

The *k* nearest neighbour search problem [16] consists of finding the *k* nearest neighbours of each target point in a set of *N* targets with *D* dimensions. This is used in machine learning for finding the *k* nearest neighbours to a sample among a large set of positive and negative targets. If the *k* nearest neighbours to a sample is primarily positive, then the algorithm would give the result that a sample is positive together with a likelihood value. Any measurement of the distance between each target can be used, but for this benchmark we have used the Euclidian distance in the space with *D* dimensions. To compute the distances the brute force approach is used, which has the complexity of $O(N^2D)$ but is easy to parallelize. To compute in parallel the targets are split into *T* sets and divided among *T* worker processes connected in a ring. The CSP network in Fig. 5 shows the workers connected in a ring, where every worker is given a local set which it will pass around the ring. In each pass the worker executes a *k*NN algorithm on the local set and the received set, until finally all results are collected and joined to the end result. Listing 8 shows the code for creating the ring network and how to compute the result for each worker. The ring approach is too fine-grained for a grid architecture but is well suited for a closely connected parallel architecture such as cluster-computers.

In Fig. 6 we have plotted the speedup for computing the 5 nearest neighbours in 72 dimensions in a set of 10,000 targets (8 core host) and a set of 60,000 targets (8 node cluster). For the 8 core host we get a speedup of 9.0 with 8 workers, which is caused by the better cache usage when the local set-size for each worker gets smaller. The poorer performance for the 8 node cluster can be explained by the necessary transfers of the actual arrays, and the associated serialization, but a reasonable speedup of 12.4 is still achieved for 16 workers.

```
@process
def build_ring(proc, N, B, pargs):
  channels=Channel(buffer=B) * N
  processes=[]
  for i in range(N):
    processes.append(proc(*pargs,
      ring_in=channels[i-1].reader(),
      ring_out=channels[i].writer(),
      ring_size=N, ring_id=i))
  Parallel(processes)

@process
def kNN(job_ch, result_ch, D, k,
        ring_in=None, ring_out=None,
        ring_size=0, ring_id=0):

  # channel -> channel end
  job=job_ch.reader()
  result=result_ch.writer()

  work=data=job()

  best = numpy.zeros((len(data), k))
  for i in range(ring_size):
    # Invoke a C impl.
    kNN_in_C(data, work, best, D, k)
    ring_out(work)
    work=ring_in()
  result(best)
  retire(result)

job_ch=Channel(); result_ch=Channel()
Parallel(
  producer(<N*D input array>, 4,
           job_ch.writer()),
  build_ring(kNN, size=4, buffer=1,
             (job_ch, result_ch, D, k)),
  consumer(result_ch.reader()))
```

Listing 8. Source code for the *k* nearest neighbour search application where the producer and consumer process have been omitted. Fig. 5 shows a visualized trace of this application.

6.3. Neutron scattering simulation

Neutron based imaging is a powerful tool in several sciences, including solid state physics and biology, where neutron imaging

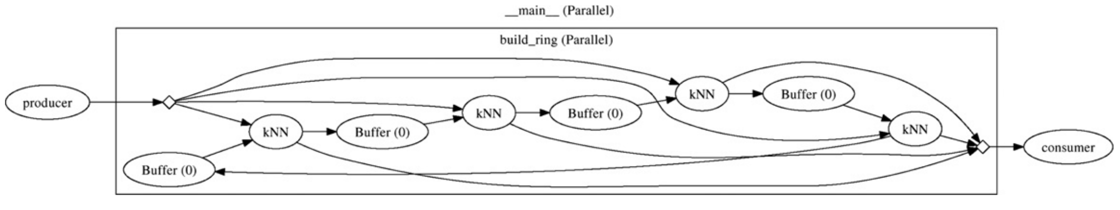


Fig. 5. Nearest neighbour search with four worker (*k*NN) processes connected in a ring.

Please cite this article in press as: R.M. Friborg, B. Vinter, Rapid development of scalable scientific software using a process oriented approach, J. Comput. Sci. (2011), doi:10.1016/j.jocs.2011.02.001

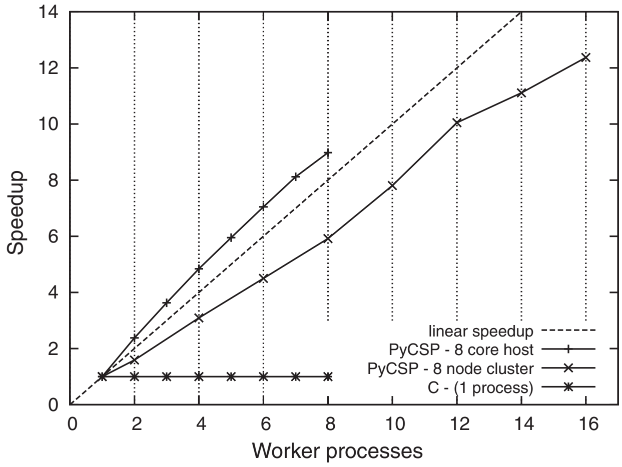


Fig. 6. Speedup of nearest neighbour search (*k*NN). The sequential C implementation computed the result for 10,000 bodies in 23.6 s (avg. of three) on the 8 core host and for 60,000 bodies in 442.9 s (avg. of three) on the 8 node cluster.

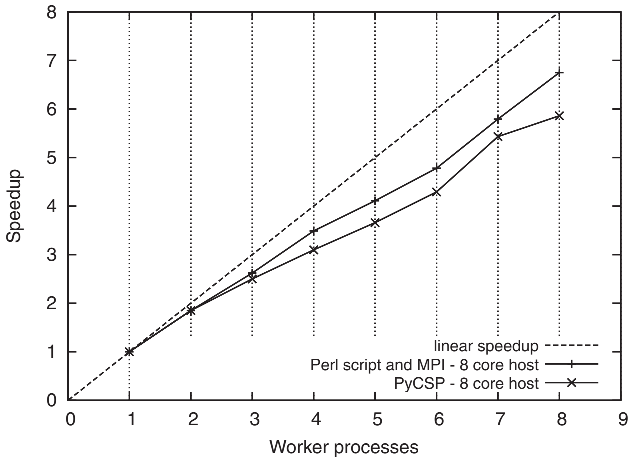


Fig. 8. Speedup of neutron scattering simulation (McStas). The sequential execution simulated 50,000,000 rays in 61.2 s (avg. of three) on the 8 core host.

is used to produce high resolution non-intrusive images of samples. However, neutron based imaging is not as simple as using an optical microscope or an X-ray imaging and before a neutron image is produced the imaging process must first be simulated to tune several parameters. This simulation is hugely time-consuming and the quality of the simulation is often dictated by the available time for simulation. The de-facto tool for such simulations is McStas [17], performing Monte Carlo simulations of neutron instruments. McStas comes with a complicated Perl script that enables parallel execution with MPI. We replace this Perl script with PyCSP, to use a process-oriented model, that executes on any number of grid resources through many levels of parallism, and finally we merge the results and save them for the user. The directed graph (Fig. 7) is generated from a trace of an actual execution. This is made possible by the compositional nature of CSP which is easily traced and visualized using PyCSP.

The first task is to load the description of the experiment in a domain specific language. This language is then compiled into a C source file which in turn is compiled into an executable. This two-phase compilation is quite demanding and requires a non-trivial installation of the McStas package.

To improve the productivity of the user we have enabled the possibility to move the compilation to a service resource by putting it into a separate process. This allows the user to configure the simulation without having to install McStas on the local computer. The resulting executable is passed on to the worker (simulate) processes that run the simulation numerous times over a set of parameters. The final merged result is sent back to the user for presentation.

In Fig. 8 we compare the MPI enabled Perl wrapper with PyCSP. The PyCSP speedup suffers slightly from the extra overhead of handling jobs. For the PyCSP executions the work was divided into 50 jobs.

The benefit of having a dynamic orchestration of workload becomes apparent when the executing resources differ. This is the case in grid computing, and by changing the configuration of the worker processes as described in Listing 9 we can utilize many more resources.

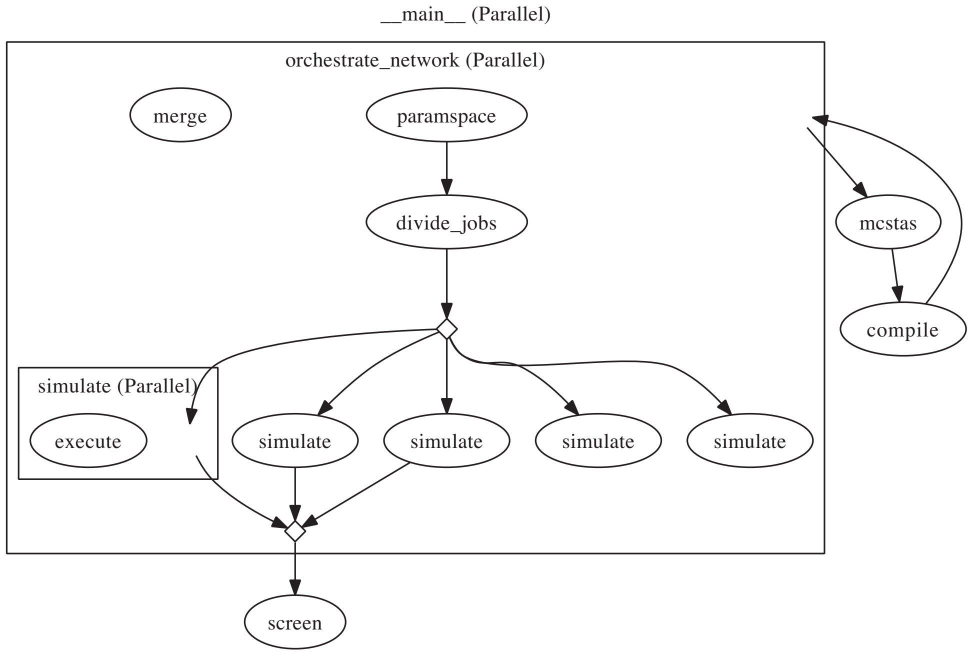


Fig. 7. Neutron scattering simulation with five worker (simulate) processes. A dynamic orchestration of workload is in place to be able to adapt to uneven resources.

```
@grid_process(vgrid='DCSC', cputime=600)
def simulate(job_in, result_out,
             exec_file):
  while True:
    ncount, params = job_in()
    cmd=tuple(['./' + exec_file,
               '--ncount=' + str(ncount),
              ] + params )
    Parallel(
      execute(cmd, retire_on_eof=False)
    )
    output=open('mcstas.sim').readlines()
    result_out(output)
```

Listing 9. Worker (simulate) process. The @grid_process decorator configures the process to behave as a grid process.

Table 1
Execution times for using grid computing. These numbers are indicative for grid executions, but will vary greatly due to large variations in resources, size of job queue and grid overhead.

System	Simulated rays	Time (s)
Dual core laptop with 2 workers	2.5×10^9	5928
Grid with up to 64 workers	2.5×10^9	411
Grid with 1 worker	1	43

When executing a PyCSP network with processes submitted to a grid system, the workload must be split dynamically to cope with inactive processes, since it is unknown when a worker process might be moved from 'queued' to 'executing' status. One execution might experience that only half of the workers are executing the entire simulation while another execution will have all the workers in 'executing' state early on.

We have performed two large neutron simulations where one was performed on a dual core laptop and the other was run from the same laptop but with up to 64 simulate processes running in a grid system. From the execution times in Table 1 we demonstrate that the simulation finished in under 7 min instead of 1.5 h.

The results produced by the three applications are not expected to scale linearly, but even a decent speedup is a good result, since the cost for producing the PyCSP application is limited and the scientist is still involved directly in the programming. We expect that all three applications scale to at least 64 workers for larger problem sizes.

7. Conclusion

If an application is organized into concurrent sequential Python processes with a reasonable work to communication ratio, we have shown that an application can scale on both multi-core and cluster systems. The degree of scaling requires a certain work to communication ratio, to hide the overhead of ≈0.1 ms per local communication and ≈2 ms per remote communication.

The flexibility of PyCSP has been demonstrated through executions in four different environments; single-core, multi-core, cluster and grid. We argue that PyCSP is a candidate to handling heterogeneous architectures in a single application, since PyCSP is portable and can run on most architectures and operating systems, e.g. the Nvidia Tegra running Android OS.

The close mapping between the graphical representation of CSP programs and the PyCSP source code makes it easy to compare design documents and implementations, helping scientific users manage the complexity that is often introduced by parallel architectures. The neutron scattering simulation example shows how PyCSP can be used to structure the execution of binaries into a concurrent Python application, which can be traced and visualized for a better understanding.

To organize a scientific application into processes, we suggest using the scientific workflow as a starting step and then evolve a CSP network from there. It is at all times possible to generate a trace of a PyCSP execution and subsequently replay this trace with a visual representation, as illustrated in this paper. This may be used to view the behavior of the CSP network and thus help to debug.

The compositional nature of CSP enables any researcher to take any process of a CSP network and optimize it in isolation, since all processes are isolated and have no side effects. This has huge benefits on cost and time since dedicated programmers can be hired to optimize the bottle necks in a workflow of any size. The scientific user still has control of the rest of the application since it remains unchanged from the original design. This results in an improved situation, where the scientific user is still in control and able to make real changes to the high performance application, even if it is running on multiple architectures and clusters at remote locations.

We believe that we present a strong tool for the scientific user to be able to produce software for the varying parallel hardware available today, and more importantly will run on new upcoming hardware.

References

[1] M.R. Berthold, N. Cebron, F. Dill, T.R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, B. Wiswedel, Knime: the konstanz information miner, in: Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007), Springer, 2008, pp. 319–326.

[2] C. Sieb, T. Meinl, M. Berthold, Parallel and distributed data pipelining with knime, Mediterranean Journal of Computers and Networks 3 (2) (2007) 43–51.

[3] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M.R. Pocock, P. Li, T. Oinn, Taverna: a tool for building and running workflows of services, Nucleic Acids Research 34 (2006) 729–732.

[4] S.M. Hajo, N. Krabbenhft, D. Bayer, Integrating ARC grid middleware with Taverna workflows, Bioinformatics 24 (9) (2008) 1221–1222.

[5] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, G. Gombás, P-grade: a grid programming environment, Journal of Grid Computing 1 (2) (2003) 171–197.

[6] P. Kacsuk, T. Kiss, G. Sipos, Solving the grid interoperability problem by p-grade portal at workflow level, Future Generation Computer Systems 24 (7) (2008) 744–751.

[7] C. Hoare, Communicating sequential processes, Communications of the ACM (1978) 260.

[8] F.A.C. Polack, P.S. Andrews, T. Ghetiu, M. Read, S. Stepney, J. Timmis, A.T. Sampson, Reflections on the simulation of complex systems for science, in: ICECCS 2010: Fifteenth IEEE International Conference on Engineering of Complex Computer Systems, IEEE Press, 2010, pp. 276–285.

[9] M. Cieslik, C. Mura, Papy: parallel and distributed data-processing pipelines in Python, in: Proceedings of the 8th Python in Science Conference (SciPy 2009), 2009, pp. 41–47.

[10] J.M. Bjørndalen, B. Vinter, O. Anshus, PyCSP – Communicating Sequential Processes for Python, in: A.A. McEwan, S. Schneider, W. Ifill, P. Welch (Eds.), Communicating Process Architectures 2007, 2007, pp. 229–248.

[11] B. Vinter, J.M. Bjørndalen, R.M. Friborg, PyCSP revisited, in: CPA, 2009, pp. 263–276.

[12] R.M. Friborg, J.M. Bjørndalen, B. Vinter, Three unique implementations of processes for PyCSP, in: CPA, 2009, pp. 277–292.

[13] B.H. Sputh, A.R. Allen, JCSP-Poison: safe termination of CSP process networks, in: CPA, Communicating Process Architectures, 2005, pp. 71–107.

[14] B. Vinter, The architecture of the minimum intrusion grid (MiG), in: Communicating Process Architectures 2005, 2005, pp. 189–201.

[15] H. Hoos, T. Sttzle, Stochastic Local Search: Foundations & Applications, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[16] B.V. Dasarathy, Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques, IEEE Computer Society Press, 1990.

[17] A.T.K. Lefmann, K. Nielsena, B. Lake, Mcstas 1.1: a tool for building neutron monte carlo simulations, Physica B: Condensed Matter (2000) 152–153.

[18] H.P. Langtangen, Python Scripting for Computational Science, Springer, 2008.

Please cite this article in press as: R.M. Friborg, B. Vinter, Rapid development of scalable scientific software using a process oriented approach, J. Comput. Sci. (2011), doi:10.1016/j.jocs.2011.02.001

Rune M. Friborg is a PhD student at the Department of Computer Science, University of Copenhagen. His current PhD project includes elements from scientific workflow modelling, concurrent systems programming, formal verification and high performance computing. Research interests are Communicating Sequential Processes (CSP), programming of next generation processors, heterogeneous systems and high performance computing.

Brian Vinter is professor at the Department of Computer Science, University of Copenhagen. His primary research areas are grid computing, supercomputing and multicore architectures. He has done research in the field of High Performance Computer since 1994. Current research includes methods for utilization of parallelism in scientific applications with special focus on exploiting heterogeneous systems.

Please cite this article in press as: R.M. Friborg, B. Vinter, Rapid development of scalable scientific software using a process oriented approach, J. Comput. Sci. (2011), doi:10.1016/j.jocs.2011.02.001

A.8 Verification of a Dynamic Channel Model using the SPIN Model Checker

Communicating Process Architectures 2011, WoTUG-33, Proceedings of the 33rd WoTUG Technical Meeting, University of Limerick, Ireland, June 19-22, 2011
ISBN: 978-1-60750-773-4, Concurrent Systems Engineering 68, IOS Press, pp. 35 – 54

Rune Møllegaard Friborg, Brian Vinter: Verification of a Dynamic Channel Model using the SPIN Model Checker

Verification of a Dynamic Channel Model using the SPIN Model Checker

Rune Møllegaard FRIBORG [a,1] and Brian VINTER [b]
[a] *eScience Center, University of Copenhagen*
[b] *Niels Bohr Institute, University of Copenhagen*

Abstract. This paper presents the central elements of a new dynamic channel leading towards a flexible CSP design suited for high-level languages. This channel is separated into three models: a shared-memory channel, a distributed channel and a dynamic synchronisation layer. The models are described such that they may function as a basis for implementing a CSP library, though many of the common features known in available CSP libraries have been excluded from the models. The SPIN model checker has been used to check for the presence of deadlocks, livelocks, starvation, race conditions and correct channel communication behaviour. The three models are separately verified for a variety of different process configurations. This verification is performed automatically by doing an exhaustive verification of all possible transitions using SPIN. The joint result of the models is a single dynamic channel type which supports both local and distributed any-to-any communication. This model has not been verified and the large state-space may make it unsuited for exhaustive verification using a model checker. An implementation of the dynamic channel will be able to change the internal synchronisation mechanisms on-the-fly, depending on the number of channel-ends connected or their location.

Keywords. CSP, PyCSP, Distributed Computing, Promela, SPIN.

Introduction

Most middleware designers experience situations where they need to choose between generality and performance. To most experienced programmers this dilemma is natural since high performance implementations are typically based on assumptions that from the usage point translates into limitations. The PyCSP project has since the beginning had a strict focus on generality, attempting to present the programmer with only one channel type and one process type. The single channel type has succeeded while the single process type has been attempted through individual PyCSP packages with separate process types. These process types are: greenlets (co-routines), threads and processes, as seen from the operating system view. The reason for the three different packages were to enable PyCSP applications to have up to 100,000 CSP processes (co-routines), posix threads for cross-platform support and OS processes executing in parallel while not being limited by the CPython Global Interpreter Lock [1].

We would like to reach the point of only one process type and one channel type, but for this to be possible we need a channel type which preserves the qualities of the previous channel type, i.e. be of the kind any-to-any and support external choice in both directions as well as offer both channel poisoning and retirement like the existing PyCSP channels. The new channel must support three possible process locations: within the same PyCSP process,

[1] Corresponding Author: *Rune Møllegaard Friborg, eScience Center, University of Copenhagen, DK-2100 Copenhagen, Denmark.* Tel.: +45 3532 1421; Fax: +45 3521 1401; E-mail: `runef@diku.dk`.

on a different PyCSP process within the same compute-node and, finally, a different PyCSP process on a different compute-node. It is of course trivial to make a common channel type based on the lowest common denominator, i.e. a networked channel, since this will also function within a compute-node and even within a process. However, the downside is self-evident since the overhead of deploying an algorithm based on shared-nothing mechanisms is much slower than algorithms that can employ shared state.

The solution to the proposed problem is a channel that can start out with the strongest possible requirements, i.e. exist within a single process, and then dynamically decrease the requirements as needed, while at the same time employing more complex, and more costly, algorithms.

The present work is a presentation of such a dynamic channel type. The algorithms that are employed grow quite complex to ensure maximum performance in any given scenario, thus much work has been put into verifying the different levels a channel may reach. Using the SPIN model checker, we perform an exhaustive verification of the local and distributed channel levels for a closed set of process configurations. These include any-to-any channels with input and output guards in six different combinations.

Background

PyCSP is currently a mix of four implementations providing one shared API, such that the user can swap between them manually. The four implementations do not share any code and the different channel implementations can not function as guards for a single external choice, since they are not compatible.

Listing 1. A simple PyCSP example demonstrating the concurrent nature in CSP upholding to unbounded non-determinism and protected against race conditions during termination.

```
# Select implementation
import pycsp.threads as pycsp
# (alternatives: pycsp.processes, pycsp.greenlets, pycsp.net)

@pycsp.process
def source(chan_out, N):
    for i in range(N):
        chan_out(”Hello (%d)\n” % (i))
    pycsp.retire(chan_out) # The channel has one less writer

@pycsp.process
def sink(chan_in):
    # The loop terminates on the signal that announces that all
    # writers have retired
    while True:
        sys.stdout.write(chan_in())

ch = pycsp.Channel()
pycsp.Parallel(                        # Run in parallel
    5 * source(ch.writer(), 10),     # Five source processes
    5 * sink(ch.reader())            # Five sink processes
)
```

In listing 1, we show a simple application using the current PyCSP where channels are any-to-any, synchronous and uni-directional. Processes can commit to reading or writing from single channels or they can commit to a set of distinct channels using the alt (external choice) construct. Committing to the alt construct means that exactly one of the channel operations will be accepted and all others are ignored. The alt construct allow a mix of read and write operations. The first model, which we present in section 2.1, is a model of the current channel implementation for threads in PyCSP.

In [2] we presented PyCSP for scientific users as a means of creating scalable scientific software. The users of a CSP library should not have to think about whether they might be sending a channel-end to a process that might be running in a remote location. Or how they work around an external choice on channels, that does not support it. One of the powerful characteristics of CSP is that every process is isolated, which means that we can move it anywhere and as long as the channels still work, the process will execute. Because of this, processes can easily be reused, since all the inputs and outputs are known.

The network-enabled PyCSP implementation is a prototype and uses a single channel server to handle all channel traffic. The single channel server runs the thread implementation of PyCSP internally, which creates a temporary thread for every request. The server is a serious bottle-neck for the channel communication and has limited the type of parallel applications implemented in PyCSP. In this paper we present a distributed channel model and check its correctness using the SPIN Model Checker.

Promela and the SPIN Model Checker

Promela (Process Meta Language) is a process modeling language whose purpose is to verify the logic in concurrent systems. In Promela models, processes can by created dynamically and can communicate through synchronous or asynchronous message channels. Also the processes are free to communicate through shared memory. If variables or channels are created globally, then they are available through shared memory to all Promela processes. Promela has a basic set of types for variables: bit, bool, byte, mtype (similar to enum), short and int. The models presented in this paper use shared memory when modeling internal communication, and message channels when modeling distributed communication.

In Promela, every statement is evaluated to one of two states, it is either *enabled* or *blocked*. Statements as assignments, declarations, **skip** or **break** are always enabled, while conditions evaluated to false are blocked. When a statement is blocked, the execution for that process halts until the statement becomes enabled. The following is an example of a blocked statement following an enabled statement:

```
value = 1;        /* enabled */
value == 0;       /* blocked */
```

When executing (simulating) a Promela model, the statements in concurrent processes are selected randomly to simulate a concurrent environment. To allow modeling synchronisation mechanisms, a sequence of statements can be indicated as atomic, by using the **atomic** keyword and enclosing the statements in curly brackets:

```
atomic {
   /* The first statement in an atomic region is allowed to block. */
   process_lock == 0;
   process_lock = 1;
}
```

To organise the code in Promela we use inline functions. When declaring inline functions, the parameters in the parameter list have no types. The inline functions are exclusively used as a replace-pattern, when generating the complete model with a single body for each thread. All values passed to an inline function is pass-by-reference. There is no return construct in Promela, thus values must be returned by updating variables through the parameter list.

Control flows in Promela can be defined using either **if** .. **fi** or **do**..**od** constructs. The latter executes the former repeatedly until the **break** statement is executed. Listing 2 shows two examples of the constructs. The first where **else** is taken only when there is no other enabled guards and another where an always enabled condition might never be executed.

Listing 2. Example of control flows in Promela.

```
if
:: (A == true) -> printf(”A is true, B is unknown”);
:: (B == true) -> printf(”B is true, A is unknown”);
:: else -> printf(”A and B are false”);
fi

do
:: (skip)->
      printf(”If A is always true, then this may never printed.”);
      break; /* breaks the do loop */
:: (A == true) ->
      printf(”A is true”);
      i = i + 1;
od
```

If the SPIN model checker performs an automatic verification of the above code, then it will visit every possible state until it aborts with the error: “max search depth too small”. The reason is that, there is no deterministic set of values for `i`, thus the system state space can never be completely explored. It is crucial that all control flows have a valid end-state otherwise SPIN can not verify the model.

The SPIN model checker can verify models written in Promela. In 1986, Vardi and Wolper [3] published the foundation for SPIN, an automata-theoretic approach to automatic program verification. SPIN [4] can verify a model for correctness by generating a C program that performs an exhaustive verification of the system state space. During simulation and verification SPIN checks for the absence of deadlocks, livelocks, race conditions, unspecified receptions and unexecutable code.

The model checker can also be used to show the correctness of system invariants, find non-progress execution cycles and linear time temporal constraints, though we have not used any of those features for the model checking in this paper.

1. Related Work

Various possibilities for synchronous communication can be found in most network libraries, but we focus exclusively on network-enabled communication libraries that support Hoare’s CSP algebra [5,6]. Several projects have investigated how to do CSP in a distributed environment. JCSP [7], Pony/occam-π [8] and C++CSP [9] provide network-enabled channels. Common to all three is that they use a specific naming for the channels, such that channels are reserved for one-to-one, one-to-any, network-enabled and so on. JCSP and C++CSP2 have the limitation that they can only do external choice (alt) on some channel types.

Pony enables transparent network support for occam-π. Schweigler and Sampson [8] write: “As long as the interface between components (i.e. processes) is clearly defined, the programmer should not need to distinguish whether the process on the other side of the interface is located on the same computer or on the other end of the globe”. Unfortunately the pony implementation in occam-π is difficult to use as basis for a CSP library in languages like C++, Java or Python, as it relies heavily on the internal workings of occam-π. Pony/occam-π does not currently have support for networked buffered channels. The communication overhead in Python is quite high, thus we are especially interested in fast one-to-one buffered networked channels, because they have the potential to hide the latency of the network. This would, for large parallel computations, make it possible to overlap computation with communication.

2. The Dynamic Channel

We present the basis for a dynamic channel type that combines multiple channel synchronisation mechanisms. The interface of the dynamic channel resembles a single channel type. When the channel is first created, it may be an any-to-any specialised for co-routines. The channel is then upgraded on request, depending on whether it participates in an alt and on the number of channel-ends connected. The next synchronisation level for the channel may be an optimised network-enabled one-to-one with no support for alt. Every upgrade stalls the communication on the channel momentarily while all active requests for a read or write are transformed to a higher synchronisation level. The upgrades continue, until the lowest common denominator (a network-enabled any-to-any with alt support) is reached.

This paper presents three models that are crucial parts in the dynamic channel design. These are: a local channel synchronisation model for shared memory, a distributed synchronisation model and the model for on-the-fly switching between synchronisation levels. We have excluded the following features to avoid state-explosion during automatic verification: mobility of channel ends, termination handling, buffered channels, skip / timeout guards and a discovery service for channel homes. Basically, we have simplified a larger model as much as possible and left out important parts, to focus on the synchronisation model handling the communication.

The different models are written in Promela to verify the design using the SPIN model checker. The verification phase is presented in section 3 where the three models are model-checked successfully. The full model-checked models are available at the PyCSP repository [10]. After the following overview, the models are described in detail:

- the local synchronisation model is built around the two-phase locking protocol. It provides a single CSP channel type supporting any-to-any communication with basic read / write and external choice (alt).
- the distributed synchronisation model is developed from the local model, providing the same set of constructs. The remote communication is similar to asynchronous sockets.
- the transition model enables the combination of a local (and faster) synchronisation model with more advanced distributed models. Channels are able to change synchronisation mechanisms, for example based on the location of channel ends, making it a dynamic channel.

For all models presented we do not handle operating system errors that cause threads to terminate or lose channel messages. We assume that all models are implemented on top of systems that provide reliable threads and message protocols.

2.1. Channel Synchronisation with Two-Phase Locking

The channel model presented here is similar to the PyCSP implementation (threads and processes) from 2009 [11] and will work as a verification of the method used in [11,12]. It is a single CSP channel type supporting any-to-any communication with basic read / write and external choice (alt).

In figure 1 we show an example of how the matching of channel operations comes about. Four processes are shown communicating on two channels using the presented design for negotiating read, write and external choice. Three requests have been posted to channel A and two requests to channel B. During an external choice, a request is posted on multiple channels. Process 2 has posted its request to multiple channels and has been been matched. Process 1 is waiting for a successful match. Process 3 has been matched and is going to remove its request. Process 4 is waiting for a successful match. In the future, process 1 and process 4 are going to be matched. The matching is initiated by both, but only one process marks the match as successful.

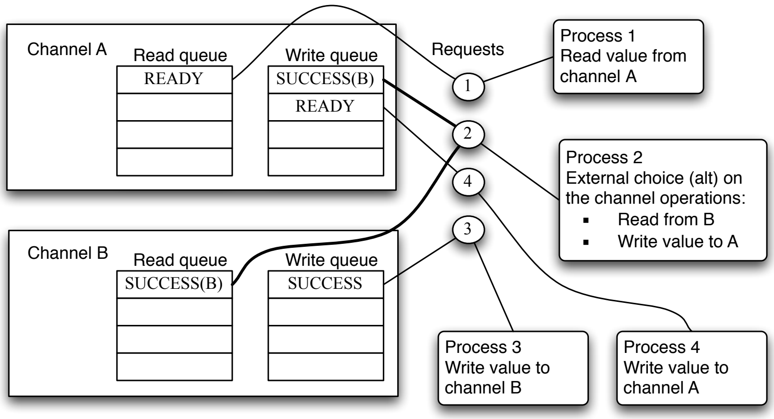


Figure 1. Example of four processes matching channel operations on two channels.

Listing 3. Simple model of a mutex lock with a condition variable. This is the minimum functionality, which can be expected from any multi-threading library.

```
typedef processtype {
        mtype state;
        bit lock;
        bit waitX;
};

processtype proc[THREADS];

inline acquire(lock_id) {
        atomic { (proc[lock_id].lock == 0); proc[lock_id].lock = 1; }
}
inline release(lock_id) {
        proc[lock_id].lock = 0;
}
inline wait(lock_id) {
        assert(proc[lock_id].lock == 1); /* lock must be acquired */
        atomic {
                release(lock_id);
                proc[lock_id].waitX = 0;     /* reset wait condition */
        }
        (proc[lock_id].waitX == 1); /* wait */
        acquire(lock_id);
}
inline notify(lock_id) {
        assert(proc[lock_id].lock == 1); /* lock must be acquired */
        proc[lock_id].waitX = 1;        /* wake up waiting process */
}
```

We use the two-phase locking protocol for channel synchronisation. When two processes are requesting to communicate on a channel, we accept the communication by first acquiring the two locks, then checking the state of the two requests and if successful, updating and finally the two locks are released. This method requires many lock requests resulting in a large overhead, but it has the advantage that it never has to roll-back from trying to update a shared resource.

To perform the local synchronisation between threads, we implement the simple lock model shown in listing 3. This is straight-forward to model in Promela, as every statement in Promela must be executable and will block the executing thread until it becomes executable. The implemented lock model is restricted to single processes calling wait. If multiple processes called wait, then the second could erase a recent notify. For the models in the paper, we never have more than one waiting process on each lock.

Now that we can synchronise processes, the process state proc[id]. state can be protected on read and update. When blocked, we wait on a condition lock instead of wasting cycles using busy waiting, but the condition lock adds a little overhead. To avoid deadlocks, the process lock must be acquired before a process initiates a wait on a condition lock and before another process notifies the condition lock. The process calls wait in write (Listing 4) and is blocked until notified by offer (Listing 6). The offer function is called by the matching algorithm, which is initiated when a request is posted. To provide an overview, figure 2 shows a pseudo call graph of the model with all inline functions and the call relationship. A process can call read, write or alt to communicate on channels. These then posts the necessary requests to the involved channels and the matching algorithm calls offer for all matching pairs. Eventually a matching pair arrives at a success and the waiting process is notified.

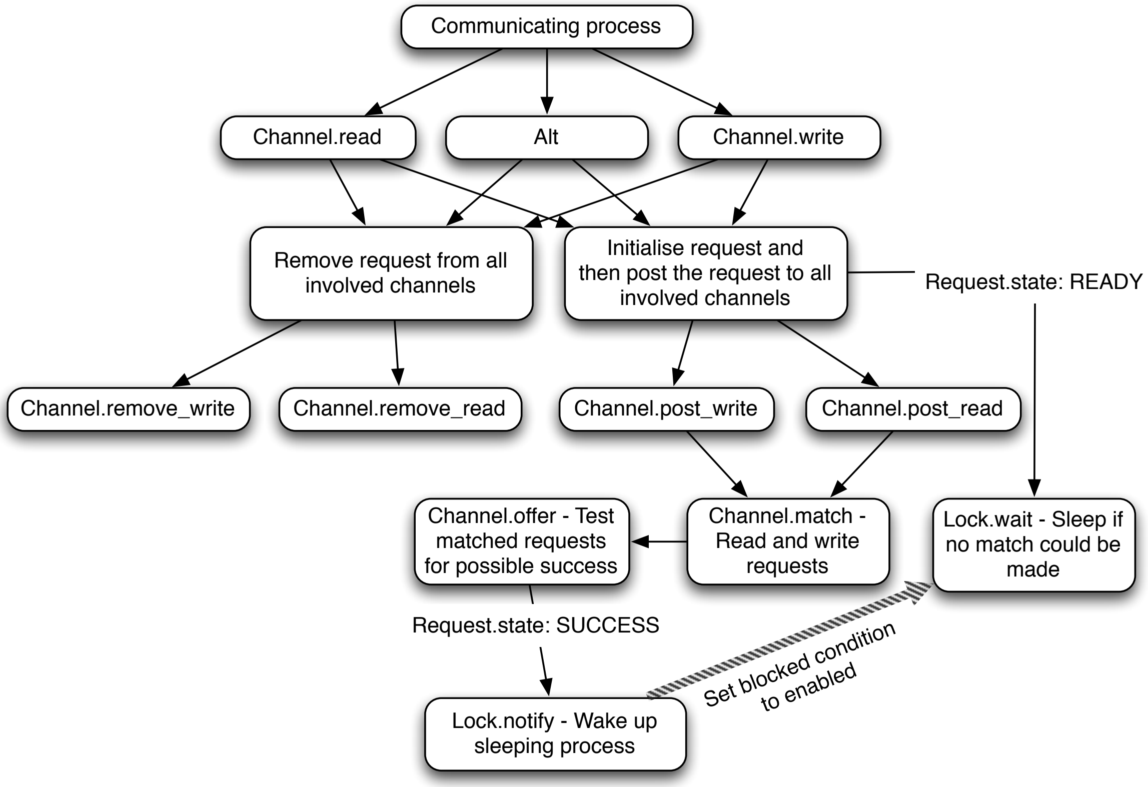

Communicating process
Channel.read
Alt
Channel.write
Remove request from all involved channels
Initialise request and then post the request to all involved channels
Request.state: READY
Channel.remove_write
Channel.remove_read
Channel.post_write
Channel.post_read
Channel.offer - Test matched requests for possible success
Channel.match - Read and write requests
Lock.wait - Sleep if no match could be made
Request.state: SUCCESS
Set blocked condition to enabled
Lock.notify - Wake up sleeping process

Figure 2. Pseudo call graph for the local channel synchronisation.

In write (Listing 4), a write request is posted to the write queue of the channel and again removed after a successful match with a write request. The corresponding functions read, post_read and remove_read are not shown since they are similar, except that remove_read returns the read value.

Listing 4. The write construct and the functions for posting and removing write requests. The process index _pid contains the Promela thread id.

```
inline write(ch_id, msg) {
  proc[_pid].state = READY;
  post_write(ch_id, msg);
  /* if no success, then wait for success */
  acquire(_pid);
  if
    :: (proc[_pid].state == READY) -> wait(_pid);
    :: else skip;
  fi;
  release(_pid);
  assert(proc[_pid].state == SUCCESS);
  remove_write(ch_id)
}
inline post_write(ch_id, msg_to_write) {
  /* acquire channel lock */
  atomic { (ch[ch_id].lock == 0) -> ch[ch_id].lock = 1; }

  <add process id, msg_to_write to ch[ch_id].wqueue>
  match(ch_id);
  ch[ch_id].lock = 0; /* release channel lock */
}
inline remove_write(ch_id) {
  /* acquire channel lock */
  atomic { (ch[ch_id].lock == 0) -> ch[ch_id].lock = 1; }

  <remove process id, msg from ch[ch_id].wqueue>
  ch[ch_id].lock = 0; /* release channel lock */
}
```

When matching read and write requests on a channel we use the two-phase locking protocol where the locks of both involved processes are acquired before the system state is changed. To handle specific cases where multiple processes have posted multiple read and write requests, a global ordering of the locks (Roscoe's deadlock rule 7 [13]) must be used to make sure they are always acquired in the same order. In this local thread system we order the locks based on their memory address. This is both quick and ensures that the ordering never changes during execution. An alternative index for a distributed system would be to generate an index as a combination of the node address and the memory address.

Listing 5. Matching pairs of read and write requests for the two-phase locking.

```
inline match(ch_id) {
        w = 0; r = 0;
        do              /* Matching all reads to all writes */
        :: (r<ch[ch_id].rlen) ->
                w = 0;
                do
                :: (w<ch[ch_id].wlen) ->
                        offer(ch_id, r, w);
                        w = w+1;
                :: else break;
                od;
                r = r+1;
        :: else break;
        od;       }
```

The two-phase locking in offer (Listing 6) is executed for every possible pair of read and write requests found by match (Listing 5). The first phase acquires locks and the second phase releases locks. Between the two phases, updates can be made. Eventually when a matching is successful, three things are updated: the condition lock of both processes is notified, the message is transferred from the writer to the reader and proc[id].state is updated.

One disadvantage of the two-phase locking is that we may have to acquire the locks of many read and write requests that are not in a ready state. The impact of this problem can easily be reduced by testing the state variable before acquiring the lock. Normally, this behaviour results in a race condition. However, the request can never change back to the ready state once it has been committed and remains posted on the channel. Because of this, the state can be tested before acquiring the lock, in order to find out whether time should be spent acquiring the lock. When the lock is acquired, the state must be checked again to ensure the request is still in the ready state. PyCSP [10] uses this approach in a similar offer method to reduce the number of acquired locks.

Listing 6. The offer function offering a possible successful match between two requests.

```
inline offer(ch_id, r, w) {
        r_pid = ch[ch_id].rqueue[r].id;
        w_pid = ch[ch_id].wqueue[w].id;
        if /* acquire locks using global ordering */
        :: (r_pid < w_pid) ->
                acquire(r_pid); acquire(w_pid);
        :: else skip ->
                acquire(w_pid); acquire(r_pid);
        fi;
        if /* Does the two processes match? */
        :: (proc[r_pid].state == READY && proc[w_pid].state == READY) ->
                proc[r_pid].state = SUCCESS;
                proc[w_pid].state = SUCCESS;

                /* Transfer message */
                ch[ch_id].rqueue[r].msg = ch[ch_id].wqueue[w].msg;
                ch[ch_id].wqueue[w].msg = NULL;
                proc[r_pid].result_ch = ch_id;
                proc[w_pid].result_ch = ch_id;

                notify(r_pid);
                notify(w_pid);
                 /* break match loop by updating w and r */
                w = LEN; r = LEN;
        :: else skip;
        fi;
        if /* release locks using reverse global ordering */
        :: (r_pid < w_pid) ->
                release(w_pid); release(r_pid);
        :: else skip ->
                release(r_pid); release(w_pid);
        fi;
}
```

The alt construct shown in listing 7 is basically the same as a read or write, except that the same process state is posted to multiple channels, thus ensuring that only one will be matched.

The alt construct should scale linearly with the number of guards. For the verification of the model we simplify alt to only accept two guards. If the model is model-checked successfully with two guards we expect an extended model to model-check successfully with more than two guards. Adding more guards to the alt construct in listing 7 is a very simple task, but it enlarges the system state-space and is unnecessary for the results presented in this paper.

Listing 7. The alt construct.

```
inline alt(ch_id1, op1, msg1, ch_id2, op2, msg2, result_chan, result) {
  proc[_pid].state = READY;
  result = NULL;
  if :: (op1 == READY) -> post_read(ch_id1);
     :: else              post_write(ch_id1, msg1);
  fi;
  if :: (op2 == READ) -> post_read(ch_id2);
     :: else              post_write(ch_id2, msg2);
  fi;
  acquire(_pid);       /* if no success, then wait for success */
  if :: (proc[_pid].state == READY) -> wait(_pid);
     :: else skip;
  fi;
  release(_pid);
  assert(proc[_pid].state == SUCCESS);
  if :: (op1 == READ) -> remove_read(ch_id1, result);
     :: else              remove_write(ch_id1);
  fi;
  if :: (op2 == READ) -> remove_read(ch_id2, result);
     :: else              remove_write(ch_id2);
  fi;
  result_chan = proc[_pid].result_ch;
}
```

2.2. *Distributed Channel Synchronisation*

The local channel synchronisation described in the previous section has a process waiting until a match has been made. The matching protocol performs a continuous two-phase locking for all pairs, thus the waiting process is constantly being tried even though it is passive. This method is not possible in a distributed model with no shared memory, instead an extra process is created to function as a remote lock, protecting updates of the posted channel requests. Similar to the local channel synchronisation, we must lock both processes in the offer function and retrieve the current process state from the process. Finally, when a match is found, both processes are notified and their process states are updated.

In figure 3, an overview of the distributed model is shown. The communicating process can call read, write or alt to communicate on channels. These then post the necessary requests to the involved channels through a Promela message channel. The channel home (channelThread) receives the request and initiates the matching algorithm to search for a successful offer amongst all matching pairs. During an offer, the channel home communicates with the lock processes (lockThread) to ensure that no other channel home conflicts. Finally, a matching pair arrives at a success and the lock process can notify the waiting process.

In listing 8 all Promela channels are created with a buffer size of 10 to model an asynchronous connection. We have chosen a buffer size of 10, as this is large enough to never get filled during verification in section 3. Every process communicating on a channel is required to have a lock process (Listing 9) associated, to handle the socket communication going in on proc_* **chan** types.

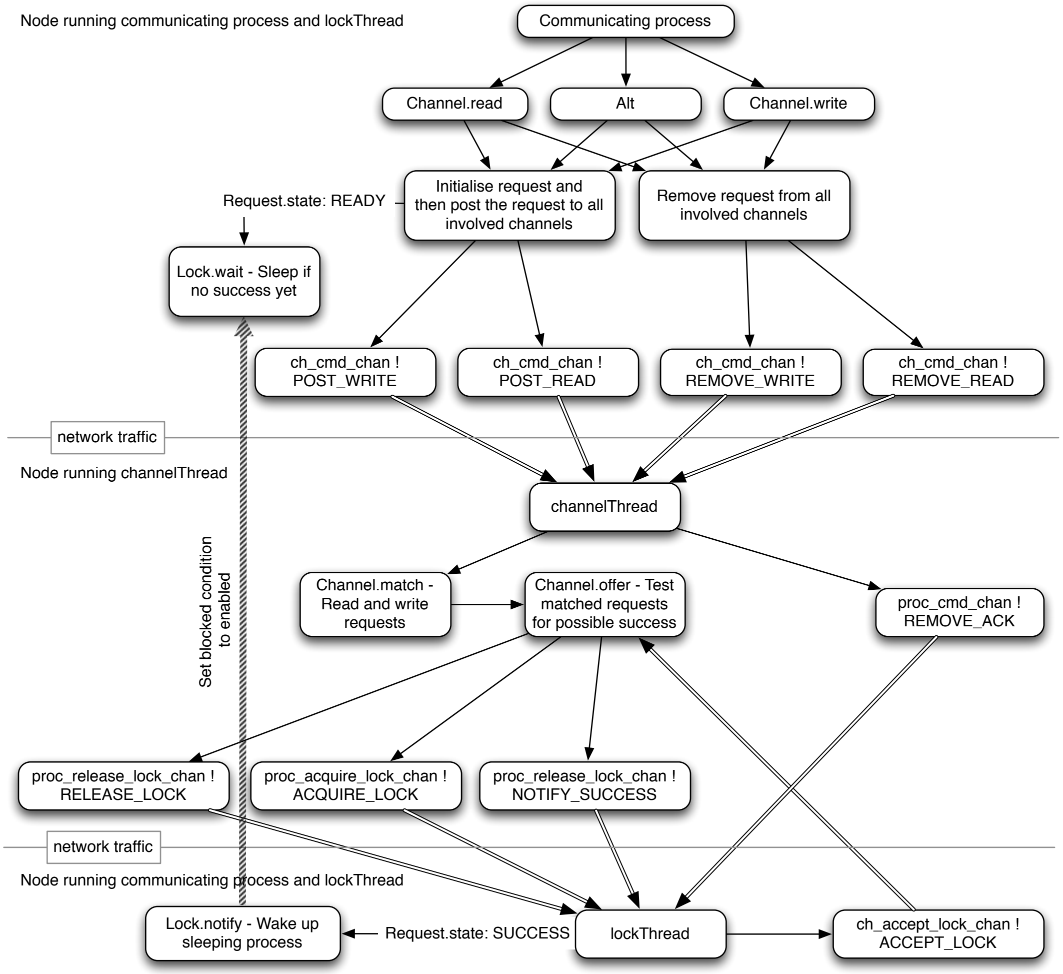

Node running communicating process and lockThread
Communicating process
Channel.read
Alt
Channel.write
Request.state: READY
Initialise request and then post the request to all involved channels
Remove request from all involved channels
Lock.wait - Sleep if no success yet
ch_cmd_chan ! POST_WRITE
ch_cmd_chan ! POST_READ
ch_cmd_chan ! REMOVE_WRITE
ch_cmd_chan ! REMOVE_READ
network traffic
Node running channelThread
channelThread
Set blocked condition to enabled
Channel.match - Read and write requests
Channel.offer - Test matched requests for possible success
proc_cmd_chan ! REMOVE_ACK
proc_release_lock_chan ! RELEASE_LOCK
proc_acquire_lock_chan ! ACQUIRE_LOCK
proc_release_lock_chan ! NOTIFY_SUCCESS
network traffic
Node running communicating process and lockThread
Lock.notify - Wake up sleeping process
Request.state: SUCCESS
lockThread
ch_accept_lock_chan ! ACCEPT_LOCK

Figure 3. Pseudo call graph for the distributed channel synchronisation.

Listing 8. Modeling asynchronous sockets.

```
/* Direction: communicating process -> channelThread */
chan ch_cmd_chan[C]=[10] of {byte, byte, byte}; /*cmd, pid, msg*/
#define POST_WRITE 1
#define POST_READ 2
#define REMOVE_WRITE 3
#define REMOVE_READ 4

/* Direction: channelThread -> communicating process */
chan proc_cmd_chan[P]=[10] of {byte,byte,byte}; /*cmd, ch, msg*/
#define REMOVE_ACK 9

/* Direction: channelThread -> lockThread */
chan proc_acquire_lock_chan[P]=[10] of {byte}; /*ch*/

/* Direction: lockThread -> channelThread */
chan ch_accept_lock_chan[C]=[10] of {byte,byte}; /*pid, proc_state*/

/* Direction: channelThread -> lockThread */
chan proc_release_lock_chan[P]=[10] of {byte,byte,byte} /*cmd, ch, msg*/
#define RELEASE_LOCK 7
#define NOTIFY_SUCCESS 8
```

The lockThread in listing 9 handles the remote locks for reading and updating the process state from the channel home thread. The two functions remote_acquire and remote_release are called from the channel home process during the offer procedure. The lockThread and the communicating process use the mutex lock operations from listing 3 for synchronisation.

Listing 9. The lock process for a communicating process.

```
proctype lockThread(byte id) {
        byte ch_id, cmd, msg;
        byte ch_id2;
        bit locked;
        do
        :: proc_acquire_lock_chan[id]?ch_id ->
                ch_accept_lock_chan[ch_id]!id, proc[id].state;
                locked = 1;
                do
                :: proc_release_lock_chan[id]?cmd, ch_id2, msg; ->
                        if
                        :: cmd == RELEASE_LOCK ->
                                assert(ch_id == ch_id2);
                                break;
                        :: cmd == NOTIFY_SUCCESS ->
                                assert(ch_id == ch_id2);
                                acquire(id);   /* mutex lock op */
                                proc[id].state = SUCCESS;
                                proc[id].result_ch = ch_id2;
                                proc[id].result_msg = msg;
                                notify(id);    /* mutex lock op */
                                release(id);   /* mutex lock op */
                        fi;
                od;
                locked = 0;
        :: proc_cmd_chan[id]?cmd, ch_id, msg ->
                if
                :: cmd == REMOVE_ACK ->
                        proc[id].waiting_removes--;
                fi;
        :: timeout ->
                assert(locked == 0);
                assert(proc[id].waiting_removes == 0);
                break;
        od;
}
inline remote_acquire(ch_id, lock_pid, get_state) {
        proc_acquire_lock_chan[lock_pid]!ch_id;
        ch_accept_lock_chan[ch_id]?id, get_state;
        assert(lock_pid == id);
}
inline remote_release(ch_id, lock_pid) {
        proc_release_lock_chan[lock_pid]!RELEASE_LOCK, ch_id, NULL;
}
```

The offer function in listing 10 performs a distributed version of the function in listing 6. In this model we exchange the message from the write request to the read request, update the process state to SUCCESS, notifies the condition lock and release the lock process, all in one transmission to the Promela channel proc_release_lock_chan . We may still have to acquire the locks of many read and write requests that are not in ready state. Acquiring the locks are now more expensive than for the local channel model and it would happen more often, due to the latency of getting old requests removed. If an extra flag is added to a request the offer function can update the flag on success. If the flag is set, we know that the request has already been accepted and we avoid the extra remote lock operations. If the flag is not set, the request may still be old and not ready, as it might have been accepted by another process.

Listing 10. The offer function for distributed channel communication.

```
inline offer(ch_id, r, w) {
        r_pid = ch[ch_id].rqueue[r].id;
        w_pid = ch[ch_id].wqueue[w].id;
        if /* acquire locks using global ordering */
        :: (r_pid < w_pid) ->
                remote_acquire(ch_id, r_pid, r_state);
                remote_acquire(ch_id, w_pid, w_state);
        :: else skip ->
                remote_acquire(ch_id, w_pid, w_state);
                remote_acquire(ch_id, r_pid, r_state);
        fi;
        if /* Does the two processes match? */
        :: (r_state == READY && w_state == READY) ->
                proc_release_lock_chan[r_pid]!
                        NOTIFY_SUCCESS, ch_id, ch[ch_id].wqueue[w].msg;
                proc_release_lock_chan[w_pid]!
                        NOTIFY_SUCCESS, ch_id, NULL;
                w = LEN; r = LEN; /* break match loop */
        :: else skip;
        fi;
        if /* release locks using reverse global ordering */
        :: (r_pid < w_pid) ->
                remote_release(ch_id, w_pid);
                remote_release(ch_id, r_pid);
        :: else skip ->
                remote_release(ch_id, r_pid);
                remote_release(ch_id, w_pid);
        fi;
}
```

Every channel must have a channel home, where the read and write requests for communication are held and the offers are made. The channel home invokes the matching algorithm for every posted request, as the post_* functions did in the local channel model. In this model every channel home is a process (Listing 11). In another implementation there might only be one process per node maintaining multiple channel homes through a simple channel dictionary.

Listing 11. The channel home process.

```
proctype channelThread(byte ch_id) {
  DECLARE_LOCAL_CHANNEL_VARS
  do
  :: ch_cmd_chan[ch_id]?cmd, id, msg ->
     if
     :: cmd == POST_WRITE ->
        <add process id, msg to ch[ch_id].wqueue>
        match(ch_id);

     :: cmd == POST_READ ->
        <add process id, msg to ch[ch_id].rqueue>
        match(ch_id);

     :: cmd == REMOVE_WRITE ->
        <remove process id, msg from ch[ch_id].wqueue>
        proc_cmd_chan[id]!REMOVE_ACK, ch_id, NULL;

     :: cmd == REMOVE_READ ->
        <remove process id, msg from ch[ch_id].rqueue>
        proc_cmd_chan[id]!REMOVE_ACK, ch_id, NULL;

     fi;
  :: timeout -> /* controlled shutdown */
     /* read and write queues must be empty */
     assert(ch[ch_id].rlen == 0 && ch[ch_id].wlen == 0);
     break;
od;
}
```

The functions read, write and alt are for the distributed channel model identical to the local channel model. We can now transfer a message locally using the local channel model or between nodes using the distributed channel model.

2.3. Dynamic Synchronisation Layer

The following model will allow channels to change the synchronisation mechanism on-the-fly. This means that a local channel can be upgraded to become a distributed channel. Activation of the upgrade may be caused by a remote process requesting to connect to the local channel. The model presented in this section can not detect which synchronisation mechanism to use, it must be set explicitly. If channel-ends were part of the implementation, a channel could keep track of the location of all channel-ends and thus it would know what mechanism to use.

A feature of the dynamic synchronisation mechanism is that specialised channels can be used, such as a low-latency one-to-one channel resulting in improved communication time and lower latency. The specialised channels may not support constructs like external-choice (alt), but if an external-choice occurs the channel is upgraded. The upgrade procedure adds an overhead, but since channels are often used more than once this is an acceptable overhead.

Figure 4 shows an overview of the transition model. In the figure, the communicating process calls read or write to communicate on channels. These then call the functions enter, wait and leave functions. The enter function posts the request to the channel. The wait function ensures that the post is posted at the correct synchronisation level, otherwise it calls the transcend function. The leave function is called, when the request has been matched successfully. The model includes a thread that at any time activates a switch in synchronisation level and thus may force a call to the transcend function.

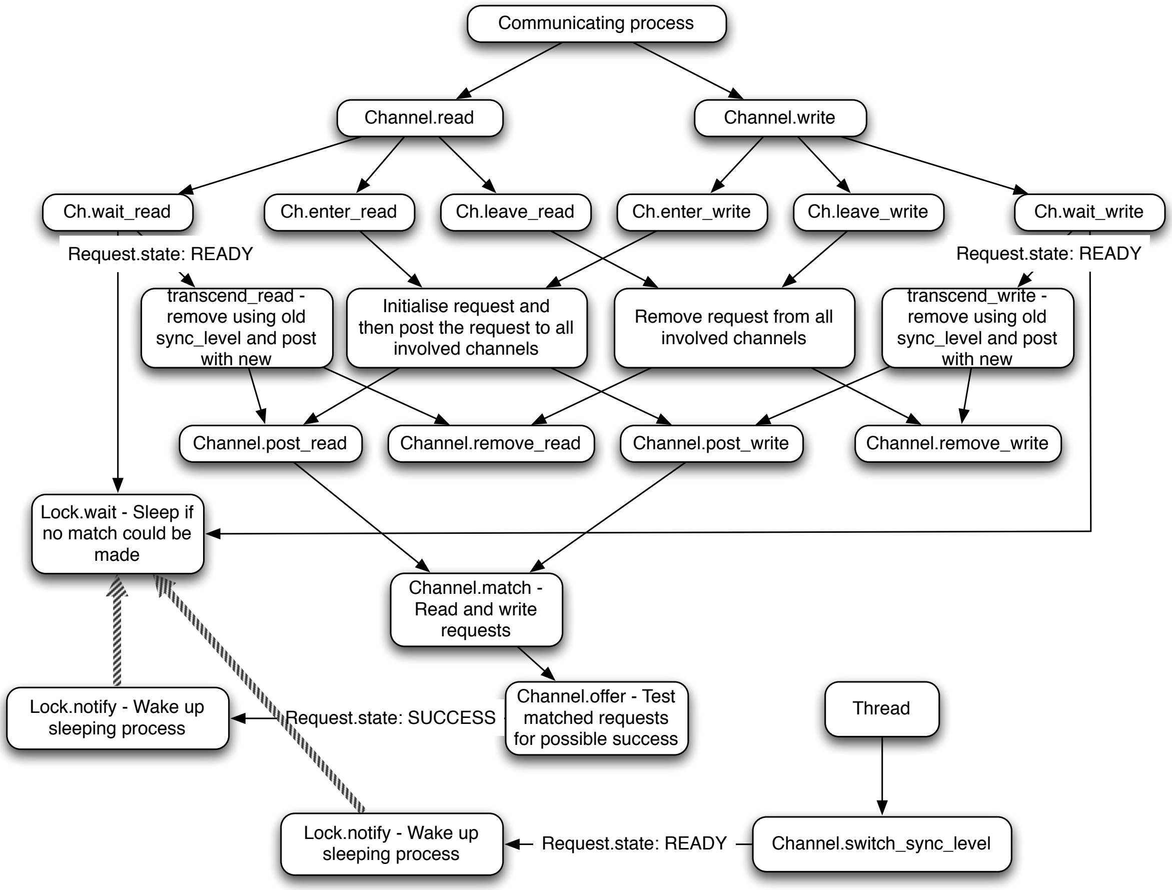

Communicating process
Channel.read
Channel.write
Ch.wait_read
Ch.enter_read
Ch.leave_read
Ch.enter_write
Ch.leave_write
Ch.wait_write
Request.state: READY
Request.state: READY
transcend_read - remove using old sync_level and post with new
Initialise request and then post the request to all involved channels
Remove request from all involved channels
transcend_write - remove using old sync_level and post with new
Channel.post_read
Channel.remove_read
Channel.post_write
Channel.remove_write
Lock.wait - Sleep if no match could be made
Channel.match - Read and write requests
Lock.notify - Wake up sleeping process
Request.state: SUCCESS
Channel.offer - Test matched requests for possible success
Thread
Lock.notify - Wake up sleeping process
Request.state: READY
Channel.switch_sync_level

Figure 4. Pseudo call graph for the dynamic synchronisation layer.

To model the transition between two levels (layers) we set up two groups of channel request queues and a synchronisation level variable per channel. Every access to a channel variable includes the channel id and the new synchronisation level variable sync_level. Every communicating process is viewed as a single channel-end and is provided with a proc_sync_level. This way the communicating process will know the synchronisation level that it is currently at, even though the sync_level variable for the channel changes. The synchronisation level of a channel may change at any time using the switch_sync_level function in listing 12.

The match and offer functions from section 2.1 have been extended with a sync_level parameter used to access the channel container. The post_∗ functions update the proc_sync_level variable to the channel synchronisation level before posting a request, while the remove_∗ functions read the proc_sync_level variable and uses the methods of that level to remove the request. Other than that, the functions match, offer, post_∗ and remove_∗ are similar to the ones from the local channel model.

The switching of synchronisation level in listing 12 works by notifying all processes with a request for communication posted to the channel. The channel sync_level variable is changed before notifying processes. In listing 14 when a process either tries to enter wait or is awoken by the notification, it will check that the proc_sync_level variable of the posted request still matches the sync_level variable of the channel. If these do not match, we activate the transcend (Listing 13) function. During a transition, the proc_state variable is temporarily changed to SYNC, so that the request is not matched by another process between release and leave_read. The leave_read function calls remove_read which uses the proc_sync_level variable to remove the request and enter_read calls post_read which uses the updated channel sync_level variable.

Listing 12. Switching the synchronisation level of a channel.

```
inline switch_sync_level(ch_id, to_level) {
        byte SL;
        byte r,w,r_pid, w_pid;
        SL = ch[ch_id].sync_level;
        atomic {        (ch[ch_id].lvl[SL].lock == 0) ->
                        ch[ch_id].lvl[SL].lock = 1; } /* acquire */
        ch[ch_id].sync_level = to_level;

        /* Notify connected processes */
        r = 0;
        do
        :: (r<ch[ch_id].lvl[SL].rlen) ->
                r_pid = ch[ch_id].lvl[SL].rqueue[r];
                acquire(r_pid);
                if
                :: proc_state[r_pid] == READY ->
                        notify(r_pid); /*Notify process to transcend*/
                :: else -> skip;
                fi;
                release(r_pid);
                r = r+1;
        :: else break;
        od;
        w = 0;
        do
        :: (w<ch[ch_id].lvl[SL].wlen) ->
                w_pid = ch[ch_id].lvl[SL].wqueue[w];
                acquire(w_pid);
                if
                :: proc_state[w_pid] == READY ->
                        notify(w_pid); /*Notify process to transcend*/
                :: else -> skip;
                fi;
                release(w_pid);
                w = w+1;
        :: else break;
        od;
        ch[ch_id].lvl[SL].lock = 0; /* release */
}
```

Listing 13. The transition mechanism for upgrading posted requests.

```
inline transcend_read(ch_id) {
        proc_state[_pid] = SYNC;
        release(_pid);
        leave_read(ch_id);
        enter_read(ch_id);
        acquire(_pid);
}
```

In listing 14 the read function from the local channel model (Section 2.1) is split into an enter, wait and leave part. To upgrade blocking processes we use the transition mechanism in listing 13 which can only be used between an enter and a leave part. We require that all synchronisation levels must have an enter part, a wait / notify state and a leave part.

Listing 14. The read function is split into an enter, wait and leave part.

```
inline enter_read(ch_id) {
        proc_state[_pid] = READY;
        post_read(ch_id);
}
inline wait_read(ch_id) {
        /* if no success, then wait for success */
        acquire(_pid);
        do
        :: (proc_sync_level[_pid] == ch[ch_id].sync_level) &&
           (proc_state[_pid] == READY) ->
                wait(_pid);
        :: (proc_sync_level[_pid] != ch[ch_id].sync_level) &&
           (proc_state[_pid] == READY) ->
                transcend_read(ch_id);
        :: else break;
        od;
        release(_pid);
}
inline leave_read(ch_id){
        assert(proc_state[_pid] == SUCCESS ||
               proc_state[_pid] == SYNC);
        remove_read(ch_id);
}
inline read(ch_id) {
        enter_read(ch_id);
        wait_read(ch_id);
        leave_read(ch_id);
}
```

The three models presented can be used separately for new projects or they can be combined to the following: a CSP library for a high-level programming language where channel-ends are mobile and can be sent to remote locations. The channel is automatically upgraded, which means that the communicating processes can exist as co-routines, threads and nodes. Specialised channel implementations can be used without the awareness of the communicating processes. Any channel implementation working at a synchronisation level in the dynamic channel, must provide six functions to the dynamic synchronisation layer: `enter_read`, `wait_read`, `leave_read`, `enter_write`, `wait_write` and `leave_write`.

3. Verification Using SPIN

The commands in listing 15 verify the state-space system of a SPIN model written in Promela. The verification process checks for the absence of deadlocks, livelocks, race conditions, unspecified receptions, unexecutable code and user-specified assertions. One of these user-specified assertions checks that the message is correctly transferred for a channel communication. All verifications were run in a single thread on an Intel Xeon E5520 with 24 Gb DDR3 memory with ECC.

Listing 15. The commands for running an automatic verification of the models.

```
spin -a model.p
gcc -o pan -O2 -DVECTORSZ=4196 -DMEMLIM=24000 -DSAFETY \\
        -DCOLLAPSE -DMA=1112 pan.c
./pan
```

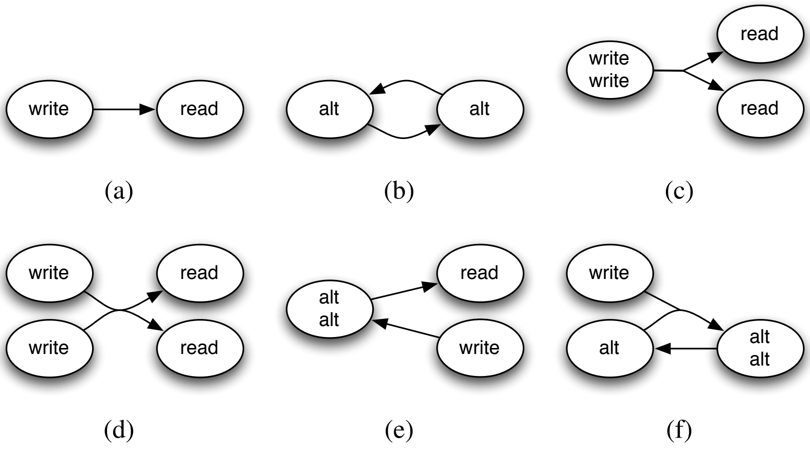
The local and the distributed channel models are verified for six process configurations and the transition model is verified for three process configurations. The results from running the SPIN model checker to verify models is listed in table 1. The automatic verification of the models found no errors. The "threads in model" column shows the threads needed for running the configuration in the specific model. The number of transitions in table 1 does not relate to how a real implementation of the model performs, but is the total amount of different transitions between states. If the number of transitions is high, then the model allows a large number of statements to happen in parallel. The SPIN model checker tries every transition possible, and if all transitions are legal the model is verified successfully for a process configuration. This means that for the verified configuration, the model has no deadlocks, no livelocks, no starvation, no race-conditions and do not fail with a wrong end-state.

The longest running verification which completed was the distributed model for the configuration in figure 5(f). This configuration completed after verifying the full state-space in 9 days. This means that adding an extra process to the model would multiply the total number of states to a level where we would not be able to complete a verification of the full state-space. The DiVinE model checker [14] is a parallel LTL model checker that should be able to handle larger models than SPIN, by performing a distributed verification. DiVinE has not been used with the models presented in this paper.

Table 1. The results from using the SPIN model checker to verify models.

Model	Configuration	Threads in model	Depth	Transitions
Local	Fig. 5(a)	2	91	1217
Local	Fig. 5(b)	2	163	10828
Local	Fig. 5(c)	3	227	149774
Local	Fig. 5(d)	4	261	2820315
Local	Fig. 5(e)	3	267	420946
Local	Fig. 5(f)	3	336	2056700
Distributed	Fig. 5(a)	5	151	90260
Distributed	Fig. 5(b)	6	245	28042640
Distributed	Fig. 5(c)	7	326	18901677
Distributed	Fig. 5(d)	9	446	1.1157292e+09
Distributed	Fig. 5(e)	8	406	6.771875e+08
Distributed	Fig. 5(f)	8	532	1.2102407e+10
Transition sync layer	Fig. 5(a)	3	162	43277
Transition sync layer	Fig. 5(c)	4	346	18567457
Transition sync layer	Fig. 5(d)	5	467	3.9206391e+09

The process configurations in figure 5 cover a wide variety of possible transitions for the local and distributed models. None of the configurations check a construct with more than two processes, but we expect the configurations to be correct for more than two processes. The synchronisation mechanisms are the same for a reading process and a writing process in the presented models. Based on this, we can expect that all the configurations in figure 5 can be mirrored and model-checked successfully. The local one-to-one communication is handled by the configuration in figure 5(a). Configurations in figure 5(c) and figure 5(d) cover the one-to-any and any-to-any cases, and we expect any-to-one to also be correct since it is a mirrored version of a one-to-any. The alt construct supports both input and output guards, thus figure 5(b) presents an obvious configuration to verify. In CSP networks this configuration does not make sense, but the verification of the configuration in figure 5(b) shows that two competing alts configured with the worst-case priority do not cause any livelocks. We must also model-check when alt communicates with reads or writes (Figure 5(e)).



(a) (b) (c)

(d) (e) (f)

Figure 5. Process configurations used for verification.

Finally, the configuration in figure 5(f) verify when alts are communicating on one-to-any and any-to-one. These configurations cover most situations for up to two processes.

4. Conclusions

We have presented three building blocks for a dynamic channel capable of transforming the internal synchronisation mechanisms during execution. The change in synchronisation mechanism is a basic part of the channel and can come about at any time. In the worst case, the communicating processes will see a delay caused by having to repost a communication request to the channel.

Three models have been presented and model-checked: the shared memory channel synchronisation model, the distributed channel synchronisation model and the dynamic synchronisation layer. The SPIN model checker has been used to perform an automatic verification of these models separately. During the verification it was checked that the communicated messages were transferred correctly using assertions. All models were found to verify with no errors for a variety of configurations with communicating sequential processes. The full model of the dynamic channel has not been verified, since the large state-space may make it unsuited for exhaustive verification using a model checker.

With the results from this paper, we can also conclude that the synchronisation mechanism in the current PyCSP [11,12] can be model-checked succesfully by SPIN. The current PyCSP uses the two-phase locking approach with total ordering of locks, which has now been shown to work correctly for both the shared memory model and the distributed model.

4.1. Future Work

The equivalence between the dynamic channel presented in this paper and CSP channels, as defined in the CSP algebra, needs to be shown. Through equivalence, it can also be shown that networks of dynamic channels function correctly.

The models presented in this paper will be the basis for a new PyCSP channel, that can start out as a simple pipe and evolve into a distributed channel spanning multiple nodes. This channel will support mobility of channel ends, termination handling, buffering, scheduling of lightweight processes, skip and timeout guards and a discovery service for channel homes.

5. Acknowledgements

The authors would like to extend their gratitude for the rigorous review of this paper, including numerous constructive proposals from the reviewers.

References

[1] David Beazly. Understanding the Python GIL. `http://dabeaz.com/python/UnderstandingGIL.pdf`. Presented at PyCon 2010.

[2] Rune M. Friborg and Brian Vinter. Rapid Development of Scalable Scientific Software Using a Process Oriented Approach. *Journal of Computational Science*, page 11, March 2011.

[3] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. *Proc. First IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986.

[4] Gerard J. Holzman. The Model Checker Spin. *IEEE Trans. on Software Engineering*, pages 279–295, May 1997.

[5] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, pages 666–676, August 1978.

[6] C.A.R. Hoare. Communicating Sequential Processes. *Prentice-Hall*, 1985.

[7] Peter H. Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, Jul 2007.

[8] M. Schweigler and A. Sampson. p0ny - the occam-π Network Environment. *Communicating Process Architectures 2006*, pages 77–108, Jan 2006.

[9] Neil C. Brown. C++CSP Networked. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 185–200, sep 2004.

[10] Pycsp distribution. `http://code.google.com/p/pycsp`.

[11] Rune M. Friborg, John Markus Bjørndalen, and Brian Vinter. Three Unique Implementations of Processes for PyCSP. In *Communicating Process Architectures 2009*, pages 277–292, 2009.

[12] Brian Vinter, John Markus Bjørndalen, and Rune M. Friborg. PyCSP Revisited. In *Communicating Process Architectures 2009*, pages 263–276, 2009.

[13] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International Series in Computer Science, 2005.

[14] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker. In *Parallel and Distributed Methods in Verification 2010*, pages 4–7, 2010.