UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD thesis

Mads Ohm Larsen

Bohrium and CSP

. . . an adventure in parallel processing and concurrent communication

Professor Brian Vinter

December 2018

Acknowledgements

I would foremost like to express my gratitude towards my supervisor, Professor Brian Vinter, for the opportunity of doing this PhD. The offer to start a PhD came at precisely the right time in my life and has been an experience I would rather not be without.

I would also like to thanks Professor Scott Baden for giving me the opportunity to work at the Lawrence Berkeley National Laboratory.

Many thanks for some great discussions both in the office and elsewhere goes to my fellow students and co-workers in the eScience Group: Klaus Birkelund Jensen, Mads R. B. Kristensen, Kenneth Skovhede, Martin Rehr, Jonas Bardino, James Avery, Troels Blum, Carsten Søgaard, Carl Johnsen, Rasmus Munk, and David Marchant.

Two other things that need acknowledgements are Codewars and the Copenhagen Ruby Brigade who both helped me stay sharp in the last three years. Without those who knows how well I would still be programming today.

Lastly, and most importantly, I would like to thank my friends and family for always being there for me. Dorthe, this is your fault! Morten, thanks for taking my mind off of work and instead concentrate on the more important stuff in life. Mom and dad, thank you for always listening, even though you might not have understood a word I was telling you. Bettina, you were immensely patient with me when I left you for half a year physically and more than a year mentally. Thank you so much.

This work is part of the CINEMA project and financial support from ***CINEMA****: The Alliance for Imaging of Energy Applications, DSF-grant no. 1305-00032B under The Danish Council for Strategic Research is gratefully acknowledged.*

Abstract

The computational sciences are generating an ever-increasing amount of data. This data needs to processed just as fast. The data comes from tomographies, images of distant star systems, gene sequencing, 3D projections, or similar real-world structures. Processing the data normally includes computing several tensor products or solving other such linear systems. Since we are processing multiple gigabytes or even terabytes of data, we really want to utilise GPGPUs, with which we can process more than one calculation of data simultaneously.

In this thesis, I show several optimisations for the modular array-programming framework Bohrium. I look into already established ways of optimising bytecode together with introducing extension methods, a way to call external libraries, for Bohrium. These extension methods are being automatically generated when Bohrium is being compiled or installed, thus we are able to link against different libraries depending on the runtime. I also discuss programmer productivity, a term which defines why it is important that the programmer is able to maintain their code and write new code. With programmer productivity, we look at the programming language Ruby and implements a new front end for Bohrium using it. This front end grants the ability to produce fast array-programming code in Ruby.

Lastly, and somewhat separate, I talk about a new Ruby framework for Communicating Sequential Processes. This framework makes it easy to emulate a CSP network with communications and is also producing fast results.

Resumé

Beregningsvidenskaberne genererer en stadigt stigende mængde data. Dette data skal behandles lige så hurtigt. Data kommer fra tomografier, billeder af fjerne stjernesystemer, gensekvenser, 3D-projektioner eller lignende strukturer fra den virkelige verden. Behandling af data omfatter normalt beregning af flere tensorprodukter eller løsning af andre sådanne lineære systemer. Da vi behandler flere gigabyte eller endda terabyte data, vil vi virkelig gerne bruge GPGPU'er, hvorved vi kan behandle mere end én beregning af data samtidigt.

I denne afhandling viser jeg adskillige optimeringer til Bohrium, et modulært array-programmeringsrammeværktøj. Jeg undersøger allerede kendte måder at optimere bytecoden på, samt kommer med metoder til at kalde eksterne biblioteker på, i Bohrium. Disse metoder genereres automatisk, når Bohrium kompileres eller installeres, så vi kan linke mod forskellige biblioteker afhængigt af kørselsmiljø. Jeg diskuterer også programmeringsproduktivitet, et begreb, der definerer, hvorfor det er vigtigt, at programmøren kan opretholde deres kode og skrive ny kode. Med programmeringsproduktivitet ser vi på programmeringssproget Ruby og implementerer en ny front end til Bohrium ved hjælp af det. Denne front end giver mulighed for at producere hurtig array-programmeringskode i Ruby.

Endelig, og lidt adskilt, kigger jeg på et nyt Ruby rammeværktøj for "Communicating Sequential Processes". Dette rammeværktøj gør det nemt at simulere et CSP-netværk med kommunikation, og det giver også hurtige resultater.

Contents

List of Figures

List of Listings

I

Extended Abstract

CHAPTER 1

Introduction

In the last three years, I have worked with optimisations for the Bohrium framework as well as creating a CSP (Communicating Sequential Processes) framework for the programming language Ruby. In this thesis, I present that work, the considerations that went into it, and results from the various optimisations.

The thesis is thus separated into two parts – Bohrium and CSP – since these are two distinct topics.

My work began with investigating various optimisations already in place in tools like `gcc` and `llvm`. From there on, I went into the world of array-programming and tried to use some of the same optimisations here. I looked into utilising already established external libraries in Bohrium because these have often been optimised already. A new templating approach was used instead of hand implementing all the wrapper functions from the various library methods.

I also wanted to spread the word of Bohrium even further than the Python community, so I developed a new front end for it using Ruby. This new front end I named *Bohrium.rb*[1]. With this new Ruby front end, all Ruby programmers can partake in increased speeds for their array-programming tasks.

[1] `.rb` is the common filename extension for Ruby files.

Parallel to my work on Bohrium, I looked into CSP and how to use it in practice. Ruby does not have a CSP framework of its own. This led me to develop my very own CSP framework, called *Emit*, in Ruby. Emit ended up being a small DSL (Domain Specific Language) for CSP code, that also performs rather well.

In the following thesis, I describe both my work and contributions to Bohrium, with the Ruby front end and templating of library methods, as well as my work on CSP with Emit. All of this is laid out in the diagram in Figure 1.1.

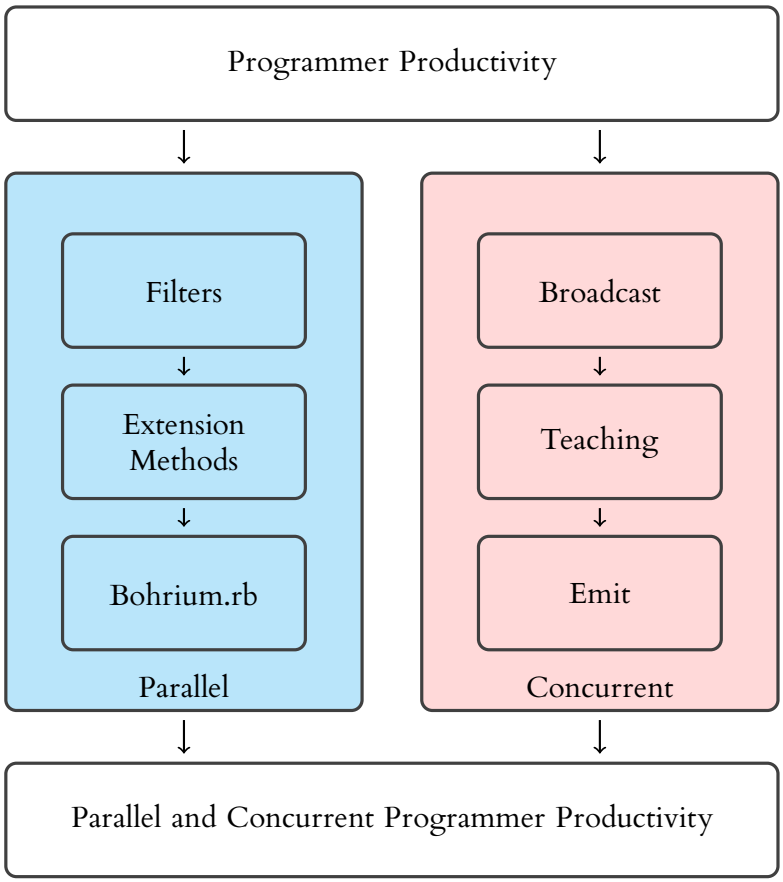


Figure 1.1: Project overview.

1.1 Contributions

This thesis begins by displaying three very different contributions to the automatic acceleration framework Bohrium. First I show a couple of *filters* I have added to Bohrium. These filters clean up some of the mathematical no-ops that other optimisations might produce.

The second Bohrium contribution is *extension methods* and how I have automated the code generation for some of them. This code generation is viable even for the general case.

Thirdly, a new Ruby front end for Bohrium has been made. Ruby is a programming language very similar to Python, which was the first front end for Bohrium. However, Ruby has only recently gotten some traction in the scientific fields with various new scientific packages, one of which I will compare against in Chapter 5.

I investigated various algebraic methods for speeding up the Bohrium intermediate language, essentially speeding up both the generation and execution of kernels from within Bohrium. The methods are described in Chapter 2 together with a background for other optimisation techniques.

Another contribution I made, has been the implementation of the CSP framework Emit in Ruby. As already hinted at Emit seems to work nicely as a CSP framework and have been shown to be quite fast as well.

Furthermore, I co-authored a paper defining what is meant by *broadcast* in CSP and introduced it into algebra, as well as showing that it works with the formal verification tool FDR.

Because of my knowledge in CSP, I also co-advised a master student in her master's project as well as with an article, that we co-authored. The topic of both was transpiling SME (Synchronous Message Exchange) to CSP_M (Machine-readable CSP).

1.1.1 Bohrium Filters

The filters try to look at the bytecode generated at runtime within Bohrium to optimise the execution time. As Bohrium is built from disjoint components we can easily create filters that can be turned on and off at runtime. These filters each lean on various already established compiler optimisations, but can go further,

since we have better knowledge when trying to create kernels from the bytecode. We can use the filters to contract mathematical statements and to remove no-ops from the bytecode sequence.

1.1.2 Automatic Extension Methods

Extension methods are the Bohrium way of saying external library functions. Since Bohrium works with tensors[1] in an array-programming paradigm, it can utilise external libraries for working with these in a linear algebra fashion. A common library for linear algebra is BLAS. BLAS exposes roughly 145 different functions, which, if you want to utilise the entire library, will need to be implemented within Bohrium with various unpacking and repacking of the arrays. This can be done by hand but will be a tedious job. Automating this task, and similar tasks for other external libraries, with the help of templating, has helped out Bohrium a lot.

1.1.3 Bohrium.rb: The Ruby Front End

With Bohrium we aim a lot for programmer productivity, but also want performance in the end. The Ruby front end makes it easy to program Ruby with Bohrium without losing said performance. Since Ruby is quite new in the scientific field, the competition for being fast and easy is still ongoing. As shown in [1], I manage to outperform both the standard library and another, already known, array-programming library, by implementing a new array API (Application Programming Interface) utilising Bohrium for large computations.

The work done with Bohrium.rb is detailed in Chapter 5.

1.1.4 Emit: Communicating Sequential Processes for Ruby

A different aspect of high-performance computing is verification. Communicating Sequential Processes is an algebra used to describe process networks. Networks written with CSP can be verified using tools such as FDR [2]. With my contributions, Ruby has now got a CSP framework of its own, that is also quite fast. In it, you can execute and emulate your CSP networks. This framework borrows a lot from the already established PyCSP framework [3, 4], which has also been written at the University of Copenhagen.

This work is discussed in Chapter 7.

[1] A tensor is a n-dimensional matrix. Vectors can be thought of as being 1-dimensional matrices.

1.2 Publications

This section contains an overview and abstracts of the ten articles I have written or contributed to. I am the first author on six of them.

One paper for the *19th Workshop on Compilers for Parallel Computing*, one for the *5th Exascale Applications and Software Conference*, six papers for the *WoTUG Conference on Concurrent and Parallel Systems* (over several years), one workshop paper for the *1st International Workshop on Next Generation of Cloud Architecture at ACM EuroSys 2017*, one positional paper for the *Doctoral Symposium at the ACM/IFIP/USENIX Middleware Conference 2017*.

The papers are all included in the appendix, Part III, beginning on page 97.

1.2.1 Current Status and Directions for the Bohrium Run-time System

Mads Ohm Larsen, **Kenneth Skovhede**, **Mads R. B. Kristensen**, and **Brian Vinter**, *CPC 2016, 19th Workshop on Compilers for Parallel Computing, Valladolid, Spain*

In this paper, we present the current status of the Bohrium run-time system for automatic parallelization of array-programming languages and libraries. We demonstrate how the design of Bohrium makes it possible to utilise different hardware platforms – from simple multi-core systems to clusters and GPU enabled systems – without any changes to the original user program.

1.2.2 Broadcasting in CSP-Style Programming

Brian Vinter, **Kenneth Skovhede**, and **Mads Ohm Larsen**, *Proceedings of Communicating Process Architectures 2016, the 38th WoTUG conference on concurrent and parallel systems, Niels Bohr Institute, University of Copenhagen*

While CSP-only models process-to-process rendezvous-style message passing, all newer CSP-type programming libraries offer more powerful mechanisms, such as buffered channels, and multiple receivers, and even multiple senders, on a single channel. This work investigates the possible variations of a one-to-all, broadcasting, channel. We discuss the

different semantic meanings of broadcasting and show three different possible solutions for adding broad-casting to CSP-style programming.

1.2.3 Algebraic Transformation of Descriptive Vector Bytecode Sequences

Mads Ohm Larsen, *Middleware 2016, Proceedings of the Doctoral Symposium of the 17th International Middleware Conference, Trento, Italy*

Both high-productivity and high-performance are two often sought after aspects of scientific programming. Python gives the programmer high-productivity, but even with NumPy, it is often not high-performant because of the GIL, which makes it inherently single threaded. Bohrium intercepts NumPy calls and generates an intermediate language, Bohrium bytecode, before being compiled to OpenCL kernels. It thus grants Python/NumPy the ability to be easily run on multi-core systems or GPUs, without changing the source code. The Bohrium bytecode can be optimised, by transforming bytecode sequences into more performant ones. This way, the scientific programmer will not need to change her code to utilise special performant constructs.

1.2.4 Imaging Data Management System

Brian Vinter, **Jonas Bardino**, **Martin Rehr**, **Klaus Birkelund Jensen**, and **Mads Ohm Larsen**, *EuroSys '17, Proceedings of the 1st International Workshop on Next generation of Cloud Architecture, Belgrade, Serbia*

In this work, we present an integrated system aimed at data management and processing for scientific areas that work with very large data-sets. The Imaging Data Management System, IDMS, seeks to support researchers in all steps of their research, starting with the transfer of data from the lab, over managing and analyzing the data, to final archiving of the essential research project results. While IDMS is in fact hosted locally at the university we seek to provide a user experience that is as close as possible to a generic cloud system, in order to allow users to share and collaborate on their data seamlessly from anywhere.

1.2.5 Automatic Code Generation for Library Method Inclusion in Domain Specific Languages

Mads Ohm Larsen, *Proceedings of Communicating Process Architectures 2017, the 39th WoTUG conference on concurrent and parallel systems, Department of Computer Science, University of Malta, Malta.*

Performance is important when creating large experiments or simulations. However, it would be preferable not to lose programmer productivity. A lot of effort has already been put into creating fast libraries used for for example linear algebra based computations (BLAS and LAPACK). In this paper, we show that utilising these libraries in a DSL made for productivity will solve both problems. This is done via automatic code generation and can be extended to other languages, libraries, and features.

1.2.6 Teaching Concurrency: 10 Years of Programming Projects at UCPH

Brian Vinter and **Mads Ohm Larsen**, *Proceedings of Communicating Process Architectures 2017, the 39th WoTUG conference on concurrent and parallel systems, Department of Computer Science, University of Malta, Malta.*

While CSP is traditionally taught as an algebra, with a focus on definitions and proofs, it may also be presented as a style of programming, that is process-oriented programming. For the last decade, the University of Copenhagen (UCPH) has been teaching CSP as a mix of the two, including both the formal aspects and process-oriented programming. This paper summarised the work that has been made to make process-oriented programming relevant to students, through programming assignments where process orientation is clearly simpler than an equivalent solution in imperative programming style.

1.2.7 Case study: Bohrium – Powering Oceanography Simulation

Mads Ohm Larsen and **Dion Häfner**, *EASC 2018, 5th Exascale Applications and Software Conference, Edinburgh, Scotland*

A rundown of how Veros uses Bohrium was supplied as a one-page abstract.

1.2.8 Emit – Communicating Sequential Processes in Ruby

Mads Ohm Larsen and **Brian Vinter**, *Proceedings of Communicating Process Architectures 2018, the 40th WoTUG conference on concurrent and parallel systems, University of Dresden, Germany.*

CSP is an algebra for reasoning about concurrent systems of processes. Being able to do so has become a necessity for computer scientists. Having to think about abstractions like mutexes and threads in practice can be cumbersome, complex, and erroneous. Ruby as a programming language has been described as fun to program in. It is, however, missing a CSP framework that it can call its own. Emit*, which is presented in this paper, tries to mitigate this by providing such a CSP framework. As a CSP framework, Emit makes it easy to think about processes, channels, and communication. It is not yet feature-complete, however comparing it to its nearest peer, PyCSP, shows good performance for the* COMMSTIME *benchmark, where Emit is 100 times faster.*

1.2.9 Bohrium.rb – The Ruby Front End

Mads Ohm Larsen and **Brian Vinter**, *Proceedings of Communicating Process Architectures 2018, the 40th WoTUG conference on concurrent and parallel systems, University of Dresden, Germany.*

The acceptance of Ruby in the scientific community lags a bit behind, partly because it is missing a good library for linear algebra and vector programming. It has a `matrix` *class in its standard library, but its execution tends to be rather slow. Only a couple of actual scientific computing libraries like NumPy for Python exist for Ruby. In this paper, we introduce a new library called* Bohrium.rb*. Bohrium.rb acts as a front end for the Bohrium framework, which generates and runs JIT-compiled OpenMP/OpenCL kernels. It currently supports Python/NumPy and C++, however as it is built of processes communicating hierarchically to each other, we can replace the front ends with new ones. This new Ruby front end is described with examples and is then compared to the standard*

library and an already established Ruby library Numo/Narray, where Bohrium.rb seems to be faster for still larger matrix calculations. This is also the trend we have seen in similar areas with Bohrium, being faster once its overhead has been amortised.

1.2.10 Towards Automatic Program Specification

Alberte Thegler, **Mads Ohm Larsen**, **Kenneth Skovhede** and **Brian Vinter**, *Proceedings of Communicating Process Architectures 2018, the 40th WoTUG conference on concurrent and parallel systems, University of Dresden, Germany.*

This paper introduces a method to simplify hardware modelling and verification thereof in order for software programmers to, more easily, meet the demands of the growing embedded device industry. We describe a simple method for transpiling from the new SME Implementation Language into CSP_M and using formal verification to verify properties within the generated program. We present a small example consisting of a seven segment display clock network and introduce how to verify the widths of the channels in the network.

This paper won the best paper award at Communicating Process Architectures 2018.

CHAPTER 2

Background

2.1 Computational Science

A trend we have seen through the last couple of decades are large amounts of data being produced from the various sciences [5]. This amount of data has been ever increasing. In some sciences, this data is several terabytes of tensors, which could all stem from a single or a couple of input values. We also see large images or projections, for example in astronomy or from tomographies [6].

The data is often manipulated with linear algebra techniques, for example, solving linear systems, tracing, summing or otherwise calculating from the data. Having terabytes or more of tensor data will become difficult for most applications and computers to work with. Often, for example with tomographies, you can work on reconstruction or segmentation in one layer or maybe a few layers at a time, slicing the tensors into small matrices or tensors with a smaller dimensionality. These data sets, however, will still be fairly large and thus not pleasant to work with because of an inherent drop in programmer productivity, a term we will come back to later.

One thing we can do is to lighten the burden for the programmer to create *clever* ways of manipulating data for various reasons. Often we try to optimise for L1 and L2 caches by reading and writing sequentially. For example, when

we read data, if we read it like the caches believe we do, we will get a speed-up, since we might have already preloaded the next cells in our tensor automatically. Instead, if we read randomly from a large tensor, we might skip around and read a bit here and a bit here, invalidating the caches for every read.

2.2 Compiler Optimisation

Instead of handling the various difficult areas of the code by hand, we often rely on the compilers to help us. That is, the tools we use for our code should help us out by changing minimal things behind the scene, before giving the end result to us as programmers.

This section lists and explains some general purpose optimisation techniques for imperative programming languages, especially some of the optimisations used in `gcc` (GNU C Compiler) [7].

Local and Global Optimisations We start with the two main methods of optimisation. When we optimise we can either optimise locally or globally. What that means is that we can optimise a single function call or block of code, or we can try to optimise the entire program as a whole. What we do will largely impact the end result, since a more global optimisation approach will be better but also requires more computing and thus time for the compiler to figure out.

There is no one optimisation solution to fit all, so the following optimisation techniques all fall into the category of either local or global optimisation, some can even be both, but each time we have to choose how much code to look at.

Peephole Optimisations Peephole optimisations are usually one of the last steps in the compilation process, just as we go into machine code. I describe it first since it is the simplest of the optimisations. With peephole optimisation, we look at a single or very few instructions following each other. These might be swapped, combined, or even deleted for faster and more specialised execution.

One such optimisation would be left shifting an integer instead of multiplying it by factors of 2 or right shifting instead of dividing by factors of 2. This works only with integers but is faster since we do not have to involve an ALU (Arithmetic Logic Unit) in the process of shifting.

We can delete operations that do not do anything anyway. We call these operations "no-ops". Such an instruction could be an addition of 0 or multiplications with 1.

Loop Optimisations Several optimisation techniques fall under the category of loop optimisations, because, for most programs, the loops are where most of the time is spent computing. This also means that loop optimisations might be the most important of the optimisations.

Here follow some subparagraphs with examples of different ways to optimise loops.

Induction Variable Analysis If a variable inside a loop is a linear function of the index variable of the loop, such as seen in Listing 2.1, we can update that variable together with the index variable every time.

```
int j;
for(int i = 0; i < 10; ++i) {
    j = 2 * i + 2;
    // Use j for something...
}
```

(a) Before.

```
for(int i = 0, int j = 2; i < 10; ++i, j += 2*i) {
    // Use j for something...
}
```

(b) After.

Listing 2.1: Linear induction variable.

If the index variable is only used for this computation, we might be able to factor it out, via dead code elimination, as well.

Loop Fission and Fusion One loop computing multiple different things can both speed-up or slow-down the code execution, depending on what is going on in said loop. An example of loop fission, where we split a loop into two loops for increased performance can be seen in Listing 2.2a. Here we assume that A and B are already initialised with some values. These values are then read from the arrays and are summed together in two separate integers. A and B might not be close to each other in memory, which can cause several jumps back and forth in memory to read all values. These jump might invalidate cache or disrupt prefetching on

a hardware level. Splitting it into two loops, that loop over each array might be faster as seen in the example in Listing 2.2b.

```
#define N 1000
int t, s, A[N], B[N]; // Initialise A and B with integer values

for (int i = 0; i < N; i++) {
    t += A[i];
    s += B[i];
}
```

(a) Before.

```
for (int i = 0; i < N; i++) {
    t += A[i];
}

for (int i = 0; i < N; i++) {
    s += B[i];
}
```

(b) After.

Listing 2.2: Loop fission.

Fusion is the opposite of fission. Here we instead merge loops, when possible, to have a lot of work done in each iteration. A classic example is shown in Listing 2.3a. Here we have a couple of arrays that are each indexed in a similar way in two loops. We can instead merge the two loops. However, just merging them means that we will index the same `T[i]` twice in each loop iteration as seen in Listing 2.3c. Instead, if we can guarantee that `T` will not be used anywhere else in the program, we can just update the `A` array directly, thus saving indexing into `T`. This is done because the actual data of `A`, `B` and `T` might be far from each other in memory, and therefore might not be easy to fetch and update together.

Optimise the Common Case Let us say that you have a function with many branches (if-statements). If it is always the first path that is being taken, we might just create a separate function, that only contains these, without the actual branch look-ups. That is, imagine having the code in the first part of Listing 2.4. Here we see two branches that, depending on the argument, will take the same path. We can imagine larger functions or maybe several functions stitched together, that share this behaviour. Instead, we can split it into two functions. In the second part of Listing 2.4 I show two functions where the first is what is being done if the argument is `1` and the second when it is something else.

```
#define N 1000
double A[N], B[N], T[N];

for(int i = 0; i < N; ++i) {
    T[i] = B[i] * A[i];
}

for(int i = 0; i < N; ++i) {
    A[i] += T[i];
}
```

(a) Before (1).

```
for(int i = 0; i < N; ++i) {
    T[i] = B[i] * A[i];
    A[i] += T[i];
}
```

(b) Before (2).

```
for(int i = 0; i < N; ++i) {
    double t = B[i] * A[i];
    A[i] += t;
}
```

(c) After.

Listing 2.3: Loop fusion.

Avoid Redundancy Avoiding redundancy is both a discipline for the programmer as well as the compiler. If you as a programmer are computing the same result twice or more, the compiler can reuse the result from the first calculation instead of recomputing it. We thus store the initial result in a register somewhere and load it from there, when we are asked to recompute it.

Of course, as can be seen in Listing 2.5 the calculation needs to be without side-effects, or else we would indeed have to redo them. Simple redundancy avoidance comes from maths terms, that are condensed to simpler terms.

Less Code When optimising for less code, we want to remove unnecessary intermediate computations. This usually results in less code being executed, because we can optimise better, when not storing intermediate values. Listing 2.6 shows a very simple program, where if `a` is not used anymore, then removing it completely will result in less code. Since we no longer need to save a copy of `a`, our stack is also smaller.

Of course, here we can go even further and use constant folding on the rest and reduce the statement down to `int b = 16;`.

```
int function(int arg) {
  if (arg == 1) {
      // (1a) Do something
  } else {
      // (1b) Do something else
  }

  // (2) Do more stuff

  if (arg == 1) {
      // (3a) Do something
  } else {
      // (3b) Do something else
  }
}
```

(a) Before.

```
int function1(/* int arg = 1 */) {
    // (1a) Do something
    // (2) Do more stuff
    // (3a) Do something
}

int function2(int arg) {
    // (1b) Do something else
    // (2) Do more stuff
    // (3b) Do something else
}
```

(b) After.

Listing 2.4: Optimising the common case.

```
double a = f(7); // Some heavy calculation without side-effects
double b = f(7);
```

(a) Before.

```
double a = f(7); // Some heavy calculation without side-effects
double b;
memcpy(&b, &a, sizeof b); // Copy the contents of a into b
```

(b) After.

Listing 2.5: Avoid redundancy.

Strength Reduction Strength reduction can be a powerful optimisation on some architectures. Sometimes using a simpler instruction will require less computation time. Instead of dividing by a constant, we can multiply with its reciprocal, or instead of calculating the exponentiation with an integer we can multiply that many times.

```
int a = 2 * 2;
int b = a * 4;
// a is not used more in this program
```

(a) Before.

```
int b = 2 * 2 * 4;
```

(b) After.

Listing 2.6: Less code.

The following equation shows this relation:

$$x^n = \underbrace{x \cdot x \cdot \ldots \cdot x}_{n} = \prod^{n} x \quad \text{if } n \in \mathbb{N}$$

Because multiplication with integers is a simpler instruction than exponentiation, it will compute faster on some architectures and thus we might be able to do multiple of these in the same amount of time.

As an example, we can change x^{10} into several multiplications of smaller powers

$$\begin{aligned} x^{10} &= x^8 \cdot x \cdot x \\ &= x^4 \cdot x^4 \cdot x \cdot x \\ &= x^2 \cdot x^2 \cdot x^2 \cdot x^2 \cdot x \cdot x \end{aligned}$$

This means, that if we first compute x^2 we can use this further on, and instead of doing 10 multiplications we can get away with only 5 without the need for extra variables. We can do with 4 integer multiplications if we can have more temporary variables. In pseudocode, this would look like Listing 2.7.

The reasons that we might want to limit ourselves to fewer temporary variables come in memory size. For integers, it might not be a problem, but as we will see later (subsection 2.3.2 on page 25) the thing we are doing exponentiation on could be a large array of integers instead. Copying a large data structure around in memory, just to throw it out afterwards might be a bad idea.

Common Sub-expression Elimination Much like optimising for less code or redundancy avoidance, common sub-expression elimination will find calculations that have already been done and reuse the results.

```
double x = 123.4;           // Any number
double result = pow(x, 10); // Calculate x to the 10th power

// Instead your compiler will calculate (with no extra variable)
double result = x * x;     // x^2
result = result * result; // x^4
result = result * result; // x^8
result = result * x;      // x^9
result = result * x;      // x^10

// ... or even (with one extra variable)
double result = x * x;          // x^2
double tmp1 = result * result; // x^4
tmp1 = tmp1 * tmp1;             // x^8
result = result * tmp1;         // x^10
```

Listing 2.7: Strength reduction with exponentiation.

Examples of common sub-expression elimination can be seen in Listing 2.8. Here Listing 2.8a has `a + b` twice. This can be factored into a separate variable as seen in Listing 2.8b. We also see a bit of algebraic rewriting in Listing 2.8c, as the compiler figures out, that a variable divided by `4` is the same as it is divided by `2` twice.

```
int r = (a + b) / 2 + (a + b) / 4;
```

(a) Before (1).

```
int c = (a + b);
int r = c / 2 + c / 4;
```

(b) Before (2).

```
int c = (a + b) / 2;
int r = c + c / 2;
```

(c) After.

Listing 2.8: Common sub-expression elimination.

Constant Folding Again, like the less code optimisation, constant folding will look into precompiling constants into the most condensed form.

When having constant values the right hand side of the expression could be computed at compile time, thus eliminating the need to recompute it for every use. In Listing 2.9 we have merged `2 + 2` into a single `4`. This could propagate

through a lot of variables and thus end in multiple variables becoming obsolete. These obsolete variables will be removed with dead code elimination later on.

```
const int a = 2 + 2;
```

(a) Before.

```
const int a = 4;
```

(b) After.

Listing 2.9: Constant folding.

Constant folding is useful for computing constants at compile-time. Thus, we end up with a program, that has the result already built in, instead of having to evaluate the code each and every time we run the code. Since this is a constant, we should be safe doing so.

Code Factoring Some functions can be optimised with common sub-expression elimination however if they are not identical, we cannot just replace it in this manner. If a block of code only differs by some parameters, we can however still do something like common sub-expression elimination. The compiler can automatically wrap the code block in functions and afterwards parameterize those functions with the differences between two code blocks.

```
int a = ... + 2 + ...; // Some complex calculation
int b = ... + 4 + ...; // The same complex calculation
```

(a) Before.

```
int f(int a) {
    return ... + a + ...; // again the same complex calculation
}

int a = f(2);
int b = f(4);
```

(b) After.

Listing 2.10: Code factoring.

Shown in Listing 2.10 is code factoring. Here we move a complex calculation into a function, which is parameterized with the difference between the two calculations. Now the function can be optimised on its own, and both the calculations

will benefit from it.

The examples shown in this section are all done directly in the C-code, however, this might not be how the compiler sees the code. The compiler can optimise both before and after converting the code into some intermediate representation (IR), assembly language, or bytecode. When we later look into optimising the Bohrium IR, we will only look at the bytecode, since the Bohrium JIT-compiler does not have access to the actual script being executing, but is rather just handed the bytecode to optimise.

2.3 Bohrium

Bohrium is a very modular framework with both multiple front ends and back ends and many components in between. It provides a way to speed-up array-programming.

Array-programming [8] is a way of thinking that exploits data properties where elements are similar. With array-programming, we look at groups of data instead of the traditional object-oriented view. We can thus perform a uniform function on the data. Usually, we perform these operations without the need to explicitly state loops of scalar operations.

The name Bohrium comes from the element of the same name, discovered by Danish physicist Niels Bohr. Since the framework was created at Niels Bohr Institute for use by computational sciences the name Bohrium seems fitting.

This section will contain a bird's eye view of Bohrium and only some components will be explained in detail.

Bohrium was created by Mads R. B. Kristensen et al. [9, 10, 11] in 2013 and have since undergone several improvement phases.

2.3.1 Overview

Bohrium is made to lazily record array operations, such as those from NumPy [12], and combine them into a bytecode instruction set, in an internal intermediate representation (IR). This IR is then compiled into architecture specific kernels and executed.

Internally Bohrium consists of a number of components, that communicate in a sequential way, passing on their output and options to the next component. These

components can be interchanged and are all specialised for different purposes. You do not need to use all components for all executions since there exists multiple front and back ends and various optional components for optimisation. An overview of these components can be seen in Figure 2.1 and will be outlined in the following paragraphs.

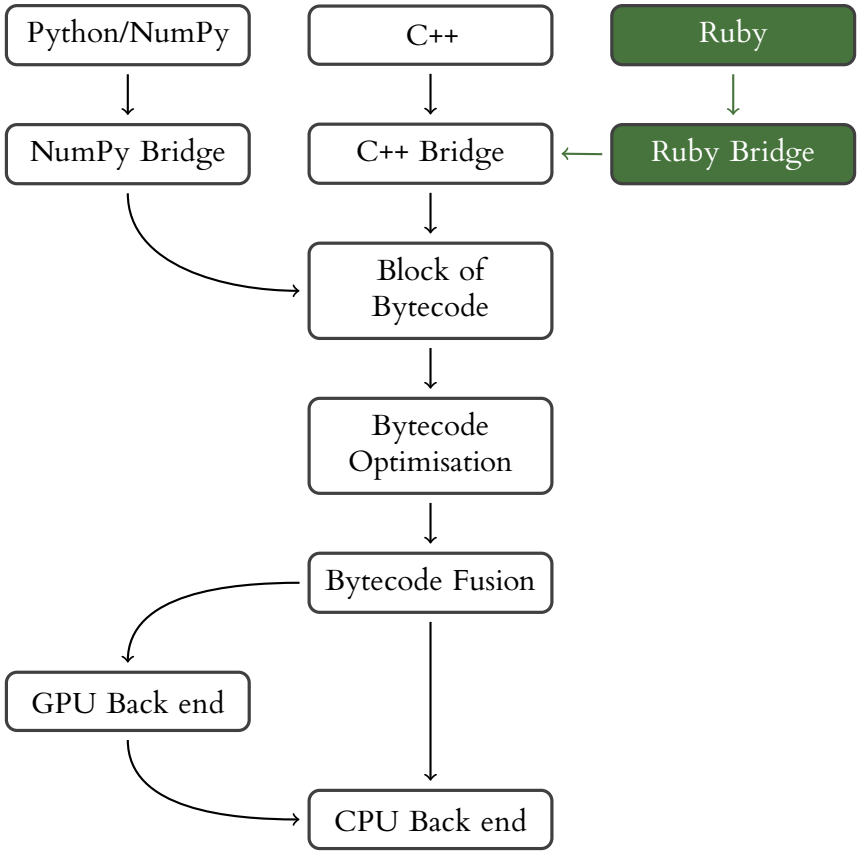


Figure 2.1: Bohrium component overview.

Front End The front ends are the access points. This is what the user sees and programs for. These are for example C++, Python and Ruby (more on the Ruby front end in Chapter 5). Bohrium itself does not favour any language or library but can have any number of front ends, as long as there is some kind of array-programming involved.

Bridge For the front ends to be able to enter Bohrium "land", we have a bridge. The bridge is unique for each front end, as it is the bridges job to translate between the front end (the host language) and Bohrium.

Bytecode Optimisation Before we execute the bytecode generated by the bridge, we can optimise the IR. Bohrium does this in a number of ways. A description of the optimisations being done can be found in Chapter 3.

Bytecode Fusion After we are done with bytecode optimisations, Bohrium will seek out to fuse as many instructions into one kernel as possible.

The fusibility is determined by the back end since some back ends might have, for example, a data-parallel criterion. Another common criterion is that shapes of arrays being worked on must match. Fusing together bytecodes will make sure that we compute as much as possible in each kernel, instead of wasting time transferring data back and forth between the front and back end.

Back end The back end will take the fused IR and actually generate and execute the kernels. Bohrium supports multiple back ends and can thus generate various kinds of kernels. These kernels will be architecture specific and can target CPUs or GPUs.

2.3.2 Bohrium Bytecode

The internal representation of Bohrium's bytecode is a simple language without conditionals and loops. At the time of writing it consists of 79 different opcodes. These all take one, two, or three operands as arguments, depending on the opcode. The operands can be either scalars or multi-dimensional tensors. The size of the tensor depends on the operation being computed as some operations require operands of the same size while other, for example, the reductions reduce the number of elements in the result tensor.

When we print the IR we have exactly one opcode per line. These lines should be read as follows: First the name of the opcode followed by the first operand and its possible start and end index together with a stride. The second and possibly third opcode follows the same structure. For example, `BH_IDENTITY a1[0:5:1] 1` computes the identity of `1` and puts into the result operand, which is always the first operand, starting from `0` going to `5` with a stride of `1`.

A simple example of a Bohrium trace which prints the bytecode sequence can be seen in Listing 2.11. Here we have two tensors `a1` and `a2`. They both have all their elements set to `1` with `BH_IDENTITY`. Afterwards, they are added with `BH_ADD` and the result is stored again in the first tensor, `a1`, overwriting it.

```
BH_IDENTITY a1[0:5:1] 1
BH_IDENTITY a2[0:5:1] 1
BH_ADD a1[0:5:1] a1[0:5:1] a2[0:5:1]
```

Listing 2.11: An example of Bohrium IR.

Of course, this IR is not executed as is, but rather it is a representation of what is stored inside Bohrium prior to it generating kernels for the currently selected architecture and runtime environment. These kernels are generated and compiled just-in-time. We can thus utilise maximal knowledge about what is being generated and executed because we know all the indices for all the tensors as well as the system it should run on. This can both be used to know where to place the generated kernel, be it the CPU or GPGPU, as well as optimise for the current case.

2.4 Library Methods

A lot of work has been put into various libraries to make them perform faster calculations. These are said to be highly *hand-optimised*. One such library specification is BLAS [13] (Basic Linear Algebra Subprograms), which have been around since the late 70's. The reference implementation of BLAS describes how to implement fast linear algebra methods, both for solving systems or for multiplying matrices.

These types of libraries have been optimised so much, that the compilers cannot optimise anymore. If they did, the programmers would change the reference implementation to be this new highly optimised version. This is a good thing since the compile-time will then also be optimised.

The BLAS implementation, of course, cannot be the fastest when run everywhere, so we specify the implementation even further, having, for example, libraries such as `clBLAS` [14], which is the implementation meant for being run from within OpenCL programs. This implementation is hand-optimised for fast execution on graphics cards using OpenCL.

In order to multiply two matrices in BLAS with the C programming language, we need the code presented in Listing 2.12. In this listing, two matrices (A and B) are created, having 3000×2000 respectively 2000×4000 random elements. A result matrix (C) is also created with room for 3000×4000 elements. The equation being computed here is simply

$$\mathbf{C} = \mathbf{AB}$$

or, when using the mandatory α and β variables

$$\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$$

In Listing 2.12 we, however, use $\alpha = 1$ and $\beta = 0$, thus making them irrelevant.

```c
#include <stdlib.h>
#include <stdio.h>
#include "util.hpp"
#define LOW 0
#define HIGH 100

float sgemm_(const char *transA, const char *transB,
             const int *m, const int *n, const int *k,
             const float *alpha, const float *A, const int *lca,
             const float *B, const int *lcb,
             const float *beta, const float *C, const int *lcc);

int main() {
    int n = 3000, k = 2000, m = 4000;

    float A[n*k];
    float B[k*m];
    float C[n*m];
    char transA = 'N', transB = 'N';
    float one = 1.0, zero = 0.0;

    for(int i = 0; i < n*k; ++i) {
        A[i] = (float) randint(LOW, HIGH);
    }

    for(int i = 0; i < k*m; ++i) {
        B[i] = (float) randint(LOW, HIGH);
    }

    sgemm_(&transA, &transB, &n, &m, &k, &one, A, &n, B, &k, &zero, C, &n);

    // defined in util.hpp to print the result matrix
    print_matrix(C, n, m);

    return 0;
}
```

Listing 2.12: An example program using BLAS for multiplying two large matrices.

One should think that the amount of lines of code is rather high for a simple case of multiplying two matrices, however, we must remember that the BLAS library is optimised for many kinds of matrix multiplications. With `sgemm` (the method used in Listing 2.12 to multiply) we can also change how the matrices are indexed as well as transposing none, one, or both matrices prior to multiplying them. We can indeed also choose to only multiply some of one matrix with some of the other, by not given the correct sizes for `n`, `m`, and `k`. This, of course, can also lead to hard-to-find bugs in your programs. We will look more into making this easier for the programmer in Chapter 4.

2.5 Programmer Productivity

Having the fastest executing program for a certain problem comes at a cost. Looking back at Listing 2.12, we need more than 30 lines of code to randomly create and multiply two matrices. That means that every time we multiply two matrices we need to write around 30 lines of code. This also means that every time we multiply two matrices, we might have a bug on one of 30 lines of code. The API for BLAS is especially, and subjectively, not intuitive and "friendly" towards the programmer. Many abstractions exist, but of course, they each have their trade-offs. In a high-level programming language as Python, we can, with the help of NumPy [12], instead use `matmul` to multiply two matrices as seen in Listing 2.13. This listing is both the randomization and multiplication and is fewer lines than the C example from before.

```
import numpy as np
n, k, m = 3000, 2000, 4000
a = np.random.rand(n*k).reshape((n, k))
b = np.random.rand(k*m).reshape((k, m))
c = np.matmul(a, b)
```

Listing 2.13: An example program using `matmul` for multiplying two large matrices.

This all boils down to a term called "programmer productivity" [15]. That is the ratio of output versus input. How much do you gain from what you put into it? Measuring this kind of productivity in a non-manual working context is hard [16]. We cannot just measure lines of code written or products output as you would be able to do with a manufacturing job.

From Casper Jones' book on programmer productivity [17] we get a few pointers towards what this term could be defined as. Jones lists metrics such as *"Maintaining and enhancing existing programs or systems"* and *"The complexity of the program and its data"*.

I believe that BLAS is very complex and it is not easy to maintain and enhance existing programs utilising BLAS. Instead, we should look into methods of wrapping this high-performance library so that we can still gain from its performance without having to worry about its internals. Indeed, this is the topic of Chapter 4.

2.6 Communicating Sequential Processes

Communicating Sequential Processes was created by Sir Charles Antony Richard Hoare [18] to be an algebra used to model networks with process-to-process communication. It can be verified with tools such as FDR [2].

I have some history with CSP, as I did my master's thesis on a theoretical part of exception handling within the algebra [19, 20]. The topic then was a theoretical inclusion of exceptions and exception handling mostly within the algebra, but also implemented in PyCSP. Since the CSP algebra does not allow you to model failure this seemed much needed to be able to model proper networks. A failure here could be hardware or invalid I/O, such as using unsanitised user input, for example, in computing simple division using the input as the denominator. This, of course, is invalid if the user inputs a zero.

During my PhD work, I have taken a slightly different path. Here I will discuss the findings regarding introducing a broadcast type channel in the CSP algebra in Chapter 6 on page 60 as well as a newly implemented CSP framework for the programming language Ruby, called Emit, in Chapter 7 on page 68.

The following sections will contain a brief introduction to CSP algebra and how to work with it.

2.6.1 CSP glossary

Before we look at what CSP is we need some common ground to start from. Table 2.1 contains the notation and meaning of the same that we will use throughout this section and the rest of the thesis.

All of this notation is described in more detail in the following sections.

Notation	Meaning
$a \rightarrow P$	Event a then process P.
$P \parallel Q$	Process P and process Q in parallel.
$P \mid\mid\mid Q$	Process P interleaved with process Q.
$P \,;\, Q$	Process P in sequence with process Q.
$P \,\Box\, Q$	Event from process P or event from process Q (deterministic choice).
✓ (or SKIP)	Successful termination process.
STOP	Deadlock process.
$c!x$	Send the value x on the channel c.
$c?x$	Receive a value on the channel c and bind it to x.
$\langle a, b, c \rangle$	The trace containing the events a, b, and c.

Table 2.1: CSP notation.

2.6.2 Processes, Events and Channel Communications

A CSP network is a group of processes. These processes will agree on a common alphabet, which is their shared knowledge as the network is initiated.

An example of a process in the algebra could be the simple P process that does nothing and deadlocks (STOP).

$$P = \text{STOP}$$

This process will not be of much use, so instead, we can imagine a process Q that engages in an event, a, and then successfully terminates (SKIP)

$$Q = a \rightarrow \text{SKIP}$$

Throughout the rest of the thesis I will use ✓ instead of SKIP as this better symbolises successful termination.

A useful device for determining what a CSP network is doing is the traces of the processes. For the previous process Q the trace would be

$$\langle a \rangle$$

In CSP we also have the concept of "choice". Here we say that the process P is either a or b followed by the appropriate process.

$$P = (a \rightarrow Q \,\Box\, b \rightarrow S)$$

Sometimes we use a bar (|) instead of the square to symbolise choice. The semantic difference is that $\Box$ is deterministic (external) choice, $\sqcap$ is non-deterministic (internal) choice, and | just means any choice.

Processes in CSP can be defined as recursive processes, that is, they are processes that "become" themselves after a certain number of events, for example

$$P = a \rightarrow P$$

This process will keep on engaging in the event a indefinitely.

Two processes that run in parallel with a shared alphabet will rendezvous on the same events. Thus, having the following alphabet (α) and processes (P and Q)

$$\begin{aligned} \alpha &= \{a,\ b\} \\ P &= a \rightarrow c \rightarrow b \rightarrow \checkmark \\ Q &= a \rightarrow b \rightarrow \checkmark \\ P &\ \|_{\alpha}\ Q \end{aligned}$$

the trace of this network will be deterministic and be

$$\langle a, c, b \rangle$$

Hoare has a number of great examples of use cases and networks in [18]. One of them is a broken vending machine, recreated here

$$\begin{aligned} \text{VMC} = \Big(\text{in2p} \rightarrow \Big(& \text{large} \rightarrow \text{VMC} \\ & \Big| \ \text{small} \rightarrow \text{out1p} \rightarrow \text{VMC} \Big) \\ \Big| \ \text{in1p} \rightarrow \Big(& \text{small} \rightarrow \text{VMC} \\ & \Big| \ \text{in1p} \rightarrow \Big(\text{large} \rightarrow \text{VMC} \\ & \qquad \Big| \ \text{in1p} \rightarrow \text{STOP} \Big) \Big) \Big) \end{aligned}$$

This vending machine is broken since we can have a trace that ends in deadlock (STOP). This is done by running the VMC process in parallel with a customer process that has three in1p in a row. In a larger network with many processes, and with many choices, this flaw might be difficult to spot, which is why we verify our networks with tools like FDR [2].

Instead of only rendezvousing on events, CSP processes normally have some kind of communication via channels. With channels, we can communicate variables without knowing them beforehand and thus only having the channel name in the process alphabet. Together with recursion, we can create a network that just counts down a variable from 10 to zero like so

$$\begin{aligned} P(0) &= \checkmark \\ P(x) &= c!\,x \rightarrow P(x-1) \\ Q &= c?\,x \rightarrow Q \,\Box\, \checkmark \\ P(10) &\parallel_c Q \end{aligned}$$

This network can be drawn as seen in Figure 2.2 with P communicating with Q over c. I will use this kind of diagram to represent CSP networks throughout this thesis.

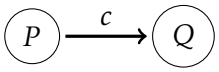


Figure 2.2: A simple network.

Hoare has many more details on this including semantics and proper formal definitions in [18].

2.7 The Programming Language Ruby

Since I will be talking a bit about Ruby [21] in this thesis I felt that a short introduction was in order.

Ruby is an object-oriented programming language, that borrows from its predecessors Perl, Smalltalk, Ada, Lisp, and more. It was created in 1995 by Yukihiro Matsumoto in Japan.

Ruby is an interpreted language with dynamic typing, which means that, as the popular Ruby saying goes: “If it looks like a duck and quacks like a duck, it is a

duck.". This is also dubbed "duck typing". It means that we never check whether objects are of a specific class, but rather look at what that object can do, to say if it fulfils a purpose. This is not type coercion, we do not cast objects to other types.

An example is that a lot of objects have a `to_s` (to string) method, so no matter what kind of object, be it a `String`, `Integer`, `Cat`, `Dog`, or something else, you get, you will still be able to call `to_s` on it.

What we need to know about Ruby for this thesis is that Ruby has modules, classes, and methods. We also need to know that everything in Ruby is an object, and can be treated as such.

We can define new classes and methods as shown in Listing 2.14. Here we create a new class, `Dog`, with a constructor, in Ruby called **`initialize`**. This class has two instance methods, `bark` and `to_s`. We create a new instance of the class with the special class method **`new`**. Notice that `Dog` here takes an argument that is passed to the constructor, namely `name`, which is used to set the instance variable `@name`, so that we can use this further on with the instance.

```ruby
class Dog
  def initialize(name)
    @name = name
  end

  def bark
    "Bow wow!"
  end

  def to_s
    "I am a Dog. My name is #{@name}."
  end
end

spot = Dog.new("Spot")
puts spot.bark # => "Bow wow!"
puts spot.to_s # => "I am a Dog. My name is Spot."
```

Listing 2.14: Introduction to Ruby.

CHAPTER 3

Bohrium Bytecode Optimisation

Because we know a lot about the environment that the code is being executed in, we can use general and various other compiler optimisations tricks on the bytecode sequence. In the Bohrium framework, we have a step called "filters". In this step, we can apply the various filters in any order we choose to. These filters can be turned on and off at runtime just like any of the other components.

3.1 Stupid Maths

One such filter is our aptly named "stupid maths" filter. It looks for statements, that might have been introduced by having run other optimisations. It is called stupid maths because it looks for multiplication with one or addition with zero. These kinds of statements could appear in the Bohrium bytecode as computation carries on.

The filter will change the following into equivalent maths statements:

$$x \cdot 1 \Rightarrow x$$
$$x \div 1 \Rightarrow x$$
$$x \pm 0 \Rightarrow x$$

This is thus just a "cleaning up" filter which removes mathematical no-op bytecodes generated mostly by other optimisations.

Another filter can detect and contract maths statements with constants, that then end up using fewer operations to complete. For example

$$2 + 3 \Rightarrow 5 \quad \text{or} \quad 3 \cdot 4 \Rightarrow 12$$

but it can also detect the use of the same variable and contract those, for example

$$2x + 3x \Rightarrow 5x$$

Here x can be a huge tensor, so contracting the two multiplications into a single one can save a lot of computation time when actually running the code. This simple filter could thus save multiple operations on the data for programmers who might not have optimised this contraction themselves. This type of programming can be seen in Listing 3.1.

For other more convoluted calculations, it is worth noting the case with exponents from Section 2.2 again. Here we had

$$x^n = \underbrace{x \cdot x \cdot \ldots \cdot x}_{n} = \prod^{n} x \quad \text{if } n \in \mathbb{N}$$

This could be turned into a series of multiplications instead of the power function.

$$\begin{aligned} x^{10} &= x^8 \cdot x \cdot x \\ &= x^4 \cdot x^4 \cdot x \cdot x \\ &= x^2 \cdot x^2 \cdot x^2 \cdot x^2 \cdot x \cdot x \end{aligned}$$

```
import bohrium as np
a = np.arange(10)
print(2*a + 3*a)
```

(a) Python code using `a` twice.

```
BH_RANGE a1[0:10:1]
BH_IDENTITY a2[0:10:1] a1[0:10:1]
BH_FREE a1[0:10:1]
BH_MULTIPLY a3[0:10:1] 2 a2[0:10:1]
BH_MULTIPLY a4[0:10:1] 3 a2[0:10:1]
BH_ADD a0[0:10:1] a3[0:10:1] a4[0:10:1]
BH_FREE a3[0:10:1]
BH_FREE a4[0:10:1]
```

(b) The Bohrium IR for Listing 3.1a without the contraction filter.

```
BH_RANGE a1[0:10:1]
BH_IDENTITY a2[0:10:1] a1[0:10:1]
BH_FREE a1[0:10:1]
BH_MULTIPLY a0[0:10:1] 5 a2[0:10:1]
```

(c) The Bohrium IR for Listing 3.1a with the contraction filter.

Listing 3.1: An example of the maths contraction filter.

In Bohrium this could be generated by the Python code shown in Listing 3.2a and the turned into Bohrium IR with Listing 3.2b. In this example, I have shown the original solution in the IR as well. The filter removes the `BH_POWER` and replaces it with five `BH_MULTIPLY`s instead.

```
import bohrium as np
a = np.arange(10)
print(a**10)
```

(a) Python code calculating a^{10}.

```
BH_RANGE a1[0:10:1]
BH_IDENTITY a2[0:10:1] a1[0:10:1]
BH_FREE a1[0:10:1]
# BH_POWER a0[0:10:1] a2[0:10:1] 10
BH_MULTIPLY a0[0:10:1] a2[0:10:1] a2[0:10:1]
BH_MULTIPLY a0[0:10:1] a0[0:10:1] a0[0:10:1]
BH_MULTIPLY a0[0:10:1] a0[0:10:1] a0[0:10:1]
BH_MULTIPLY a0[0:10:1] a0[0:10:1] a2[0:10:1]
BH_MULTIPLY a0[0:10:1] a0[0:10:1] a2[0:10:1]
```

(b) The Bohrium IR for Listing 3.2a.

Listing 3.2: Calculating a^{10}.

3.1.1 Benchmark

To show off the stupid filter we take a large tensor and add `1` to it several times. In normal Python we would have to execute all of these additions one by one, but with Bohrium we can actually collect them into a single addition that is then executed. This reduces some overhead when working with large tensors.

```
import numpy as np
from time import time
t0 = time()

z = np.ones((200, 200, 200))
for _ in range(500):
    z += 1

print(np.sum(z))
print("Ran in {}s".format(time() - t0))
```

Listing 3.3: Stupid maths benchmark.

Listing 3.3 shows a simple Python example that generates a large 200×200×200 tensor of only ones and then add `1` to it 500 times. Listing 3.4[1] shows the Bohrium IR (trace) for this both with and without the filter turned on.

The execution time reported for the two cases are 4.079*s* and 0.149*s* respectively. Thus the stupid maths filter here has increased the general execution performance by a factor 27×. NumPy without Bohrium perform as bad as Bohrium without the filter, about 4.1*s*.

[1] As to not clutter the example with all the tensor indices and strides, these have been removed from the listing.

```
BH_IDENTITY a1 1
BH_ADD a1 a1 1
... # 498 times
BH_ADD a1 a1 1
BH_ADD_REDUCE a2 a1 2
BH_ADD_REDUCE a3 a2 1
BH_ADD_REDUCE a0 a3 0
BH_FREE a3
BH_FREE a2
```

(a) Trace of benchmark without the stupid maths filter.

```
BH_IDENTITY a1 1
BH_ADD a1 a1 500
BH_ADD_REDUCE a2 a1 2
BH_ADD_REDUCE a3 a2 1
BH_ADD_REDUCE a0 a3 0
BH_FREE a3
BH_FREE a2
```

(b) Trace with the stupids maths filters.

Listing 3.4: Traces of the Python program from Listing 3.3.

CHAPTER 4

Automatic Extension Methods

Another way to extend Bohrium's capabilities, other than filters, is with external libraries. It is very common practice to take in other peoples libraries and utilise already created algorithms for other purposes.

In scientific computing, the fastest way of multiplying two matrices is with BLAS [22]. Because of this, everyone who has to multiply two matrices – or other more convoluted computations – use BLAS. BLAS is just a reference, having multiple implementations, such as `cBLAS`, Accelerate[1] [23], `clBLAS` [14], OpenBLAS [24], GotoBLAS [25], and more. The most efficient of these depends on where your data lives. In Bohrium the data from the tensors lives inside Bohrium, and thus it might be infeasible to, for example, just call the BLAS methods with the NumPy/Bohrium array. Doing so might halt the current computation pipeline and yield a copy of the data which could potentially decrease the overall performance of the code execution. Especially if the data lives in Bohrium on a GPGPU and have to be transferred back to the host device prior to being fed into a new GPGPU specific

[1] Apple's implementation of BLAS.

library that copies it back. Instead, what we can do is to have Bohrium call the external libraries internally.

In this section, we look at BLAS and LAPACK as examples. Since these libraries have been through multiple development cycles, they have existed for a long time, their APIs can be quite inflexible. This can be seen with the aforementioned implementations of BLAS. They all have seemingly identical interfaces, but some of the namings are slightly different, causing the users to only have the choice of one of them. As an example, `cBLAS` and Accelerate have the same interface for the same architecture, but two different operating systems. `clBLAS` has a widely different interface but does the same calculations as `cBLAS` albeit on GPGPUs. `LAPACKE`, a LAPACK implementation for Linux, has a different interface than Accelerate, which also implements the LAPACK methods, again on MacOS. This all means, that if the programmer would like to have a flexible implementation of their simulations, they would have to implement the calls with all these different implementations in mind.

The BLAS reference is from a time when Fortran was at the height of programming languages. Fortran is column-major in its memory management, unlike many programming languages today. This can still be seen in the BLAS implementations, where the programmer needs to take special care about how the program manages its memory.

A simple call to `cBLAS` multiplying the matrices **A** and **B** can be seen in Listing 4.1. There are a lot of parameters and things to take note of issuing such a simple operation in `cBLAS`. The equivalent statement from Python can be seen in Listing 4.2. Both of these listings assume that **A** and **B** have been initialised elsewhere. It should be fairly simple to see that the Python version is easier to follow and to later modify. In this concrete example, both will run with the same performance, since the NumPy version just translates to the C version behind the scenes.

Bohrium also implements the `matmul` method, however, here it is done the regular way, with the naïve $O(n^3)$ algorithm, first multiplying and then summing along the various columns and rows in the matrices. BLAS uses Strassen's algorithm [26] which is $O(n^{log_2 7}) \approx O(n^{2.807})$. This means that Bohrium will be slower as n increases. Instead of implementing Strassen's algorithm for Bohrium, we want to just link Bohrium against a BLAS library.

```
// C := alpha*op(A)*op(B) + beta*c
// where op(X) is either X or X^T

cblas_sgemm(
  CblasRowMajor, // Memory mangement
  CblasNoTrans,  // Transpose A?
  CblasNoTrans,  // Transpose B?
  m,             // Number of rows of op(A)
  n,             // Number of columns of op(B)
  k,             // Number of columns/rows of op(A)/op(B)
  1.0,           // Alpha
  A_data,        // Array of size m*k
  k,             // First dimension of A / Stride of A
  B_data,        // Array of size k*n
  n,             // Stride of B
  0.0,           // Beta
  C_data,        // Result array of size m*n
  n              // Stride of C
);
```

Listing 4.1: cBLAS general matrix multiplication.

```
import numpy as np
np.matmul(a, b)
```

Listing 4.2: Python (NumPy) matrix multiplication.

This is essentially what the extension methods are and what NumPy is already doing, to achieve the improved performance.

4.1 Code Generation

Since Bohrium has different back ends, we need to be able to link to the correct libraries, as well as use the correct interfaces. In practice, we create an internal API, that we can always use, and that we can then change to point to other libraries. In Bohrium this is implemented as a flush of the current Bohrium instruction set, followed by a call into the C++ back end. The API for all the extension methods are the same here, being called with a specific name and all the arrays involved.

All these wrapper APIs are generated when Bohrium is first compiled or installed on your machine. Here Bohrium will find all the paths that it needs in order to later link against libraries when JIT-compiling the kernels. If the user has many different, for example, BLAS implementations installed, Bohrium will gather them all, not choosing anything yet, only choosing at runtime.

The automatically generated wrapper functions are the heart of the extension methods. They are generated in three steps. First, we define a JSON-file with all the methods and their options. Second, we create various templates for the code generator to fill out. Lastly, we run the actual generator, which combines several templates into a single source file comprising the extension method.

We could write all the methods for BLAS by hand, however, there are many and this will quickly become a tedious job, as they are quite similar.

4.1.1 JSON

First, we create a JSON-file will all the options that we need. A snippet of such a JSON-file can be seen in Listing 4.3. The dots here represent the other methods of BLAS.

```
{
  "methods": [
    {
      "name": "gemm",
      "types": ["s", "d", "c", "z"],
      "options": [
        "layout", "notransA", "notransB",
        "m", "n", "k",
        "A", "B", "C"
      ]
    },
    "..."
  ]
}
```

Listing 4.3: JSON-file with options for `gemm`.

When turning all the options into booleans for later use, we can create the four different types of General Matrix Multiplication (`gemm`) methods for the different single- and double-precision as well as complex and double complex numbers, that is `sgemm`, `dgemm`, `cgemm`, and `zgemm`.

Later, we can create shortcuts to these generated methods that can then be accessed from the Bohrium front end, here Python.

4.1.2 Templating

After reading in the JSON-file, we parse the various template files that might accompany it. In this system, we have four different templates. We are abusing the string interpolation capabilities of a Python library, called `pyratemp` [27], in

order to have file inclusion in their templating language. More files could easily be added. The files in question here are: `header.tpl`, `body.tpl`, `body_func.tpl`, and `footer.tpl`.

`pyratemp` uses a syntax like `@!VAR!@` to substitute in the contents of that variable. Conditionals are done similar to HTML comments that is `<!--(if BOOL)-->` ... `<!--(end)-->`.

The header and footer contain the boilerplate code needed for Bohrium to understand the extension methods. These are the C inclusions, the namespace definitions, and so on. A sample of a header template can be seen in Listing 4.4.

```
#include <bh_extmethod.hpp>

#if defined(__APPLE__) || defined(__MACOSX)
  #include <Accelerate/Accelerate.h>
#else
  #include <cblas.h>
#endif

#include <stdexcept>

using namespace bohrium;
using namespace extmethod;
using namespace std;

namespace {
  @!body!@
} /* end of namespace */
@!footer!@
```

Listing 4.4: The `header.tpl` Bohrium uses for BLAS.

In this template we have `@!body!@` which tells `pyratemp` to insert the `body.tpl`, which can be seen in Listing 4.5. The same is done with `@!footer!@`. In Listing 4.5 we first load in the two matrices `A` and `B` from the instruction operands. The `B` matrix is only allocated if actually present and used. This is where we use the options from the JSON-file, which has given us the `if_B` boolean.

The lines with the switch statement have been omitted from the example, but here we have a switch case statement, that includes the template for the `body_func.tpl` file seen in Listing 4.6. There is a lot of conditionals in this example, but if we look at the JSON-file from Listing 4.3, we see that only `layout`, `notransA`, and `notransB` are set for `gemm`. This means, that only those lines will be generated, the rest will be omitted from the end result.

```
struct @!uname!@Impl : public ExtmethodImpl {
public:
  void execute(bh_instruction *instr, void* arg) {
    // All matrices must be contigous
    assert(instr->is_contigous());

    // A is m*k matrix
    bh_view* A = &instr->operand[1];
    // We allocate the A data, if not already present
    bh_data_malloc(A->base);
    void *A_data = A->base->data;

    <!--(if if_B)-->
    // B is k*n matrix
    bh_view* B = &instr->operand[2];
    // We allocate the B data, if not already present
    bh_data_malloc(B->base);
    void *B_data = B->base->data;
    <!--(end)-->

    ... // switch
  } /* end of execute method */
} /* end of struct */
```

Listing 4.5: The `body.tpl` Bohrium uses for BLAS.

Listing 4.7 contains the result of running the above code generation, with the JSON and template files already mentioned. It should look fairly similar to Listing 4.1 because that is essentially what we aim to automatically generate.

4.2 Results

The results in this section all come from experiments run on an Intel® Core™ i7-3770 3.4 GHz processor with a NVIDIA GeForce® GTX 680 graphics card for the OpenCL parts. The system was running Ubuntu Server 14.04 with Python 2.7 and NumPy 1.12.1. OpenBLAS 0.2.19 was used for the `cBLAS` interface and `lapacke` 3.5.0 for the LAPACK interface. Both NumPy and Bohrium were linked against these same packages. All the numbers come from an average of ten consecutive runs.

4.2.1 General Matrix Multiplication

We have already seen how the `gemm` from BLAS works and how I have automatically generated wrapper methods for Bohrium to use.

```
case @!utype!@: {
  @!alpha!@
  @!beta!@
  cblas_@!t!@@!name!@(
    <!--(if if_layout)-->    CblasRowMajor, <!--(end)-->
    <!--(if if_side)-->      CblasLeft,     <!--(end)-->
    <!--(if if_uplo)-->      CblasUpper,    <!--(end)-->
    <!--(if if_notransA)--> CblasNoTrans,  <!--(end)-->
    <!--(if if_transA)-->    CblasTrans,    <!--(end)-->
    <!--(if if_notransB)--> CblasNoTrans,  <!--(end)-->
    <!--(if if_transB)-->    CblasTrans,    <!--(end)-->
    <!--(if if_diag)-->      CblasUnit,     <!--(end)-->
    <!--(if if_m)-->         m,             <!--(end)-->
    <!--(if if_n)-->         n,             <!--(end)-->
    <!--(if if_k)-->         k,             <!--(end)-->
    @!alpha_arg!@,
    (@!blas_type!@*)(((@!type!@*) A_data) + A->start),
    k,
    <!--(if if_B)-->
    (@!blas_type!@*)(((@!type!@*) B_data) + B->start),
    n<!--(if if_C)-->,<!--(end)-->
    <!--(end)-->
    <!--(if if_C)-->
    @!beta_arg!@,
    (@!blas_type!@*)(((@!type!@*) C_data) + C->start),
    n
    <!--(end)-->
  );
  break;
}
```

Listing 4.6: The `body_func.tpl` Bohrium uses for BLAS.

Here we will multiply two large matrices. In this example we let **A** be a matrix of size 2000×3000 and **B** have size 3000×4000. The result matrix $\mathbf{C} = \mathbf{AB}$ will thus have size 2000×4000. All of this is done in 32-bit floating point numbers and both **A** and **B** will be filled with random numbers. Thus, **A** will be 24MB, **B** will be 48MB and the result matrix **C** will consist of 32MB of random numbers.

For the NumPy version, we use the `matmul` method to compute the matrix multiplication. The same code is used for Bohrium, with the import switched to Bohrium's. We will have both a version with the old setup as well as one with the new extension methods enabled. For the OpenCL version of Bohrium we just use a different stack, namely the OpenCL one and Bohrium will automatically link against the new OpenCL extension method instead. All the results can be seen in Figure 4.1.

It should come as no surprise that the NumPy and OpenBLAS version perform the same since internally NumPy is just calling on OpenBLAS to compute the multiplication. Also, the first Bohrium is without this new extension method, so it

```
cblas_sgemm(
  CblasRowMajor,
  CblasNoTrans,
  CblasNoTrans,
  m,
  n,
  k,
  1.0,
  (bh_float32*)(((bh_float32*) A_data) + A->start),
  k,
  (bh_float32*)(((bh_float32*) B_data) + B->start),
  n,
  0.0,
  (bh_float32*)(((bh_float32*) C_data) + C->start),
  n
);
```

Listing 4.7: The automatically generated cBLAS code for Bohrium's extension method.

should perform much worse as already described. The second Bohrium column is with the extension enabled, so this too should perform just as well as NumPy and OpenBLAS.

The clBLAS version should be much faster, as we are computing the matrix multiplication on a graphics card now. The Bohrium version with OpenCL is worse because we are naïvely doing the same computations, which does not yield good results on the GPU. The last column has Bohrium with OpenCL and the clBLAS extension method linked. This is running the same code as the NumPy version, but Bohrium now knows to run it on the GPU with clBLAS linked thus it is running at a much faster speed.

All in all, we get a speed-up of a factor of about 4× without changing the original code.

4.2.2 LAPACK Solver

Since not all science is multiplying matrices, I will also show, that the Bohrium extension methods work for other libraries. Let us take a look at the gesv solver from LAPACK. This solver solves the equation $\mathbf{A}x = \mathbf{B}$ for x, where $\mathbf{A}$ and $\mathbf{B}$ are matrices and x is a vector. The results shown in Figure 4.2 solves a system of 5000 random equations that are assumed to be linearly independent.

With NumPy we can use `numpy.linalg.solve(...)` from the `linalg` package included in NumPy. The LAPACK version is hand-coded in C and the Bohrium extension method just uses the same techniques as already described to incorporate

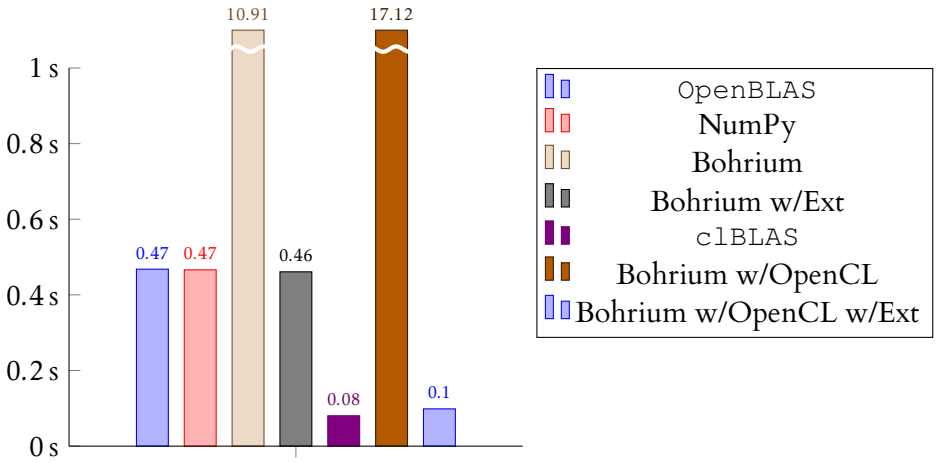


Figure 4.1: General matrix multiplication (**C** = **AB**) for multiple implementations.

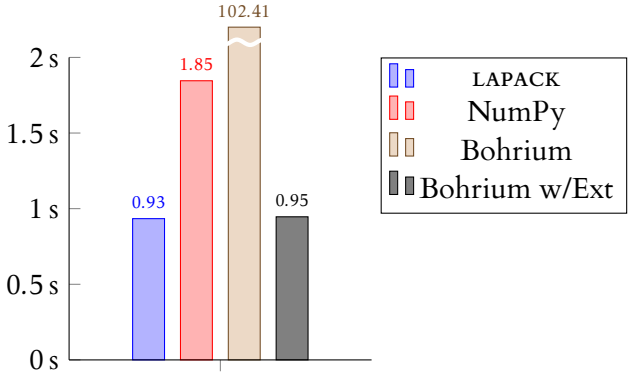


Figure 4.2: System of equations solver (**A***x* = **B**) for multiple implementations.

LAPACK into extension methods. Bohrium again is as fast as LAPACK when these are enabled. It seems that NumPy is doing some pre-computations that slow it down a bit.

When using Bohrium for array-programming we can thus use the same NumPy code, but gain performance without losing the programmer productivity aspects.

CHAPTER 5

Bohrium.rb: The Ruby Front End

Ruby has always lagged behind in scientific circles. I believe this to be a shame since Ruby is often cited as being a fun language to program in. This, together with programmer productivity, should help unfamiliar programmers program more advanced simulations. Other than NumPy, Ruby has a lot of the same capabilities as Python, which has seen a great boost in scientific areas in the last couple of years [28].

Ruby does have scientific packages, however, they are not as feature complete and as fast as NumPy is for Python. More time should be invested in this branch of programming in the Ruby community if we want Ruby to be used in broader circles.

In the Ruby standard library (STL) there is a `matrix` class with the most basic matrix operations, such as matrix products, determinants, and LU decomposition to name a few. There is also a Ruby gem[1] called Numo/Narray that tries to emulate NumPy for Ruby. This seems to be the most current and still maintained gem.

[1] A self-contained Ruby library.

In this chapter, I will present my own gem for handling tensors as well as comparing its performance to the STL and Numo/Narray. For this, I have implemented a front end for Bohrium in Ruby, so that we can have Ruby generate JIT-compiled OpenMP and OpenCL kernels, and thus run them faster than STL and Numo/Narray. This is all done by implementing a Ruby bridge using C++ that then uses the Bohrium C++ bridge as seen back in Figure 2.1 on page 24.

5.1 Bohrium.rb

The Ruby front end is called `bohrium`, so is the framework and the Python package. In order to distinguish these, I will call the Ruby gem Bohrium.rb in the following sections.

The Python package for Bohrium overwrites already established NumPy classes, namely the array class. For Bohrium.rb instead I have chosen to create a new array class, named `BhArray`, which works similar to NumPy's arrays. This is done because the original array class in Ruby does not have the same interface and behaviour as the NumPy ones do. For example, as we will see later, you cannot directly add two arrays elementwise, as you would expect to be able to do with matrices and vectors in other maths settings. In the future, I might choose to "monkey patch"[1] the actual `array` or `matrix` class from the STL, so that a transition from STL to Bohrium.rb will be more seamless.

Of course, you lose some of the ease-of-use that I have already written about with regards to Bohrium and its NumPy version, but for now, the Bohrium array lives in the `BhArray` class in the Ruby gem.

In order to use Bohrium from Ruby we simply use the Ruby require statement: `require "bohrium"`.

5.2 Compiling and Executing

For now, Bohrium.rb only lives on the `ruby-frontend` branch[2] of the original Bohrium project[3]. When you compile Bohrium, you have to add `-D RUBY_BRIDGE=ON` in order for CMake to enable the Ruby Bridge and thus the Ruby front end. This

[1] The common term used in the Ruby community about opening a class and adding new features to it.

[2] https://github.com/omegahm/bohrium/tree/ruby-frontend

[3] https://github.com/bh107/bohrium

setting is off by default, so as to not compile more than necessary for people who do not need Ruby.

When executing Bohrium and Ruby together we first run the Ruby code. As Ruby is an interpreted language this is done line by line. If we encounter Bohrium.rb operations we save these for later executing within the Bohrium runtime to be lazily evaluated when needed. We collect as many operations as we can until we get a side effect that uses the result. When this happens, we let the Bohrium runtime do all the computations needed which are then passed back the result we need. These operations might be split into several kernels within Bohrium, but that is of no concern to the user. When the result is given back, it can be utilised as a regular Ruby `array`. Figure 5.1 gives an overview of this pipeline as well.

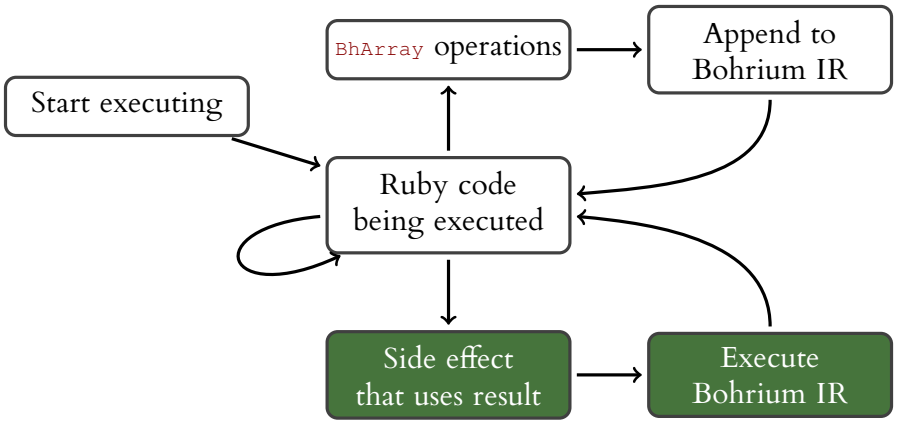


Figure 5.1: How Bohrium works with Ruby.

In all the examples in this section, we print the result of the array operations. This in itself is a side effect, which will then trigger Bohrium to execute said operations.

5.3 Initialising a `BhArray`

The constructor method for `BhArray` takes an STL array as an argument and will convert it into a proper Bohrium array. The argument array is copied into Bohrium memory, element by element, and can be used as a Bohrium array afterwards. An example of the constructor method, which is invoked with `new`, can be seen in Listing 5.1.

```
ruby_ary = [1, 2, 3]
bh_ary = BhArray.new(ruby_ary)

puts ruby_ary # => [1, 2, 3]
puts bh_ary   # => [1, 2, 3]
```

Listing 5.1: Initialise `BhArray` with Ruby STL array.

It should be noted, that the bottom print out of the `bh_ary` here actually converts the data and exposes a string representation of its internals when using the print method `puts` from Ruby. Thus, this is not just the same array, but rather a similar representation of the internals of it. Bohrium.rb does not retain a reference to the original array object, which means that if you later change the original array, the `BhArray` will not have the same changes applied and vice versa.

Instead of supplying Bohrium.rb with an already instantiated array, which we then have to loop through to copy, we can have Bohrium.rb create the array itself. This will, of course, be faster for large tensors and should therefore always be used, unless you have a static array that you need to feed into the system. In Listing 5.2 we see three different ways of creating a `BhArray`. The first fills the array with ones, the second creates a matrix of zeros and the last creates a sequence from 0 to 9.

```
# Create 5x1 array of ones
ary1 = BhArray.ones(5, 1)
puts ary1 # => [1, 1, 1, 1, 1]

# Create 3x2 array (matrix) of zeros
ary2 = BhArray.zeros(3, 2)
puts ary2 # => [[0, 0], [0, 0], [0, 0]]

# Create a range of length 10 starting from 0
ary3 = BhArray.arange(10) # => could also be BhArray.seq(10)
puts ary3 # => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listing 5.2: Initialising `BhArray` using various class methods.

Bohrium.rb uses `arange` here in the same way that NumPy does, however, the method is also aliased as `seq`, which seems to be a more sensible name.

5.4 Using `BhArray`

All the common maths operations that are available in Bohrium is also present in Bohrium.rb. We can thus add two `BhArray`s elementwise, subtract them, multiply,

or other such operations. We can also take the cosine of every element, bitwise and them, or similar. None of these is directly available in the Ruby STL.

Since Bohrium can overwrite the argument array, I have added so-called *bang methods*[1] for the common maths operations. The maths operations are also aliased to their appropriate symbols where available. We can thus look at Listing 5.3 and see both `+`, `add`, and `add!` in action. As seen in this Listing, this cannot be done in the Ruby STL, where we instead will concatenate the arrays.

```
bh_ary1 = BhArray.ones(5, 1)
bh_ary2 = BhArray.ones(5, 1)

result_ary = bh_ary1.add(bh_ary2)
puts result_ary # => [2, 2, 2, 2, 2]

result_ary = bh_ary1 + bh_ary2
puts result_ary # => [2, 2, 2, 2, 2]

bh_ary1.add!(bh_ary2)
puts bh_ary1 # => [2, 2, 2, 2, 2]

# This cannot be done in Ruby STL
rb_ary1 = [1, 1, 1, 1, 1]
rb_ary2 = [1, 1, 1, 1, 1]
rb_ary3 = rb_ary1 + rb_ary2
puts rb_ary3 # => [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Listing 5.3: Adding `BhArray`s.

Other than regular maths operations, Bohrium.rb also supports reduction methods, such as `add_reduce`, which we will use for the performance results.

5.5 Views into Arrays

One of the common tasks in array-programming involves looking at views or windows of a tensor. This usually means, that you would like a section of the entire tensor, that you can then work with and "put back". We thus want a reference to a piece of memory, without having to pull out the entire array.

In Bohrium.rb this is done simply by indexing into the `BhArray`. You can use a single index, to return a single element of a one-dimensional tensor, or ranges to get multiple values, as seen in Listing 5.4.

Here we also see the essential method `reshape` that take an already initialised `BhArray` and modifies it to a different shape. Since all arrays are stored in one-

[1] Called so because of the exclamation mark, or the *bang*.

```
bh_ary = BhArray.seq(10)
puts bh_ary[4] # => [4]

bh_ary = BhArray.seq(25).reshape([5, 5])
puts bh_ary             # => [[ 0,  1,  2,  3,  4],
                        #     [ 5,  6,  7,  8,  9],
                        #     [10, 11, 12, 13, 14],
                        #     [15, 16, 17, 18, 19],
                        #     [20, 21, 22, 23, 24]]

puts bh_ary[0..1, 0..1] # => [[ 0,  1],
                        #     [ 5,  6]]
```

Listing 5.4: Indexing with a single index.

dimensional memory, this is only for the programmers to be able to reason about the tensors in a regular way. You can add as many dimensions as you would like with `reshape`. Here we will only concern ourselves with one- and two-dimensional arrays, that is vectors and matrices. In the view part, we index with two ranges, the first one specifying the first dimension and the second the second dimension. In a standard matrix, the first dimension is the rows and the second is the columns.

If we want, for example, all of the elements in one dimension but only some in the second, we can use **`true`** instead of a range or integer. This instructs Bohrium.rb to create a view with all the elements.

We can also use views to set the array data as seen in Listing 5.5. The first example sets a 2×2 view to the constant value 2. The second example also sets a 2×2 view, but this time we use another `BhArray` object to set it. This second array needs to be the same size and shape as the view or else you would get a runtime error.

```
bh_ary1 = BhArray.seq(9).reshape([3, 3]) # => [[0, 1, 2],
                                         #     [3, 4, 5],
                                         #     [6, 7, 8]]
bh_ary1[0..1, 0..1] = 2
puts bh_ary1 # => [[2, 2, 2],
             #     [2, 2, 5],
             #     [6, 7, 8]]

bh_ary2 = BhArray.seq(9).reshape([3, 3])
bh_ary2[0..1, 0..1] = BhArray.new([0, -1, -3, -4]).reshape([2, 2])
puts bh_ary2 # => [[ 0, -1,  2],
             #     [-3, -4,  5],
             #     [ 6,  7,  8]]
```

Listing 5.5: Setting a view.

5.6 Performance Results

In this section, we will be performing two performance benchmarks. These benchmarks are run on a MacBook Pro with a 3.1 GHz Intel® Core™ i7 processor and 16 GB 1867 MHz DDR3 RAM. The benchmarks are an average of 10 consecutive runs and are all run with the default settings for both Bohrium and Numo/Narray.

5.6.1 Add

In this first benchmark, we will simply add a large set of numbers. This is done by creating two vectors and elementwise adding them, as we saw back in Listing 5.3 on page 54.

In Ruby, we usually have many ways of doing a simple operation. Adding numbers is no different. I asked the local Copenhagen user group for Ruby "Copenhagen Ruby Brigade" what their preferred way of adding numbers in arrays was. They came up with Listing 5.6, which uses the `zip` method in Ruby to join two arrays elementwise and then the `reduce` method to sum the sub-arrays.

```
ary1 = [2] * n # Create an array of length n comprised of 2s
ary2 = [3] * n # Create an array of length n comprised of 3s
ary1.zip(ary2).map { |i| i.reduce(&:+) }
```

Listing 5.6: Adding with the Ruby STL (`zip`).

When using the Ruby STL matrix class, we will define a matrix with only one row. The same is true for both Numo/Narray and Bohrium.rb. Listing 5.7, Listing 5.8, Listing 5.9, and Listing 5.10 shows the solution using matrices in the three different libraries. I show two different ways of doing it with Bohrium.rb, the last one being the more reasonable because we generate the arrays in Bohrium memory.

```
require "matrix"
m1 = Matrix[[2] * n]]
m2 = Matrix[[3] * n]]
m1 + m2
```

Listing 5.7: Adding with STL matrix.

Figure 5.2 shows the speed-ups compared to the original `zip` solution from Ruby. For the very small cases of having $n = 10^1$ or $n = 10^2$ (not shown here),

```
require "numo/narray"
m1 = Numo::Int64.new(n, 1).fill(2)
m2 = Numo::Int64.new(n, 1).fill(3)
m1 + m2
```

Listing 5.8: Adding with Numo/Narray.

```
require "bohrium"
ary1 = [2] * n
ary2 = [3] * n
a = BhArray.new(ary1)
b = BhArray.new(ary2)
a + b
```

Listing 5.9: Adding with Bohrium.rb (init).

```
require "bohrium"
a = BhArray.ones(1, n, 2)
b = BhArray.ones(1, n, 3)
a + b
```

Listing 5.10: Adding with Bohrium.rb (ones).

both Numo/Narray and Bohrium.rb are slower than the Ruby STL. This is because of the inherent overhead that both libraries have when moving data into their own data structures. For the largest case, $n = 10^8$, we see that Numo/Narray is 2.2× faster than `zip`, while Bohrium.rb is 2.8× faster.

5.6.2 Sum

The other benchmark involves summing along an axis instead of just summing two arrays. This is thus summing along multiple columns (vertical arrays) instead. In this benchmark, I have fixed the number of columns to 1000 and then have n be the number of rows again, increasing it as the experiment goes on. The numbers in the matrix we are creating will be consecutive starting from 0. This benchmark thus tests both the creation of sequential numbers as well as the ability to sum memory, that lives in row-major order.

Listing 5.11, Listing 5.12, and Listing 5.13 shows the various implementation of this benchmark. The Ruby version is again a result of asking the Copenhagen Ruby Brigade how they would do it.

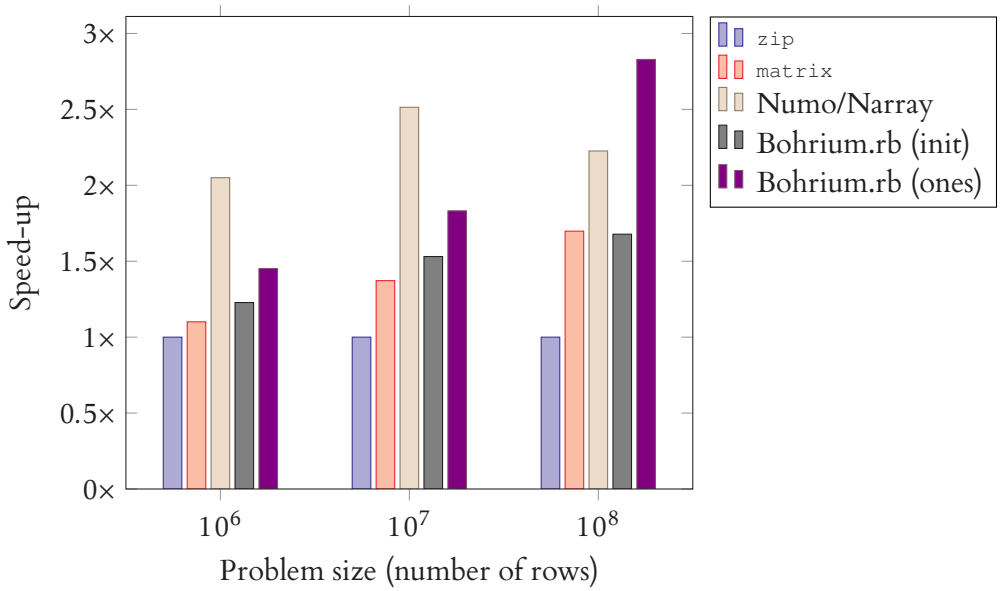


Figure 5.2: ADD benchmark results.

The Numo/Narray and Bohrium.rb solutions are almost identical, as they both agree on how matrices should be created and summed. In Bohrium.rb we have a method called `add_reduce` that sums over the axis given, here 0, as this is the first.

```
i = -1
ary = Array.new(r) { Array.new(c) { i += 1 } }
result = Array.new(c) { 0 }
c.times do |col_idx|
  r.times do |row_idx|
    result[col_idx] += ary[row_idx][col_idx]
  end
end
```

Listing 5.11: SUM benchmark – Ruby STL.

```
require "numo/narray"
result = Numo::Int32.new(r * c).seq.reshape(r, c).sum(axis: 0)
```

Listing 5.12: SUM benchmark – Numo/Narray.

The results from Figure 5.3 show that Bohrium.rb is approximately 43 times faster than the Ruby STL for $n = 10^5$. The reason we see a decrease in speed-up

```
require "bohrium"
result = BhArray.seq(r * c).reshape([r, c]).add_reduce(0)
```

Listing 5.13: SUM benchmark – Bohrium.rb.

at $n = 10^6$ could be the fact that working with $10^6 \times 1000$ 32-bit integers takes up more than the 16 GB of memory available on the machine.

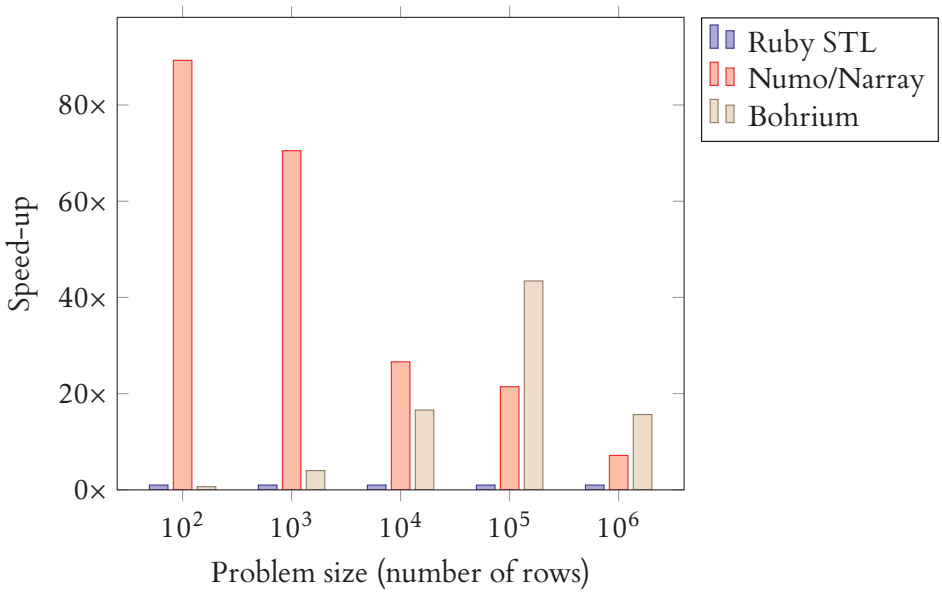


Figure 5.3: SUM benchmark results.

So again, we see that where Numo/Narray *only* outperform the Ruby STL by a factor of 7×, Bohrium.rb is actually 15.5× faster, double the speed. For the second largest matrix summed we see the aforementioned 43.4× from Bohrium.rb, but only 16.6× from Numo/Narray.

CHAPTER 6

Broadcasting

Broadcasting is the act of emitting a message from one place and then potentially receiving it in several other places. In this chapter, we will discuss what this means and whether these processes may choose not to receive the message if they deem it unimportant.

When rejecting messages, we can think of it as physical broadcasting. Think of a man standing in a town square. He is yelling his opinions for all to hear. Most people will ignore him and only some will listen to what he is saying and use the message.

As a technical term "broadcast" is also used in radio and television to mean something that may or may not be received by other devices. The radio or TV show is there to be received if you so choose.

In computers, we tend to say that broadcast means communicate with *everybody else*. If I am to broadcast something, I would like everybody else to at least hear the message.

In the next sections, I will give some examples of models that use broadcast in different ways as well as theory and verification that uses broadcast with different models in CSP.

6.1 Models

Broadcasting can mean a lot of things. As already stated, we have a difference already between physical and technical broadcasts. We can have several different models of broadcast sources. Some of these are listed here, where A and B denote broadcast sources:

6.1.1 Simple broadcast

This is what we mean when we talk about the physical broadcast. A simple broadcast is one participant broadcasting and many who are able to listen with no delivery guarantees. A broadcast that is received by no one is still considered a valid and correct broadcast.

6.1.2 Reliable broadcast

A reliable broadcast *must* be received by all recipients to be correct. We do not guarantee ordering that is, if several participants are broadcasting at the same time, we do not care if messages from A arrive before or after messages from B.

6.1.3 Atomic broadcast

When we want to guarantee ordering we use atomic broadcasts. Two participants can still both be broadcasting at the same time, but all processes must either receive as A then B or B then A. This will be hard to do in hardware without some form of lock or barrier. In our CSP algebra, we will see this emulated with a broadcast controller, that will not accept new messages before everybody has received the old one.

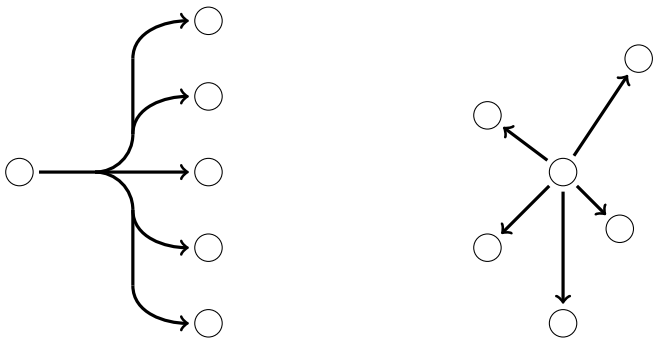
6.1.4 Causal broadcast

The causal broadcast [29] ensures that if the broadcaster of message B has received message A, then no other participants can receive message B before A.

6.1.5 Synchronous and asynchronous broadcasts

Apart from ordering, we might want all the messages to arrive at the same time, that is synchronous. A synchronous broadcast is rarely used in distributed systems, as this would, in fact, be a one-to-all message. An asynchronous broadcast seems

more intuitive. I have tried to illustrate these two types of broadcasts within a network in Figure 6.1.



(a) A synchronous broadcast network. (b) An asynchronous broadcast network.

Figure 6.1: The two types of broadcasts.

6.2 Broadcasting in CSP

In Hoare's CSP [18] only two processes can communicate at a time. If we disregard this, we can very simply model broadcast over n processes in CSP as follows.

$$
\begin{aligned}
S &= m!\,x \rightarrow S' \\
P_i &= m?\,x \rightarrow P_i' \\
S &\parallel \left(\mathop{\Vert}_{i=0}^{n-1} P_i \right)
\end{aligned}
$$

In this and the following algebraic equations, primed processes are processes that go on to live on afterwards, without us specifying exactly what they do. They may become the ✓ process or they may become something else.

If we want to model broadcasting as described with CSP terms, we must do so with a two-phase commit. We can create a network of processes, with one sender, one controller and n receivers.

The CSP algebra needed for this to work is as follows

$$
\begin{aligned}
S(x) &= m!\,x \rightarrow m_{\text{ACK}} \rightarrow S' \\
C &= m?\,x \rightarrow \left(\Vert_{i=0}^{n-1} c_i!\,x \rightarrow \checkmark \right); m_{\text{ACK}} \rightarrow C' \\
P_i &= c_i?\,x \rightarrow P_i' \\
S(\texttt{"hello"}) &\parallel C \parallel \left(\Vert_{i=0}^{n-1} P_i \right)
\end{aligned}
$$

A diagram of the communication can be seen in Figure 6.2.

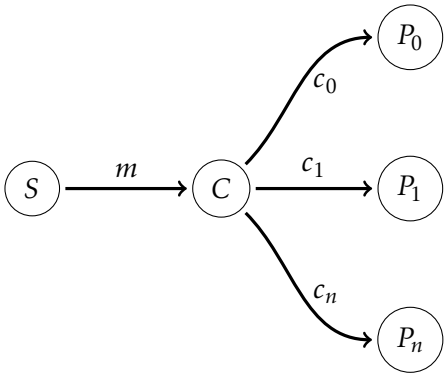


Figure 6.2: Broadcast in CSP.

Here the S process will start by sending a message, `"hello"`, over the m channel. The controller C will receive this, and immediately start interleaving messages on all the c_i channels. The P_i processes will synchronise on these. Once all P_i have received the message, an ACK will be sent back to the S process, signifying the end of the communication and the two-phase commit.

This two-phase commit broadcast model can be verified to be deadlock-free and behave as stated with FDR[1]. The first assertion verifies the model to be deadlock-free while the second assertion, with the `\\`, verifies that the trace of the system, with the events `m` and `mack` hidden, is as stated. The CSP code necessary to verify this for three P_i processes is presented in Listing 6.1. Since we interleave the communications FDR's state diagram will contain all the permutations of ways to communicate in order to verify that it is indeed deadlock-free. This, of course,

[1] In the published paper [30] we had an additional c_{ACK} channel after the c_i channels. This is unnecessary and furthermore actually introduced an error in the later FDR verification.

grows with n, so the larger n the longer time FDR will have to use to verify the system.

The processes in Listing 6.1 all go back to being themselves again. This will still work, as with our primed processes, because remember, we said they could be anything, even themselves. It should be obvious that this works with larger n as well, the only downside is the state explosion as already discussed.

```
N = 3
PNAMES = {0..N-1}
MSG = {"hello"}

channel mack
channel m:MSG
channel c:PNAMES.MSG

S(x) = m!x -> mack -> S(x)
C = m?x -> (||| i:PNAMES @ c.i!x -> SKIP) ; mack -> C
P(i) = c.i?x -> P(i)

SYSTEM(x) = S(x) [|{|m, mack|}|]
            (C [|{|c|}|] || i:PNAMES @[{|c.i|}] P(i))

assert SYSTEM("hello") :[deadlock free [F]]
assert SYSTEM("hello") \ {|m, mack|}
  :[has trace [T]]: <c.0."hello", c.1."hello", c.2."hello">
```

Listing 6.1: CSP algebra used to verify our broadcast semantic in FDR.

This semantics, of course, means that the broadcast controller, here C, must know how many processes are receiving the message, thereby having their c_i channels in its alphabet.

Instead of having only S send a message we could want all processes to both be able to send and receive a broadcasted message and thus avoiding the need for an explicit sender. This can be done with the following algebra:

$$P_i(x) = \left(c_i!\,x \to c_{i,\text{ACK}} \to P_i'\right) \Box \left(c_i?\,x \to P_i''\right)$$

$$C = \Box_{i=0}^{n-1} \left(c_i?\,x \to \left(\parallel_{\substack{j=0 \\ j \neq i}}^{n-1} c_j!\,x \to \checkmark \right); c_{i,\text{ACK}} \to C' \right)$$

$$C \parallel \left(\parallel_{i=0}^{n-1} P_i(\texttt{"hello"}) \right)$$

A diagram illustrating this, with P_0 sending a message, can be seen in Figure 6.3. Note that the arrow of P_0 points the other way, signalling that it is passing its message to C.

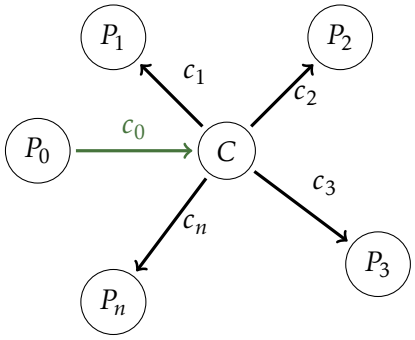


Figure 6.3: Broadcast in CSP without a sender.

Each P_i process is able to write to the controller at any time and can likewise read from the controller, if there is a message ready, at any time. This will work since each P_i process is the only one allowed to write on c_i before a broadcast is initiated. Once a broadcast has begun, no other P_i will be able to write on c_i, because only C was listening and willing to participate in the communication.

One could imagine that P_i' and P_i'' would have the same read and write incorporated into their definitions. Of course, this now means that all P's wants to communicate at any time which might not always be the case. Special care should be used when using the above two-phase commit for describing a network.

6.2.1 Broadcasting with Mailboxes

A different scenario for broadcasting, where the message is stored for later reading, is mailboxes. Here the message is broadcast to a mailbox that stores the message. A mailbox is dedicated to each process and can only be accessed by that process. This is how message passing works in programming languages like Erlang [31] and Elixir [32].

If we allow these mailboxes not to block, they are essentially endless buffers, that the process can read from at any time it would like. When working with mailboxes in CSP we can have the writer process be blocked until all mailboxes have acknowledged that they have received the message with the following algebra:

$$
\begin{aligned}
S(x) &= b!\,x \to b_{\text{ACK}} \to S' \\
B &= b?\,x \to \left(\Vert_{i=0}^{n-1} m_i!\,x \to \checkmark \right); b_{\text{ACK}} \to B' \\
M_i(\emptyset) &= m_i?\,y \to M_i\,(y) \\
M_i(x : xs) &= (m_i?\,y \to M_i\,(x : xs : y)) \,\Box\, (c_i!\,x \to M_i\,(xs)) \\
P_i &= c_i?\,x \to P_i' \\
S\,(\texttt{"hello"}) &\parallel B \parallel \left(\parallel_{i=0}^{n-1} M_i(\emptyset) \parallel P_i \right)
\end{aligned}
$$

Figure 6.4 shows a diagram of the mailboxed communications within CSP.

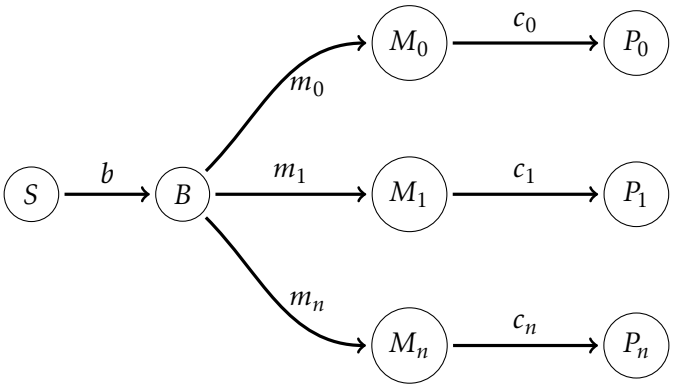


Figure 6.4: Broadcast in CSP with mailboxes.

Here : means concatenation and $x : xs$ means the element x concatenated with the list of elements xs, which in turn could be empty ($\emptyset$). Listing 6.2 shows that this variant of mailboxed broadcasting can be verified to include a correct trace and be deadlock-free with FDR. Again we let the primed processes be the original process, which is equivalent with having them become whatever they want.

In Listing 6.2 we need a buffer size, here set to 5, to set an upper limit on how many cases FDR will try to find counterexamples for. This buffer size can be arbitrarily large and in a real-world scenario can be infinite. With a buffer size of 5 FDR checks it almost instantly while adding 100 for the buffer size, it will check more than 10 million states, which it completes in about a minute.

```
N = 3
MAXBUFFER = 5
PNAMES = {0..N-1}
MSG = {"hello"}

channel back
channel b:MSG
channel c, m:PNAMES.MSG

S(x) = b!x -> back -> S(x)
B = b?x -> (||| i:PNAMES @ m.i!x -> SKIP) ; back -> B

M(i, <>)       = m.i?y -> M(i, <y>)
M(i, xss)      = if #xss > MAXBUFFER then Ml(i, xss) else Ms(i, xss)
Ml(i, <x>^xs) = c.i!x -> M(i, xs)
Ms(i, xss)     = Ml(i, xss) [] (m.i?y -> M(i, xss^<y>))

P(i) = c.i?x -> P(i)

MAILBOX = ||| i:PNAMES @ M(i, <>)
RECV    = ||| i:PNAMES @ P(i)
COMM(x) = S(x) [|{|b, back|}|] B

SYSTEM(x) = (COMM(x) [|{|m|}|] MAILBOX) [|{|c|}|] RECV

assert SYSTEM("hello") :[deadlock free [F]]
assert SYSTEM("hello") \ {|b, back, m|}
  :[has trace [T]]: <c.1."hello", c.0."hello", c.2."hello">
```

Listing 6.2: Mailboxes verified with CSP and FDR.

6.3 Broadcasts in Other Libraries

Both JCSP [33] (Java Communicating Sequential Processes) and CHP [34] (Communicating Haskell Processes) has a kind of broadcast channel already. JCSP implements a "one-to-many" channel type while in CHP multiple processes can enrol to accept a message from the same sender.

Both of these kinds of broadcasts are blocking, and in fact not really a broadcast, since the process needs prior knowledge about the other processes, that is, it needs to know that at some point somebody might broadcast a message.

With the methods and algebra presented here, instead, we treat broadcasts as ordinary channel communications.

CHAPTER 7

Emit

Continuing the trend of CSP libraries, I wanted to create one that allowed me to model CSP algebra using the Ruby programming language. Several attempts for creating such a library already exists, as well as an emulation of Go's channel and process model [35, 36], but this seems to be more of an emulation of Go rather than CSP, and even though Go leans on CSP for its channel and process model, it is not quite the same. Thus, I created my own and called it *Emit*.

Emit borrows ideas from the PyCSP implementation [3], which itself looked at JCSP [37, 38, 39] and C++CSP [40]. PyCSP was developed at the University of Copenhagen and University of Tromsø.

The two main components of any CSP framework is, of course, the processes and channels. Emit has, like JCSP and PyCSP, channel objects that allow for read and write channel-ends to be extracted and used for communication. The programmer can get as many channel-ends as needed from a single channel object. These channel-ends allow for any-to-any communication, where only one reader and one writer are active in the rendezvous. Subjectively, it makes it easier to reason about a CSP implementation, if we always know that a channel is only being read from or written to in a given process. This makes it easier programming-wise, however, the programmers are able to pass along entire channel objects instead of channel-ends if the need arises.

7.1 Implementation

As mentioned, Emit is written in pure Ruby. This means that all you need to run Emit is a Ruby interpreter, which is present on most systems today [41]. It also means that you will not have to compile external libraries or dependencies in order for Emit to function on your system since Emit does not make use of any external form of locks or barrier libraries.

Since Ruby is a highly dynamic programming language, there are several ways of setting up the processes, which can be seen in Listing 7.1.

```ruby
# Import the Emit framework
require "emit"

# Setup a process with a block. arg1 is passed to the block.
p1 = Emit::Process.new(arg1) { |arg1| ... }

# Give the 'new' method a proc. arg1 is passed to the proc.
p2_proc = proc { |arg1| ... }
p2 = Emit::Process.new(arg1, &p2_proc)

# Use a lambda and a shortcut for the 'new' method.
# arg1 is passed to the lambda
p3 = Emit.process(arg1, ->(arg1) { ... })

# Use a regular Ruby method. arg1 is passed to the method.
def p4_method(arg1)
  ...
end
p4 = Emit.process(arg1, &method(:p4_method))

# Use a regular Ruby method defined in the same namespace as Emit.
# arg1 is passed to the method.
def p5_method(arg1)
  ...
end
p5 = Emit.p5_method(arg1)
```

Listing 7.1: Various ways of setting up processes in Emit.

We also see the various ways that syntactic sugar can be used to create processes. Normally the programmer would have to write out `Emit::Process.new(...)` every time they wanted to create a new process, but the shortcuts make it possible to just write `Emit.process(...)` or even `Emit.my_own_method(...)` which makes Emit almost invisible and unobtrusive. One of the main goals of Emit is to make it easy to write CSP networks and pass messages between processes. Using Emit, the programmer should never have to think about locks, network exceptions, and so on.

Channels are likewise very easy to work with. You just tell Emit you want a channel and then grab the channel-ends with unary plus or minus. This can be seen in Listing 7.2. These are again just syntactic sugar to make Emit as low profile as possible.

```
require "emit"

# Create a new channel object
chan = Emit.channel

# output (writer) channel-end
cout = -chan # or 'cout = chan.writer'

# input (reader) channel-end
cin = +chan # or 'cin = chan.reader'
```

Listing 7.2: Channel-ends in Emit.

7.1.1 Processes

The processes in Emit are, as already shown, created from simple Ruby procs, lambdas, or globally defined methods. Inside Emit the processes are emulated using fibers [42]. The instances of the `Process` class in Emit will have all the necessary information in order to run a process. When the programmer creates a new process, it is spawned but not yet run, and thus does not have a fiber yet. When using the `parallel` module method of Emit all the processes mentioned will get a fiber attached and then start to run, once the scheduler picks them up and transfers control to their fiber.

Since everything in Ruby is an object this essentially means that you can have a process in a variable that is then sent over channel communication to other processes and then started a separate place from where it was created thus giving us mobile processes and channels for free. As of the writing of this thesis, there is, however, no way of starting the newly spawned mobile process in the same scheduler.

7.1.2 Scheduler

In order for the processes to run we need a scheduler. The Ruby fibers work by having the main thread transferring control to the next fiber that is then activated. The newly activated fiber can yield the control to yet another fiber, or back to

the scheduler, and thus all the processes can run. The scheduler is at all time maintaining a list of processes (fibers) that are still queued to run.

We are not running fibers in parallel, but internally sequentially, thus we avoid all the regular problems that arise in parallel programs. This unfortunately also means that we cannot make use of multiple cores in a machine. Some concurrent problems still remain. One such problem is deadlocks, as shown in Figure 7.1. The following algebra specifies a communication that will always deadlock, as both processes are waiting to send on different channels at the same time.

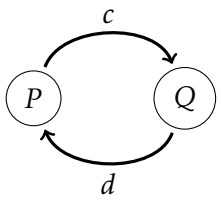


Figure 7.1: A deadlocked network.

$$
\begin{aligned}
P(x) &= c!\,x \to d?\,x \to P(x) \\
Q(x) &= d!\,x \to c?\,x \to Q(x) \\
P(1) &\underset{\{c,\,d\}}{\parallel} Q(1)
\end{aligned}
$$

Here both P and Q are ready to input on channel c and d respectively. This will never work, and thus will deadlock if programmed in Emit. The Emit scheduler is clever enough to see these kinds of patterns and throws a `DeadlockException` that you can catch in your program if you would like to.

This particular deadlock is detected by the scheduler since the first process engages in communication and is placed in a write queue then its fiber gives control back to the scheduler. The scheduler then transfers control to the second process that also places itself into a write queue. The scheduler now gets control again, but cannot transfer control to any other process, since all processes are in write queues waiting for readers. Thus we throw a `DeadlockException`.

Listing 7.3 shows the deadlocked network implemented with Emit.

```
require "emit"

def deadlock_process(cout, cin)
  cout << 1
  cin.()
end

c = Emit.channel
d = Emit.channel

Emit.parallel(
  Emit.deadlock_process(-c, +d),
  Emit.deadlock_process(-d, +c)
)
# => DeadlockException
```

Listing 7.3: Deadlocked network in Emit.

7.2 Communication

We have already discussed some of Emit's communication scheme. Channels are just Ruby objects from which you can extract as many read and write ends as you need. All channels are therefore any-to-any channels. A normal way of using the channel ends is to give them as arguments when instantiating the processes before running them with `parallel`.

When a process wants to communicate, either read or write, on a channel it puts itself into the appropriate read or write queue on that channel. If the opposite queue is empty, the process will give the control back to the scheduler. Hopefully, the scheduler will return control at some point in the future, if not, we have a deadlock. When another process comes along that also wants to communicate on the same channel, that is read what the first process wanted to write, the writer will be activated and allowed to write the message to the current process. Afterwards, the writing process is placed back into the scheduler's main queue and activated at some point in the future. If the reading process arrives at the rendezvous first, it will also suspend itself and wait for the writer, but here it will be allowed to read and then continue, again leaving the writer in the scheduler's main queue. The message will thus be moved from the writer to the reader on this rendezvous channel communication.

If multiple readers or writers are present that is if, for example, two writers arrive before a single reader, the writer is chosen at random. The same is true if multiple readers have arrived before a single writer.

7.2.1 Termination, Poison and Retirement

Since Emit, and also PyCSP, JCSP, and others, are built without the proper CSP events, we need a way to signal, that the parallel construct is done. In the algebra, we would normally just have all processes synchronise on ✓, but in real life, we can do an even simpler thing. We say that the network is terminated when all processes are terminated. We also assume, that if the process itself does not explicitly loop, then it terminates after it has performed its process body. That is, the processes in Listing 7.4 is actually emulating the following algebra:

$$P = c!\texttt{"Hello, world!"} \rightarrow ✓$$
$$Q = c?x \rightarrow ✓$$
$$P \underset{\{c\}}{\parallel} Q$$

```
require "emit"

c = Emit.channel

Emit.parallel(
  Emit.process(-c) { |cout| cout << "Hello, world!" },
  Emit.process(+c) { |cin| puts cin.() }
)
# => Hello, world!
```

Listing 7.4: A simple Emit network that terminates.

Notice the ✓s in the algebra and the lack of a terminating term in the actual implementation.

This, however, will not do for recursive processes. Imagine having a worker process that keeps listening on a channel for more work. How is it to know that no more work will come. To solve this problem, we introduce poison into the network. Poison was first introduced in C++CSP [40] and JCSP [43]. Poison works by disallowing further communication on a channel. This is done, by having all future reads and writes on a channel throw an exception that can be caught and treated in the process. This behaviour can be emulated with CSP as follows:

$$
\begin{aligned}
P(0) &= poison \rightarrow \checkmark \\
P(x) &= c!\texttt{"hello"} \rightarrow P(x-1) \\
Q &= (c?x \rightarrow Q) \,\square\, (poison \rightarrow \checkmark) \\
P(10) &\underset{\{c,\ poison\}}{\parallel} Q
\end{aligned}
$$

Here we output the string `"hello"` ten times on the channel before engaging in the *poison* event. When we engage in *poison* the Q process can also only engage in that, so the network synchronises on ✓ and terminates.

In Emit we go a step further. Instead of the programmer having to handle the poison exception, we just propagate the poison down all the channel-ends given as arguments to the current process. However, if the programmer would like, they can still catch the exception and handle it differently. Listing 7.5 shows a network, with a looping process that keeps on reading from a channel and another process that sends only ten items before poisoning it.

```
require "emit"

def source(cout)
  10.times { |i| cout << i }
  Emit.poison(cout) # or 'cout.posion'
end

def sink(cin)
  loop { print cin.() }
end

c = Emit.channel

Emit.parallel(
  Emit.source(+c)
  Emit.sink(-c)
)
# => 0123456789
```

Listing 7.5: Poisoning a channel in Emit.

While the poison construct is nice to have, it actually yields a new problem. Because the channels are any-to-any, we can have several readers and writers at any point. If a process is done and poisons this channels, all the other readers and writers still using that channel will also die. Listing 7.6 shows this in Emit. Here it might just be that the first process gets to write ten times in succession to the

channel before anybody else. If it does, it will poison the channel and subsequent writes will be poisoned without having given off their values.

```
...

Emit.parallel(
  10.times.map { Emit.source(+c) },
  10.times.map { Emit.sink(-c) }
)
```

Listing 7.6: Poisoning a channel that is still in use in Emit.

Instead of using poison, Emit also supports channel retirement as introduced in PyCSP [4]. Channel retirement works the same way as poison, except instead of poisoning, and thus ending all communication, immediately, we have a counter for both types of channel-ends. When a process wants to terminate, it retires its channel-ends and decrements the counters. If a counter is ever zero, that is no more readers or writers are present on the channel, we can poison the channel and terminate the last readers or writers that might still be present. This means that instead of the first poison killing the channel, it is instead the last retirement, for a channel-end type, that kills it. Retirement in Emit is used by switching out `poison` for `retire` in Listing 7.5.

7.2.2 Choice

Another important aspect of any CSP library is the ability to emulate choice. If we have several ready channels to read from, which channel do we then proceed to actually read from? There exist many ways of determining this. One is to just loop through all channels and take the first one available. This might be okay, but could also pose a problem, if say the first channel is always ready to write, then none of the other channels will get their turn.

Another way is to choose at random, but even random has its disadvantages. With random, we have no guarantees that all channels are chosen at some point. Here fairness comes into play. We can have a fair select [40] that always chooses the one that has been chosen the least.

In Emit, I have chosen to implement the standard choice as a random select. Emit looks at the list of choices it has, shuffles it, and runs through them from the top down, choosing the first one that is available. Listing 7.7 has an example process, `flip_coin`, that makes a choice, whether to read from `cin1` or `cin2`. The _,

in this case, is to signify that we are ignoring which channel was chosen, but only care about the message passed on. This message is saved in the global result array and lastly output at the end. This result array consists of a random mixture of ones and zeros, as we read either a one or zero from the channels.

```
require "emit"

$result = []

def flip_coin(cin1, cin2, n)
  n.times do
    _, msg = Emit.choice(cin1, cin2)
    $result << msg
  end
end

ch1, ch2 = Emit.channel, Emit.channel

n = 10
Emit.parallel(
  Emit.process { n.times { -ch1 << 0 } },
  Emit.process { n.times { -ch2 << 1 } },
  Emit.flip_coin(+ch1, +ch2, 2*n)
)

puts $result.inspect
# => [1, 0, 1, 0, 0, 1, 0, 0, 1, 1]
```

Listing 7.7: Choice between two different channels.

7.3 Results

7.3.1 COMMSTIME

An already well-established benchmark for CSP style libraries are the COMMSTIME benchmark [44]. In COMMSTIME we have four processes, often called: PREFIX P, DELTA D, SUCC S, and CONSUME C. Figure 7.2 shows a diagram of the communications in the network. PREFIX is given an initial value of 0. All it does is read from its input channel and write to its output channel, having first output the initial value. DELTA copies the message in two and passes in to each of its output channels. SUCC will increment the value by one and pass it on.

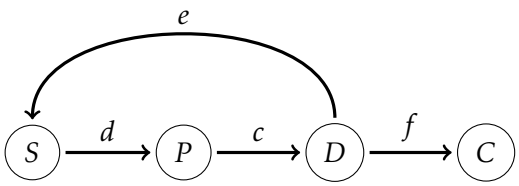


Figure 7.2: The COMMSTIME network.

These communications follow the algebra:

$$
\begin{aligned}
P(x) &= (c!\,x \rightarrow d?\,y \rightarrow P(y)) \,\square\, \checkmark \\
D &= (c?\,x \rightarrow f!\,x \rightarrow e!\,x \rightarrow D) \,\square\, \checkmark \\
S &= (e?\,x \rightarrow d!\,(x+1) \rightarrow S) \,\square\, \checkmark \\
C(0) &= \checkmark \\
C(n) &= f?\,x \rightarrow C(n-1) \\
P(0) &\| D \| S \| C(500000)
\end{aligned}
$$

Here I have added a ✓ to terminate the network. One could also imagine an alternative where the CONSUME process did not count up or down, but just consumed the value.

Listing 7.8 shows the FDR verification of COMMSTIME. This is, of course, deadlock-free as well as having a specific trace. In this example I have replaced the counting with a modulo operation, so the number just flips between 0 and 1, as FDR can then continue ad infinitum[1].

7.3.2 Implementation

With COMMSTIME defined and verified, we can take a look at an implementation in Emit and compare that to implementations in other CSP libraries. In the appendix, starting on page 91, Listing 1, Listing 2, Listing 3, and Listing 4 all contain a COMMSTIME implementation. In order we have Emit, Go, JCSP [45][2], PyCSP [46][3].

[1] I have seen countless problems with FDR at compile time, when trying to get it to simply add one to a number that is bounded.

[2] This code is modified from the JCSP repository.

[3] This code is modified from the PyCSP repository.

```
channel c, d, e, f : { 0..1 }

P(x) = c!x -> d?y -> P(y)
D = c?x -> f!x -> e!x -> D
S = e?x -> d!((x+1)%2) -> S
C = f?x -> C

SYSTEM = (P(0) [{| c, d |} || {| c, e, f |}] D)
         [{|c, d, e, f|} || {|e, d, f|}]
         (S [{|e, d|} || {| f |}] C)

assert SYSTEM :[deadlock free [F]]
assert SKIP [F= SYSTEM \ Events

assert SYSTEM
  :[has trace [T]]: <c.0, f.0, e.0, d.1, c.1, f.1, e.1>
```

Listing 7.8: CSP algebra used to verify COMMSTIME in FDR.

Table 7.1: COMMSTIME results averaged over 10 runs.

Framework	Result (μs/communication)
PyCSP	346.30
PyCSP (greenlets)	6.04
JCSP	22.55
Emit	3.79
Go	0.28

When using COMMSTIME as a benchmark, what we are looking for is the number of microseconds (μs) per communication. In it, we have 500,000 iterations being run, where each of the four processes does one communication. Thus, we have 2,000,000 communications in a run. Timing the entire benchmark and then dividing by 2,000,000 gives us the sought after time per communication. Table 7.1 shows the various results of running the specific COMMSTIME implementation. The "PyCSP (greenlets)" benchmark is the same code as Listing 4 in the appendix, but with `from pycsp.greenlets import *`, thus importing the greenlet version of PyCSP. Obviously, in Table 7.1, Go wins using only a quarter of a microsecond per communication. This might be because the channels and processes are built into the language itself and the code is compiled, however, Emit follows close by and is only a factor ten behind, which is quite impressive for an interpreted language such as Ruby. Emit is also roughly twice as fast as PyCSP using greenlets and 100 times faster than regular PyCSP.

CHAPTER 8

Ongoing & Future Work

Even after three years of work on the various parts of Bohrium and Emit there are some things that are still ongoing. There are also several features of both frameworks that I would love to see in the future. These are all described in this chapter.

8.1 Bohrium Filters

The "stupid maths" filter is great, it does what it says on the box, however, it is only one of these such filters that could be made for Bohrium. Other, more resilient and strong, filters could also be made, to contract even more maths into simpler terms. There are a lot of pitfalls that programmers need to consider when creating these filters.

For example, it would be nice to have a filter where multiply and divide negate each other, however, this will not always work for integer types. For example,

$$\frac{r}{2} \neq \frac{z}{2}, \text{ when } r \in \mathbb{R},\ z \in \mathbb{Z},\ r \equiv z$$

is not always true, since, if $r \equiv z = 3$, but if $\frac{z}{2}$ still need to be in $\mathbb{Z}$, we would need to round or floor the fraction. Bohrium filters that take this into account and still produce faster kernels would be a nice addition.

8.2 Automatic Extension Methods

At first, I wanted to have a more programmatic approach to the templating schemes. I wanted to create a DSL (Domain Specific Language) in order to express the templates and then have the entire suite automatically generate itself. This is feasible, but I do not know whether it would be a feature worth creating. Splitting the templates into even smaller pieces might be nice for reusability, but I cannot say that there were much in my experiments. More often then not small changes to each piece of the template had to be made. In the DSL, this could be made by having more and more arguments for each command, but soon you would have much of the original code polluting the programmers' use of the DSL, which was not the point of creating the DSL, to begin with.

I only tried to implement this with BLAS, LAPACK, and OpenCV, but additional external libraries could be interesting to have in Bohrium as well.

8.3 Bohrium.rb

The C++ part of the Ruby front end can be optimised a lot. There are still several pointers floating around and I believe that the Ruby garbage collector is being ignored, instead of helping it along. Thus, figuring out how to mark the Bohrium internals for garbage collection by the Ruby garbage collector is a must. Unfortunately, there is not very much documentation on the C++ interface for Ruby, which is one of the reasons why I have not invested more time in it myself.

As already discussed earlier, the Ruby front end gets its methods from a JSON-file containing the Bohrium methods. This could possibly be done easier and better with less sanitising used before actually generating the methods. It could also be, that it would be beneficial to actually hand-implement these 79 methods instead of automatically generating them thereby having much more control.

As also stated earlier, it would be nice, if we monkey patch, for example, the `Matrix` class instead of creating our own `BhArray` class to encapsulate the Bohrium arrays. Having them in an STL matrix means that we can just plug our newly optimised Bohrium matrix into the already established STL matrix, with little overhead from the programmer. Thus, giving us the same benefit as with NumPy and Python where the programmer can just rename "numpy" to "bohrium".

8.4 Emit

An implementation of the scheduler in Emit that is actually able to run the threads in parallel would be very nice. This could be done by having the scheduler spin up a new thread or OS process for each fiber. Doing so might, however, decrease the performance, as the overhead of spawning new processes could get quite large. Figuring out how to do this properly is the next step for Emit.

Proper nested parallel structures are also needed, for example, I would like to have a CSP process be able to start new parallel processes with a new scheduler or possibly even the same scheduler. For example, for the case of COMMSTIME, we might like DELTA to be able to communicate with SUCC and CONSUME in parallel, instead of now having them communicate one and then the other. This could be implemented with a choice as well, but for the results back in Chapter 7 this does not matter.

CHAPTER 9

Conclusion

Throughout the duration of my PhD studies, I have worked with many different technologies, frameworks, libraries, programming languages, and theories. With my work on both Bohrium and Emit I have shown, not only that Ruby can be used for scientific purposes, but also that Bohrium and CSP in themselves have a lot of potentials.

I have published ten scientific articles and have personally presented the work at five different conferences. These articles were on different topics, some further explained in this thesis, some not explained here. I have also presented several posters at poster sessions that I have neither attached nor explained here, as their topics are different than the topics of this thesis.

For Bohrium I have created filters that optimise the generated bytecode and further optimises the generated kernels. These filters show that we can transform the bytecode with already known compiler techniques thus increasing the performance for each kernel.

Bohrium can now use BLAS directly to multiply matrices thus allowing it to speed up its execution when computing matrix products or similar functions taken from the various BLAS libraries. This also works for the OpenCL generated kernels. This is done by automatically generating wrapper functions from templates when Bohrium is first compiled.

I have also created a Ruby front end for Bohrium that outperforms the standard library for Ruby by up to 43×, in a benchmark summing along columns in a large matrix. These results are better than the current scientific package of choice for Ruby, Numo/Narray, however, Numo/Narray has still many more features. The Ruby front end makes it very easy to use Bohrium from Ruby in much the same way as NumPy works for Python.

For CSP we defined broadcast and how broadcast could work in the algebra. This, and a version that uses mailboxes to handle the messages, was verified with FDR.

Emit is my own CSP framework that I built just to see if I could outperform Python. It turned out that the implementation of the scheduler made it faster by a factor of 2 for the most common benchmark commstime. This scheduler runs the spawned processes concurrently and rendezvous on communications just like the CSP algebra.

With both of these cases in hand, I would argue that Ruby still has a lot of potential in the scientific community. Since both Bohrium.rb and Emit are so low profile, I would likewise argue that utilising these gems would come at no cost to the programmer productivity.

Bibliography

[1] Mads Ohm Larsen and Brian Vinter. Bohrium.rb – The Ruby Front End. In *Communicating Process Architectures*, 2018.

[2] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[3] Brian Vinter, John Markus Bjørndalen, and Otto Johan Anshus. PyCSP – Communicating Sequential Processes for Python. In *Communicating Process Architectures*, 2007.

[4] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP Revisited. In *CPA*, pages 263–276, 2009.

[5] Saint John Walker. *Big data: A revolution that will transform how we live, work, and think*. Taylor & Francis, 2014.

[6] CINEMA: the allianCe for ImagiNg and Modelling of Energy Applications. http://www.cinema-dsf.dk/, 2018. Accessed: 2018-11-20.

[7] Richard M Stallman and GCC Developer Community. Using the GNU compiler collection: a GNU manual for GCC version 4.3. 3. *CreateSpace, Paramount, CA*, 2009.

[8] Kenneth E Iverson. Notation as a Tool of Thought. *ACM SIGAPL APL Quote Quad*, 35(1-2):2–31, 2007.

[9] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[10] Mads RB Kristensen, Simon AF Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: A Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[11] Mads Ohm Larsen, Kenneth Skovhede, Mads RB Kristensen, and Brian Vinter. Current Status and Directions for the Bohrium Runtime System. *Compilers for Parallel Computing*, 2016:25, 2016.

[12] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[13] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[14] clBLAS. https://github.com/clMathLibraries/clBLAS. Accessed: 2018-06-11.

[15] Stefan Tangen. Demystifying productivity and performance. *International Journal of Productivity and performance management*, 54(1):34–46, 2005.

[16] Peter F Drucker. Knowledge-worker productivity: The biggest challenge. *California management review*, 41(2):79–94, 1999.

[17] Capers Jones. *Programming productivity*. McGraw-Hill, Inc., 1985.

[18] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[19] Mads Ohm Larsen and Brian Vinter. Exception Handling and Checkpointing in CSP. *Communicating Process Architectures*, pages 201–212, 2012.

[20] Mads Ohm Larsen. Exception Handling in Communicating Sequential Processes. Master's thesis, The University of Copenhagen, 2012.

[21] Ruby Programming Language. https://www.ruby-lang.org/en/, 2018. Accessed: 2018-12-10.

[22] BLAS Reference Implementation. http://www.netlib.org/blas/. Accessed: 2018-10-29.

[23] Accelerate. https://developer.apple.com/documentation/accelerate, 2018. Accessed: 2018-11-20.

[24] OpenBLAS. http://www.openblas.net, 2018. Accessed: 2018-11-20.

[25] Kazushige Goto and Robert A Geijn. Anatomy of High-performance Matrix Multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[26] Eric Ziegel. Numerical Recipes: The Art of Scientific Computing, 1987.

[27] Roland Koebler. Pyratemp templating framework. http://www.simple-is-better.org/template/pyratemp.html, 2018. Accessed: 2018-11-20.

[28] Fernando Perez, Brian E Granger, and John D Hunter. Python: An Ecosystem for Scientific Computing. *Computing in Science & Engineering*, 13(2):13–21, 2011.

[29] Robbert van Renesse. Why bother with CATOCS? *ACM SIGOPS Operating Systems Review*, 28(1):22–27, 1994.

[30] Brian Vinter, Kenneth Skovhede, and Mads Ohm Larsen. Broadcasting in CSP-Style Programming. *Communicating Process Architectures 2016*, pages 99–108, 2016.

[31] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Citeseer, 1993.

[32] Jose Valim. Elixir. https://elixir-lang.org/development.html, 2012. Accessed: 2019-01-29.

[33] 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.

[34] Neil Christopher Charles Brown. *Communicating Haskell Processes*. PhD thesis, 2011.

[35] Ilya Grigorik. Agent. https://github.com/igrigorik/agent/, 2010. Accessed: 2018-10-15.

[36] Ilya Grigorik. Concurrency with Actors, Goroutines & Ruby. https://www.igvita.com/2010/12/02/concurrency-with-actors-goroutines-ruby/, 2010. Accessed: 2018-10-15.

[37] James Moores. Native JCSP – the CSP for Java Library with a Low-Overhead CSP Kernel. In *Communicating Process Architectures*, volume 58, pages 263–273, 2000.

[38] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In *International Conference on Computational Science (2)*, volume 687, 2002.

[39] Peter H. Welch and Paul Austin. The JCSP (CSP for Java) Home Page. https://www.cs.kent.ac.uk/projects/ofa/jcsp/, 1999. Accessed: 2018-10-15.

[40] Neil . Brown and Peter H. Welch. An introduction to the Kent C++ CSP Library. volume 61, pages 139–156. IOS Press, 2003.

[41] Re: hah, check these errors. http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/170, 1999. Accessed: 2019-01-29.

[42] Ruby Docs: Fibers. https://ruby-doc.org/core-2.5.0/Fiber.html, 2018. Accessed: 2018-10-15.

[43] Bernhard HC Sputh and Alastair R Allen. JCSP-Poison: Safe Termination of CSP Process Networks. In *CPA*, volume 63, pages 71–107, 2005.

[44] Frederick RM Barnes and Peter H Welch. Prioritised Dynamic Communicating Processes – Part I. *Communicating Process Architectures*, 2002:321–352, 2002.

[45] JCSP SVN. http://projects.cs.kent.ac.uk/projects/jcsp/svn/jcsp/, 2018. Accessed: 2018-11-20.

[46] PyCSP Github. https://github.com/runefriborg/pycsp/, 2018. Accessed: 2018-11-20.

II

Appendix

```
require "emit"

def prefix(cin, cout, item)
  loop do
    cout << item
    item = cin.()
  end
end

def delta2(cin, cout1, cout2)
  loop do
    t = cin.()
    cout1 << t
    cout2 << t
  end
end

def successor(cin, cout)
  loop { cout << cin.() + 1 }
end

def consumer(cin, n)
  cin.()

  t1 = Time.now
  n.times { cin.() }
  dt = Time.now - t1
  tchan = dt.fdiv(4 * n)

  puts "Total time elapsed          = %.6fs" % dt
  puts "Avg. time per communication = %.6fs
        = %.6fµs\n" % [tchan, tchan * 1_000_000]

  Emit.poison(cin)
end

n = 3
comms = 500_000
n.times do |i|
  begin
    puts "---------- run: #{i+1} / #{n} ----------"

    a, b, c, d = Emit.channel, Emit.channel, Emit.channel, Emit.channel

    Emit.parallel(
      Emit.prefix(+c, -a, 0),
      Emit.delta2(+a, -b, -d),
      Emit.successor(+b, -c),
      Emit.consumer(+d, comms)
    )
  rescue Emit::ChannelPoisonedException
    Emit::Scheduler.reset!
  end
end
```

Listing 1: COMMSTIME using Emit.

```go
package commstime

func prefix(cin, cout chan int, v int) {
  cout <- v
  for v := range cin {
    cout <- v
  }
}

func id(cin, cout chan int) {
  for v := range cin {
    cout <- v
  }
}

func delta(cin, cout1, cout2 chan int) {
  for v := range cin {
    cout1 <- v
    cout2 <- v
  }
}

func succ(cin, cout chan int) {
  for v := range cin {
    cout <- v + 1
  }
}

func consume(cin chan int, n int) {
  for i := 0; i < n; i++ {
    <-cin
  }
}
```

```go
package commstime

import "testing"

func BenchmarkCommstime(bench *testing.B) {
  a := make(chan int)
  b := make(chan int)
  c := make(chan int)
  d := make(chan int)

  go prefix(c, a, 0)
  go delta(a, b, d)
  go succ(b, c)

  consume(d, bench.N)
}
```

Listing 2: COMMSTIME implemented as a benchmark in Go.

```
import org.jcsp.lang.*;
import org.jcsp.plugNplay.ints.*;
import org.jcsp.demos.util.*;

public class CommsTime {
  public static void  main(String argv []) {
    int nLoops = 500000;
    System.out.println(nLoops + " loops ...\n");

    One2OneChannelInt a = Channel.one2oneInt();
    One2OneChannelInt b = Channel.one2oneInt();
    One2OneChannelInt c = Channel.one2oneInt();
    One2OneChannelInt d = Channel.one2oneInt();

    new Parallel (
      new CSProcess[] {
        new PrefixInt(0, c.in(), a.out()),
        new Delta2Int(a.in(), d.out(), b.out()),
        new SuccessorInt(b.in(), c.out()),
        new Consume(nLoops, d.in())
      }
    ).run ();
  }
}
```

```
import org.jcsp.lang.*;

class Consume implements CSProcess {
  private int nLoops;
  private ChannelInputInt in;

  public Consume(int nLoops, ChannelInputInt in) {
    this.nLoops = nLoops;
    this.in = in;
  }

  public void run() {
    int x = -1;

    while(true) {
      long t0 = System.currentTimeMillis();
      for(int i = 0; i < nLoops; i++) {
        x = in.read();
      }

      long microseconds   = (System.currentTimeMillis() - t0) * 1000;
      long timePerLoop_us = microseconds /((long)(4*nLoops));
      System.out.println("    " + timePerLoop_us +
                         " microseconds / communication");
    }
  }
}
```

Listing 3: (Some of) COMMSTIME implemented using JCSP.

```
# -*- coding: utf-8 -*-
from pycsp import *
import time

@process
def prefix(cin, cout, item):
    while True:
        cout(item)
        item = cin()

@process
def delta2(cin, cout1, cout2):
    while True:
        t = cin()
        cout1(t)
        cout2(t)

@process
def successor(cin, cout):
    while True:
        cout(cin()+1)

@process
def consumer(cin, n):
    cin()
    t1 = time.time()
    for i in xrange(n):
        cin()
    dt = time.time() - t1
    tchan = dt / (4 * n)
    print("Total time elapsed          = %.6fs" % dt)
    print("Avg. time per communication = %.6fs" \
          "= %.6fµs\n" % (tchan, tchan * 1000000))

    poison(cin)

N = 500000

for i in xrange(3):
    print("----------- run %d/3 -------------" % (i+1))

    a, b, c, d = [Channel(), Channel(), Channel(), Channel()]

    Parallel(
        prefix(+c, -a, 0),
        delta2(+a, -b, -d),
        successor(+b, -c),
        consumer(+d, N)
    )
```

Listing 4: COMMSTIME with PyCSP.

III

Publications

Current Status and Directions for the Bohrium Runtime System

Mads Ohm Larsen, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
{ohm, skovhede, madsbk, vinter}@nbi.ku.dk

***Abstract*—In this paper, we present the current status of the Bohrium runtime system for automatic parallelization of array programming languages and libraries. We demonstrate how the design of Bohrium makes it possible to utilize different hardware platforms – from simple multi-core systems to clusters and GPU enabled systems – without any changes to the original user program.**

I. Introduction

In the scientific community, array programming[5] is a popular programming paradigm[13], [11]. It provides a natural way to express linear algebra problems without using pointer arithmetic or other low-level language constructs. Thus, array programming languages and libraries such as Matlab, Python/NumPy, R, and Fortran are very popular.

Bohrium[1][9], [10] defines a virtual machine, which executes an instruction from a bytecode instruction set that operates on arrays. This approach exploits the popularity of array programming by translating array operations into bytecodes, performing optimizations on the bytecodes, and then compiling the bytecodes into architecture specific binary kernels, and finally executing them.

In the rest of this paper, we will provide an overview and status of the different component of Bohrium.

A. Target audience

Bohrium is not built for speed, however as we will see in section VII it can be fast. Bohrium is rather meant to help scientific personal easily parallelize their programs, without having to know about special annotations such as `pragma`. Thus Bohrium can help people write fast, parallelized code, without rewriting their programs. To run with Python all the user needs to do, is switch the `import numpy` for `import bohrium` or even easier, launch their Python program with a `-m bohrium` command flag, which will substitute NumPy for Bohrium.

B. Interoperability

As already stated and as will be discussed in the following, Bohrium works with multiple languages and libraries. The main languages/libraries are NumPy, C++ and CIL[2], however it is possible to use Bohrium from any environment that can call C libraries.

[1]Available at http://www.bh107.org.
[2]Common Intermediate Language, also called .NET, MSIL or CLR

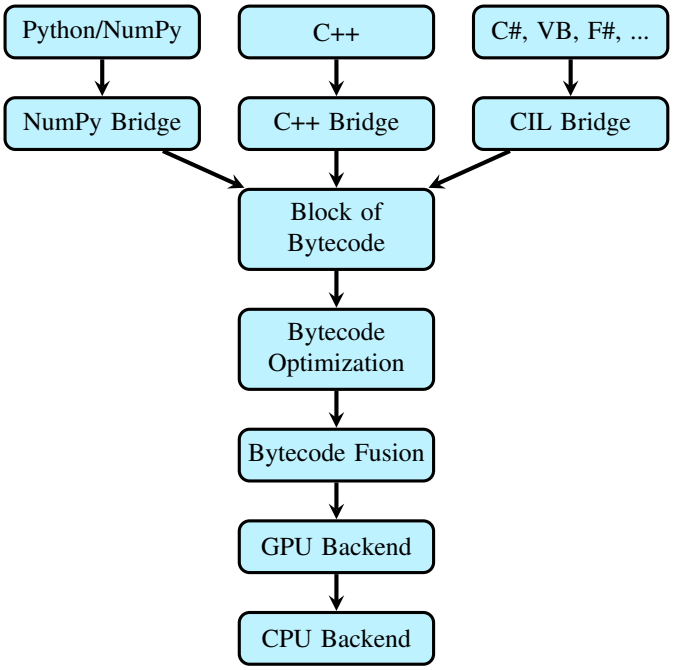


Fig. 1. Component Overview

II. Overview

Bohrium provides the mechanics to seamlessly couple an array-programming language or library with an architecture-specific implementation. It lazily records array operations, such as NumPy array operations, compiles them into architecture-specific binaries, e.g. GPGPU kernels, and executes them.

Bohrium consists of a number of components that operate on hardware agnostic array bytecodes. Components can be architecture-specific but they all use the same bytecode and communication protocol and can be interchanged. This design makes it possible to combine components in a setup that matches a specific execution environment without changing the user program.

The following component types are available for Bohrium:

Frontend At the highest level, we have the frontend programming language and library. Bohrium is not biased towards any specific choice of programming language or library as long as it is compatible with the array-programming model.

Bridge Connected to the frontend is a *Bridge* component. Its job is to translate the frontend language into Bohrium bytecode.

Bytecode Optimization Between the bridge and the execution backend, Bohrium supports a number of components that make bytecode-to-bytecode transformations. The specific component setup will vary depending on which optimizations and fuse strategies one wants to apply.

Bytecode Fusion After bytecode-to-bytecode transformations, Bohrium will fuse array bytecode into kernels that satisfies specific criteria given by the backend. A common criteria is data-parallelism, which makes it possible to calculate all array element individually without any communication between calculating threads. Another common criteria is that the shape of the arrays within a kernel must match.

Backend Given the kernels of array bytecode, the backend will compile the array bytecode into binary kernels that targets a specific architecture such as a multi-core CPU or a GPU.

Figure 1 shows an example of a Bohrium runtime setup that fits a system with both a CPU and a GPU. Notice that the GPU is the primary backend but may pass some array bytecodes to the CPU. The exact component setup depends on the runtime system e.g. if the system has no GPU, we can simply connect the fusion component directly to the CPU backend.

To make Bohrium as flexible a framework as possible, we manage the setup of all the components at runtime through a configuration file. The idea is that the user or system administrator can specify the hardware setup of the system through a configuration file. Thus, it is just a matter of editing the configuration file when changing or moving to a new hardware setup and there is no need to change the user's programs. For compiled languages, the same compiled binary can be used with multiple configuration files.

III. Frontend

As a running example for each frontend we use an implementation that solves the heat equation iteratively using the Jacobi Method.

A. C++

The C++ bridge provides an interface to Bohrium as a domain-specific embedded language (DSEL) providing a declarative, high-level programming model. Related libraries and DSELs include Armadillo[14], Blitz++[16], Eigen[1] and Intel Array Building Blocks[12]. These libraries have similar traits: declarative programming style through operator-overloading, template metaprogramming, and lazy evaluation for applying optimizations and late instantiation.

A key difference is that the C++ bridge applies lazy evaluation at runtime by delegating all operations on arrays to the Bohrium runtime, whereas the other libraries apply lazy evaluation at compile-time via expression-templates. This is a general design-choice in Bohrium – evaluation is improved by a shared component and not in every language bridge. A positive side effect of avoiding expression-templates in the C++ bridge are better compile-time error messages for the user.

```
#include <bh/bh.hpp>
double solve(multi_array<double> grid, size_t epsilon)
{
  multi_array<double> center,north,south,east,west,tmp;
  center  = grid[_(1,-1,1)][_(1,-1,1)];
  north   = grid[_(0,-2,1)][_(1,-1,1)];
  south   = grid[_(2, 0,1)][_(1,-1,1)];
  east    = grid[_(1,-1,1)][_(2, 0,1)];
  west    = grid[_(1,-1,1)][_(0,-2,1)];

  double delta = epsilon+1;

  while(delta > epsilon){
    tmp = 0.2*(center+north+east+west+south);
    delta = scalar(sum(abs(tmp-center)));
    center(tmp);
  }
}
```

Listing 1: Bohrium C++ implementation of the heat equation solver. The `grid` is a two-dimensional Bohrium array and the `epsilon` is a regular C/C++ scalar.

Listing 1 illustrates the heat equation solver implemented in Bohrium/C++, a brief clarification of the semantics follows. Arrays along with the type of their containing elements are declared as `multi_array<T>`. The function `_(start, end, skip)` creates a slice of every `skip` element from `start` to (but not including) `end`. The generated slice is then passed to the overloaded `operator[]` to create a segmented view of the operand. Overload of `operator=` creates aliases to avoid copying. To explicitly copy an operand the user must use a `copy(...)` function. Overload of `operator()` allows for updating an existing operand; as can been seen in the loop-body.

B. CIL

The NumCIL library introduces the declarative array programming model to the CIL languages [15] and, like ILNumerics, provides an array class that supports full-array operations. In order to utilize Bohrium, the CIL bridge extends NumCIL with a new Bohrium backend.

The Bohrium extension to NumCIL, and NumCIL itself, is written in C# but with consideration for other languages. Example benchmarks are provided that shows how to use NumCIL with other popular languages, such as F# and IronPython. An additional IronPython module is provided which allows a subset of NumPy programs to run unmodified in IronPython with NumCIL. Due to the nature of the CIL, any language that can use NumCIL can also seamlessly utilize the Bohrium extension. The NumCIL library is designed to work with an unmodified compiler and runtime environment and supports Windows, Linux and Mac. It provides both operator overloads and function-based ways to utilize the library.

Where the NumCIL library executes operations when requested, the Bohrium extension uses both lazy evaluation and lazy instantiation. When a side effect can be observed, such as

accessing a scalar value, any queued instructions are executed. To avoid problems with garbage collection and memory limits in CIL, access to data is kept outside CIL. This allows lazy instantiation, and allows the Bohrium runtime to avoid costly data transfers.

```
using NumCIL.Double;
using R = NumCIL.Range;

double Solve(NdArray grid, double epsilon)
{
  var center = grid[R.Slice(1,-1), R.Slice(1,-1)];
  var north  = grid[R.Slice(0,-2), R.Slice(1,-1)];
  var south  = grid[R.Slice(2, 0), R.Slice(1,-1)];
  var east   = grid[R.Slice(1,-1), R.Slice(2, 0)];
  var west   = grid[R.Slice(1,-1), R.Slice(0,-2)];

  var delta  = epsilon+1;

  while(delta > epsilon){
    var tmp = 0.2*(center+north+east+west+south);
    delta = (tmp-center).Abs().Sum();
    center[R.All] = tmp;
  }
}
```

Listing 2: NumCIL C# implementation of the heat equation solver. The `grid` is a two-dimensional NumCIL array and `epsilon` is a regular scalar value.

The usage of NumCIL with the C# language is shown in listing 2. The `NdArray` class is a typed version of a general multidimensional array, from which multiple views can be extracted. In the example, the `Range` class is used to extract views on a common base. The notation for views is influenced by Python, in which slices can be expressed as a three-element tuple of offset, length and stride. If the stride is omitted, as in the example, it will have the default value of one. The length will default to zero, which means "the rest", but can also be set to negative numbers which will be interpreted as "the rest minus N elements". The benefit of this notation is that it becomes possible to express views in terms of relative sizes, instead of hardcoding the sizes.

In the example, the one line update actually reads multiple data elements from same memory region and writes it back. The use of views simplifies concurrent access and removes all problems related to handling boundary conditions and manual pointer arithmetic. The special use of indexing on the target variable is needed to update the contents of the variable, instead of replacing the variable.

C. Python/NumPy

The implementation of the Python/NumPy bridge consists primarily of a new bohrium-array that inherits from NumPy's numpy-array. The bohrium-array is implemented in C and uses the Python-C interface to inherit from numpy-array. Thus, it is possible to replace bohrium-array with numpy-array both in C and in Python – a feature we need in order to support third party projects such as matplotlib.

As is typical in object-oriented programming, the bohrium-array exploits the functionality of numpy-array as much as possible. The original numpy-array implementation handles

```
import numpy as np

def solve(grid, epsilon):
  center = grid[1:-1,1:-1]
  north  = grid[-2:,1:-1]
  south  = grid[2:,1:-1]
  east   = grid[1:-1,:2]
  west   = grid[1:-1,2:]

  delta  = epsilon+1

  while delta > epsilon:
    tmp = 0.2*(center+north+south+east+west)
    delta = np.sum(np.abs(tmp-center))
    center[:] = tmp
```

Listing 3: Python/NumPy implementation of the heat equation solver.

metadata manipulation, such as slicing and transposing; only the actual array calculations will be handled by Bohrium. The bohrium-array overloads arithmetic operators, thus an operator on bohrium-arrays will use Bohrium.

However, NumPy functions in general will not make use of the Bohrium backend since many of them uses the C-interface to access the array memory directly. In order to address this problem, Bohrium has to re-implement some of the NumPy API. The result is that the Bohrium implements all array creation functions, matrix multiplication, random, FFT, and all ufuncs for now. All other functions, which accesses array memory directly, will simply get unrestricted access to the memory.

In order to detect and handle direct memory access to arrays, Bohrium uses two address spaces for each array memory: a user address space visible to the user interface, and a backend address space visible to the backend interface. Initially, the user address space of a new array is memory protected with `mprotect` such that subsequent accesses to the memory will trigger a segmentation fault. In order to detect and handle direct memory access, Bohrium can then handle this kernel signal by transferring array memory from the backend address space to the user address space.

Similarly to the other bridges, the Python/NumPy bridge uses lazy evaluation where it records instruction until a side effect can be observed.

IV. Optimizations on Bytecode Level

The bytecode used by Bohrium is a descriptive array bytecode. This can be used to record additional information about the instructions at compile-time. One can optimize such bytecodes in several ways, e.g. if multiple `BH_ADD`s are done for the same view, we can combine these into one operation. Doing so will decrease the number of steps for the fuser and code-generator.

Due to the distributive property of multiplication we can also do the following rewriting

$$ax + bx \rightarrow (a + b)x$$

Generalizing this, for some array x and scalar values c_i, we want to rewrite

$$\sum_i (x \cdot c_i) \rightarrow x \cdot \sum_i c_i$$

to give us the least amount of multiplication operations.

Multiplying and dividing with the identity element can be removed from the bytecode program, and the views can be replaced throughout the rest of it. The same is true for addition and subtraction. That is, we can do the following rewrite

$$x * e \rightarrow x$$

when $*$ is an operator for which e is the identity.

Another interesting bytecode to look at is `BH_POWER`. If the exponent of the power function is an integer, it is actually faster to do a series of multiplications instead [6], that is

$$x^n \rightarrow \underbrace{x \cdot x \cdot \cdots \cdot x}_{n} = \prod^{n} x \quad if\ n \in \mathbb{N}_+$$

Another optimization can be applied to this. In practice we do not want to generate a new temporary array in memory, to hold the result, so we are only allowed to operate on two arrays, the input and output arrays. We could just copy the input array to the output array and then multiply the output array with the input array $n-1$ times, however there is a better way. Instead we copy the input array to the output array, and then multiply the output array with itself $\lfloor \log_2(n) \rfloor$ times. This will give us the closest array to the result, which we can calculate with the least amount of multiplications. The rest can be done by multiplying with the input array again.

Let $input$ be the input array and $output$ be the output array and let us as an example calculate x^{10}. Since $\lfloor \log_2(10) \rfloor = 3$, we need to do three self multiplications of $output$.

$$\begin{aligned}
output &= input && (x)\\
output &= output \cdot output && (x^2)\\
output &= output \cdot output && (x^4)\\
output &= output \cdot output && (x^8)\\
output &= output \cdot input && (x^9)\\
output &= output \cdot input && (x^{10})
\end{aligned}$$

There are other faster ways to do this [4], however this is the most generic scheme, that works for all $n \in \mathbb{N}_+$. This optimization helps us speed up various benchmarks, especially Black Scholes, which we will discuss in section VII.

Other more complex patterns, that we will look for in the future, could be solving linear systems without actually creating the inverse matrix, e.g. solving

$$Ax = b$$

without figuring out A^{-1}. This can be done with LU factorization, but we would need to actually detect this pattern in the bytecode, again easing the use of Bohrium, since the user do not have to optimize their linear system solving themselves.

V. Array Bytecode Fusion

Array operation fusion is a program transformation that combines (fuses) multiple array operations into a *kernel* of operations. When it is applicable, the technique can drastically improve cache utilization through temporal data locality and enables other program transformations, such as streaming and array contraction [3].

Consider the two for-loops in listing 4a, which are fused into one for-loop, listing 4c, with the result of much improved cache utilization since array `T` and `A` are only traversed once instead of two times. For the next level of improvement, the for-loop in listing 4d does not allocate the array `T` at all. Instead, it uses the scalar `t` to stream the intermediate result of `B[i] * A[i]`, which is possible because `T` is only used within the for-loop – it is a temporary array local to the for-loop.

Not all fusion of array operation are allowed. Consider the two loops in listing 4b: the second loop traverses the result from the first loop in reverse, we must compute the complete result of the first loop before continuing to the second loop. This prevents fusion of the two for-loops and streaming of `T`, since it is not temporary to any one for-loop.

A. Fusibility

Array streaming depend on fusing array operations, so it is necessary to determine which operations we can legally fuse, and which we can profit from fusing. Generally, it is useful to fuse two array operations when the result of each output array element can be calculated independently without any communication between threads or processors:

Definition 1 (Fusibility)**.** *Two array operations, f, g, are fusible when there are no horizontal dependencies between:*

- *The output arrays of f and the input arrays of g*
- *The output arrays of g and the input arrays of f*
- *The output arrays of f and the output arrays of g*

where two arrays have a horizontal dependency when they access the same memory in different order.

Bohrium further restrict the fusibility of array operations by requiring that the shape of the involved arrays is the same. However, the number of dimensions in reduction operations is allowed to differ.

B. Fusion of Array Operations

[8] describes methods for finding a partition of operations such that a cost function is optimized, or near-optimized using a fast approximation heuristic. In Bohrium, we apply these methods to generate kernels that optimize for array streaming.

The problem of finding the optimal operation partitions is called the *Fusion of Array Operations Problem* (FAO problem), and is defined as follows:

```c
#define N 1000
double A[N], B[N], T[N];
for(int i=0; i<N; ++i) {
    T[i] = B[i] * A[i];
}
for(int i=0; i<N; ++i) {
    A[i] += T[i];
}
```

(a) Two forward iterating loops.

```c
#define N 1000
double A[N], B[N], T[N];
int j = N;
for(int i=0; i<N; ++i) {
    T[i] = B[i] * A[i];
}
for(int i=0; i<N; ++i) {
    A[i] += T[--j];
}
```

(b) A forward and a reverse iterating loop.

```c
for(int i=0; i<N; ++i) {
    T[i] = B[i] * A[i];
    A[i] += T[i];
}
```

(c) Loop fusion: the two loops from 4a fused into one.

```c
for(int i=0; i<N; ++i) {
    double t = B[i] * A[i];
    A[i] += t;
}
```

(d) Array contraction: the temporary array `T` from 4c is contracted into the scalar `t`.

Listing 4: Loop fusion and array contraction in C.

Definition 2. *Given a set of array operations, A, equipped with a strict partial order imposed by the data dependencies between them, $(A, \overset{d}{<})$, find a partition, P, of A for which:*

1) *All operations within a block in P are fusible (Def. 1).*
2) *For all blocks, $B \in P$, if $a_1 \overset{d}{<} a_2 \overset{d}{<} a_3$ and $a_1, a_3 \in B$ then $a_2 \in B$. (I.e. the partition obeys dependency order).*
3) *The cost of the partition is minimal.*

We will not go further into the detail of array operation fusion but instead refer to [8] that describe the theoretical groundwork and [7] that demonstrates its uses in Bohrium.

VI. Bohrium Processing Unit

The current backends for Bohrium support both CPU, GPGPU and even cluster based setups. This illustrates the flexibility in the programming model, and indicates that the Bohrium runtime system can target different types of hardware. While commodity hardware, such as CPUs and GPGPUs, have a good **price-to-performance** ratio, they do not offer the best possible **flops-per-watt** ratio, nor the lowest possible latency.

To achieve a lower latency and a lower power consumption, the ASIC[3], or the related FPGA[4] are more promising approaches. Unfortunately, both of these approaches require designing hardware circuits, which is many times more complicated than writing software, and thus entirely out of reach for the average Bohrium user.

We have designed and implemented the core for a Bohrium Processing Unit, which can execute Bohrium bytecodes, and utilize the memory layouts used. A schematic overview of the core unit is shown in figure 2.

The BPU is designed to work in a triple buffer setup, where dedicated hardware units perform three actions in parallel: read, execute, and write. Since the Bohrium bytecode is highly regular, we know in advance what memory to pre-fetch, and we have no need for branch prediction logic.

[3]Application Specific Integrated Circuit
[4]Field Programmable Gate Array

Programming the BPU would be difficult, as the user needs to keep track of what data is stored in the local scratch space, while balancing this with the need to issue memory reads and writes ahead of time, similar to how a user-controlled cache would work. We have written a rudimentary compiler that transforms a kernel from Bohrium into the instruction format defined for the BPU, such that all these requirements can be handled.

With this setup, it is possible execute a NumPy program on an FPGA without knowing anything about hardware design, or even modifying the program to fit an FPGA.

The BPU core shown in figure 2 is implemented in VHDL and can be simulated and tested with existing FPGA design tools. We have not yet implemented floating-point support, and emulate access to an external memory bus. The next step is to build a memory controller and connect it to a real memory interface, such that we can feed multiple BPU cores with a memory source. The memory controller will resemble the GPGPU approach, where each core can access the memory with varying offsets, but unlike the GPGPU, we know which offsets each core will request in advance, due to the regularity of the Bohrium bytecode.

The transpiler that converts bytecode to BPU instructions is implemented in a very simple manner, such that it only attempts to emit memory operations as much ahead of time as possible. Rather than implement optimizations in the transpiler, we are investigating filter transformations as the optimization step. The cost function for determining the optimal BPU program is different than for most others, as we have a need to keep kernel memory usage small enough to be in the scratch memory. If the kernels use too much memory, we need to swap to the attached memory interface, which decreases performance. Instead, it might be beneficial to partition kernels into identical sub-kernels, that fit in the limited storage and then stream the sub-kernels back-to-back.

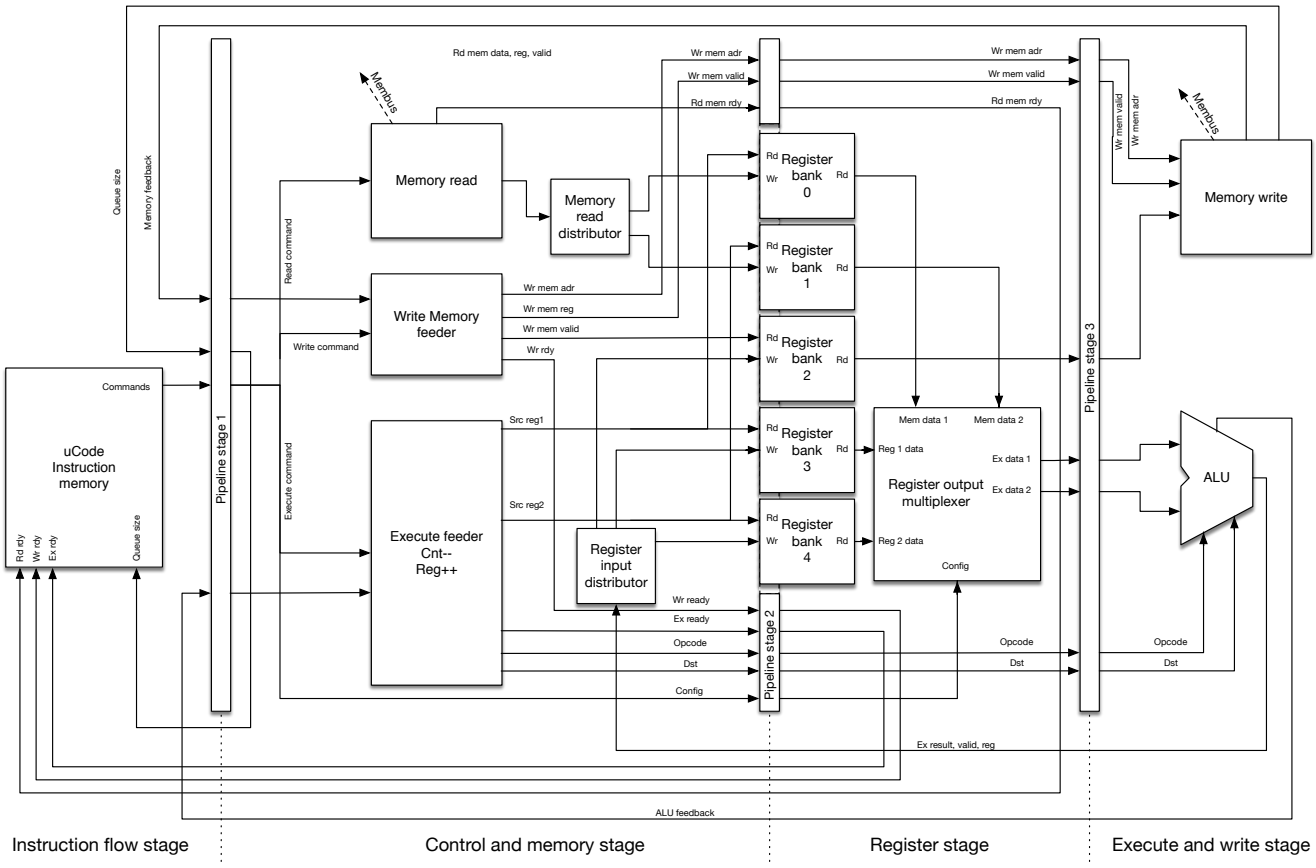


Fig. 2. Overview of the cBohrium Processing Unit.

VII. Performance

In this section, we will present the current performance results of Bohrium. We use Benchpress, which is an open source benchmark tool and suite. The source code for the implementations and the Benchpress tool is available online[5]. For reproducibility, the exact version used can be obtained from the source code repository[6].

A. The CPU backend

In order to evaluate the CPU backend, we compare a serial C implementation, a C++/OpenMP implementation, and a Python/NumPy implementation of each benchmark. The C and C++ implementations are handwritten and compiled with GNU Compiler Collection using `"-O3 -march=native"`. The Python/NumPy implementation is regular Python/NumPy code without any hand tuning or other low-level optimizations. We use the CPython 2.7 interpreter with the `"-m bohrium"` option in order to utilize the Bohrium CPU backend.

We run all benchmarks on a machine with 32-cores divided between four NUMA nodes (Table I). Figure 3 shows the speedup results with the serial C implementation as the baseline.

Machine:	
Processor:	AMD Opteron 6272
Clock:	2.1 GHz
#Cores:	32
L3 Cache:	16MB
Memory:	128GB DDR3
Compiler:	GCC 4.8.4
Software:	Ubuntu 14.04, Linux 3.13, Python 2.7.6, NumPy 1.8.2

TABLE I
Machine Specifications

The C implementation of the Black Scholes benchmark is compute-bound as the C++/OpenMP implementation show by achieving a near perfect linear speedup using 32 threads. The numbers reported by the Python/NumPy implementations using Bohrium obtain super-linear speedup of $\times 67.3$ using 32 threads and a speedup of about $\times 4.8$ using a single thread/core.

The Black Scholes benchmark relies heavily on exponentiations. As seen in section IV we optimize the power function (x^n) when $n \in \mathbb{N}_+$, which is exactly what we have here. This

[5] http://benchpress.readthedocs.org/
[6] https://github.com/bh107/benchpress.git revision `0aa2942`

	Hand-tuned C++/OpenMP		Bohrium Python/NumPy	
Threads	1	32	1	32
Black Scholes	0.9	29.1	4.8	67.3
Heat Equation	0.6	7.1	0.7	7.0
Leibnitz PI	1.0	22.6	0.6	14.6
Monte Carlo PI	1.0	29.8	1.0	27.8
Mxmul	1.0	9.5	1.0	14.9
Rosenbrock	1.0	21.0	1.2	15.8
Shallow Water	0.5	9.1	0.7	6.6

Fig. 3. Speedup results, serial C implementation used as baseline.

Processor:	Intel Core i7-3770	
Clock:	3.4 GHz	
#Cores:	4	
Peak performance:	108.8 GFLOPS	
L3 Cache:	16MB	
Memory:	128GB DDR3	
Vendor:	AMD	NVIDIA
Model:	HD 7970	GTX 680
#Cores:	2048	1536
Clock:	1000 MHz	1006 MHz
Memory:	3GB GDDR5	2GB DDR5
-bandwidth:	288 GB/s	192 GB/s
Peak perf.:	4096 GFLOPS	3090 GFLOPS

TABLE II
SYSTEM SPECIFICATIONS

is what gives the super-linear speedup.

B. The GPU backend

We have conducted a performance study in order to evaluate how well the GPU-backend performs, compared to regular sequential Python/NumPy execution. This study has been previously published [2] and is by no means a study of how well Bohrium with the GPU-backend, or NumPy utilizes the hardware, it is simply an illustration of the magnitude of speedup the end user can expect to experience, when using Bohrium with the GPU-backend. Keeping in mind that the transition from native Python/NumPy to Bohrium is completely seamless and requires no effort of the user. Wall clock time is measured for all benchmark executions, which include data transfers between the CPU and GPU.

We run all benchmarks on a Intel machine with both a AMD and NVIDIA GPU (Table II).

The Black-Scholes application is embarrassingly parallel, which makes it perfect for running on the GPU. Even with the relatively simple scheme for kernel generation, the GPU-backend currently implements; it generates only *one* kernel per iteration of the main loop. The result is a very effective execution that achieves a speedup of 834 times (ATI) and 643 times (NVIDIA) respectively for the largest 32bit float problems (figure 4). Additionally, it clearly demonstrates the comparably poor 64bit performance of the Kepler architecture (NVIDIA). Note that the GTX 680 delivers 1/24 double precision operation per single precision operation according the specifications, which is worse than the ratio of 1:14 seen in the Black-Scholes benchmark. This indicates that even in

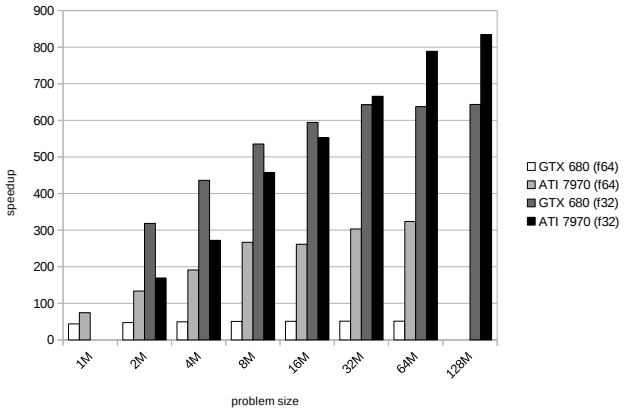


Fig. 4. Relative speedup of the Black-Scholes application running on the workstation

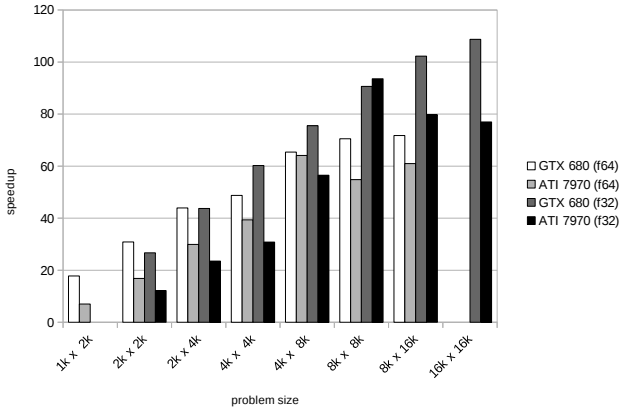


Fig. 5. Relative speedup of the SOR application running on the workstation

the embarrassingly parallel Black-Scholes application, which generates the largest kernel and has the best operation to calculation ratio, memory bandwidth still plays a role as a limiting factor.

The SOR application is the most memory bound and the least compute intensive of the four applications. Still, it is clearly beneficial to utilize the GPU through Bohrium (figure 5). For the largest problem size, it achieves a speedups of 109 and 94 times for the single precision versions and 72 and 61 times for the double precision versions. Even for the smallest problem size, it achieves a significant speedup. The drop-off in performance for the ATI GPU for single precision from 8k $\times$ 8k to 16k $\times$ 16k is something that needs further investigation.

The shallow water application works on several distinct arrays and has more complex computational kernels, compared to the SOR application. The more complex kernel is why we are able to get better performance. Again we observe

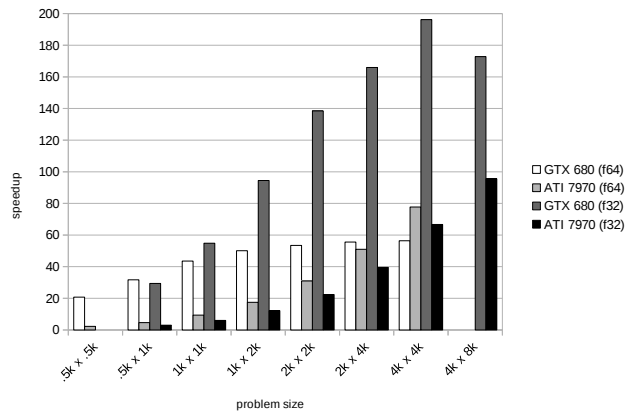


Fig. 6. Relative speedup of the Shallow water application running on the workstation

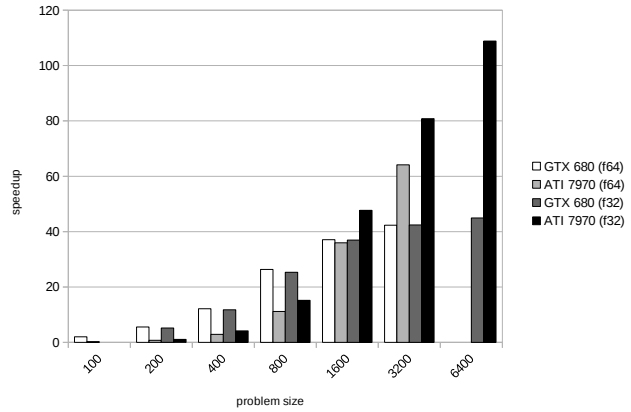


Fig. 7. Relative speedup of the N-body application running on the workstation

the same drop off in performance for the largest problem size – though this time on the NVIDIA GPU (see figure 6). The more curious observation one can make from figure 6 is that the ATI GPU performs much poorer than NVIDIA. ATI has better specifications in both memory bandwidth and peak performance. We will have to investigate whether the code we generate favors NVIDIA GPUs, and if we can do something to remedy this.

The straight forward algorithm used in the N-body simulations computes distances of all pairs, expanding the N-body data to $O(n^2)$ data points. While calculating the forces, the data is reduced back to the original $O(n)$ size. Due to the simple algorithm used in the GPU-backend, the reduction will force a kernel boundary, resulting in the expanded data being written back to global memory, before being read again by another reduction kernel – thereby being reduced. The large space requirements, due to the all pairs expansion, also puts an unfortunate limitation on the problem sizes NumPy is able to run. Only the two largest problem sizes are theoretically able to use all the core on the two GPUs, which leaves little room for latency hiding. Still, the Bohrium GPU backend is able to achieve up to 40–100 times speedup as figure 7 illustrate.

It is clear from figure 4–7 that the bigger the problem size, the better suited it is for execution on the GPU. This is no surprise since a bigger problem, will instantiate more threads, better utilizing the many cores of the GPUs, and at the same time enabling better latency hiding for the memory fetches. It is also expected, that there is a certain initialization cost for calling an external library, generating and decoding the bytecode, generation kernels and source code and invoking the GPU kernels. All of the experiments above have been run for a small, but sufficient number of iterations that the initial costs are amortized. To illustrate that the initialization costs are not excessively large, all four benchmark applications were run for just a single iteration. The Black-Scholes application still shows a speedup of 10–500 times dependent on the problem size for a single iteration. The SOR and Shallow water applications show speedup for all, but the two smallest problem sizes (up to 30 times). Finally, the N-body application only shows speedup for the two largest problem sizes with a single iteration – keeping in mind that it is only these problem sizes that theoretically are able to utilize all cores. All the experiments that do not show a speedup for a single iteration has a total execution time of less than 0.4 seconds.

VIII. Conclusion

We have shown how Bohrium can be used as an easy way of creating parallel programs without much fuzz. This is mainly due to its tight collaboration with various array-programming libraries.

Bohrium gets its interoperability from being component based. These components are interchangeable and thus provide freedom of use for the user. It is easy to change the code from running on CPU to run on GPU instead, by just changing the backend component.

Dedicated hardware for running Bohrium, the BPU, is being investigated. With this we hope to achieve a better flops-per-watt ratio than conventional hardware. This allows the user of Bohrium to run their e.g. NumPy programs on dedicated hardware, without knowing about how to actually program for this hardware.

After code-generation Bohrium does various bytecode optimizations as well as array bytecode fusion. These optimizations and fusions allow for Bohrium to run faster and sometimes even faster than hand coded OpenMP code. Even though Bohrium is not build for speed, it can be fast. In case of the Black Scholes benchmark, Bohrium is actually 67.3 times faster than a serial C implementation, while a hand-tuned C++/OpenMP implementation only gives a speedup of 29.1 for 32 threads.

Bohrium is thus an easy way to parallelize, and speedup, your array programming code.

REFERENCES

[1] Eigen. http://eigen.tuxfamily.org/. [Online; accessed 12 March 2013].

[2] Troels Blum, Mads R. B. Kristensen, and Brian Vinter. Transparent GPU Execution of NumPy Applications. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2014 IEEE 28th International*. IEEE, 2014.

[3] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 281–295. Springer Berlin Heidelberg, 1993.

[4] Daniel M Gordon. A survey of fast exponentiation methods. *Journal of algorithms*, 27(1):129–146, 1998.

[5] Herbert Hellerman. Experimental personalized array translator system. *Communications of the ACM*, 7(7):433–438, 1964.

[6] Donald E Knuth. Seminumerical algorithms. 2007.

[7] Mads R. B. Kristensen, James Avery, Troels Blum, Simon A. F. Lund, and Brian Vinter. Battling memory requirements of array programming through streaming. *First International Workshop on Performance Portable Programming Models for Accelerators (P^3MA)*, 2016.

[8] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and James Avery. Fusion of array operations at runtime. *Arxiv preprint, arXiv:1601.05400*, 2016.

[9] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[10] Mads R. B. Kristensen, Simon A. F. Lund, T. Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[11] Philippe Mougin and Stéphane Ducasse. Oopal: integrating array programming in object-oriented programming. In *ACM SIGPLAN Notices*, volume 38, pages 65–77. ACM, 2003.

[12] C.J. Newburn, Byoungro So, Zhenying Liu, M. McCool, A. Ghuloum, S.D. Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 224–235, 2011.

[13] Cherri M Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers? *Computer*, 23(12):13–23, 1990.

[14] Conrad Sanderson et al. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, Technical report, NICTA, 2010.

[15] Kenneth Skovhede and Brian Vinter. NumCIL: Numeric operations in the Common Intermediate Language. *Journal of Next Generation Information Technology*, 4(1), 2013.

[16] ToddL. Veldhuizen. Arrays in Blitz++. In Denis Caromel, RodneyR. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer Berlin Heidelberg, 1998.

Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

Broadcasting in CSP-Style Programming

Brian VINTER [1], Kenneth SKOVHEDE, and Mads Ohm LARSEN
University of Copenhagen, Niels Bohr Institute

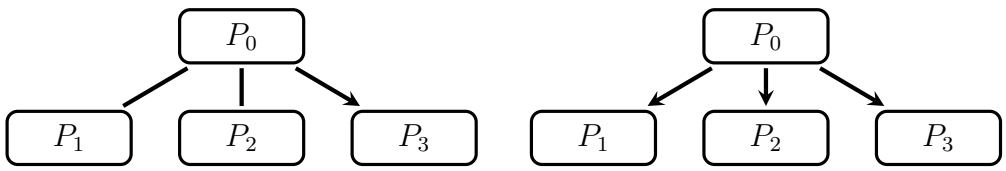
Abstract. While CSP-only models process-to-process rendezvous-style message passing, all newer CSP-type programming libraries offer more powerful mechanisms, such as buffered channels, and multiple receivers, and even multiple senders, on a single channel. This work investigates the possible variations of a one-to-all, broadcasting, channel. We discuss the different semantic meanings of broadcasting and show three different possible solutions for adding broadcasting to CSP-style programming.

Keywords. CSP, JCSP, broadcasting

Introduction

The concept of broadcasting — emitting a message from one process and receiving in all other processes in a group — has been debated in [1] and [2]. The work on Synchronous Message Exchange, SME [3] stems from a lack of broadcasting in CSP. In this work the authors will seek to establish the meaning of broadcasting and the possible benefit of a broadcast mechanism in CSP-style programming.

Perhaps the easiest way to introduce the meaning of a broadcast mechanism is to compare it with the well established one-to-any or any-to-any mechanism which all modern CSP-style libraries offer [4,5,6,7]. With the any-to-any channel a message sent by one process is delivered to any process that is ready to receive. Neither one-to-any nor any-to-any channels are part of the CSP-theory, however they can be emulated using multiple channels [8]. Contrarily a broadcast would be a one-to-all/any-to-all operation where every receiving process on the channel would get a copy of the message. Figure 1a and 1b seek to sketch the difference between a to-any and a to-all send operation.



(a) With the one-to-any model a message ends up at one process in the group

(b) The one-to-all delivers the message to all members of the group.

Figure 1. One-to-any and one-to-all communication.

Broadcasting is a common feature in message based parallel programming libraries, such as MPI [9] and PVM [10]. The need for broadcasting stems from algorithms that

[1]Corresponding Author: *Brian Vinter, Blegdamsvej 17, 2100 Copenhagen OE* E-mail: vinter@nbi.ku.dk.

need global synchronization. Broadcast may be used directly, that is for distributing a new global bound value in a parallel branch-and-bound algorithm, or it may be used in combination with a reduction, that is for determining the global change in a system after an iteration in a converging algorithm.

In SME, broadcasts were needed for a set of processes to all know a given value for the next simulated time-step, that is as a stand-alone broadcast. The use of reductions followed by a broadcast is mostly known in performance oriented parallel programs, though applications in the concurrency domain should not be entirely ignored. This work however, only investigates the feasibility of a pure broadcasting mechanism in a CSP-style library.

1. Broadcasting

Broadcasting as a concept appears deceptively simple however, several variations exists that have a slightly different semantics. Fundamentally broadcasting is based on physical broadcasts, that is a sender transmits a message for anybody to receive, as shown in Figure 2.

Figure 2. Physical broadcast mechanism.

We may define *simple broadcast* as the basic mechanism shown in Figure 2; one process may broadcast, any other process may receive. Thus, the *simple broadcast* mechanism has no delivery guarantee, a broadcast that is received by no process is still defined as a correct broadcast. Such a broadcast mechanism is of little use in real-world scenarios, and is thus often not provided. UDP/IP datagrams may be broadcast and if so it is done as *simple broadcast*, that is any layer in the network stack may choose to not propagate the broadcast. The only guarantee that the *simple broadcast* provides is message integrity, that is the message is either delivered in full and as originally sent, or not at all.

Simple broadcast may be improved to be a *reliable broadcast*. A reliable broadcast mechanism has the added semantics that when a message is broadcasted, all processes must receive a correct version of that message. Reliable broadcast does not guarantee any ordering, that is two concurrent broadcasts by two different processes may be received in different order by different processes in the system. While reliable broadcasting may be physically intuitive, as sketched in Figure 3, the lack of total ordering is still a limitation that makes programming harder in various cases.

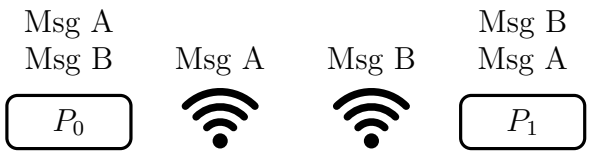


Figure 3. Two senders transmits messages A and B at the same time, because of physical proximity process zero will receive the messages in the order A then B, while process one will receive them in the order B then A. This is still a correct reliable broadcast.

An example where reliable broadcast is not sufficient is described as follows: a system consists of a set of fire detectors, fire alarms, and control boards. Imagine that a detector, A, detects a false fire, that is a forgotten toaster, the person using the

toaster immediately cancels the alarm using the control board A', but then another fire detector, B, detects a real alarm. If the broadcast message from B is received before the cancelation from A' then a non-counting alarm would falsely turn off, even though a fire was indeed spreading.

Reliable broadcasting may be further improved upon to guarantee total ordering; this type of broadcast is known as *atomic broadcast*. In the *atomic broadcast* all broadcasts are received by all processes, and in the same order. This also implies that a process that has failed, that is been unable to receive for some reason, cannot recover, but must leave the system and, if possible, rejoin. If a system implements *atomic broadcasts* the two processes in Figure 3 will agree on which order the messages A and B are received. Whether they are received as A then B or as B then A is still not defined, only that all processes will receive them in the same order.

A fourth broadcast mechanism which is well researched is the *causal broadcast* [11]. In this model a broadcast message B cannot be delivered to a receiver ahead of another message, A, if message A was received by the broadcaster of message B prior to that broadcast. Causal broadcasts however, has turned out to be very complex to implement and harder to work with than *atomic broadcasts*, thus we will not investigate *causal broadcasts* in this work.

Apart from delivery guarantees and message order broadcasts may be defined as synchronous or not. A *synchronous broadcast* will only be delivered to a receiver once all receivers are ready to receive, while an asynchronous delivery simply guarantees either reliable or *atomic broadcasts*. In distributed systems *synchronous broadcasts* are not available in any widespread libraries as the message cost becomes prohibitive, but in a CSP-library context a *synchronous broadcast* could be more efficiently implemented.

Finally, conventional broadcast literature differentiates between broadcasts in open and closed groups [12]. Open group broadcast means that a process, which is not on the list of recipients may still broadcast a message to the group, while closed group broadcasts require the sender to be a member of the recipient group.

2. Broadcasting in Message Passing Systems

2.1. Parallel Virtual Machines

The Parallel Virtual Machines library, PVM, supports group communication from version 3. Since PVM has a rather low level approach to message passing messages must first be packed into a message buffer and can then be broadcast to a group. The below example in listing 1, adapted from the PVM `man` page, packs 10 integers from a variable called array and then broadcasts the values, using the tag 42 to a group of processes, called tasks in PVM, named worker.

```
info = pvm_initsend(PvmDataRaw);
info = pvm_pkint(array, 10, 1);
info = pvm_bcast("worker", 42);
```

Listing 1. PVM broadcast sender.

The receiver will then have to issue an ordinary receive and then unpack the data as shown below in listing 2.

```
buf_id = pvm_recv(&tid, &tag)
info   = pvm_upkint(array, 10, 1)
```

Listing 2. PVM broadcast receiver.

In PVM the messages are received by the individual recipients as ordinary messages. PVM broadcasts are open group and provides only reliable broadcasts. In addition to the broadcast operation PVM also provides a multicast operation where a group is dynamically created from a list of recipients.

2.2. Message Passing Interface

Message Passing Interface, MPI, manages broadcasts rather differently than PVM. Groups are defined as subgroups of the overall set of processes, called `MPI_COMM_WORLD`. Message contents is defined like ordinary point-to-point messages, and addressing is simply to a subgroup. The major difference from PVM is that the broadcast is issued by all participants in the group, and a parameter in the broadcast specifies which process is the sending and all others are then receivers. This approach means that broadcasting in MPI is in closed groups only. A ten integer dense array broadcast example, from process zero to all other processes in the program, then looks as below in listing 3.

```
result = MPI_Bcast(data, 10, MPI_Int, 0, MPI_COMM_WORLD)
```

Listing 3. MPI broadcast.

3. Models for Broadcasting in CSP

From the existing systems that features a broadcast method, it is clear that such a mechanism has a use and this would be interesting to provide in a CSP-style library as well. Merging broadcast with CSP-semantics is not trivial, however, as we have previously shown in the SME work [3]. In this section we sketch the various broadcast styles one may imagine for CSP, and try to evaluate their feasibility for CSP.

3.1. Broadcast Messages

The simplest approach would be to introduce a broadcast message command, `channel.bcast(msg)`, similar to the semantics in PVM. With this approach, the library has to know about all processes that holds a reading end of the channel and then relay the message to each such process as they issue read commands from the channel. This approach raises a set of difficulties however; CSP dictates that the broadcast operation does not return until after the operation has finished, that is when all processes has received the message that was broadcast. However, it is not intuitive what should happen to the channel while the broadcast completes, if we require the channel to be blocked until the message is received everywhere then deadlocks may arise as other processes may have agreed to exchange a point-to-point message on the channel. If, on the other hand, we do not freeze the channel while the broadcast completes, other patterns that are non-intuitive may occur. If one process, that holds a receiving end of a channel, never issues a read on that channel then the channel will require infinite buffer-capacity to remain functional. A broadcast may be received by all but one process, and other messages have been passed as point-to-point afterwards, but if the last process then terminates, the broadcast has never truly been completed and the meaning of a broadcast becomes weaker.

An emulation of a broadcast message could be implemented as in Figure 4. This naïve network, and the theory behind, will be discussed in the next sections.

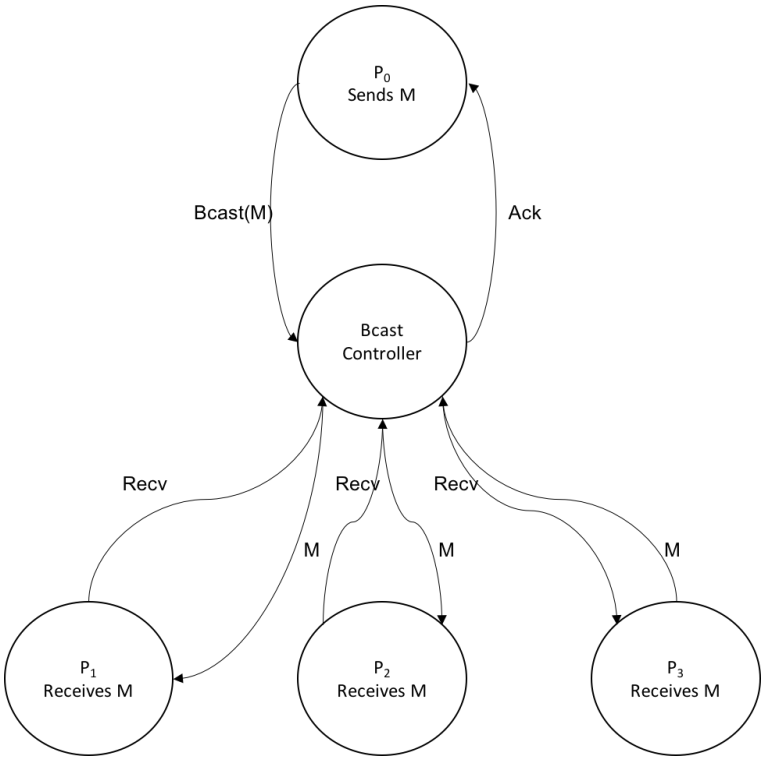


Figure 4. A naïve network for emulating synchronous broadcasting in CSP.

3.2. Mailboxes

A slightly more complex approach would be to introduce a mailbox system, similar to mailboxes as they are found in Erlang, but with a subscription service so that a set of processes can read the same message. A mailbox model is very intuitive for a programmer to comprehend; a message is sent to a central position and all processes may read it. However, this model does not include any synchronization as a CSP-style programmer would expect. If the model is extended to synchronize — either by announcing to the sender when all processes have read the message, or by introducing a fully synchronous model — then the mailbox model is identical to the broadcast message as described above.

A network using mailboxes for broadcast is shown in section 3.4.

3.3. Broadcast channels

A third approach to CSP-style broadcasting is to introduce dedicated broadcast channels. Such a broadcast channel has semantics that are identical to the synchronous broadcast message in the first scenario. The difference is that with a broadcasting channel there are no point-to-point messages that may be interleaved with broadcasts. This way the deadlock scenario from the broadcast message cannot occur, the channel provides synchronized communication as a point-to-point channel and may be used in guarded expressions just as an ordinary channel.

The downside of this approach is that programmers must use another type of channel. In JCSP this inconvenience is small as there is by design a large number of channel types, in PyCSP the change is more radical as the single channel type must be abandoned.

3.4. Broadcast in CSP-theory

In the following section, primed processes (i.e. S') will be used to show that the process can live on afterwards.

Emulating broadcast channel in CSP-theory can be done with n processes. Here S can broadcast a message to all of $P_{0..n}$:

$$S = m!x \rightarrow S'$$
$$P_i = m?x \rightarrow P_i'$$

$$S \parallel \left(\mathop{\Vert}_{i=0}^{n} P_i \right)$$

However, this is possible only, if we disregard the fact, that only two processes are allowed to communicate with each other at a given time according to Hoare CSP [13].

If we instead want to broadcast, but adhere to the original theory, we have to do something else. We will create a network of processes, where S will act as sender, B_c will act as a broadcast controller and $P_{0..n}$ will be receivers. They will pass messages using a two-phase commit protocol.

$$S = m!x \rightarrow m_{\text{ACK}} \rightarrow S'$$
$$B_c = m?x \rightarrow \left(\mathop{|||}_{i=0}^{n} c_i!x \rightarrow c_{i,\text{ACK}} \rightarrow \checkmark \right) ;\ m_{\text{ACK}} \rightarrow B_c'$$
$$P_i = c_i?x \rightarrow c_{i,\text{ACK}} \rightarrow P_i'$$

$$S \parallel B_c \parallel \left(\mathop{\Vert}_{i=0}^{n} P_i \right)$$

Looking at the system with FDR [14] in Listing 4 we see that it is deadlock free. A trace were verified where the message was received out-of-order.

```
N = 3
PNAMES = {0..N-1}
MSG = {"MSG"}

channel mack
channel m:MSG
channel cack:PNAMES
channel c:PNAMES.MSG

S(x) = m!x -> mack -> S(x)
B = m?x -> (||| i:PNAMES @ c.i!x -> cack.i -> SKIP) ; mack -> B
P(i) = c.i?x -> cack.i -> P(i)

SYSTEM(x) = S(x) [|{|m, mack|}|]
            (B [|{|c, cack|}|] || i:PNAMES @[{|cack, c.i|}] P(i))

assert SYSTEM("MSG") :[deadlock free [F]]
```

```
assert SYSTEM("MSG") \ {|m, mack, cack|}
  :[has trace [T]]: <c.0."MSG", c.1."MSG", c.2."MSG">
```

Listing 4. FDR3 verified Broadcasting CSP-system.

This of course means that the broadcast controller must know how many receivers there are present in the system and be able to pass messages to all of them on their respective channels.

If we want to allow all processes to be the writer at a given time, we can model this as alternation on the communication:

$$P_i = \Big(c_i!x \rightarrow c_{i,\text{ACK}} \rightarrow P_i'\Big) \Box \Big(c_i?x \rightarrow c_{i,\text{ACK}} \rightarrow P_i''(x)\Big)$$

$$B_c = \mathop{\Box}_{i=0}^{n} \left(c_i?x \rightarrow \left(\mathop{\|}_{\substack{j=0 \\ j \neq i}}^{n} c_j!x \rightarrow c_{j,\text{ACK}} \rightarrow \checkmark \right) ; c_{i,\text{ACK}} \rightarrow B_c' \right)$$

$$B_c \parallel \left(\mathop{\|}_{i=0}^{n} P_i \right)$$

Here all the processes either write on their channel or read from their shared channel with B_c.

The equivalent mailboxing scenario can be made with a mailbox process for each receiving process. Here the writer will be able to continue, once all mailboxes have ACK'ed back that they have received the message:

$$S = b!x \rightarrow b_{\text{ACK}} \rightarrow S$$

$$B = b?x \rightarrow \left(\mathop{|||}_{i=0}^{n} m_i!x \rightarrow m_{i,\text{ACK}} \rightarrow \checkmark \right) ; b_{\text{ACK}} \rightarrow B$$

$$M_i(x : xs) = \Big(m_i?y \rightarrow m_{i,\text{ACK}} \rightarrow M_i(x : xs : y)\Big) \Box \Big(c_i!x \rightarrow c_{i,\text{ACK}} \rightarrow M_i(xs)\Big)$$

$$P_i = c_i?x \rightarrow c_{i,\text{ACK}} \rightarrow P_i$$

$$S \parallel B \parallel \left(\mathop{\|}_{i=0}^{n} M_i(\emptyset) \parallel P_i \right)$$

where : means CONS and $x : xs$ is the head and tail of the message list. If one does not need the guarantee on each mailbox having received the message, the ACK steps can be omitted. This has been tested with FDR and found to be deadlock free. A trace was also found with each process having received the message.

The mailboxing works for arbitrarily big buffers; in Listing 5 it is shown with a buffer size of 5.

```
N = 3
MAXBUFFER = 5
PNAMES = {0..N-1}
MSG = {"MSG"}
```

```
channel back
channel b:MSG
channel cack, mack:PNAMES
channel c, m:PNAMES.MSG

S(x) = b!x -> back -> S(x)
B = b?x -> (||| i:PNAMES @ m.i!x -> mack.i -> SKIP) ; back -> B

M(i, <>)       = m.i?y -> mack.i -> M(i, <y>)
M(i, xss)      = if #xss > MAXBUFFER then Ml(i, xss) else Ms(i, xss)
Ml(i, <x>^xs) = c.i!x -> cack.i -> M(i, xs)
Ms(i, xss)     = Ml(i, xss) [] (m.i?y -> mack.i -> M(i, xss^<y>))

P(i) = c.i?x -> cack.i -> P(i)

MAILBOX = ||| i:PNAMES @ M(i, <>)
RECV    = ||| i:PNAMES @ P(i)
COMM(x) = S(x) [|{|b, back|}|] B

SYSTEM(x) = (COMM(x) [|{|m, mack|}|] MAILBOX) [|{|c, cack|}|] RECV

assert SYSTEM("MSG") :[deadlock free [F]]
assert SYSTEM("MSG") \ {|b, back, m, mack, cack|}
  :[has trace [T]]: <c.1."MSG", c.0."MSG", c.2."MSG">
```

Listing 5. FDR3 verified Mailboxing CSP-system.

4. Related work

Both JCSP [1] (Java Communicating Sequential Processes) and CHP [2] (Communicating Haskell Processes) offers a form of broadcast channel. In the former, a "one-to-many" channel is implemented. This must know the number of readers when initialized, and works by having two barriers, one before read and one after, so that all readers are done reading, before the writer is released. The latter is implemented in a similar way, where each reader enrolls to receive the same value from a single writer.

5. Conclusion

Broadcasting in CSP-style has been frequently discussed, and in the SME work replaced by a bus-style channel [15]. While adding broadcasting to CSP-style programming appears appealing and straight forward it has yet not been added to any CSP-style library. In this work we have outlined three approaches to adding broadcasting to CSP, and concluded that while they are all possible, only the explicit broadcasting channel is able to provide what the authors consider a CSP-style behavior and the common expected functionality of a broadcast operation. The need for broadcasting in CSP-style libraries is still an issue that must be investigated further, it is not obvious that broadcasting has a general use in application where CSP is commonly used, but might be reserved for fore traditional HPC style applications.

References

[1] 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.

[2] Neil Christopher Charles Brown. *Communicating Haskell Processes*. Citeseer, 2011.

[3] Brian Vinter and Kenneth Skovhede. Synchronous message exchange for hardware designs. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2015*, pages 201–212, aug 2014.

[4] Nan C. Schaller, Gerald H. Hilderink, and Peter H. Welch. Using Java for Parallel Computing - JCSP versus CTJ. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 205–226, Sep 2000.

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

[6] Neil C. 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 183–205, jul 2007.

[7] Kenneth Skovhede and Brian Vinter. CoCoL: Concurrent Communications Library. In *Communicating Process Architectures*, 2015.

[8] Mads Ohm Larsen and Brian Vinter. Exception Handling and Checkpointing in CSP. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 201–212, aug 2012.

[9] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[10] Al Geist. Pvm (parallel virtual machine). In *Encyclopedia of Parallel Computing*, pages 1647–1651. Springer, 2011.

[11] Robbert van Renesse. Why bother with catocs? *ACM SIGOPS Operating Systems Review*, 28(1):22–27, 1994.

[12] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[13] Charles Antony Richard Hoare et al. *Communicating Sequential Processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.

[14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika brahm and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[15] Brian Vinter and Kenneth Skovhede. Bus centric synchronous message exchange for hardware designs. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2015*, pages 245–257, aug 2015.

Algebraic Transformation of Descriptive Vector Byte-code Sequences

Mads Ohm Larsen
Niels Bohr Institute, Copenhagen University
ohm@nbi.ku.dk

ABSTRACT

Both high-productivity and high-performance are two often sought after aspects of scientific programming. Python gives the programmer high-productivity, but even with NumPy it is often not high-performant because of the GIL[1], which makes it inherently single threaded.

Bohrium intercepts NumPy calls and generates an intermediate language, Bohrium byte-code, before being compiled to OpenCL kernels. It thus grants Python/NumPy the ability to be easily run on multicore systems or GPUs, without changing the source code.

The Bohrium byte-code can be optimized, by transforming byte-code sequences into more performant ones. This way, the scientific programmer will not need to change her code to utilize special performant constructs.

CCS Concepts

•**Applied computing** → Physical sciences and engineering; •**Computing methodologies** → **Parallel computing methodologies;** Symbolic and algebraic manipulation;

Keywords

bohrium; numpy; algebraic transformation; byte-code

1. INTRODUCTION

Python is an interpreted, high-level, general purpose programming language, which enables high-productivity and readability. For scientific applications Python programmers utilize NumPy [4], which has become the de-facto standard for vectorized code[2] for Python.

Bohrium [2, 3] extends NumPy, by allowing the code to be run on multicore CPUs, clusters, or GPUs, without interfering with the high-productivity aspect of Python. The programmer only has to change the `import` from `numpy` to `bohrium`, and the Bohrium runtime will take over, still utilizing all NumPy calls internally.

The byte-code language operates on tensors[3] of varying size and shape. Every time the programmer invokes a NumPy method, Bohrium intercepts it and, if possible, does the computation on e.g. the GPU instead.

Since Bohrium has multiple front-ends, e.g. Python, CIL, C++, this project will focus on optimizing the intermediate language, that the Bohrium runtime generates. An optimization would be transforming byte-code sequences that can be rewritten into more efficient code, thus allowing the programmer high-productivity as well as fast running code.

2. ALGEBRAIC TRANSFORMATIONS

A transformation can be thought of as a rewriting of elements from one set to another.

An example of such a transformation could be to realize that

$$x^n \mapsto \underbrace{x \cdot x \cdot \cdots \cdot x}_{n} = \prod^{n} x \quad \text{if } n \in \mathbb{N} \tag{1}$$

We can thus exchange the power-function for a series of multiplications or vice versa. This also holds true for tensors, such that $\vec{x}^{10} = \prod^{10} \vec{x}$.

Another example of a transformation, one that requires a more context-aware transformation, could be transforming a solution of

$$\vec{A}x = \vec{B}$$

, where $\vec{A}$ and $\vec{B}$ are tensors. Usually we first have to find the inverse $\vec{A}^{-1}$ tensor to solve for x.

$$\begin{aligned} \vec{A}x &= \vec{B} && \Leftrightarrow \\ \vec{A}^{-1}\vec{A}x &= \vec{A}^{-1}\vec{B} && \Leftrightarrow \\ x &= \vec{A}^{-1}\vec{B} \end{aligned} \tag{2}$$

Instead one could do a LU-factorization [1] of the same problem, which would usually be faster to compute. Note that this is of course only faster, if we do not use the $\vec{A}^{-1}$ tensor for anything else in our computations.

The transformations can thus be small loop-fusion-like contractions of byte-codes, or it can detect the semantic meaning of the code, thus being more specialized and context-aware.

[1]Global Interpreter Lock
[2]Also known as array-programming.
[3]Multi-dimensional matrices.

Middleware Doctoral Symposium '16, December 12-16 2016, Trento, Italy

ISBN 978-1-4503-4665-8/16/12…$15.00
DOI: http://dx.doi.org/10.1145/3009925.3009926

3. BOHIRUM BYTE-CODE

Even though Python is an interpreted language, with Bohrium we actually get a JIT-compiled intermediate language. We can thus manipulate the byte-code before executing it.

In this byte-code language a single line encapsulates one byte-code. A byte-code consists of an op-code, e.g. `BH_ADD`, a result register, and up to two parameter registers or constants.

In Listing 1 we have a Python program adding three ones together in a one-dimensional vector of size 10.

```
import bohrium as np
a = np.zeros(10)
a += 1
a += 1
a += 1
print a
```

Listing 1: Adding three ones in Python.

In Listing 2 the same program is shown in the Bohrium byte-code langauge.

```
BH_IDENTITY a0[0:10:1] 0
BH_ADD a0[0:10:1] a0[0:10:1] 1
BH_ADD a0[0:10:1] a0[0:10:1] 1
BH_ADD a0[0:10:1] a0[0:10:1] 1
BH_SYNC a0[0:10:1]
```

Listing 2: Adding three ones with Bohrium.

Here, `a0` is a vector register where our view is always from `0` to `10` with a step of `1`. In further listings I assume the view is the same for all registers, and thus will not write it out.

What this means is that we three times add 1 to all elements of our `a0` tensor's full view and store it back into the full view.

3.1 Transforming the Byte-code

In the code example in Listing 2, we see each of the three additions being their own byte-code. In reality our `a0` tensor could be very large, so large in fact that adding one to each element would take a long time. Instead the constants of the three byte-codes can be merged into one by simply adding them together. After transforming the byte-code sequence, we thus end up with only one `BH_ADD` op-code adding 3 to each element, as shown in Listing 3.

```
BH_IDENTITY a0 0
BH_ADD a0 a0 3
BH_SYNC a0
```

Listing 3: Optimized adding three ones with Bohrium.

By transforming we can also generate more byte-codes. In the power-to-multiplication example we start with one byte-code, `BH_POWER`, and can potentially end up with many. Here it is important to know, that we usually only have access to the origin and result tensors, since copying data to create temporary tensors would be time consuming for large tensors. For the power example from (1), we can get better performance, still only using `BH_MULTIPLY`, by utilizing the result tensor multiple times, instead of only seeing it as the end-result. To do this, we can realize that

$$\begin{aligned} x^{10} &= x^8 \cdot x \cdot x \\ &= x^4 \cdot x^4 \cdot x \cdot x \\ &= x^2 \cdot x^2 \cdot x^2 \cdot x^2 \cdot x \cdot x \end{aligned}$$

If we thus first calculate $x^2 = x \cdot x$ and store that in our result tensor `a1`, we can then multiply it with itself to get x^4. Multiplying this with itself grants us x^8 and then multiplying this with x twice gives us the result x^{10}.

In Listing 4 we have calculated x^{10} by multiplying first our origin tensor by x and then 9 more times multiplying the result tensor with x.

```
BH_IDENTITY a0 ...      # initialize the tensor, x
BH_MULTIPLY a1 a0 a0    # x^2
... (x 7)               # x^3..x^9
BH_MULTIPLY a1 a1 a0    # x^10
BH_SYNC a1
```

Listing 4: $\vec{x}$ to the power of ten, using nine `BH_MULTIPLY`s.

We could actually do better. Since we own the result tensor, we are allowed to use it as we see fit. In Listing 5 a better x^{10} is shown.

```
BH_IDENTITY a0 ...      # x
BH_MULTIPLY a1 a0 a0    # x^2
BH_MULTIPLY a1 a1 a1    # x^4
BH_MULTIPLY a1 a1 a1    # x^8
BH_MULTIPLY a1 a1 a0    # x^9
BH_MULTIPLY a1 a1 a0    # x^10
BH_SYNC a1
```

Listing 5: $\vec{x}$ to the power of ten, using just five `BH_MULTIPLY`s.

4. CONCLUSIONS

Some of these rudimentary transformations have already been implemented into the Bohrium runtime system. Bohrium already supports merging integer addition, by adding the constants, before adding it to the actual tensors. It also does power expansion by default, since benchmarks have shown, that for values close to a power of 2, multiplying multiple times is faster than doing an actual `BH_POWER`.

A further study of real examples, such as (2), from imaging software and benchmark suites is planned. If resulting sequences is found to be too slow, a study on how to make them faster will be made.

5. ACKNOWLEDGMENTS

This work is part of the CINEMA project and financial support from CINEMA: the allianCe for ImagiNg of Energy MAterials, DSF-grant no. 1305-00032B under The Danish Council for Strategic Research is gratefully acknowledged.

Thanks goes to Sarah Haas for her Writer Development course as well as tools on how to write in protected environments.

6. REFERENCES

[1] J. R. Brunch and J. Hopcroft. Triangular factorization and inversion by fast matrix multiplication, 1974.

[2] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[3] M. O. Larsen, K. Skovhede, M. R. B. Kristensen, and B. Vinter. Current Status and Directions for the Bohrium Runtime System. In *Compilers for Parallel Computing 2016*, 2016.

[4] T. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

Imaging Data Management System *

Brian Vinter Jonas Bardino Martin Rehr Klaus Birkelund Mads Ohm Larsen
Niels Bohr Institute, University of Copenhagen
{vinter,bardino,rehr,birkelund,ohm}@nbi.ku.dk

Abstract

In this work we present an integrated system aimed at data management and processing for scientific areas that work with very large datasets. The Imaging Data Management System, IDMS, seeks to support researchers in all steps of their research, starting with transfer of data from the lab, over managing and analysing the data, to final archiving of the essential research project results. While IDMS is in fact hosted locally at the university we seek to provide a user experience that is as close as possible to a generic cloud system, in order to allow users to share and collaborate on their data seamlessly from anywhere.

1. Introduction

We see an enormous increase in data output from large research facilities, e.g. supercomputers and synchrotrons, which challenges the researchers in several aspects, including just storing and moving the data around. Processing the data also becomes non-trivial once you exceed the memory or computational capacities of a conventional workstation, since the researchers can then no longer just rely on their own computers and familiar tools. While the computational power of conventional workstation has grown and enabled researchers to do more analysis, so has the size of scientific data sets.

Since the data grow at an even faster rate than the compute power, so does the need for researchers with scalable computing skills. Yet, the common researcher appears to have fewer computing skills than in the past, and even basic acquaintance with batch-processing on remote resources often poses a major challenge.

* This material is based upon work supported by Innovation Fund Denmark.

CloudNG'17, April 23, 2017, Belgrade, Serbia.

http://dx.doi.org/10.1145/3068126.3071061

1.1 Motivation

An increasing number of scientific areas are collecting data at exponential rates . The Large Hadron Collider at CERN is a well-established facility producing more than 25 PB of data per year[1], and it has a dedicated grid computing system set up to store and process that data. However, several other areas are beginning to produce data at similar rates. For example astronomic observations, X-Ray/neutron imaging and climate research all produce very large datasets by now. A research area such as high energy physics has a long tradition for data management, and the users have grown comfortable with managing complex data structures through custom interfaces, such as a Globus-based grid system[2]. Researchers in other areas, such as X-Ray imaging, are far less comfortable with complex data management and processing systems. A researcher who needs to use a synchrotron for X-Ray imaging of an object, will first go to the facility with the object and perform the imaging experiment, which stores data at the local system. The researcher then copies the data to a set of USB hard drives and carries the data home that way. Back home the researcher then opens the data, typically in MATLAB, and performs data analysis. Sharing the data with other researchers is done by physically passing the USB hard drives, and loss of data due to hard drives being lost or failing is not uncommon. Thus the current modus operandi is not a reliable approach: data can be lost in transit or during sharing, processing is slow, and for new and very large datasets a conventional workstation with MATLAB is simply not sufficient.

To address these challenges we have developed the Imaging Data Management System, IDMS, that supports the entire life-cycle of the data: from the time it leaves the data production facility, through collaborative analysis and all the way to the final research data archiving at the end of the project. IDMS is run as a private cloud[3] system at the University of Copenhagen. There are several reasons to keep the system internal, most importantly cost but also legal concerns with e.g. medical images. However, the users are able to use IDMS from anywhere in the world as with public cloud systems. While internal users are able to log in with their university credentials, the system is not limited to local users, and IDMS allows researchers to seamlessly collaborate with people outside University of Copenhagen.

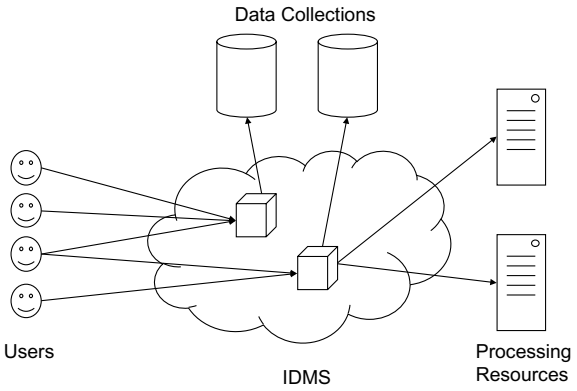


Figure 1. An IDMS schematic

2. The Imaging Data Management System

The Imaging Data Management System can in principle handle any kind of data, but is optimized for data that can be seen as images, either in 2D or as 3D representations. The researcher can preview data, as images, directly in a web-interface, and easily manage groups for transparent data sharing.

2.1 Design

IDMS is based on the Minimum intrusion Grid, MiG[4]. In MiG each user has a home directory where files and directories are stored. Sharing is done in workgroups, which are implemented as Virtual Grids, VGrids[5], where the researcher can create and administer groups without any administrator involvement. In addition to sharing files in a workgroup it is also possible to add computing resources to a workgroup and thus share processing power as well, in essence allowing a research project to define an internal project cloud. Individual researchers thus have a single entry point to all their data, and projects are represented as directories in their home directory. It is possible to hierarchically define projects, to create sub-projects and thus refine the granularity of sharing data and resources. Figure 1 shows a schematic of IDMS with four users, connected to two workgroups, one of which has two computing resources.

2.2 Interfaces

Access to data is offered through a number of interfaces that serve different purposes. The primary interface is the web presentation, through which users can up- and download data and where images can be previewed in 2D or 3D. Figures 2 and 3 show examples of the web interface with both views. Users can manage workgroups and configure the remaining interfaces there, too. Additionally it comes with archiving facilities and it also offers an import portal for data from known sources. This way a user can eliminate the need for migrating data to USB drives, and instead import data directly from the remote facility. This pull feature ensures

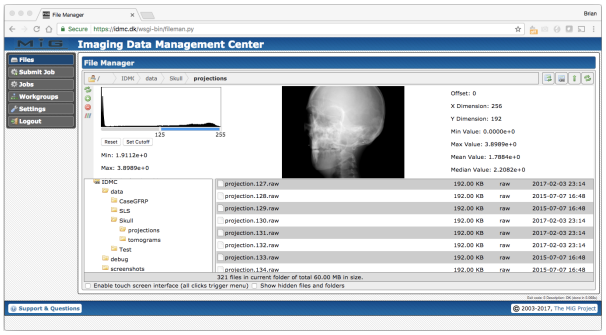


Figure 2. A 2D Image view snapshot

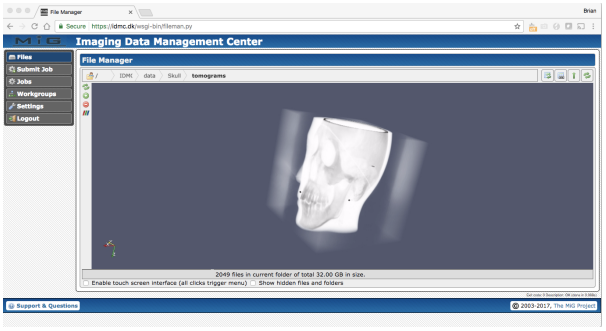


Figure 3. A 3D Image view snapshot

that large datasets can be transferred into IDMS even after the researcher has physically left the data acquisition site, which would not be possible with only a push model.

In addition to the basic web interface researchers have two more cloud-like interfaces in IDMS: their entire datahome can be mounted as a networked drive, using SFTP or WebDAVS. In addition to this personal home data mount point, a user can also create anonymous mount points, which rely on an auto-generated random string for access control. These anonymous mount points refer to a directory in the user's space, and only expose data in that subset of the user-home. The anonymous mount points are typically used e.g. for lab PCs and for instruments like microscopes, to automatically upload data to IDMS without exposing any other user data if the instrument should get compromised. A researcher may also choose to have parts of the dataset available in the local filesystem on their notebook and/or workstation, which can then be synchronised to IDMS with the built-in synchronisation solution. As in similar synchronisation systems a researcher may also use this feature to keep two or more machines synchronised in addition to the copy on IDMS. If a researcher needs to exchange data with a collaborator without explicitly sharing the data, IDMS offers direct share-links. They can be used to allow a collaborator to download a specific file or files from a specific directory, or upload data to a specific directory. This is particularly con-

venient for exchanging data with project partners without an account.

IDMS also supports the final archiving stage of the scientific data management life-cycle. Once ready for that the researcher simply uses the corresponding functionality on the web interface to select files and create an archive of them for long-term persistent storage. The researcher is then asked to fill in metadata that describe the scientific data, and decide if the data should be made publicly available or not. If the data is made available IDMS offers a digital object identifier, DOI, so that the researcher can receive citations on the dataset.

2.3 Infrastructure

IDMS is running on a highly elastic and fault-tolerant infrastructure. Primary storage is based on JBODs with 60 SAS hard drives of 6TB each. Every JBOD is connected to two servers. Currently each server has three connected JBODs. Both servers operate half the disks which it combines into five ZFS[6] RAIDZ-2 sets of 18 disks, each capable of sustaining a 10Gbps link. This setup allows a server to act as a redundancy server if the other server should fail by taking over responsibility of the 90 disks that become orphaned. The nodes are joined in a GlusterFS[7] filesystem that presents the entire storage system as a flat namespace. This approach is chosen over a parallel filesystem because is allows IDMS to recover datasets directly from individual JBODs in a disaster scenario. In parallel with the disk based storage IDMS hosts a tape-storage system for long term archives. Tape storage is run through Tapr[8] which allows us to manage tape without going through expensive backup systems. At the time of writing IDMS has 1.6PB disk-storage online, which hosts 1.1PB of user data. In addition there is 1.5PB tape storage, of which only a small fraction is in use.

2.4 Processing

IDMS relies heavily on the underlying MiG infrastructure for the automatic processing of tasks. MiG comes with a number of services to handle everything from job queuing and scheduling to basic web and advanced file access interfaces.

Event handling is a just another MiG service which allows efficient detection of file changes and then acts on those changes based on a set of rules. The rules are defined on a per-user/workgroup basis and they can be configured to do just about anything that a user could manually do on the Grid. In the IDMS context we focus on just two kinds of rules, namely those to handle actual processing and those to cascade events. The latter work only locally to trigger other rules, e.g. to force re-processing of data when analysis tools are updated. The processing rules submit a grid job for execution on a remote resource to offload the computational work. IDMS uses both kinds of event rules under the hood to provide the automatic previews of image data in particular. In addition users can explicitly configure their own rules to handle their analysis flow(s) as one or more chains of data-driven processes. For flexibility the flow paths can additionally be set up to branch and merge if needed.

Similar, though not cloud-enabled, solutions include CSP-builder[9], where a scientific workflow is defined in CSP, and Taverna[10] which has no underlying formalism. Many other workflow systems exist, but all are end-to-end descriptions and not as flexible as the trigger system in IDMS.

Any actual data processing maps to the submit action, which takes a job description and the file event details as input. It creates and submits a job specific to the particular rule. In that way the required functionality is provided, without ever allowing arbitrary user code execution inside the core IDMS system, but only on the dedicated compute resources.

The actual event handling is implemented in an event manager daemon, which monitors file changes and reacts accordingly as outlined in Figure 4. It relies on the Python *watchdog* [11] module, which in turn uses the *inotify* feature when run on a Linux kernel. It detects and reacts to relevant file changes without the need for continuously searching through the entire file system; repeatedly traversing the entire file system looking for changes would not only take a long time, but also put a heavy load on the disk subsystem of a server, and thus make ordinary user I/O slower. By using inotify, on the other hand, immediate detection of file changes is achieved without the disk search overhead. The event manager monitors the file system and keeps a database or dictionary of registered trigger rules to match file changes against. The rules are read from disk during start up, and then it uses another watchdog-listener to register any new ones along the way. When the daemon receives a notification about a file change through the listener, it first looks up the path in the trigger rule dictionary, which maps file patterns to actual rules. Thus, it is simple to run through the rule keys to detect any matching patterns. In case of a match, the corresponding rule is further inspected to see if the actual state change also matches the rule, and if so the kind of action determines the further handling.

Each rule is bound to a VGrid and associated with a path or path pattern to match. The path is always considered relative to the VGrid shared folder and any file changes that match the path (pattern) will be considered in trigger handling. Trigger events additionally come with one of four file state changes: *created*, *modified*, *deleted* and *moved*. Users can set up trigger rules to match one or more of those state changes for specific files, or for all files that match a given pattern with wildcards.

When a file change matches both a trigger file pattern and the associated state change, the resulting trigger action will be invoked for the file. If the trigger rule is set up with a cascading action the corresponding file state change trigger

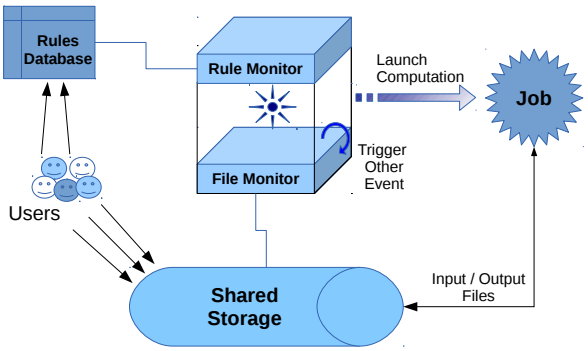


Figure 4. Event Manager Overview

event will be dispatched for the file(s) provided as trigger arguments.

If on the other hand a trigger rule is set up with the submit action, a hit will result in the supplied job template being filled and submitted based on the file, that triggered the event. For ordinary user trigger rules the user provides a job template used by the resulting jobs. For preview generators IDMS supplies the job template file on behalf of the user. In any case the job template resembles the usual grid job descriptions, only using variable keywords in places where the job depends on the triggering file. It is currently possible to use the following variable keywords to be expanded in the job template:

- +TRIGGERPATH+ : Full trigger file path
- +TRIGGERRELPATH+ : Relative trigger file path
- +TRIGGERDIRNAME+ : Directory part of trigger file path
- +TRIGGERRELDIRNAME+ : Directory part of relative trigger file path
- +TRIGGERFILENAME+ : Filename part of trigger file path
- +TRIGGERPREFIX+ : Base filename of the trigger file
- +TRIGGEREXTENSION+ : File extension of the trigger file
- +TRIGGERCHANGE+ : Kind of file state change
- +TRIGGERVGRIDNAME+ : VGrid defining the rule
- +TRIGGERRUNAS+ : Rule owner ID

All occurrences of those variables are replaced by the values for the trigger file, and then the filled job template is submitted on behalf of the user who created the trigger rule explicitly - or implicitly by enabling IDMS previews for a folder.

When a submit action is handled for a file, the event manager generates all the keyword variables mentioned above in a lookup dictionary for that file and rule. Then it opens the associated job template using the saved rule argument, and expands all keyword variables there using the lookup dictionary. Finally it saves the filled template in a new temporary job file, and submits it to the grid queue on behalf of the rule owner.

Processing flows are basically a set of rules that have input and output in common in a way that output from one rule is the target input for the next.

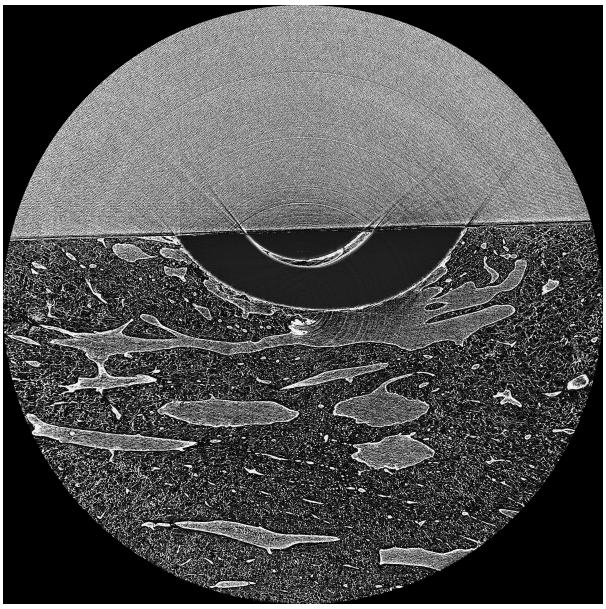


Figure 5. X-Ray image of a titanium implant in a goat jaw-bone, it is easy to see that the upper part of the image is corrupted during image acquisition

3. Use Cases

3.1 X-Ray imaging

This X-Ray imaging case uses data from a dental implant research project. The overall flow in the project starts by a goat having a tooth replaced by a titanium implant, after a while the jaw holding the implant is removed and imaged on a synchrotron to produce a 3D volume image. The purpose is to determine the connectivity between the titanium implant and the jaw-bone. Figure 5 shows an example of the images that are produced.

3.1.1 Acquisition

Data is acquired with very high resolution on the European Synchrotron Radiation Facility, ESRF. Imaging is done by acquiring one image, rotating the object one third of a degree and repeating. Each image is 3487 by 3487 pixels stored at 32-bit floating point, thus one image takes up just over 46MB, and all images for an object sum up to more than 54GB. One hundred samples then accumulate to more than 5TB in total.

3.1.2 Data Transfer

The actual data we work on was transferred on 6 external USB drives, but with the data import feature now ready in IDMS, we can eliminate the use of USB drives for the next set of experiments. Back at the university the data from the USB drives were uploaded to IDMS by the user. This was simply done by mounting IDMS as a net drive and using drag-and-drop for the data transfer. As a USB drive can

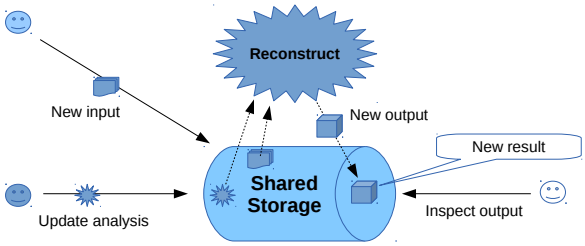


Figure 6. Automatic Processing and Reprocessing

basically transfer one picture per second, it took 20 minutes to transfer the pictures for one sample and 30 hours to upload data for 100 samples.

3.1.3 Automatic Processing

While the data coming from the synchrotron are technically images, they are not available in any *common* image data format. Each facility has its own data format and they are stored uncompressed. Thus, individual images are 46MB in size, and they are not directly viewable. When images are uploaded to IDMS they trigger a job that generates preview images in the PNG format, as well as a small program that collects statistics on the images, min, max, median and variance. This automatic processing corresponds to the New input part of the flow in Figure 6 only with the Reconstruct part replaced by a Preview generation.

3.1.4 Reprocessing

The experiment is the first of its kind in the world and thus no established method for analysis exists. This means that the analysis program frequently changes, and new analysis must be done very often. To facilitate this a trigger-rule was set up for the analysis files, and when one changes, all analysis is automatically re-run for all images. This makes the iterative method of producing the best possible analysis method far more efficient than if a researcher were to download one or two images and optimise the analysis for those and then test if the total dataset is fit for that analysis code. The automatic re-processing corresponds to the Update analysis part of the flow in Figure 6.

3.2 Climate Modelling

Climate modelling is done in pure simulation environments, and the result is not images per se, but all data are presented as visualisations in the form of pictures. Since the work flow consists of a simulate step followed by a process results step, a climate simulation is in fact similar to a physical imaging facility.

3.3 Acquisition

Climate models are long-running simulations that produce very large output-files. A common simulation will run on about 16000 processor-cores and produce output for each simulated three days. The researchers in the field would

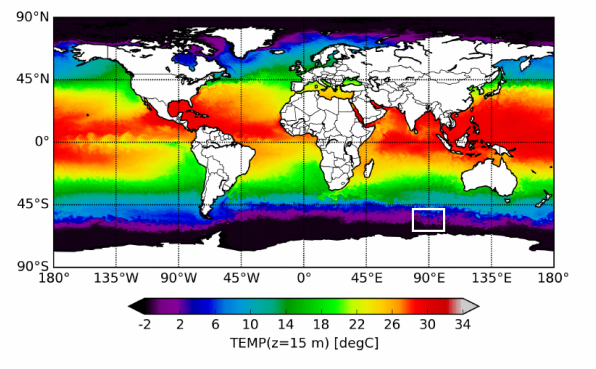


Figure 7. Global surface temperature from a climate simulation

optimally prefer to save the state for every simulated six hours, but the output size usually makes that infeasible. A simulation is typically a system with a spatial resolution of 3600x2400 grid points, and with 62 layers. Each layer represents a single parameter and one such parameter can be visualised as a picture. As an example Figure 7 is the visualisation of global surface temperature layer. When using 32-bit floating point numbers, a single parameter layer alone takes up 3600x2400x62x4 = 2GB. Since a simulation may involve as many as 74 parameters, it means that saving one full state takes approximately. 150GB. Depending on the speed of the supercomputer the three-day simulation may be done in as little as one hour. Thus a 30-day simulation produces approximately 100 TB of data.

3.3.1 Data Transfer

Supercomputer facilities always support remote data access, so it was trivial to apply the data import feature for the data transfer to IDMS.

3.3.2 Automatic Processing

The first step in post-processing of the data involves creation of time series for each parameter - such as temperature, velocity, salinity, etc. The simulations are saved in NetCDF format, using a layout where each saved time step is composed of a sequence of parameters. This interleaved layout is sketched in Figure 8. The associated analysis programs work on exactly one parameter at a time, thus building the time series for surface temperature alone requires traversing all 100TB of data. Once that is done the process can be repeated for salinity and so forth. Even with a dedicated 10Gb connection this is 28 hours per parameter, and 85 days for a complete simulation. I.e. close to three times the run-time of the simulation. IDMS presents the time-series as images to the researcher, which helps select interesting datasets for further analysis without downloading all data; something that would essentially also be infeasible due to the data sizes.

Figure 8. Sketch of climate data in NetCDF format.

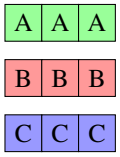


Figure 9. Simulation results reformatted to individual parameters.

No analysis approach uses all parameters at one time - in fact most analyses are on only a single parameter, and only a few exceed the use of three parameters. So IDMS is set up for processing the data for a very simple data split, where the entire dataset is read through once and extracted into individual parameters. They are then saved to individual files for each parameter. This means that the 28h job of reading through data is now done exactly once, and all time series analysis is then reduced to reading through the actual data for the parameter in question. The transformation changes the data as seen in Figure 8 to a layout as sketched in Figure 9. We are currently investigating solutions for performing the transformation at upload time in a streaming manner. Then we can eliminate even the one 28h read-through of a dataset.

3.4 Reprocessing

More advanced processing takes place as well, making advanced statistics and aligning with known physics. This is done with the trigger system in IDMS and utilises mostly the re-processing feature, where changes in an analysis program triggers re-processing of all datasets.

4. Conclusion

The Imaging Data Management System is an internally hosted cloud system from the University of Copenhagen. It can be categorised as a private cloud system, but it offers full access for users from outside the university in order to facilitate collaboration between employees and external partners. In principle there would be no problem in moving the complete software stack to a commercial cloud vendor. However, with the current size and traffic numbers this solution would not be economically feasible with the price levels the market has today. The system currently hosts more than 1PB of data, and allows users to seamlessly view their images directly from a web interface, even for images that do not follow standard image formats.

The system is continuously expanding its feature set, but presently the features that users are offered are; a web-interface for file upload and download as well as image preview and management. The users may mount directly into the storage using SSHFS or WebDAVS, and they may have parts of their local computer mirrored in IDMS, or have explicit backups done to IDMS. In addition, IDMS offers permanent archiving of research data, including an option for enabling public access to the data.

Acknowledgements

The authors would like to thank Professor Else Marie Pinholt and Professor Markus Jochum for their input to the use of IDMS and the permission to use their use cases, the X-Ray imaging and Climate use cases respectively.

References

[1] Cian O'Luanaigh. Cern data centre passes 100 petabytes. `https://home.cern/about/updates/2013/02/cern-data-centre-passes-100-petabytes`, 2013.

[2] Ian T Foster. The globus toolkit for grid computing. In *ccgrid*, volume 1, page 2, 2001.

[3] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[4] Jost Berthold, Jonas Bardino, and Brian Vinter. A principled approach to grid middleware status report on the minimum intrusion grid. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'11, pages 409–418, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Henrik Hoey Karlsen and Brian Vinter. Vgrids as an implementation of virtual organizations in grid computing. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2006. WETICE'06. 15th IEEE International Workshops on*, pages 175–180. IEEE, 2006.

[6] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, 2003.

[7] GlusterFS. https://www.gluster.org/.

[8] Klaus Birkelund Jensen. High performance tape streaming in tapr. In *Proceedings of the 38th WoTUG conference on concurrent and parallel systems*. Open Channel Publishing Ltd., 2016.

[9] Rune Møllegård Friborg and Brian Vinter. Cspbuilder-csp based scientific workflow modelling. In *CPA*, pages 347–363, 2008.

[10] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, page gkt328, 2013.

[11] Yesudeep Mangalapilly. watchdog. `https://pypi.python.org/pypi/watchdog`, 2015.

Communicating Process Architectures 2017
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2017

Automatic Code Generation for Library Method Inclusion in Domain Specific Languages

Mads Ohm LARSEN [1]

Niels Bohr Institute, University of Copenhagen, Denmark

Abstract. Performance is important when creating large experiments or simulations. However it would be preferable not to lose programmer productivity. A lot of effort has already been put into creating fast libraries for for example linear algebra based computations (BLAS and LAPACK). In this paper we show that utilizing these libraries in a DSL made for productivity will solve both problems. This is done via automatic code generation and can be extended to other languages, libraries, and features.

Keywords. Code Generation, Performance, Bohrium, GPGPU

Introduction

Many experimental sciences utilize high-performance libraries for simulations and experiments. A lot of time and effort have been put into making these libraries specially suited for various tasks, such as matrix manipulation (BLAS[2] [1]) or linear equation solving (LAPACK[3] [2]). Since many of these libraries have been through multiple development iterations, their APIs (Application Programming Interface) might not be very rigid. This can be seen with the various implementations of for example BLAS such as `cBLAS`, `Accelerate`[4], `clBLAS`, OpenBLAS, GotoBLAS, etc. These all have seemingly similar interfaces, but their naming conventions might be different, and even two different versions of the same shared object file might not link in the same way. As an example `cBLAS` and `Accelerate` have the same interface for the same architecture, but for different operating systems. `clBLAS` has a different interface, but is also for a different architecture, namely GPGPUs. `LACPAKE` has a different interface for the same architecture as `Accelerate`, but again, for a different operating system. Thus, calling the same LAPACK method with the either `LAPACKE` or `Accelerate` installed, will demand two different implementations of your simulation.

The reference BLAS implementation, which was created in 1979, is written in FORTRAN and is, subjectively, a bit out-dated. For example when calculating matrix multiplications, the programmer needs to be aware of FORTRAN being column-major in its memory management. The same is true for various other FORTRAN specifics. This can be seen in Listing 1, where a standard matrix multiplication call is being issued.

[1]Corresponding Author: *Mads Ohm Larsen, Blegdamsvej 17, 2100 Copenhagen OE.* E-mail: `ohm@nbi.ku.dk`.

[2]Basic Linear Algebra Subprograms

[3]Linear Algebra PACKage

[4]Apple's BLAS-implementation.

```
// Calculates
//   C := alpha*op(A)*op(B) + beta*C
// where op(X) is either X or X^T

cblas_sgemm(
  CblasRowMajor, // Memory management
  CblasNoTrans,  // Transpose A?
  CblasNoTrans,  // Transpose B?
  m,             // Number of rows of op(A)
  n,             // Number of columns of op(B)
  k,             // Number of columns/rows of op(A) and op(B)
  1.0,           // Alpha argument
  A_data,        // Array of size m*k
  k,             // First dimension of A / Stride of A
  B_data,        // Array of size k*n
  n,             // Stride of B
  0.0,           // Beta argument
  C_data,        // Array of size m*n
  n              // Stride of C
);
```

Listing 1. cBLAS general matrix multiplication.

```
import numpy as np
np.matmul(a, b)
```

Listing 2. Matrix multiplication with NumPy.

Listing 2 shows the same $\mathbf{AB}$ matrix multiplication, but with a higher level of abstraction, here written in Python using the library NumPy [3].

As a scientist using these libraries, you need to be aware of the different implementations currently available on your system. You might also not have the same setup on your desktop machine as the supercomputer on which you will later be running the code.

Since BLAS affords the greatest performance, programmers are forced to use it if they want their simulations to perform. Various attempts at implementing BLAS have been made [4,5,6], most of which just mimic the FORTRAN interface (cBLAS, Accelerate, clBLAS) and some that implements whole matrix libraries around it such as ViennaCL [7].

This is of course not only true for BLAS and LAPACK, but also for other libraries that speed up performance or implement hard to do algorithms.

In this paper we extend our current programming suite with the ability to automatically choose which library to link against and which API to use based on what is currently available on the system. We will focus on Python and NumPy, and the fact that the user should be unaware of the underlying library being used. Bohrium[5] [8,9,10], as a DSL (Domain Specific Language), has this performance and productivity relationship, however it loses some of NumPy's BLAS methods in the midst. This paper will rectify that by introducing *extension methods*, that can be called from Python and which, in the end, we will wrap around NumPy's methods, so that the user will not even realize that they are calling specific BLAS methods.

The technique shown in this paper can be used for other languages and libraries as well; to easily choose between already installed libraries and auto-generate code for their inclusion and usage.

[5]Available at http://www.bh107.org.

1. Domain Specific Languages

DSLs are a way to minimize the particular knowledge needed by the programmer to write domain specific code. For example with the previously mentioned BLAS method calls, one would need to know about column-major versus row-major memory management and other intrinsic parameters of the library methods. With a DSL we can circumvent that knowledge gap. The programmer will still have to be taught a new concept, namely the DSL in question, but the knowledge gap here might be smaller.

When creating a DSL you need to include a lot of different methods in order to still encompass the same performance as with a more native programming language. Writing all the boilerplate code for including these library methods in a meaningful way will quickly become a tedious task. We should auto-generate as much code for these inclusions in the DSL as possible.

1.1. Bohrium

In the experimental sciences, array programming [11] have become a popular programming paradigm [12,13]. With it, one can express linear algebra problems without using pointer arithmetic and other such low-level constructs.

NumPy implements the array programming paradigm for use in Python. Bohrium wraps or overrides NumPy's methods and defines a virtual machine, which executes instructions on arrays. Bohrium can be configured to run on single CPU, multi-core CPUs, or GPGPUs, and should thus perform faster than NumPy, since Python is inherently single-threaded.

The rest of this paper will use Bohrium as an example of a DSL that needs extension methods.

1.1.1. Kernels

In order to run on multiple architectures, Bohrium generates kernels and compiles them just-in-time (JIT) from an accumulated instruction set. These kernels can be specialized to run on CPUs (OpenMP) or GPGPUs (OpenCL). This means that inside these kernels, we can call other C or OpenCL library methods, like we normally would, had we coded everything by hand. When computing simple things, for example summing up a vector, this will not be necessary, however when we want to do more specialized things, for example matrix multiplication, this might improve performance significantly.

1.1.2. NumPy's BLAS *Implementation*

NumPy links against an existing BLAS implementation when first installing it onto a new system. From then on, when calling upon matrix-matrix, vector-matrix, or vector-vector operations, NumPy will call BLAS to accelerate the code. This greatly improves the performance of these matrix-matrix and other operations. Unfortunately, Bohrium has to do the matrix-matrix operations as a reduction of multiplications, thus doing so as multiple nested for-loops in the CPU/GPGPU kernels. The nested for-loops are of course slow compared to optimized BLAS calls, which leads us to the topic at hand, namely extension methods for Bohrium, allowing us to specialize these kind of method calls and have Bohrium call the same BLAS methods as NumPy.

2. Extension Methods

Bohrium's extension methods are methods that go a bit beyond the regular NumPy calls and implementations for Bohrium. These are implemented as a flush in the Bohrium instruction

set, followed by a call into the C++ backend. Here all the extension methods have the same function-signature, namely they get passed the instruction at hand, with all of its attributes, for example the arrays, the sizes, and so on, together with an engine when utilizing OpenCL.

They are in essence library methods that get called from Bohrium byte-code instructions, which is a DSL.

This paper will focus on a subset of methods from BLAS, LAPACK, and OpenCV.

2.1. BLAS

The BLAS extension methods are divided into two parts: `cBLAS/Accelerate` and `clBLAS`. Both `cBLAS` and `Accelerate` implement the BLAS interface verbatim in C. `clBLAS` are used for running the kernels with OpenCL and thus its interface is quite different.

It is important to note, that even though they have similar names, coding for `clBLAS` is vastly different from coding for `cBLAS/Accelerate`, as you, the programmer, need to know many OpenCL details prior to setting up these calls.

2.2. LAPACK

LAPACK is composed of linear algebra solvers, which are all useful when having a lot of simultaneous linear equations to be solved. The reference implementation is again written in FORTRAN and wrappers are then made for other languages, such as C. `Accelerate` implements these for macOS in a similar way to `lapacke`, which is the standard implementation on Linux systems.

2.3. OpenCV

A completely different library is OpenCV [14,15], which works mostly with images and are able to do various transformations and computations with them. One such operation is thresholding, which looks at the individual pixels and caps them at a certain color value. Another operation is eroding and dilating, which looks at neighboring pixels in order to figure out if the current pixel needs to change. These operations are implemented in C++ around OpenCV's own matrix representation.

3. Code Generation

The code generation for Bohrium's extension methods are done together with compiling Bohrium itself, thus allowing it to link with the current libraries installed on the system. This is however not JIT-compiled, as talked about previously, as it is only the actual kernels that are JIT-compiled. Bohrium will automatically find an installed BLAS and LAPACK when installing it and put the location into its configuration file, for linking when JIT-compiling.

We do code generation in three steps. First we define a JSON-file with all the methods and their options. Second we create a skeleton/template, for the code generator to fill. And lastly, we run the actual generator, which combines these two (or more) files to a full-fledged extension method.

As already stated, we can write all of these extension methods by hand, however, this will quickly become a tedious task, as most of them are just composed of loading matrices and then calling the various BLAS methods.

3.1. JSON

First we create the JSON-file. This file (Listing 3) will contain all the methods, that we want to generate as well as their options. The options will later be turned into boolean values where only the ones specified are true.

```
{
  "methods": [
    {
      "name":    "gemm",
      "types":   [ "s", "d", "c", "z" ],
      "options": [
        "layout", "notransA", "notransB",
        "m", "n", "k",
        "A", "B", "C"
      ]
    },
    ...
  ]
}
```

Listing 3. JSON-file with options.

Here, all methods are listed in the `methods`-array, each with a name, types, and options. The `name` paramenter is the name of the function in the library, here BLAS. `type` are the four BLAS types, namely single- and double-precision, complex, and double complex. Not all methods have all four types, so we specify which ones apply.

From this one code example we can generate the four different `gemm` methods, one for each type, and later let it be called from Bohrium and thus from Python.

3.2. Templating

In the templating step, we take four different files, namely `header.tpl`, `body.tpl`, `body_func.tpl`, and `footer.tpl`, and compile them into the final library. The chosen templating library, `pyratemp`, uses the syntax `@!VAR!@` for substitutions. This is abused for file inclusions in a separate Python script, where we load all of the contents from the previously mentioned files and substitute in various variables. Likewise, `if`-statements, that we will use heavily, have the syntax `<!--(if BOOL)--> ... <!--(end)-->`.

The `header.tpl` and `footer.tpl` files contain the boilerplate, that needs to go before and after all of the functions. These could be the C inclusions, the namespace definitions and so on. A sample of `header.tpl` from Bohrium can be seen in Listing 4. At some point, we write `@!body!@`. This tells the code generator, that we want the contents of the `body.tpl` to appear here. Likewise this is done with `@!footer!@` for the footer content.

In the Bohrium example footer we have extern C declarations for all the generated functions. This includes a constructor (`blas_@!name!@_create()`) and a destructor (`blas_@!name!@_destroy()`). The `@!name!@` again comes from the JSON-file.

The body, which is presented in Listing 5, is the actual method implementation, which does the setup of various components. For this example, we have omitted some of the setup in this file.

```
#include <bh_extmethod.hpp>

#if defined(__APPLE__) || defined(__MACOSX)
    #include <Accelerate/Accelerate.h>
#else
```

```
    #include <cblas.h>
#endif

#include <stdexcept>

using namespace bohrium;
using namespace extmethod;
using namespace std;

namespace {
    @!body!@
} /* end of namespace */

@!footer!@
```

Listing 4. The `header.tpl` used in Bohrium for BLAS.

```
struct @!uname!@Impl : public ExtmethodImpl {
public:
    void execute(bh_instruction *instr, void* arg) {
        // All matrices must be contigous
        assert(instr->is_contiguous());

        // A is a m*k matrix
        bh_view* A = &instr->operand[1];
        // We allocate the A data, if not already present
        bh_data_malloc(A->base);
        void *A_data = A->base->data;

        <!--(if if_B)-->
        // B is a k*n matrix
        bh_view* B = &instr->operand[2];
        // We allocate the B data, if not already present
        bh_data_malloc(B->base);

        assert(A->base->type == B->base->type);
        void *B_data = B->base->data;
        <!--(end)-->

        ... // Switch
    } /* end execute method */
}; /* end of struct */
```

Listing 5. Implementation snippet.

We first load the A and B matrix pointers from the instruction operands. The B-matrix is only loaded if the `B` option was set in the JSON, which is the `if if_B` scope in the above snippet. Various other setup steps, that are identical for the methods we are generating, can be present here.

In the `body.tpl`-file we have a `switch`-statement, that looks at the types given. This is because Bohrium work with two different integer types, `BH_INT32` and `BH_INT64`, as well as two different floating point types, `BH_FLOAT32` and `BH_FLOAT64`. We also operate with two different complex types, 64-bit and 128. All of the ones needed can be specified in the JSON-file. The body of this switch, in Listing 6, comes from the last file, the `body_func.tpl`-file.

```
switch(A->base->type) {
  @!func!@
```

```
  default:
    std::stringstream ss;
    ss << bh_type_text(A->base->type) << \
          " not supported by BLAS for '@!name!@'.";
    throw std::runtime_error(ss.str());
} /* end of switch */
```

Listing 6. Switch snippet.

`@!func!@` will be substituted in looped over each type. That is, for example for `gemm` we give four cases, because we have four different types.

We can now look at the `gemm` method call from BLAS, which is in Listing 7. This should look at lot like the actual call from Listing 1, just with more added for other cases.

```
case @!utype!@: {
  @!alpha!@
  @!beta!@
  cblas_@!t!@@!name!@(
    <!--(if if_layout)-->   CblasRowMajor, <!--(end)-->
    <!--(if if_side)-->     CblasLeft,     <!--(end)-->
    <!--(if if_uplo)-->     CblasUpper,    <!--(end)-->
    <!--(if if_notransA)--> CblasNoTrans,  <!--(end)-->
    <!--(if if_transA)-->   CblasTrans,    <!--(end)-->
    <!--(if if_notransB)--> CblasNoTrans,  <!--(end)-->
    <!--(if if_diag)-->     CblasUnit,     <!--(end)-->
    <!--(if if_m)-->        m,             <!--(end)-->
    <!--(if if_n)-->        n,             <!--(end)-->
    <!--(if if_k)-->        k,             <!--(end)-->
    @!alpha_arg!@,
    (@!blas_type!@*)(((@!type!@*) A_data) + A->start),
    k,
    <!--(if if_B)-->
    (@!blas_type!@*)(((@!type!@*) B_data) + B->start),
    n<!--(if if_C)-->,<!--(end)-->
    <!--(end)-->
    <!--(if if_C)-->
    @!beta_arg!@,
    (@!blas_type!@*)(((@!type!@*) C_data) + C->start),
    n
    <!--(end)-->
  );
  break;
}
```

Listing 7. Inner part of switch.

There is a lot of `if`s in this method call, however, if we look at the JSON-file (Listing 3) we only set `layout`, `notransA`, and `notransB` for `gemm`. That means that only these lines will get generated in the final library, for this method and type. The lines not surrounded by `if`s will of course get generated for all instances. Listing 8 contains the generated example for `gemm` with `BH_FLOAT32` type. This listing is almost identical to Listing 1, thus it should behave the same as the actual BLAS calls.

```
  case bh_type::FLOAT32: {
    cblas_sgemm(
      CblasRowMajor,
      CblasNoTrans,
```

```
            CblasNoTrans,
            m,
            n,
            k,
            1.0,
            (bh_float32*)(((bh_float32*) A_data) + A->start),
            k,
            (bh_float32*)(((bh_float32*) B_data) + B->start),
            n,
            0.0,
            (bh_float32*)(((bh_float32*) C_data) + C->start),
            n
        );
        break;
    }
```

Listing 8. Generated `gemm` for one case.

3.3. Generation

The generation in itself is handled by a short Python script, which reads in the template files and fills in all the variables and `if`-statements.

To generate these BLAS calls alongside Bohrium, we add them to the `CMake`-files, which will make sure to call this Python generator and generate the necessary libraries. The `CMake`-files will check if you have for example BLAS installed on your system, and then generate the BLAS extension methods. If you have `clBLAS`, the OpenCL version for GPGPUs, installed, it will also generate these methods and will let them overload the others, so that you are actually running OpenCL on your GPGPU.

4. Examples and Results

All of the following experiments have been run on an Intel Core i7-3770 3.4 GHz processor and a GeForce GTX 680 graphics card. The system was running Ubuntu Server 14.04 with Python 2.7 and NumPy 1.12.1.

`OpenBLAS` 0.2.19 was used for the `cBLAS` interface and `lapacke` 3.5.0 was used for the LAPACK interface. Both NumPy and Bohrium were linked to these.

In the following sections, the run times presented are an average of 10 consecutive runs.

4.1. General Matrix Multiplication

For general matrix multiplication we multiply two large matrices. For this example, we define $\mathbf{A}$ as a matrix of size 2000×3000 filled with random 32-bit floating point numbers and $\mathbf{B}$ as a random matrix with size 3000×4000. The resulting matrix $\mathbf{C} = \mathbf{AB}$ will then be the matrix product and of size 2000×4000.

For NumPy the `matmul` method will calculate the matrix multiplication. The same is true for Bohrium, both with and without the extension enabled. The only difference between the CPU and GPGPU/OpenCL Bohrium timings, is an environment variable `BH_STACK=opencl` which has been set. This instructs Bohrium to run with OpenCL and thus run on a GPGPU if possible.

The results for this general matrix multiplication can be seen in Figure 1. Here we clearly see that BLAS is fast. NumPy, which utilizes this, is just as fast. The first Bohrium bar shows the naïve n^3 algorithm. The second Bohrium bar shows that with the extension, it becomes just as fast as BLAS.

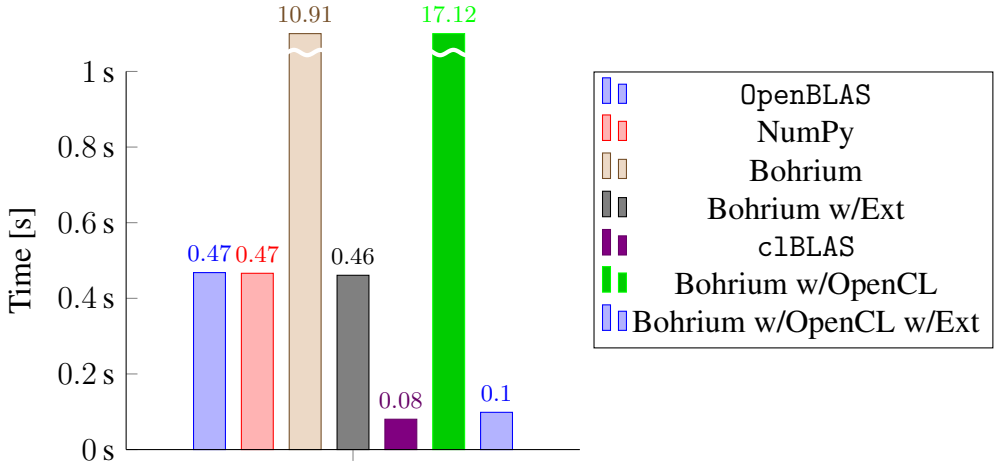


Figure 1. General matrix multiplication ($\mathbf{C} = \mathbf{AB}$) for multiple implementations.

NumPy does not utilize the GPGPU present on the system, `clBLAS` does however, and is thus much faster. Running Bohrium just with OpenCL does not speed up the code execution, however utilizing the `clBLAS` extension, it is just as fast as the native implementation.

It should be noted that the Python code used for the NumPy and all the Bohrium implementations is the same. All that is changed are environment variables and module imports. The implementation for `OpenBLAS` and `clBLAS` had to be handwritten. We gain a performance speed-up of $4.7x$ when running Bohrium on the GPGPU as opposed to NumPy on the CPU, with identical code, without having to hand write the kernels for the GPGPU, and thus not losing productivity or performance.

4.2. Specialized Matrix Multiplication

BLAS has a couple of specialized matrix multiplication methods, one of which is symmetric matrix multiplication. Here we assume that $\mathbf{A}$ is a symmetric matrix and then we just do the same calculation as before $\mathbf{C} = \mathbf{AB}$.

To ensure that our random matrix is actually symmetric, we first obtain a random matrix $\mathbf{R}$ and then let $\mathbf{A} = \frac{\mathbf{R}+\mathbf{R}^T}{2}$, where the addition and division are done element-wise. Since $\mathbf{A}$ is symmetric it has to be square, so instead of the dimensions from the previous test we let $\mathbf{A}$ be a 3000×3000 matrix. $\mathbf{B}$ is still a 3000×4000 random matrix.

NumPy does not have a specialized method for doing this symmetric multiplication, so we use the `matmul` from before again. The first Bohrium is also doing this `matmul`, while the second is calling our new extension method, `bohrium.blas.symm`.

From the various run times in Figure 2 we see that you would not get much speed-up using this specialized method, however running the same code on a GPGPU will grant you around a factor $3x$ speed-up. Again we note that the two Bohrium times with the extension method on only differ in an environment variable, while the `cBLAS` and `clBLAS` are two completely different programs that both had to be written from scratch.

4.3. LAPACK Solver

For the next example, we will look at a LAPACK solver, namely `gesv`, which is the one NumPy uses when calling `numpy.linalg.solve(...)`. This solves the equation $\mathbf{A}x = \mathbf{B}$ for x.

The tests are solving a system of 5000 random equations and assume them to be linearly independent.

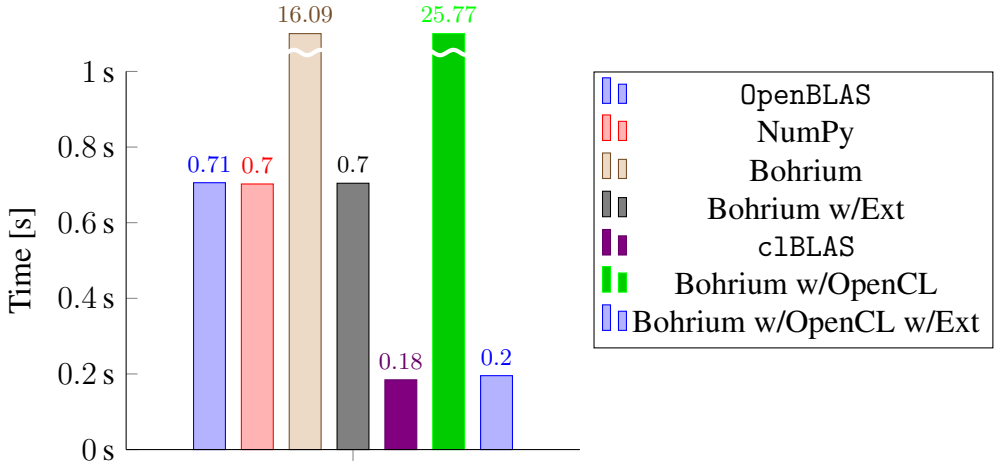


Figure 2. Symmetric matrix multiplication ($\mathbf{C} = \mathbf{AB}$) for multiple implementations.

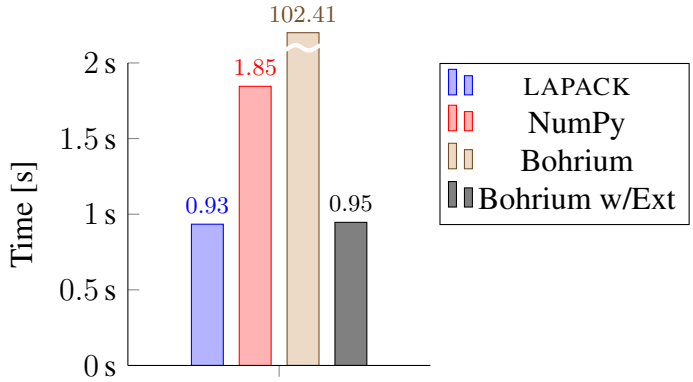


Figure 3. System of equations solver ($\mathbf{A}x = \mathbf{B}$) for multiple implementations.

Figure 3 shows the elapsed time for LAPACK, Numpy, and Bohrium with the LAPACK extension. Again, Bohrium is just as fast as the implementation that it extends.

When utilizing Bohrium for array programming, we can thus use the same Python/NumPy calls, but actually gain performance speed-ups, without losing the productivity gained from Python.

4.4. OpenCV

A more real-world scenario, instead of just computing matrix-multiplications or solving equations, could be utilizing some of OpenCV's functionality. Here we look at `erode`, which is a method that takes a binary image and an erosion kernel. The kernel is superimposed on the image. Everywhere the kernel is completely contained inside the image, that is, the binary sum is the same, we keep the center pixel, otherwise we remove it. This will give the effect of eroding the image.

Figure 4 show the running times for OpenCV, NumPy, Bohrium, and Bohrium with the OpenCV erode extension. Here we erode a random 10000×10000 pixel binary image. This time Bohrium on its own performs rather nice, but with the extension method we can again perform as fast as the library method.

Again, the code run by NumPy and Bohrium is identical, except for the module inclusion, while the Bohrium extension implementation just calls the extension method directly, instead of implementing it by hand.

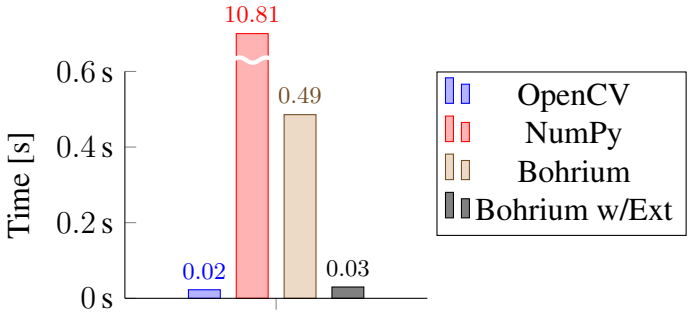


Figure 4. Erode for multiple implementations.

5. Conclusion

Productivity and performance are both important aspects of programming for the experimental sciences. Utilizing already implemented fast libraries are essential for having both. Most times however, you cannot use the same piece of software on both your desktop machine, whilst testing, and on a supercomputer, whilst doing the actual experiment or simulation.

Using DSLs that compile to your specific setup will improve productivity, however you usually cannot include the fast library calls.

In this paper it is shown that using templating we can get both performance and productivity out of DSLs, here shown with Bohrium.

Implementing the linear algebra examples, that is matrix multiplication and general solvers in Bohrium is easier than doing the same in C or C++, where the high-performance is. Letting Bohrium call into these high-performance libraries grants Bohrium both performance and productivity at no extra cost.

This approach can be further extended to other DSLs, which will then also gain these performance boosts.

6. Future Work

6.1. Boolean Attributes

Being able to specify boolean attributes in the JSON-file would mean we do not need both `transA` and `notransA`, as we could just specify that we want both generated. `transA` and `notransA` should never occur at the same time, at least in the examples. This should be done in some kind of inclusion/exclusion pattern.

6.2. DSL

A completely different approach to this would be to create a new DSL that allows us to generate all the code from one file. This will be a DSL for creating methods in other DSLs. So instead of having the JSON-file together with the template files, we could instead have a DSL, where we could specify all the various snippets and options. This way, all will be in one place and expanding this to multiple libraries might be even easier. This would also make the code generation even more customizable, as opposed to always having to have a header, footer, body, and body-function template file.

Acknowledgements

This work is part of the CINEMA project and financial support from **CINEMA**: The Alliance for Imaging of Energy Applications, DSF-grant no. 1305-00032B under The Danish Council for Strategic Research is gratefully acknowledged.

Thanks to Mads R.B. Kristensen for his valuable comments on this paper.

References

[1] BLAS Reference Implementation. `http://www.netlib.org/blas/`. Accessed: 2017-04-24.
[2] LAPACK - Linear Algebra PACKage. `http://www.netlib.org/lapack/`. Accessed: 2017-04-24.
[3] Travis E. Oliphant. *A Guide to NumPy*. Trelgol Publishing USA, 2006.
[4] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009.
[5] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997.
[6] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):4, 2008.
[7] Karl Rupp, Florian Rudolf, and Josef Weinbub. ViennaCL - A high level linear algebra library for GPUs and multi-core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*, pages 51–56, 2010.
[8] Mads O. Larsen, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter. Current Status and Directions for the Bohrium Runtime System. *19th Workshop on Compilers for Parallel Computing*, 2016.
[9] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.
[10] Mads R. B. Kristensen, Troels Blum, Simon A. F. Lund, Kenneth Skovhede, and Brian Vinter. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.
[11] Herbert Hellerman. Experimental Personalized Array Translator System. *Communications of the ACM*, 7(7):433–438, 1964.
[12] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers? *Computer*, 23(12):13–23, 1990.
[13] Philippe Mougin and Stéphane Ducasse. Oopal: integrating array programming in object-oriented programming. In *ACM SIGPLAN Notices*, volume 38, pages 65–77. ACM, 2003.
[14] Gary Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
[15] Itseez. Open Source Computer Vision Library. `https://github.com/itseez/opencv`, 2015.

Communicating Process Architectures 2017
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2017

Teaching Concurrency: 10 Years of Programming Projects at UCPH

Brian VINTER [1] and Mads Ohm LARSEN

Niels Bohr Institute, University of Copenhagen

Abstract.
While CSP is traditionally taught as an algebra, with a focus on definitions and proofs, it may also be presented as a style of programming, that is process-oriented programming. For the last decade University of Copenhagen (UCPH) has been teaching CSP as a mix of the two, including both the formal aspects and process-oriented programming. This paper summarized the work that has been made to make process-oriented programming relevant to students, through programming assignments where process orientation is clearly simpler than an equivalent solution in imperative programming style.

Keywords. Teaching, Concurrency

Introduction

Concurrency is an important aspect of computing, especially today with multiple cores in desktop machines, and, thus, it is naturally an important topic of computer science education. For more than 10 years, we have been teaching some kind of concurrency course[2] at the University of Copenhagen (UCPH). In this course, we teach both theory and practice about classical concurrency problems, such as race conditions and deadlocks.

At the end of the course, students are expected to know about CSP (Communicating Sequential Processes) concepts and notation, as well as some CSP-based libraries, such as the occam programming language [1] and PyCSP [2]. The students are also expected to be able to prove formal properties about processes in CSP-algebra, and design concurrent programs using the CSP paradigm.

These topics are taught using an array of assignments, both theoretical and practical, as well as lectures focusing on both of these aspects separately.

This paper focuses on the various programming projects, the practical assignments, that the students have received during the last 10 years of teaching concurrency at UCPH.

As the title indicate the work here represents ten years of assignments and the curriculum was not static in this period. Thus the workload is not necessarily comparable across the assignments, and the expected knowledge of formal CSP also varies significantly throughout the years. In the future work section, at the end of the paper, we try to outline how to remedy this.

[1]Corresponding Author: *Brian Vinter, Blegdamsvej 17, 2100 Copenhagen*. E-mail: `vinter@nbi.ku.dk`.
[2]Called ”eXtreme Multiprogramming”.

1. Related Work

Various systems have been developed for use in teaching concurrency. Most of these have to do with CSP, thought of by C.A.R. Hoare in 1985 [3]. Some examples of this is the occam programming language [1], C++ CSP [4], JCSP [5], and our own PyCSP [2]. Recently, the Go programming language [6] have also seen an increase in use. These systems all implement Hoare's theory to some degree, and can thus be used after having been taught some of the CSP-theory.

All of these are fairly similar in how they function with regards to setup and communicating via channels, which can be seen in listings 1, 2, and 3.

```
PROC writer (CHAN INT out!)
  out ! 42
:
```

Listing 1: occam process.

```
class Writer : public CSProcess
{
  private:
    Chanout<int> out;
  protected:
    void run();
  public:
    Writer(const Chanout<int>& i);
};

void Writer::run()
{
  out << 42;
}
```

Listing 2: C++ CSP process.

```
@process
def writer(chan_out):
    chan_out(42)
```

Listing 3: PyCSP process.

2. Projects

When teaching concurrency at UCPH, we make it clear from the start, that this is not *parallelism*. The distinction between concurrency and parallelism is often lost on some students, but it will be emphasized that using the techniques taught might not increase your performance, but you will instead be able to actually prove correctness about your programs. We also emphasize the ease of expanding on a process-oriented solution.

The projects we have given the students have always been projects that offers a large degree of freedom so that students may design their solutions more freely than what may be done with a strict specification. This, of course, will increase the time spent on the work, having to grade each student's assignment in a different way, since there are no one particular solution.

The following sections will contain a plethora of assignments and exams that we have given our students at UCPH. They have been divided into three categories: Games, Simulations, and Control Systems.

2.1. Learning Goals

When the assignments were originally made we did not formally specify the learning goals for each assignment, but merely adapted them to the progress in the class at the time the assignment was given. In order to make the assignments more easily usable we have reevaluated all assignments and marked the learning goals for each of the assignments.

Obviously deadlock avoidance and absence of race-conditions are common learning goals. Many assignments are deliberately programming language agnostic, but in a few assignments there are explicit targets for a specific language, either Python/PyCSP or Go. A common problem is the connection of processes with channels, i.e. wiring. Wiring processes in the assignments can have three different learning goals, simple wiring to a know topology, dynamic wiring to match a given input, or to use process mobility. Simpler learning goals includes network termination and the use of subnets for compositionality. Several assignments also have process mobility as a learning goal.

3. Games

The idea of programming computer games is inherently attractive to many students and several game projects have been considered over the years, but most often dropped due to the lack of an easy-to-use graphics library that works with a CSP-style design. PyGame is an example of a library that meets all the requirements of game-writing, but also requires the programmer to use the concurrency mechanism within the library, which is not CSP-compliant in any way. Thus the game based assignments we have offered have been text based, using curses for simple ASCII graphics.

3.1. Multi User Dungeon (MUD)

A MUD game consists of multiple users and autonomous program agents. The purpose of this assignment is to test the students ability to design and implement a concurrent design, not to test their skills as a game-writer. A MUD game consists of a number of rooms a player may move between, through designated doors. In a room there may be objects that can be picked up; similarly a player can leave objects. A real MUD game will also have an action component that allows fighting and dying; this is not required in this assignment. The assignment stipulates that the solution should include channel and process mobility. The learning goals are:

- Deadlock avoidance
- Absence of race-conditions
- Python/PyCSP
- Simple wiring
- Channel mobility
- Network termination
- Process mobility

Thus a game implementation must include:

- A few rooms.
- Doors the player can pass through to move from one room to another.
- A few objects that may be picked up and left again.

- At least one autonomous agent that moves between the rooms as a programmed agent; it needs not do anything but move around.

A player should be able to issue the following commands:

- `look` - should return a description of the current room, what is in the room, who is in the room and a list of items that the player is holding.
- `take` - should pick up an object in the room, that is `take gold` should pickup the gold in the room and the player now holds the gold (this is assuming there is any gold in the room).
- `leave` - should taken any object the player is currently holding and leave it in the room (this is the inverse of the `take` operation).
- `N, E, S, W` - should move the player through a door to the north, east, south, or west, if such a door exist.
- `exit` - should end the game and terminate all processes.

A room in the game should:

- Be able to hold objects.
- Be able to host more than one player or agent at the same time.
- When a player or agent enters or leaves the room all other players and agents in the room should be informed to the fact.
- A room may never be blocked for an extended time, that is only trivial operations are allowed before the room again accepts commands (i.e. the room may execute any of the supported player commands but nothing more advanced).

The game should support mutable users, programmed agents, and object mobility. To validate the assignment we produced a small reference solution, sketched in Figure 1, that takes input from a user and hosts an autonomous agent. The assignment is easily solved if the students comprehend channel and process mobility. The reference implementation is written in PyCSP and is implemented in roughly 150 lines of Python.

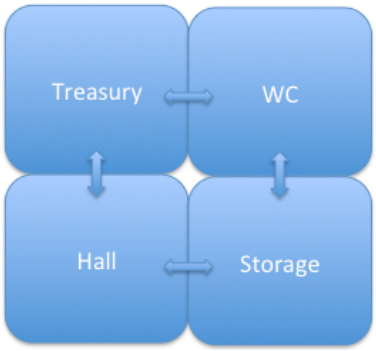


Figure 1. Schematic of the MUD reference implementation.

Observation. *While hardly any students had tried MUD-style games previously the assignment worked very well, and have been given twice now. The worst students tend to design a centralized solution. The best students however utilize channel and process mobility efficiently and end up with very small implementations. The most common problem that is observed is termination where mobility issues may cause processes to not receive termination signals.*

3.2. Light Cycles

In this assignment, the students are asked to write a small version of the '80s arcade game *Light Cycles*. The game should be written in the Go programming language, though PyCSP

would probably work nicely as well. Since we are primarily interested in the concurrency issues the graphics will be replaced by text. Students are encouraged to use the `termbox` library for printing at a specific point on the screen. The learning goals are:

- Absence of race-conditions
- Go
- Simple wiring
- Network termination

The rules of Light Cycles are very simple; a field, which is surrounded by a barrier, hosts two players, one of which is an AI player. As the cycles drive they leave behind a barrier. If a Light Cycle hits a barrier the player dies and the other one wins.

The project in split into three problems:

Problem 1. *Make a Go implementation of the basic Light Cycle game, players run at a constant speed.*

Problem 2. *Improve your game to increase speed at time progresses, either continuously or as discrete steps.*

Problem 3. *Improve your game to add difficult by adding a random piece of barrier in the game area every 5 seconds.*

Figure 2 is screen dump of a solution to Problem 1. The assignment lends itself strongly to a CSP-style solution, and students who struggle with the communicating process approach end up with very large solutions, and often erroneous. In addition to testing the students understanding of communicating processes it also tests interfacing with external devices, screen and keyboard, and the use of realtime injections to a process network.

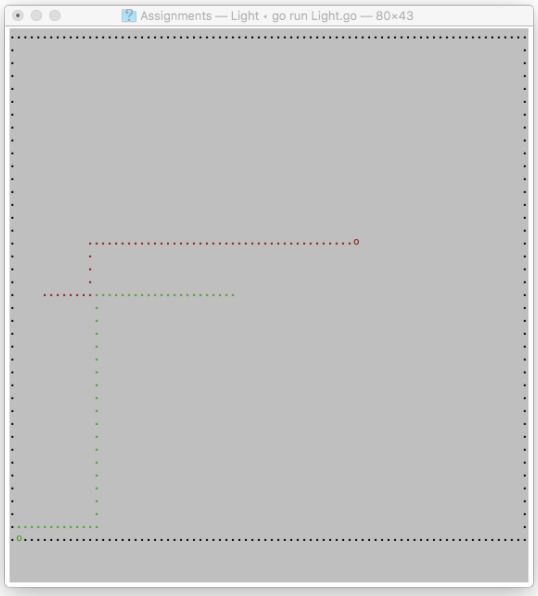


Figure 2. Screendump from the reference implementation.

Observation. *The Light Cycles game has only been tried once and the primary finding is that the project is not well suited for a first experience with the Go programming language. Overall the project works fine, here too the worst students tend to centralize their solution. The need for real time progress of the game combined with the need to receive input from the*

user tests the concurrency mechanisms very well, and most students' intuitively acknowledge that the same task would be very hard in a non-concurrency environment. We will seek to refine the assignment and use it again in the coming class.

4. Simulations

Simulations are often inherently concurrent, for example as in [7], and from a teaching perspective simulations are extremely well suited for assignments that test the students ability to work with concurrency. While it is easy to model simple communication and compositionality in simulation based assignments, process and channel mobility are often harder to think in.

4.1. Train Simulation

In this assignment, students should design a system that simulates a railway system. Throughout the assignment distances and time are unitless, that is a distance may be stated as 40 and a speed as 10. This could be 40 km and 10 km/h, but may also be any other metrics, the distance unit is shared between the two units however, this approach is chosen to stop students from moving into details that are not relevant for the assignment. While there are constants in the assignment that allows them to solve the problems in closed form, they are required to make their solution generic enough that these constants, and the rail-design can be changed. The learning goals are:

- Deadlock avoidance
- Absence of race-conditions
- Python/PyCSP
- Dynamic wiring
- Network termination
- Process mobility

The assignment is split into three problems:

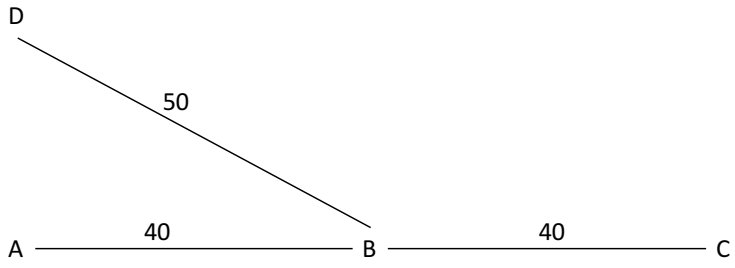


Figure 3. Simple train simulation setup.

Problem 1. *For the system that is shown in Figure 3, determine if there are any collisions in the following two setups:*

- *Train T leaves station A with destination station C, at time* 0*, running at speed* 10*.*
- *Train U leaves station D with destination station C, at time* 0*, running at speed* 10*.*

and

- *Train T leaves station A with destination station C, at time* 0*, running at speed* 10*.*
- *Train U leaves station C with destination station A, at time* 5*, running at speed* 10*.*

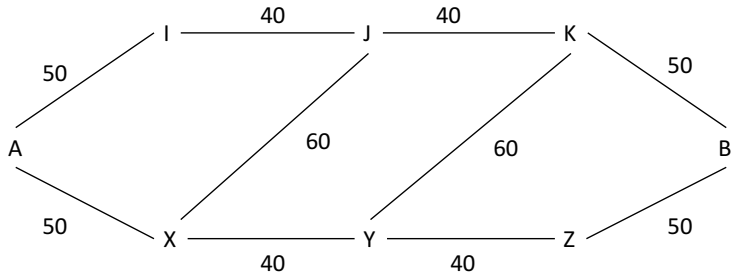


Figure 4. Advanced train simulation setup.

Problem 2. *For the system that is shown in Figure 4, determine if there are any collisions in the following two setups.*

- *Train T leaves station A at time* 0*, taking the path through I, J, K terminating in B, running at speed* 10*.*
- *Train U leaves station A at time* 0*, taking the path through X, Y, Z terminating in B, running at speed* 10*.*

and

- *Train T leaves station A at time* 0*, taking the path through I, J, K terminating in B, running at speed* 10*.*
- *Train U leaves station A at time* 0*, taking the path through X, J, K terminating in B, running at speed* 11*.*

Problem 3. *Now assume that the stations can hold any number of stationary trains. Extend your simulation system to be a control system, in such a way that a train is never released from a station to a track unless it will not collide with another train on that section of track. If two trains at the same station wish to move down the same section of track they should do so in the order they first requested to go, if the time is identical a random train is chosen over another.*

For this problem, the student is asked to generate their own flow of trains.

Observation. *This assignment produced a surprising variety of solutions. There were solutions where both tracks and stations were processes, some where only stations were processes and one where only tracks were processes. Many students ended up discretizing time in solutions that turned out to be less than optimal. The largest challenges tended to be the representation of the rail-topology, and too many students spent much time on this. We consider using the assignment again and provide the students with a sparse matrix structure that describes the rail models to eliminate the time spent on that.*

4.2. Bean-machine

The Bean-machine [8] was designed by Sir Galton to visualize the central limit theorem [9]. In this exercise students are asked to build a simulator of the bean-machine using CSP.

The student is asked to implement each "pin" in the triangle as a process, which may receive balls from any of it two upper neighbors and passes it to any of the two neighbors below. The bottom level sends to a counting bin. In the original form the pins should be fair, that is equal chance of going left and right.

In addition, the simulation will need additional processes to simulate, bean/ball injection and bin-counting. The learning goals are:

- Absence of race-conditions
- Dynamic wiring

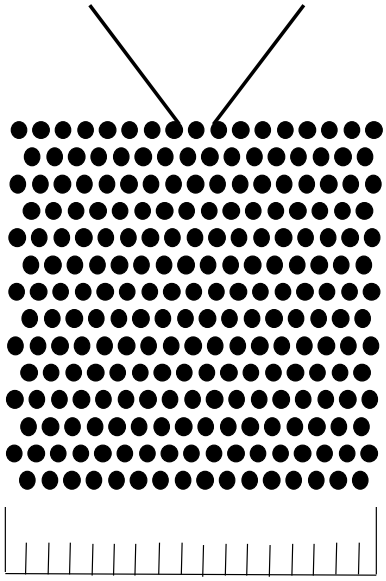


Figure 5. A simulated bean-machine.

- Network termination
- Compositionality

Problem 1. *Write a bean-machine with* 2 *layers, that is* 3 *bins, in the pin-pyramid.*

Problem 2. *Write a bean-machine that accepts an arbitrary number of levels in the pyramid. Set it up to run* 10 *layers of pins, that is* 11 *bins. Setup the simulator to terminate after a number of balls are either*

- *Put into the simulation.*
- *Received in the counting bins.*

The students are then asked to run the 50 layer simulation to count 1000 balls/beans with both termination options (i.e., injected balls and counted balls). Note, that in both cases the number of counted balls should be exactly 1000.

Problem 3. *Finally the students are asked to extend the bean-machine from Problem 2 with a bias, that is if the beans were iron balls we could apply a weak magnetic field to skew the probability to one side or the other. Add a process that includes an oscillator that sends bias-values in the range* $[0 : 0.5]$ *with increment/decrement of* 0.01 *at the injection of each new bean/ball, to the bean-machine bias. Rerun the* 1000 *ball experiments from Problem 2.*

Observation. *The bean-machine has been used several times and in general works well. The project tests most aspects of concurrency programming, though not all students manage to include mobility. The assignment is also a fine test for the students understanding of composability, the best students will wire up the simulation with recursive components, where the students build a preocess that is merely a line of processes and then form bloks of lines until the full systems is build, while the less strong will do so with trivial nested for-loops.*

4.3. Pedestrian Simulation

The project assignment is to write a cellular automaton that can simulate pedestrian behavior [10]. Since we are working with cellular automata our world will be built from cells that each work autonomously and communicate only with its Cartesian neighbors, that is up-down-left-right but no diagonals.

To keep the task simple we will make very simple pedestrian agents. Each agent has a target direction which is also Cartesian, and will try to move forward in that direction. If the cell an agent wishes to move into is occupied the agent will attempt to move orthogonally to the target direction to an empty neighboring cell, if both these neighboring cells are empty the agent is moved to one of the two at random. If the target cell and both neighbor cells are occupied the agent simply remains in the present cell. If two agents both wish to move into an empty cell one of them will make the move and the other will act as if the cell is full, which makes the move should be non-deterministic.

The students may make their world as complex as they wish but as a minimum they must implement a model of a 5-by-10 sidewalk. The learning goals are:

- Deadlock avoidance
- Absence of race-conditions
- Dynamic wiring
- Channel mobility
- Network termination
- Process mobility

Problem 1. *Make a cellular automaton with the above rules and minimum size, and implement automatic injection of agents from at least two endpoints such that agents can in principle collide. With the minimum sidewalk simulation, inject at both ends of the 10 cell long sidewalk, a new pedestrian every 4 time-steps. After each update of the overall cellular automaton you should dump a snapshot of the cellular automaton to show where the agents are placed, text is enough but graphics would be great. The simulation should be run to 200 time steps.*

Problem 2. *Add a simple control interface, text interface is sufficient but graphic is fine too. The interface should allow the person that executes the simulation to*

1. *Pause and resume the simulation.*
2. *Change the rate at which the agents are injected, allow values from* 1 *through* 10 *time-steps between injections.*
3. *Terminate the simulation.*

Problem 3. *Add support for having a tile occupied by a drunk. The addition of the drunk should be through the interface from Problem 2. A drunk stays at this cell throughout the simulation. Any pedestrian agent that comes within 3 cells of a drunk will change its preferred movement to be orthogonal to its original direction until such a time as it is more than* 3 *cells from the drunk and the resume the original preferred direction.*

Problem 4. *Make the drunken agent move around at random, if the drunk leaves the cellular automaton it remains gone for the remainder of the simulation.*

Student are asked to make sure that their report addressed how to model the cellular automaton with a CSP-programming library, and to make sure they reflect on potential concurrency and how to shut down the simulation.

Observation. *The pedestrian simulation is a project that divides students in two groups; one where pedestrians are passive messages and one where they are mobile processes. In the case of simple messages all the control in the simulation is kept within the processes that describe the sidewalk, while the solution that use mobile-processes can use a mostly generic grid of processes for the sidewalk and have the agent-processes handle the rules. This of course makes it easier to add a new agent type. The static pedestrian message is easily sufficient to model all the requirements, and we thus consider changing the project slightly to add requirements that will make the use of process mobility the easy solution.*

5. Control Systems

Assignments that ask students to write a control system for a physical system are extremely well suited for teaching concurrency. Control systems are real-world examples where concurrency is highly used and many systems have natural non-determinism that fits well with a concurrency approach to problem solving. The challenge in building assignments on control project is that the system that should be controlled usually will have to be simulated and the students must either have a simulated system provided, the design of which may limit the possible solution space, or be asked to write the simulation as well. The latter choice both increase the workload and increase the chance that a misunderstanding on the students part can make an assignment either trivial or overly complex.

5.1. An Autonomous Survey Robot

In this project the task is to program a control system for a simple robot that can navigate its way out of a maze. The robot has three sensors and two actuators. The actuators control two wheels, the sensors measure the distance to an object. The learning goals are:

- Deadlock avoidance
- Deadlock avoidance
- Absence of race-conditions
- Simple wiring
- Network termination
- Compositionality

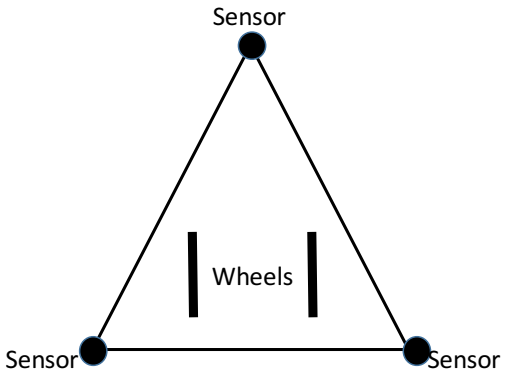


Figure 6. A robot, 3 sensors and 2 actuators.

The wheels in the robot takes an integer in the range $[-10 : 10]$, where the value represents the speed and negative numbers means going backwards. The sensors return a value in the range $[0 : 100]$, the value represents the distance to an object. Both actuators and sensors should communicate through channels.

Problem 1. *Design and implement a robot simulation that works as the robot described above, that is two wheels and three distance sensors, all of which communicate through CSP-type channels. Demonstrate that your robot can be navigated and detect obstacles.*

Problem 2. *Design and implement a small maze in your simulation.*

Problem 3. *Program your simulated robot to find its way out of a maze, using information from its sensors as the only information it has. Show that your robot can find its way out of your maze.*

Problem 4. *Extend your robot to simulate hardware imprecision. To do this allow the robot to spin one wheel slightly slower than the requested,* 10%*, that is if the wheel is asked to spin at speed* 1 *it will go only* 0.9*, requested speed of* 5 *becomes* 4.5 *and speed* 10 *becomes* 9*. Extend your control software to detect that the robot drives off the expected path and add functionality to realign the robot with maze walls.*

The solution should work as a simulation of the robot-maze setup. The students are are responsible for setting up a simple test scenario, making sure to test both positive and negative acceleration, as well as uneven speeds in between wheels. In the report they must provide CSP-diagrams that shows the final simulation setup, explains how they avoid deadlocks, live-locks and how the overall simulation is terminated.

Observation. *The robot-maze project turned out to have too many unspecified components and the simulation environment was a large challenge for the majority of the students. We consider repeating the project in a version that provides the simulated environment, while still allowing students a high degree of freedom in their solution design.*

5.2. Drone Control

A quadcopter, as shown in Figure 7, has four individual rotor. The overall exercise in this assignment is to make a rudimentary control system for these rotors and thus the quadcopter. The learning goals are:

- Deadlock avoidance
- Deadlock avoidance
- Absence of race-conditions
- Simple wiring
- Network termination
- Compositionality

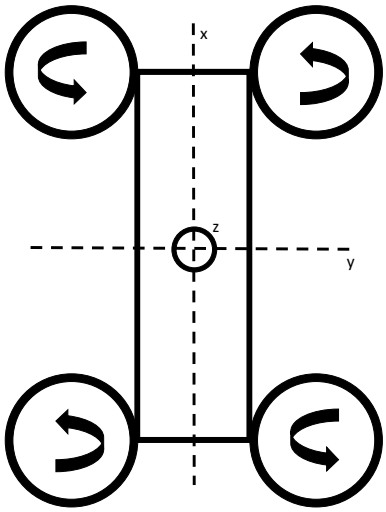


Figure 7. The Drone. The curved arrows inside the rotors indicates the rotation direction.

A quadcopter has the ability to pivot along three axes, x, y, and z. Since we do not have a physical quadcopter to work on, we will work in a simulated environment. The physical laws we simulate will be as follows. All values are unitless, we will work using

- Time units: $[0 : \infty)$.
- Distance units: $[0 : \infty)$.
- Engine speed units: $[0 : 100]$.

- Height units: $[0 : 10000]$.
- Angle units: $[0 : 359]$.

X-axis: If the engines on the left runs slower than the engines on the right, the quadcopter will each in time unit increase its pivot by one degree per difference in engine speed. That is if the right engines run at speed 50 and the left engines run at speed 48, then after 5 time units the pivot will be 10 angle units.

The consequences of pivoting along the x-axis, called roll, is not simulated in our environment. Thus pivoting in the x-axis is only used as information for correcting that pivot.

Y-axis: Pivoting along the y-axis, called pitch, follows the same rules as along the x-axis, that is the difference in engine speed on the front and back engines dictate the pivot over time, with the same ratios as with the x-axis.

If the quadcopter is pivoted along the y-axis it will move along the x- and z-axis, linear in time. Movement in the x-axis depend on the direction of the pivot, if the front engines point down then the movement along the x-axis is forward, if the front points up the quadcopter moves backwards. In both cases the quadcopter will also move downwards.

Example: If the quadcopter has been angled 3 units downwards the quadcopter will move forwards and downwards each 3 units per time unit.

Z-axis: Pivoting along the z-axis in this assignment is only defined when there is no pivot on the x and y-axis, that is we will only model rotation around the quadcopter center without any other movement. The movement is defined as with the other axis, but for pivoting in the z-axis we work in the diagonal axis, that is the axis from upper-left to lower-right (LR), and upper-right to lower-left (RL). If the engines on LR axis runs at 1 speed unit slower than than the engines on the RL axis, the quadcopter will rotate counter clockwise 1 angle unit per time unit, if the engines on the LR runs 3 speed units faster than the RL engines, rotation will be 3 angle units per time unit clock wise.

If all engines run at the same speed the quadcopter may move up or down. Height is symmetrical around 50 speed units, that is if all engines run at 55 speed units the quadcopter will raise by five height units per time unit, if they run at 48 speed units the quadcopter will drop by 2 units per time unit. If all engines run at 50 speed units, the quadcopter will stand still in the air.

Please note that the above physics simulation rules are hugely simplified and the actual reality should be modeled with differential equations rather than linear. However, from a control perspective this does not matter, so the above oversimplified rules are set up so that the student can focus on the concurrency aspects, not physics.

Problem 1. *Design and implement a simulation environment for our quadcopter simulations.*

The simulation environment should receive information from the engines, and from that input and the above rules produce the following simulated instruments:

- Altimeter - tells how high above ground the quadcopter is $[0 : 10000]$ in height units.
- Compass - tells how many angle units the front of the quadcopter is rotated relative to its initial position $[0 : 359]$.
- Inclinometer-X - tells how many angle units the quadcopter pivots along the x-axis.
- Inclinometer-Y - tells how many angle units the quadcopter pivots along the y-axis.

You can assume that engine data translates into physics perfectly. Assume also that engines are always symmetrical along some axis, that is we have only perfect relations that are described by the physics rules.

Problem 2. *Once your simulation environment is done, we can start making controls for the quadcopter. First start to design and implement a control system that allows the quadcopter to move in the z-axis, and stabilize.*

- *Support a request to move from the current height to a requested height. Limit speed to a maximum of* 5 *height units per time unit.*
- *Support rotating the quadcopter from its current angle to a required angle. Limit speed to a maximum of* 1 *angle units per time unit.*

As our quadcopter can now raise and rotate, we can start flying.

Problem 3. *Add support for moving forward* N *distance units. Since the quadcopter drops as it moves forward your control must level the quadcopter once it has dropped* 10 *height units and raise to the height at which the forward statement was originally initiated. Once the quadcopter has moved forward* N *distance units it should raise to the same height as well, and then hover.*

Problem 4. *At this point your quadcopter should be able to run the following simple command sequence:*

```
[raise to 1000; rotate 90; forward 100; rotate 90; forward 100]
```

Observation. *The quadcopter assignment turned out to be more challenging in the simulation part than expected, and several students ended up with a very sequential solution to the simulation part which made most of the assignment rather trivial. If the assignment is repeated we will provide the simulated environment as code to the students, and make the core assignment more challenging by adding imprecisions to the control operations.*

5.3. RAID System

This project is structured somewhat differently than the other projects we describe here. The point of the project is to stress composability to the extreme; students are asked to build very small elements, similar to the LEGOs from JCSP [11]. From these tiny processes students are asked to build networks and then reflect on what the network does. This approach does limit the degrees of freedom the students have for a solution, but makes network connections and compositionality very clear to them.

For each question A through E the student is required to document their work with a description and drawing(s). The point of the exercise is to make the students appreciate the power of compositional processes, but due to the strict specifications in the assignment the students are left with little freedom to design their solutions and instead spend time on analyzing the results of what they did. The learning goals are:

- Deadlock avoidance
- Python/PyCSP
- Simple wiring
- Dynamic wiring
- Compositionality

Problem 1. *Build the following micro components.*

1. Numbers (0 input(s), 1 output(s)).
   - Started with an iteration number, I.
   - Outputs the numbers $0 \ldots 255$ in sequence.
   - Repeated I times.
2. Checksum element (1 input(s), 1 output(s)).
   - Started with two constants; block-size, B, and checksum-size, C.
   - Inputs a sequence of bytes of length block-size.
   - Checksum is $\sum$ data-sequence $\mod$ checksum-size.

- Outputs a checksum every block-size bytes.

3. Message duplicator (1 input(s), 2 output(s)).
   - Inputs a message.
   - Outputs the same message on two different channels.
4. Integrator (1 input(s), 1 output(s)).
   - Started with block-size.
   - Receives block-size bytes.
   - Outputs block-size bytes as one message.
5. DataContainer (2 input(s), 2 output(s)).
   - Started with a capacity, N.
   - Inputs N data-blocks and N checksums - stores each pair.
   - Outputs N data-blocks and N checksums - pairwise.
6. Daemon (1 input(s), 1 output(s)).
   - Inputs a data-block and with probability P changes one, non-zero, element in the data-sequence to 0 (probability is only for the entire sequence, not per element).
   - Outputs the, potentially changed, sequence.
7. Checker (3 input(s), 1 output(s)).
   - Inputs a data-block, $checksum_1$ and $checksum_2$.
   - Outputs the data-block, if $checksum_1 = checksum_2$, else outputs an error.
8. Separator (1 input(s), 1 output(s)).
   - Inputs a data-block.
   - Outputs the elements in the data-block as a sequence once, one by one.
9. Elector (2 input(s), 1 output(s)).
   - Inputs two data-sequences.
   - If at least one of the data-sequences is not an error, output the data sequence, otherwise output error.
10. Distributer (1 input(s), F output(s)).
    - Started with a fan-out, F.
    - Inputs F consecutive messages.
    - Outputs the messages, one on each of F output channels.
11. Joiner (F input(s), 1 output(s)).
    - Started with a fan-in, F.
    - Inputs F data-sequences from F input channels.
    - Outputs the F data-sequences in channel order.
12. XOR (2 input(s), 1 output(s)).
    - Inputs first two data-sequences, a and b from two channels.
    - Outputs $a \oplus b$ (xor).
13. Rotator (F input(s), F output(s)).
    - Started with a fan-in and fan-out, F.
    - Inputs on F input-channels.
    - Writes to F output-channels, but starting with channel $r \mod F$, where r it a counter that says how many times the component has been active; that is first time the start channel is 1, then 2 and so forth.

Problem 2. *Build a network, $DataPipeline$, using the components from the first problem f the assignment. The network should take a sequence of bytes, store them with their checksums in a container and then read them back while verifying the checksum, to test the data should be challenged by the daemon. Use the $Integrator$ at the appropriate place to transform from a stream of bytes to a data-sequence, and the $Separator$ for the inverse.*

Use the following constants:

```
I: 20
B: 512
C: 16 bit
N: 10
P: 0,0.2
F: 2
```

Verify that all data are correctly passed through the network with $P = 0$ and that you detect errors correctly with $P = 0.2$.

Problem 3. *Now make another network, $SafePipeline$ which has two $DataPipelines$ joined with the $Elector$, so that if any of the two data-sequences are valid the correct data-sequence is output. Describe briefly how many processes and channels are in the $SafePipeline$ network and what common technology principle you have just implemented.*

Problem 4. *Build $RedundantPipeline$ by extending $DataPipeline$ to include two $DataContainers$ that are then written to using the $Distributer$. The data from the distributer should be passed through XOR and the output written to a third $DataContainer$.*

Data from the fist two containers should be run through XOR with the data from the third $DataContainer$. Then the data form each $DataContainer$ should passed through an $Elector$ with the output from the XOR from the other $DataContainer$, the outputs from the $Elector$s should then be joined through the $Joiner$.

What have you built now? How many processes?

Problem 5. *Extend the $RedundantPipeline$ by inserting a $Rotator$ before the inputs to the $DataContainers$ and right after the outputs as well.*

What have you built now? How many processes?

Observation. *The RAID system is very different from the other projects, and the limited degrees of freedom and the lack of mobility requirements makes the project less successful for teaching concurrency. It does stress compositionality and the students did like the assignment, however it is too easy and will not be used again in our class.*

5.4. Cluster Monitor

The task in this assignment is using CSP-mechanisms to design and implement a system that supports data collection and intervention from a dynamic set of machines. The learning goals are:

- Deadlock avoidance
- Absence of race-conditions
- Python/PyCSP
- Dynamic wiring
- Channel mobility
- Network termination

- Compositionality
- Process mobility

The scenario is a large set of individual computers, for example tens of thousands of nodes. Each node will send one of three kinds of status messages when an event occurs:

- `info`.
- `warning`.
- `critical`.

We will be monitoring the following components:

- CPU temperature.
- Room temperature.
- Disk temperature.
- Memory error.
- Link down.
- Progress (time).

For each component we define a set of intervals or events that give the three different message levels.

- Link down: `critical`.
- Memory error: `warning`.
- CPU temperature:.
  - Change of 1° but below 60°: `info`.
  - Change of 0.5° and value between 60° and 75°: `warning`.
  - Change of 0.1° and value above 75°: `critical`.
- Room temperature:.
  - Change of 1° but in range $20^\circ - 25^\circ$: `info`.
  - Change of 0.5° and values in range $15^\circ - 20^\circ$ or $25^\circ - 30^\circ$: `warning`.
  - Change of 0.1° and value below 15° or above 30°: `critical`.
- Disk temperature:.
  - Change of 1° but below 40°: `info`.
  - Change of 0.5° and value between 40° and 45°: `warning`.
  - Change of 0.1° and value above 45°: `critical`.
- Time: If a node has not submitted a message for a period of one minute it should send an `info` message that says alive. If a node does not send anything for two minutes produce an `info`, for four minutes a `warning` and five minutes a `critical` (repeat every five minute that the note has not sent a message).

Messages are sent as one string:

```
[TYPE] [NODE] [TIME] [COMPONENT] "STRING VALUE"
```

Examples could be:

```
Info 81 Wed Nov 9 14:19:11 2016 Disk 7.4C
Critical 5 Wed Nov 9 14:19:11 2016 Memory write error
```

Note that messages must be sent as one human readable string in this format.

As a system may be very large, any number of receivers should be able to subscribe to messages. A receiver should be able to make a selective subscription based on:

```
NODES, MESSAGE TYPE, COMPONENTS
```

For instance

```
[0:15] [Critical] [DISK, CPU]
```

could mean subscribe to critical messages concerning disk and CPU for nodes zero through fifteen. The above format is just an example; you are free to define your own protocol.

The users should be able to support each other; thus two supportive functions must be provided:

- Pass subscriptions.
- Listen in.

"Pass subscriptions" means that one listening client passes all its subscriptions to another user, for example if an administrator has a lunch break another can take over the subscriptions.

"Listen in" should allow a user to add one or more other users to receive the messages from its subscription, so that they may help locate an error.

Problem 1. *You must write your own node simulation that produce status messages. Make sure that the node simulator is also concurrent.*

Problem 2. *You should also provide your own simulated listeners; they can either be human and/or logging mechanisms.*

Your solution should be implemented using some CSP-oriented library. PyCSP or Go is highly recommended.

The students are made sure to document their design with figures and explain the concurrency issues and solution to them.

Observation. *We expected the cluster monitoring project to be easily understood by computer science students, unfortunately this was not the case and students spent a lot of time understanding the actual problem. Several students tended to centralize the solution, especially the passing of responsibility, rather than using channel mobility. If used again, we will redesign the assignment to be simpler from a context perspective and harder from a concurrency perspective, to reward the use of mobility.*

5.5. Firewall

In this project the student must design a firewall. While firewalls do in fact work on network protocols the assignment is based on CSP-type channels, where the first message sent on the channel will be a three tuple `(IP, PORT, reply_channel)`, that specifies the address and port number of the required host, that is `(192.160.0.12, 22, <channel>)` means that the channel will connect to a machines with address `192.168.0.12` and a service that listens to port `22` on that host, the service should reply on `<channel>`. To keep the service responsive the server should create a new channel to which the connecting machine should send future messages - similar to a conventional network socket-based connection.

The overall assignment seeks to design a scalable firewall where new features can easily be added. The learning goals are:

- Deadlock avoidance
- Absence of race-conditions

- Dynamic wiring
- Channel mobility
- Compositionality
- Process mobility

All programming should be done using PyCSP. There is an example of a simulated network connection between two machines, without an intermediate firewall, with this text, you can use this example as a baseline and introduce the firewall between them.

Problem 1. *The basic firewall is based on information from a table which is an positive-list of hosts and ports. Only addresses that are on that list can be connected to by incoming connections. Table 1 shows an example translation list, if an incoming connection does not match an entry in the translation list then the connection must be dropped. Dropping a connection should be simulated by poisoning the reply channel.*

You first programming part of this exam should implement a firewall that supports the connection of legal requests and dropping of illegal requests.

HOST	PORT
10.0.0.12	80
10.0.0.22	21
10.0.0.28	22

Table 1. An example translation list.

For simplicity we will assume that our network connections only pass clear text messages. To increase the safety of the system all inflowing data must be inspected.

In this problem a list of words simulates the illegal contents, thus if a word from the illegal list occurs in a stream the connection must be dropped, without forwarding the illegal word or any of the following data to the recipient. Table 2 shows an example list of illegal words. It is necessary to be able to change the list of illegal words at runtime - this means that your solution should allow an administrator to maintain the illegal word list, without stopping your firewall application.

{*objects, java, php*}

Table 2. An example illegal word list.

Problem 2. *The second programming part of this exam should extend the firewall to support packet inspection for illegal words, using a wordlist that can be changed at runtime, if an illegal word is detected the connection must be poisoned. An example of such illegal words can be found in Table 2.*

During execution a connection may be selected for monitoring. This means that an operator can ask to receive copies of all traffic on a channel for manual inspection or logging. In addition, the operator should be able to shut down an open connection. The connection may be identified in any way you feel is easiest, for example a connection may simply be given an ID and the operator ask to get a copy of the information on a connection or to shut a connection down, based on the ID.

Problem 3. *The third programming part of this exam should extend the firewall to support manual packet inspection for illegal words. The operator should be able to ask to see all data on a channel and to shut down a channel.*

Observation. *The firewall was a computer science specific context that worked very well, students had an immediate understanding of the problem. The project seems to test all aspects of concurrency and overall was very successful. We may add another question to the project but as it is, this firewall is a fine project.*

5.6. Train Break System

In this project a neck-tie-wearing boss has for some reason decided to buy an Italian made train. As it turns out the trains braking system does not work[3]. The project assignment is to design a new braking software system for the train. The learning goals are:

- Deadlock avoidance
- Absence of race-conditions
- Simple wiring
- Compositionality

On the train each wheel is attached to an electric motor, which works both as an engine and a brake. The motor is controlled by a simple computer controls. That computer can control and monitor the rotation speed of the wheel. The setup is shown in Figure 8.

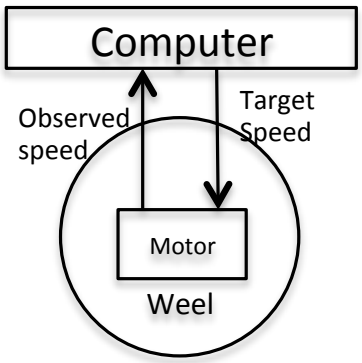


Figure 8. Configuration af a single wheel.

The computer receives an absolute target speed in the interval $[0:100]$ from a central control unit. When the computer sets a new speed at the motor it should monitor the observed speed of the wheel, if the observed change is larger than 1 between two observations the wheel is malfunctioning, that is it may be blocking during a braking operation or spinning during an acceleration. In this case the computer should split the acceleration in two steps to help avoid the problem. Note that this solution may be applied repeatedly so that a large acceleration, positive or negative, may be split into many smaller accelerations to avoid blocking or spinning the wheels. A wheel is part of a wheel-set, one on each side of the train, and in addition to controlling the correctness of its own wheel a computer is also responsible for coordinating with the other wheel in the same set to guarantee that both wheels are running at the same speed which is necessary on rails. To ensure this the computers opposite each other exchange observed speeds as they arrive. The setup is shown in Figure 9.

The solution software, based on a CSP-library of the students own choice, should implement a train with 10 sets of wheels, that is 20 wheels, that may receive a target speed and then reach that speed while compensating for blocking and spinning and ensuring that both wheels are synchronized.

The resulting software should work as a simulation of the train set. The student is responsible for setting up a simple test scenario themselves, make sure that they test both pos-

[3]Unfortunately a really real story [12]

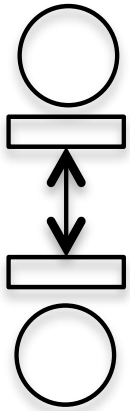


Figure 9. Configuration of a wheel-pair.

itive and negative acceleration as well as wheel blockage, spin and uneven speeds in a set. In the report they must provide CSP-diagram that shows the final simulation setup, explains how their design avoid deadlocks, live-locks and how the overall simulation is terminated.

Observation. *The train system was easily understood by the students, and most students did well, though some did end up with centralized solutions. The many boundary conditions, test the students ability to use composability and most students manage to write very scalable solutions.*

5.7. Ball Sorter

Imagine a machine for sorting balls according to color. The machine is shown below in Figure 10.

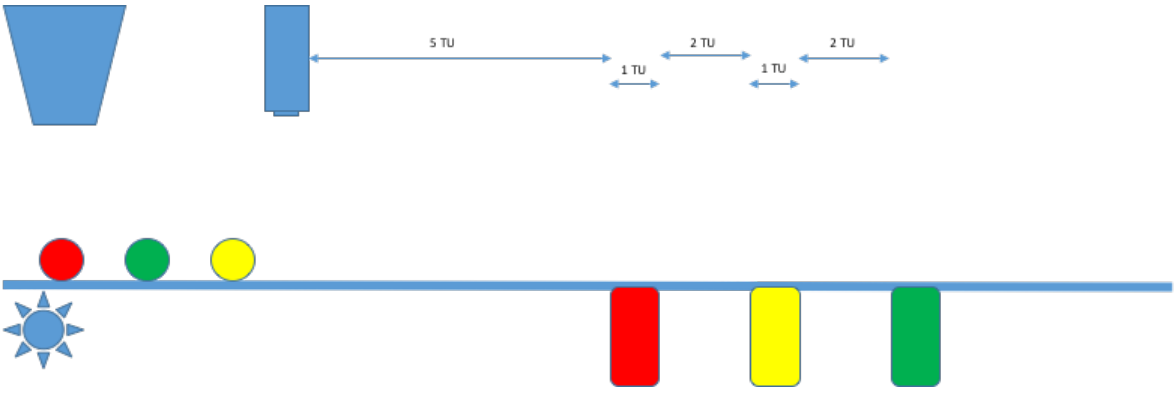


Figure 10. Schematic overview of the conveyor belt sorting machine.

Balls in three different colors fall from the input funnel every one time unit, and the conveyor belt moves at a constant speed which makes the whole system synchronous. From the time a ball passes under the camera and until it reaches the first bin, five time units pass. Passing a bin takes one time unit and traversing the space between two bins takes two time units.

The project assignment is to program a control system for the sorting machine. The learning goals are:

- Deadlock avoidance
- Absence of race-conditions
- Python/PyCSP
- Simple wiring

- Network termination
- Compositionality
- Process mobility (not required but makes a simpler solution)

The solution must tell every bin when to open so that it collects the balls of the color of the bin. Each bin can hold ten balls, when a bin is filled the conveyor belt and the funnel must be told to stop to prevent overflow, the system should be paused for five time units.

Problem 1. *Design and implements, using CSP-mechanisms, the control system that behaves as described above.*

Problem 2. *Assume now that the number of bins change from 3 to some other constant, the distance between camera and the first bin, and the distance between bins does not change. Change your solution to be able to handle an arbitrary number of bins.*

Problem 3. *Building on your solution from question 2, assume that the conveyor belt is wide enough to hold a number of balls in parallel, and each row has a bin for each color. The system is still controlled by a single camera. Extend your solution so that an arbitrary number of parallel balls can be handled.*

Problem 4. *Provide a graphic representation of your solution, remember to name both processes and channels.*

The students are encouraged to use the same names for channels and processes in the code.

Observation. *The ball sorter as described above have some challenges with the injection of balls and the communication with the camera, and some students fused the two to make the solution mostly trivial. We will reuse the assignment but will try to provide the injection and camera as code and ask the students to simulate and control only the conveyer belt and bins.*

6. Conclusion

Based on ten years of teaching concurrency using loosely defined programming projects, our overall conclusion is that the approach is fundamentally sound. There are pitfalls, most importantly the risk of defining assignments too precisely, and thus dictating a single solution, or describing assignments where the understanding of the application domain is too hard for novice students.

Overall the conclusions are that the approach works; assignments based on games are extremely popular and makes students work very hard, unfortunately the lack of a good graphics library that supports CSP-style programming is a hard obstacle. Simulation assignments are very well suited for teaching concurrency and is in our experience the most successful type of programming projects. Projects that aim at control systems are also very well suited for concurrency and represent a set of problems that is easily related to real-world problems, unfortunately they often require students to program a simulated test environment as well which increase the workload and introduces so many degrees of freedom that some students experience large challenges in designing a solution.

The primary conclusion however remains that games are in principle extremely well suited but the lack of a game engine that supports CSP-style programming is a limitation. Future work will, in addition to defining more projects and setting up a repository of projects for sharing, is to make another CSP-style programming library that includes the features that are needed for writing games.

6.1. Future Work

The authors found the exercise of collecting the assignments quite useful ourselves, the division that project may be divided into Games, Simulations, and Control Systems was an insight we ourselves did not get until we started a discussion on how to organise the assignments. At the same time, it became obvious that the assignments do not directly compare, neither in workload, prerequisites or learning goals. Thus we will move on to make the assignments more homogeneous, make assumptions of prior knowledge as well as what should be learned from the project explicit and then publish the projects for others to use as an online tool or software repository. In our collection of assignments, we often require the students to solve the problem with a given programming language, this is not necessarily the best option for a community tool and instead we will make recommendations to which programming languages are well suited for the individual assignments.

References

[1] Geraint Jones and Michael Goldsmith. *Programming in occam*. Prentice-Hall International New York, NY, 1987.
[2] John Markus Bjørndalen, Brian Vinter, and Otto J Anshus. PyCSP-Communicating Sequential Processes for Python. In *CPA*, pages 229–248, 2007.
[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.
[4] Neil CC Brown and Peter H Welch. An introduction to the Kent C++ CSP Library. *Communicating process architectures*, 2003:139–156, 2003.
[5] Peter H Welch and Jeremy MR Martin. A CSP model for Java multithreading. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 114. IEEE Computer Society, 2000.
[6] Robert Griesemer, Rob Pike, and Ken Thompson. *The Go programming language*. 2007.
[7] Klaus Birkelund Jensen and Brian Vinter. a design for interchangable simulation and implementation. *Proceedings of Communicating Process Architectures 2015*, 2015.
[8] Wikipedia. Bean machine, 2017.
[9] HF Trotter. An elementary proof of the central limit theorem. *Archiv der Mathematik*, 10(1):226–234, 1959.
[10] Jan Dijkstra, Harry JP Timmermans, and AJ Jessurun. A multi-agent cellular automata system for visualising simulated pedestrian activity. In *Theory and Practical Issues on Cellular Automata*, pages 29–36. Springer, 2001.
[11] Peter H Welch. Process oriented design for java: Concurrency for all. In *International Conference on Computational Science (2)*, volume 687, 2002.
[12] Wikipedia. Ic4, 2017.

Case study: Bohrium – Powering Oceanography Simulation

Mads Ohm Larsen and Dion Häfner

Niels Bohr Institute, University of Copenhagen, Denmark {ohm, dion}@nbi.ku.dk

In the field of oceanography numerical simulations have been used for more than 50 years [1]. These simulations often require long integration time and can run for several real-time months. They are thus often written in high-performance languages such as C or Fortran. Both of these are often thought of as being complex to use. Fortunately we have seen a shift in the scientific programming community towards focusing more on productivity as oppose to just performance [2]. We would, however, like to keep the performance from these archaic languages.

Academic code often has a limited lifespan because the developers shift as people graduate and new people arrive. Having to use a long time to understand the simulations will take away from the actual science being made. Veros [3] is a Python translation of an already existing oceanography project written in Fortran. It utilizes NumPy [4] for its vector computations. In order to rectify the performance loss Veros have chosen Bohrium [5, 6] as its computational back-end.

Bohrium is a run-time framework that allows sequential interpreted code to be easily parallelized and possibly run on GPGPUs. This is done by just-in-time (JIT) compiling generated OpenMP, OpenCL, or CUDA kernels and running them on the appropriate hardware. Bohrium also support multiple front-ends, namely Python, C, and C++. In Python this is done with minimal intrusion, thus no annotations are needed, as long as the code utilize NumPy functions Bohrium will override these and replace them with JIT-compiled kernels. Bohrium has its own intermediate representation (IR), which is an instruction set gathered from the interpreted code up to a side effect, for example I/O. Well known compiler optimizations, such as constant folding and strength reduction, can be applied to the IR prior to generating the kernels.

Other optimizations include using already established libraries such as `BLAS` [7] for low-level linear algebra. Bindings to the appropriate `BLAS` library on your system is auto generated when Bohrium is compiled. This means, that if you have for example `clBLAS` installed, Bohrium will create bindings to it, that can be utilized directly from Python. These will also overwrite the NumPy methods already using `BLAS` for even better performance.

Using Bohrium of course comes with an overhead in form of generating the kernels. Fortunately this overhead is amortized for larger simulations. For the Veros project we see that Bohrium is roughly an order of magnitude faster than the same implementation using Fortran or NumPy in the benchmarks. However, a parallel Fortran implementation using MPI for communication is faster still. In the future we would like to utilize distributed memory systems with Bohrium, so we can run even larger problem sizes, using possibly multiple terabytes of memory.

References

[1] Markus Jochum and Raghu Murtugudde "Physical oceanography: Developments since 1950", Springer Science & Business Media, 2006

[2] Cherri M Pancake and Donna Bergmark, "Do parallel languages respond to the needs of scientific programmers?", IEEE, 1990

[3] "Veros documentation", `https://veros.readthedocs.io/`, Web, December 2017

[4] Travis Oliphant, "NumPy", `http://numpy.scipy.org/`, 2006, Web, December 2017

[5] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter, "Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster", PyHPC'13, 2013

[6] "Bohrium documentation", `https://bohrium.readthedocs.io/`, Web, December 2017

[7] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh, "Basic linear algebra subprograms for Fortran usage", ACM Transactions on Mathematical Software (TOMS), ACM, 1979

Communicating Process Architectures 2017
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2018

Emit – Communicating Sequential Processes in Ruby

Mads Ohm LARSEN [1] and Brian VINTER

Niels Bohr Institute, University of Copenhagen, Denmark

Abstract. CSP is an algebra for reasoning about concurrent systems of processes. Being able to do so has become a necessity for computer scientists. Having to think about abstractions like mutexes and threads in practice can be cumbersome, complex, and erroneous. Ruby as a programming language has been described as fun to program in. It is however missing a CSP framework that it can call its own. *Emit*, which is presented in this paper, tries to mitigate this by providing such a CSP framework. As a CSP framework, Emit makes it easy to think about processes, channels, and communication. It is not yet feature complete, however comparing it to its nearest peer, PyCSP, shows good performance for the COMMSTIME benchmark, where Emit is 100 times faster.

Keywords. CSP, Ruby, Concurrency

Introduction

Being able to reason about large scale systems has become more and more of a necessity. Many newer simulations are programmed to run on hundreds of thousands of cores and in just as many different processes. CSP (Communicating Sequential Processes) [1] is an algebra for reasoning about such systems.

Creating a network of processes and having them communicate can often be a cumbersome, complex, and erroneous task. There exists many different models to choose from, when thinking about concurrency, for example: threads, mutexes, shared memory, and fork/wait just to name a few. As developers trying to reason about a network of processes and their communications, we do not want to have to think about mutexes and thread handling. Instead we would like an easy API (Application Programming Interface) to describe the network.

The Ruby programming language has long been missing a CSP framework of its own. Yukihiro Matsumoto (Matz) – the creator of Ruby – has stated in an interview for [2], in response to what feature of Ruby he would change, that: "I would remove the thread and add actors or some other more advanced concurrency features". An emulation of Go's channel and process model, which looks quite similar to CSP, already exists [3,4], but is more emulating Go than CSP. A true CSP framework has yet to emerge. This paper presents such an implementation, namely: *Emit*.

Emit is being developed at University of Copenhagen. Its goal is to show that CSP and its abstractions can be easily understood without having to program large networks with for example Go, or worry about strong types as with JCSP. PyCSP [5,6] is already used at University of Copenhagen for its Extreme Multiprogramming course, however most students opted for Go in its last run and many struggled with their solutions to the exam problems.

[1]Corresponding Author: *Mads Ohm Larsen, Blegdamsvej 17, 2100 Copenhagen OE.* E-mail: ohm@nbi.ku.dk.

Emit is still undergoing development and thus the API presented in this paper might change. It is not entirely feature complete, but early experiments shows great performance when compared to PyCSP and JCSP.

1. Programming in Ruby

Ruby is an object-oriented programming language, that borrows from, amongst others, Perl, Smalltalk, Ada, and Lisp. It is reflective and dynamically typed, which means that, as the popular Ruby saying goes: "If it looks like a duck and quacks like a duck, it is a duck.". That is to say, the objects do not need to be a specific type, to fulfill a specific purpose. It is important to note, that this is not type coercion, but just objects fulfilling the same base types. An example is that a lot of objects have a `to_s` (to string) method, so no matter what kind of object, be it a `String`, `Integer`, `Cat`, `Dog`, or something else, you get, you will still be able to call `to_s` on it.

What we need to know about Ruby for this paper is that Ruby has modules, classes, and methods. We also need to know that everything in Ruby is an object, and can be treated as such.

For CSP, this means that we do not need to concern ourselves about what type we message over a channel, and as such, we do not need say `IntegerChannel`, `FloatChannel`, and so on, but can have just one channel type. This also means, that we can transfer processes and channels via channels and thus have mobility for free.

2. Introduction to Emit

The two main components of any CSP framework are processes and channels. An Emit program will have many processes and many communications over channels. A channel in Emit is further divided into reading and writing channel-ends, which is not part of Hoare's original theory, but is how JCSP also uses channels [7].

Let us first look at a very simple network of just two processes. Here one process sends a message to the other and then the network closes down because both processes are done. In Figure 1 this network is shown as a simple diagram with circles and arrows. The circles are processes and the arrows are channel communications.

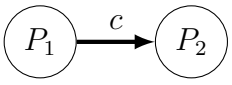


Figure 1. A simple network.

In CSP theory terms, this network would look like:

$$P_1 = c!x \rightarrow \checkmark$$
$$P_2 = c?x \rightarrow \checkmark$$
$$P_1 \parallel P_2$$

The ✓ here represents a process with successful termination. In the implementation, it could poison or retire its channels before termination, which we will look into in Section 3.3.

```
require "emit"

def p1(cout)
  cout << "Hello, world!"
end

def p2(cin)
  puts cin.()
end

c = Emit.channel

Emit.parallel(
  Emit.p1(-c),
  Emit.p2(+c)
)
# => Hello, world!
```

Listing 1: A simple network written in Emit.

```
require "emit"

p1 = Emit::Process.new(arg1) { |arg1| ... }

p2_proc = proc { |arg1| ... }
p2 = Emit::Process.new(arg1, &p2_proc)

p3_lambda = lambda { |arg1| ... } # or '->(arg1) { ... }'
p3 = Emit.process(arg1, &p3_lambda)

def p4_method(arg1)
  ...
end
p4 = Emit::Process.new(arg1, &method(:p4_method))

def p5_method(arg1)
  ...
end
p5 = Emit.p5_method(arg1)
```

Listing 2: Various ways of creating Emit processes.

The same network can be simulated with Emit shown in Listing 1. Here we create `p1` and `p2` as methods, that takes a channel-end as argument. The channel is being created in a global namespace and the channel-ends are being extracted with the unary plus, for the write end, and minus, for the read end. The two processes are being run in parallel, by giving them both as arguments to the `parallel` module method on `Emit`. Since both `p1` and `p2` are created in a global namespace, the `Emit` module will know about them, and they can thus be called with `Emit.p1` and `Emit.p2`.

Since Ruby is a highly dynamic programming language, we have various ways of setting up each process in the parallel structure as seen in the previous example. Listing 2 shows these various ways of doing so.

As a convenience for the user of Emit, we have made a module method `process` which is an alias for `Emit::Process.new`, so that you can use this shorthand every place you would

like to create new processes. There is also the extra shorthand, shown in Listing 2 as `p5`, where, if the method is created in a global namespace, the Emit module can be called with a method of the same name.

3. Emit Implementation

Emit is implemented in pure Ruby. This means that Emit can run anywhere there is a Ruby interpreter, which is virtually everywhere.

One aim of Emit is to make it easy and fun to program a CSP network, that easily passes messages between processes. Within Emit, the user should never have to consider locks, network exceptions, and the like. All of these are wrapped up inside Emit and should not be accessed by the user. However, if the user wanted to, Ruby allows for them to reopen closed classes and redefine already defined methods, so they can inject their own scheduler or similar into Emits code.

Emit borrows a lot of ideas from the PyCSP implementation [5], which in turn looks to JCSP [7,8,9] and C++CSP [10].

3.1. Processes

CSP processes are in Emit emulated with fibers [11]. Since everything in Ruby is an object, we create a `Process` class under the `Emit` module, to encompass all the information about a single process. This means that new processes are spawned, but not started, with the `new` method. As already stated, we have added aliases so that we can call an implicit new method instead, namely `process` on the `Emit` module.

An instantiated `Process` object knows a bunch of things. It has its inner code, called a block in Ruby, as well as its arguments. It also knows that it has yet to be executed and when you first instantiate it, it does not have a fiber attached either. When we get to run the processes, the `parallel` method will start all the processes, which in turn will give them all their own fiber to live in. These processes will not run, until the scheduler picks them up and transfers control to their fiber.

3.1.1. The Scheduler

In Ruby, fibers work by the main thread transferring control to them. An example of Ruby fibers can be seen in Listing 3. The `yield` class method on `Fiber` will transfer the control back to whomever gave it to you, which could be both another fiber or the main thread.

The scheduler in Emit works by manipulating fibers and giving control to them one at the time. Every process (fiber) is enqueued in the scheduler, when we run them. Then the scheduler looks at the queue and pops the process that will run next. This process can be put back at the end of the queue, if it starts communicating, and control be returned when both processes are ready to communicate.

As only one process is in control in the scheduler at a time, we avoid starvation of resources by switching away from that process as soon as it wants to communicate. Control is then resumed, if another process also wants to communicate on the same channel. If we ever get into a case where we would like to wake a process that is trying to communicate, but without a counterpart, Emit will throw a `DeadlockException` as seen in Figure 2, Listing 4, and the following algebra:

$$
\begin{aligned}
&P\left(1\right) \parallel Q\left(1\right) \\
&P\left(x\right) = c!x \rightarrow d?x \rightarrow P\left(x\right) \\
&Q\left(x\right) = d!x \rightarrow c?x \rightarrow Q\left(x\right)
\end{aligned}
$$

```ruby
require "fiber"

@f1 = Fiber.new do
  puts "F1: Hello to you too"
  Fiber.yield
  puts "F1: I'm done. (This will never print)"
end

@f2 = Fiber.new do
  puts "F2: Hello F1"
  @f1.transfer
end

puts "Main: I am going to resume @f2."
@f2.resume
# => Main: I am going to resume @f2.
# => F2: Hello F1
# => F1: Hello to you too
```

Listing 3: Fibers in Ruby.

```ruby
require "emit"

def deadlock_process(cout, cin)
  cout << 1
  cin.()
end

c = Emit.channel
d = Emit.channel

Emit.parallel(
  Emit.deadlock_process(-c, +d),
  Emit.deadlock_process(-d, +c)
)
# => DeadlockException
```

Listing 4: Deadlock in Emit.

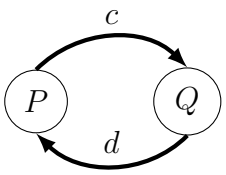


Figure 2. Deadlocked network.

3.2. Parallel and Sequence

The two main ways of running processes is `parallel` and `sequence`. In CSP terms these are the parallel operator $\|$ and the regular sequence ;.

In Emit sequence works by just starting and running each sequential process, circumventing the scheduler. Parallel on the other hand will start each process, enqueue it into the

```
1 Emit.parallel(
2   Emit.master(-ch1),
3   10.times.map { Emit.worker(+ch1, -ch2) },
4   Emit.sink(+ch2)
5 )
```

Listing 5: Parallel processes.

scheduler's queue and join the processes in the end, which in turn starts the scheduler, so it can schedule the processes to run.

Both `Emit.parallel` and `Emit.sequence` are methods that takes a list of processes. In Emit this could look like Listing 5.

The 10 times here shows how to create many of the same process in Emit. `parallel` can actually take an array of arrays of processes and will flatten this array into just an array of processes, before running everything in parallel.

3.3. Channels

The channels in Emit work like those of PyCSP, having two channel-ends, the read and write end, that can be used for only that purpose. All channels are therefore any-to-any channels, as you can get as many read and write ends as you please. These channel-ends are usually given to the processes as arguments, and can then be used in that process. A channel in Emit is just another Ruby object, with associated methods on it. A newly instantiated channel object has a write and read queue as well as the number of readers and writers. These two numbers are known, because of the read and write ends being extracted from the channels, when the processes are being instantiated. When a process needs an end, we can use the `reader` or `writer` method, or their Emit aliases the unary plus or minus.

Communication from a reader's perspective is done by putting itself into a read queue. After that, it tells the scheduler, that it is waiting. This transfers control to the next process in the scheduler's queue. This next process could or could not be the process that will also communicate on that channel. If it is, it will put itself into a write queue for that channel and will then immediately notice, that there is a reader present. It will then proceed to tell the reader to wake up with the message.

If the writer arrives to the communication first, the communication takes place in much the same way, but with the writer being placed into a write queue first and the reader just waking up the writer to get the message from it.

3.3.1. Channel Poisoning

Emit supports channel poisoning, first introduced in C++CSP [10] and JCSP [12]. If a process decides to poison its channel, every subsequent read or write on that channel will be invalid and throw an exception. This means that the poison will propagate through the network, and every process not being handled will shut down, because of an exception.

In Emit, we only propagate the poison to other channel-ends that were given as arguments to our current process. That is, if we create a new channel inside a process, Emit will not automatically propagate the poison on this channel. These channels the programmer would have to poison manually, via exception handling.

The poison itself works in the same way as in PyCSP, with an exception being thrown from the channel. This exception can then be caught to make it close down the channel in another way or do something else on a poison event.

A channel-end is poisoned in Emit by calling its `poison` method or the module method `poison` on Emit. The latter takes a list of channels to poison all at once.

```
def source(cout)
  10.times { cout << "Hello, world!" }
  Emit.poison(cout) # or cout.poison
end

def sink(cin)
  loop { puts cin.() }
end

c = Emit.channel

Emit.parallel(
  Emit.source(+c),
  Emit.sink(-c),
)
```

Listing 6: Poisoning a channel in Emit.

```
Emit.parallel(
  10.times.map { Emit.source(+c) },
  10.times.map { Emit.sink(-c) }
)
```

Listing 7: Running multiple sources and sinks in parallel.

An example of channel poisoning can be seen in Listing 6. The same example can be viewed with multiple of each type of process by changing the parallel construct to that of Listing 7.

In the Listing 7 example we might encounter a classical poisoning problem. Since multiple `source` processes can be working at the same time, not all `source` processes might be done, when the first process decides to poison the channel. To address this problem Emit supports channel retiring.

3.3.2. Channel Retiring

Channel retiring as introduced in [6] works very similar to poisoning, but works by counting the number of readers and writers. Every time you get a channel-end, this counter will increase by one. When a process retires from a channel, it decreases the respective counter by one. If one of the counters reaches zero, the channel has been retired and all subsequent requests will result in an exception being thrown, similar to poisoning the channel.

Using this instead in our example in Listing 6 and 7, by replacing the `poison` call with `retire`, we ensure that all sources are done, before the channel is poisoned and the network is shutdown.

4. Results

4.1. COMMSTIME

An already established metric for measuring networks and communications is the COMMSTIME benchmark [13]. Here we have four processes, often called: PREFIX (P), DELTA (D), SUCC (S), and CONSUME (C).

Figure 3 shows how a COMMSTIME network could look and the following algebra should encompass the entire COMMSTIME network:

$$
\begin{aligned}
&P\,(0) \parallel D \parallel S \parallel C(500000)\\
&P\,(x) = c!x \rightarrow d?y \rightarrow P(y)\\
&\quad D = c?x \rightarrow ((e!x \rightarrow f!x \rightarrow D) \;\square\; (f!x \rightarrow e!x \rightarrow D))\\
&\quad S = e?x \rightarrow d!\,(x+1) \rightarrow S\\
&C\,(0) = \checkmark\\
&C\,(n) = f?x \rightarrow C(n-1)
\end{aligned}
$$

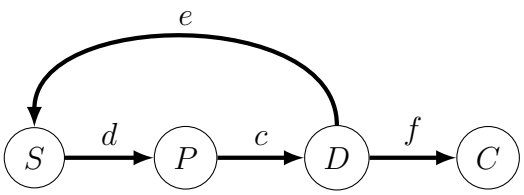


Figure 3. COMMSTIME network.

PREFIX is given 0 as an initial input and will keep reading from its input channel and outputting the same on its output channel. DELTA will read on its input channel and output the same on both its output channels. SUCC adds one to its input and outputs it.

In our results, shown in Table 1, we are using $2,000,000$ communications to test the system. This is done, by letting the CONSUME process count to $500,000$, as we have four communication per round-trip. The timings in the results are an average of 10 runs.

Table 1. COMMSTIME results.

Framework	Result (μs/communication)
PyCSP	346.30
PyCSP (greenlets)	6.04
JCSP	22.55
Emit	3.79
Go	0.28

The lower we can get the time per communication the better. Obviously Go wins, having this form of communication built into the language, but Emit is not that far off, only a factor 10, which is quite impressive for an interpreted language without these features built in. PyCSP with the default process type is a factor $100\times$ behind Emit. Using PyCSP with greenlets, Emit is still roughly twice as fast.

4.2. Monte Carlo π simulation

Another example we can look at, is the Monte Carlo π simulation. This is a classical producer-worker-consumer network, where we have one producer, a numbers (10) of workers, and one consumer at the end, collecting the results. The producer starts by producing the tasks and the workers each calculate their own estimate of π, that is then collected by the consumer and averaged. This should show that Emit is capable of having multiple active readers, the workers, waiting to read from the producer's channel. Since Emit uses fibers, these will still all run in the same process, so no true parallelism is being used, but the scheduler can still choose different processes each time it gets to a read/write.

```
def producer(job_out, bagsize, bags)
  bags.times { job_out << bagsize }
  Emit.retire(job_out)
end

def worker(job_in, result_out)
  loop do
    cnt = job_in.()
    sum = cnt.times.count { (rand**2 + rand**2) < 1 }
    result_out << (4.0 * sum) / cnt
  end
rescue Emit::ChannelRetiredException
  Emit.retire(result_out)
end

def consumer(result_in)
  cnt, sum = 0, result_in.()
  loop do
    cnt += 1
    sum = (sum * cnt + result_in.()) / (cnt+1)
  end
rescue Emit::ChannelRetiredException
  puts sum
end

jobs    = Emit.channel
results = Emit.channel

Emit.parallel(
  Emit.producer(-jobs, 1000, 10000),
  10.times.map { Emit.worker(+jobs, -results) },
  Emit.consumer(+results)
)
```

Listing 8: Monte Carlo π simulation in Emit.

An Emit Monte Carlo π simulation program is presented in Listing 8. An equivalent program was written for PyCSP. The results (time) can be seen in Table 2. These again show Emit to be faster than its inspiration: PyCSP.

Table 2. Monte Carlo π timings.

Framework	Time spend (s)
PyCSP	6.65
PyCSP (greenlets)	3.55
Emit	2.54

5. Conclusions

Emit has been developed at University of Copenhagen. This is also the home for PyCSP. PyCSP has been fortunate enough to have been used in the Extreme Multiprogramming course given at University of Copenhagen for multiple years. This course is however deprecated and a new course will take its place. It is not certain that CSP will be taught in the same way next year, so Emit might not benefit in the same way that PyCSP did from the students taking the course.

Emit has a lot of the same features as PyCSP. Many of these features are very preliminary, and thus lack further optimization. Emit is however shown to be a factor 100× faster than PyCSP and only a factor 10× slower than Go.

Emit can be downloaded from Github [14].

6. Future Work

As Emit works with fibers, we do not have true parallelism yet. An implementation with threads, OS processes, or some other parallel model could be interesting. Threads would run into the same problem that PyCSP has seen, namely the global interpreter lock (GIL), which means you cannot run multiple Ruby threads in parallel either, without going into C or another low-level language.

Some of the core features, like selective alt, are still missing, but planned to appear in a future release of Emit.

Acknowledgments

This work is part of the CINEMA project and financial support from **CINEMA**: The Alliance for Imaging of Energy Applications, DSF-grant no. 1305-00032B under The Danish Council for Strategic Research is gratefully acknowledged.

References

[1] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[2] Bruce A. Tate. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Pragmatic Bookshelf, 1st edition, 2010.

[3] Ilya Grigorik. Concurrency with Actors, Goroutines & Ruby. `https://www.igvita.com/2010/12/02/concurrency-with-actors-goroutines-ruby/`, 2010.

[4] Ilya Grigorik. Agent. `https://github.com/igrigorik/agent/`, 2010.

[5] Brian Vinter, John Markus Bjørndalen, and Otto Johan Anshus. PyCSP – Communicating Sequential Processes for Python. In *Communicating Process Architectures*, 2007.

[6] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. Pycsp revisited. In *CPA*, pages 263–276, 2009.

[7] Jim Moores. Native JCSP – the CSP for Java Library with a Low-Overhead CSP Kernel. volume 58, pages 263–273, 2000.

[8] PH Welch and PD Austin. The JCSP (CSP for Java) Home Page. `https://www.cs.kent.ac.uk/projects/ofa/jcsp/`, 1999.

[9] Peter H Welch. Process Oriented Design for Java: Concurrency for All. In *International Conference on Computational Science (2)*, volume 687, 2002.

[10] Neil CC Brown and Peter H Welch. An introduction to the Kent C++ CSP Library. volume 61, pages 139–156. IOS Press, 2003.

[11] Ruby Docs: Fibers. `https://ruby-doc.org/core-2.5.0/Fiber.html`, 2018.

[12] Bernhard HC Sputh and Alastair R Allen. JCSP-Poison: Safe Termination of CSP Process Networks. In *CPA*, volume 63, pages 71–107, 2005.

[13] Frederick RM Barnes and Peter H Welch. Prioritised Dynamic Communicating Processes – Part I. *Communicating Process Architectures*, 2002:321–352, 2002.

[14] Mads Ohm Larsen. Emit. `https://github.com/omegahm/emit`, 2018.

Communicating Process Architectures 2018
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2018

Bohrium.rb – The Ruby Front End

Mads Ohm LARSEN [1] and Brian VINTER

Niels Bohr Institute, University of Copenhagen, Denmark

Abstract. The acceptance of Ruby in the scientific community lags a bit behind, partly because it is missing a good library for linear algebra and vector programming. It has a `matrix` class in its standard library, but its execution tends to be rather slow. Only a couple of actual scientific computing libraries like NumPy for Python exist for Ruby. In this paper we introduce a new library called *Bohrium.rb*. Bohrium.rb acts as a front end for the Bohrium framework, which generates and runs JIT-compiled OpenMP/OpenCL kernels. It currently supports Python/NumPy and C++, however as it is built of processes communicating hierarchically to each other, we can replace the front ends with new ones. This new Ruby front end is described with examples and is then compared to the standard library and an already established Ruby library Numo/Narray, where Bohrium.rb seems to be faster for still larger matrix calculations. This is also the trend we have seen in similar areas with Bohrium, being faster once its overhead has been amortized.

Keywords. Ruby, Bohrium, parallel computing, OpenMP, OpenCL, JIT-compile

Introduction

Ruby, as a programming language, has always lacked impact in the scientific code community. This is a shame, since Ruby have often been said to be fun to program in. Ruby has a lot of the same capabilities as Python, but lacks a linear algebra package like NumPy [1] for Python. To the best of our knowledge, there exist only a couple of packages in the Ruby community that deal with the same problems as NumPy. One of these is Numo/Narray [2], which we will compare our newly built framework against in this paper. Ruby also has a `matrix` class in the standard library (STL), that is able to do some matrix calculations, but, as we will see in the result section of this paper, it is rather slow and can be sped up a great deal.

This paper introduces a new such framework for Ruby, that relies on the already established automatic parallelization framework Bohrium [3]. The Bohrium framework, with its Python/NumPy and C++ front ends, has already been presented and discussed several times [4,5,6,7]. The focus has mostly been on speeding up the various NumPy methods and comparing against other such Python frameworks or even hand-coded OpenMP or OpenCL kernels.

We will start by given a brief presentation of the core concepts of Bohrium and then introduce a new front end as well as performance results on using it.

1. Bohrium

Bohrium consists of a number of disjoint processes. As seen in Figure 1, each speaks with the next by passing output and options along in a serial fashion. At the very top we find the

[1]Corresponding Author: *Mads Ohm Larsen, Blegdamsvej 17, 2100 Copenhagen OE.* E-mail: `ohm@nbi.ku.dk`.

front ends. As shown in Figure 1 we currently have Python, C++, and the new Ruby front end, which this paper is introducing. The front ends speaks to a bridge, whose job it is to translate the front end code to Bohrium's intermediate representation (IR). This IR is then optimized using various optimization steps, fused together in a manner that makes sense for the back end and lastly given to one back end, that might pass it along to several others. The back ends execute the generated kernels and return the results.

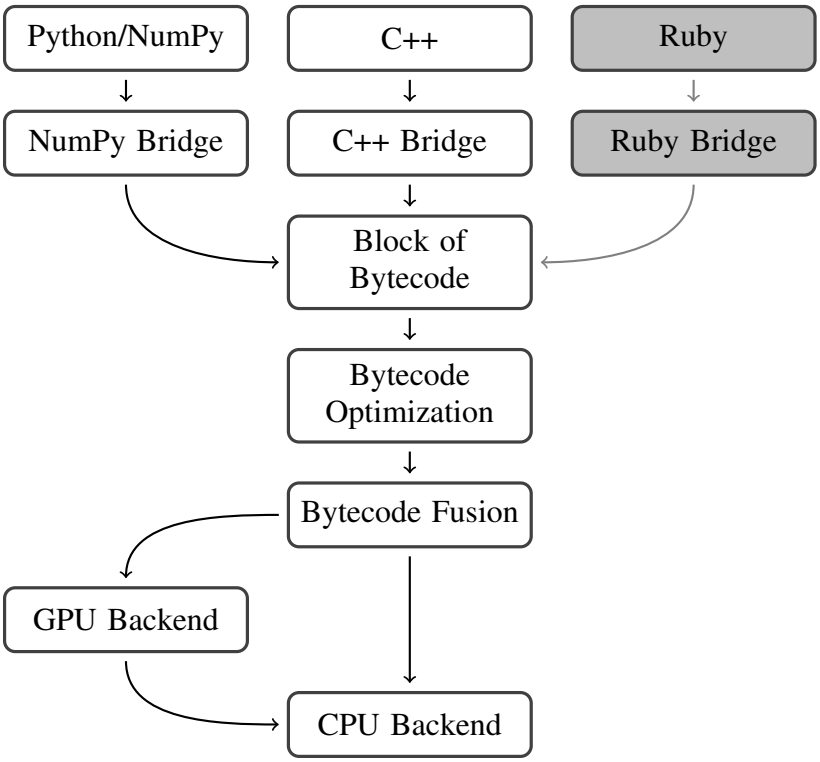


Figure 1. Process Overview.

The current back ends in Bohrium are OpenMP, OpenCL, and CUDA. These are each capable of transforming Bohrium's IR into code that can be executed on either a CPU or a GPU.

Thus, writing simple code in one of the front ends will yield high performance architecture specific code for CPUs or GPUs in OpenMP, OpenCL, or CUDA depending on your current environment variables and choices.

The claim of this paper is thus that the new Ruby front end will allow Ruby programmers to generate fast array processesing code for execution on CPUs or GPUs.

2. Bohrium.rb

The Ruby front end is called `bohrium`, but to distinguish it from *Bohrium* the framework and `bohrium` the Python package, we will discuss it here as Bohrium.rb[2].

Bohrium.rb is a Ruby gem[3] project, that allows Ruby programmers to get the performance and ease-of-use out of JIT-compiling and executing OpenMP, OpenCL, or CUDA kernels.

[2] `rb` is the common file extension for Ruby files.

[3] A self-contained Ruby library.

The Python package for Bohrium overwrites already established classes, however, for Bohrium.rb we have chosen instead to create a new array class, named `BhArray`, which works similar to that of NumPy's arrays. This takes away one of the selling-points of Bohrium: the non-obtrusiveness. You will have to switch your code to use Bohrium.rb instead of the STL. This, too, is true for the C++ bridge, where the programmer needs to use our interface. This might be rectified in future versions of Bohrium.rb, but since the array programming paradigm does not correspond with the Ruby STL, this might not be possible. That is, in Ruby, the array class will work differently than the `BhArray` class does, in the sense that you, for example, cannot add two STL arrays directly, as we will see shortly.

This new array class has many methods associated with it, some of which will be discussed in the following sections. In order to import Bohrium.rb we use the Ruby require statement – `require "bohrium"`. To keep the code examples in this paper short and to the point, this require statement is omitted from most of them.

2.1. Bohrium and Ruby

Bohrium.rb installs when you are compiling Bohrium, simply by adding `-DRUBY_BRIDGE=ON` to the CMake settings. This is off by default, since not everybody wants to install Bohrium.rb on their system (they might not be coding in Ruby).

Figure 2 shows how Bohrium.rb works with Ruby. First we have some Ruby code being executed by the Ruby runtime. In this code, we might encounter Bohrium.rb operations. These operations will not be executed immediately, but instead will be collected in Bohrium to then be lazily evaluated afterwards. We collect as many operations we can, until we need to give a result back to Ruby, which in Figure 2 is represented by a side effect. Only when we encounter a side effect does Bohrium kick in and actually execute all the commands. These commands might be split up into several kernels, depending on their respective sizes and shapes. After the side effect, the result is given back to the Ruby runtime, which can then use the result as a normal Ruby array or scalar value.

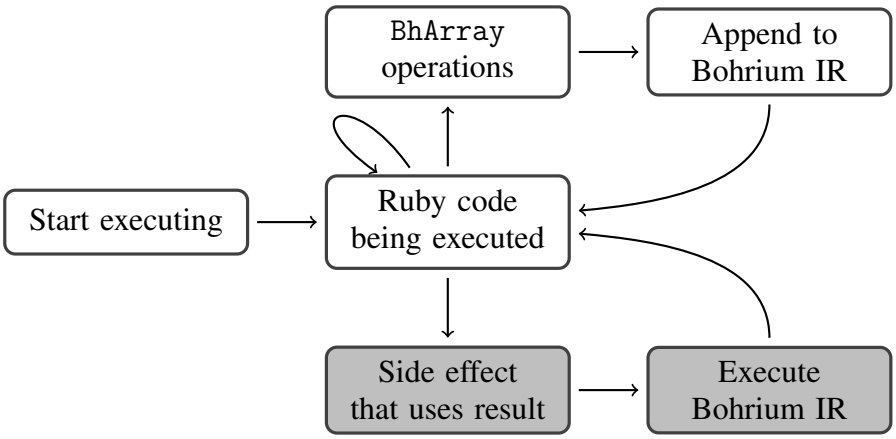


Figure 2. How Bohrium works with Ruby.

In all the following examples we print the result. This is one of the side effects, which triggers Bohrium to actually execute the operations.

2.2. Initialize a `BhArray`

The constructor – in Ruby called `initialize` – for `BhArray` takes an STL array as an argument. This array is copied into Bohrium memory and can thereafter be used in Bohrium contexts. This is shown in Listing 1, where we create a Ruby STL array, then use that to create a Bohrium.rb array. Both are then printed and, as seen in the listing, both show the same array

```ruby
ruby_ary = [1, 2, 3]
bh_ary = BhArray.new(ruby_ary)

p ruby_ary # => [1, 2, 3]
p bh_ary   # => [1, 2, 3]
```

Listing 1: Initialize `BhArray` with Ruby STL array.

```ruby
# Create 5x1 array of ones
ary1 = BhArray.ones(5, 1)
p ary1 # => [1, 1, 1, 1, 1]

# Create 3x2 array (matrix) of zeros
ary2 = BhArray.zeros(3, 2)
p ary2 # => [[0, 0], [0, 0], [0, 0]]

# Create a range of length 10 starting from 0
ary3 = BhArray.arange(10) # => could also be BhArray.seq(10)
p ary3 # => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listing 2: Initializing `BhArray` using various class methods.

printed out. The Bohrium array does not just have a reference to the Ruby STL array, but it converts its own data to a Ruby STL array in order to print it. It does not keep the reference, since we could later change the Bohrium array directly, which we would like to do, without changing the Ruby STL array. This means that initializing a `BhArray` in this way is rather heavy, since the STL array needs to be traversed and each value copied into a corresponding Bohrium array.

Another way of initializing arrays in Bohrium.rb is via the class methods for `BhArray`. This can be seen in Listing 2, where we use `ones`, `zeros`, and `arange` to create Bohrium arrays. The `arange` method is taken from NumPy, and is kept in Bohrium.rb to compare the two. It is aliased in Bohrium.rb to `seq`, which seems to be a more sensible name.

It is of course far faster to use the second type of initialization, as you do not need to first represent the array in the Ruby STL format.

2.3. *Working with `BhArrays`*

In Bohrium.rb we have all the common math operators. You can add two arrays, which will add them element-wise, subtract them, multiply them and divide two arrays with each other. You can also raise an array to a power, take the cosine of each value, or bitwise `and` them all. This is just like you are used to from the NumPy interface.

None of these things are possible directly with the Ruby STL. Here when you add two arrays, you get an array of their summed elements, which is shown in Listing 3 together with Bohrium.rb's `add` method, here shown as the aliased binary + operator.

Since Bohrium is able to overwrite the source array, Bohrium.rb also support a couple of so-called bang-methods[4], for example `add!`. These methods are the same as their non-bang counterparts, but will overwrite the source array in place. An example of this can be seen in Listing 4. Internally in Bohrium you would get an IR as seen in Listing 5, where Bohrium only allocates two arrays of size 5, instead of the two and one for the result.

[4]Called so because of the exclamation point, or the *bang*.

```
bh_ary1 = BhArray.ones(5, 1)
bh_ary2 = BhArray.ones(5, 1)
bh_ary3 = bh_ary1 + bh_ary2 # Aliased from `#add`
p bh_ary3 # => [2, 2, 2, 2, 2]

rb_ary1 = [1, 1, 1, 1, 1]
rb_ary2 = [1, 1, 1, 1, 1]
rb_ary3 = rb_ary1 + rb_ary2
p rb_ary3 # => [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Listing 3: Adding BhArrays.

```
bh_ary1 = BhArray.ones(5, 1)
bh_ary2 = BhArray.ones(5, 1)
bh_ary1.add!(bh_ary2) # => We use `add!` here
# bh_ary1 was overwritten
p bh_ary1 # => [2, 2, 2, 2, 2]
```

Listing 4: Adding BhArrays with bang-methods.

```
BH_IDENTITY a1[0:5:1] 1
BH_IDENTITY a2[0:5:1] 1
BH_ADD a1[0:5:1] a1[0:5:1] a2[0:5:1]
```

Listing 5: Bohrium IR for add!.

```
bh_ary = BhArray.arange(10)
p bh_ary[4] # => [4]
```

Listing 6: Indexing with a single index.

All the other math methods you could imagine are also present in Bohrium.rb, such as arccos, logical_not, floor, and so on. Bohrium also supports reduction methods; these are all also present in Bohrium.rb, such as add_reduce, which we will use to show some performance results later in section 3.2.

2.4. Views

When doing array programming, we often want to have just a view into our large matrices. This can be done in Bohrium.rb by indexing into the BhArray, just like you would a normal array. You can grab just a single element, by using for example the code in Listing 6. We return a new array, since we always return a view – a portion of the entire array – even though this is just a single element.

With the reshape method, we can reshape a BhArray from being one-dimensional into being many-dimensional. BhArrays are one-dimensional when they are first created, so this reshape method becomes essential for the entire program. For the sake of a simplicity we will only concern ourselves with one- and two-dimensional BhArrays here.

In Listing 7 we see a two-dimensional array being indexed with a Ruby STL range. This tells us to take the values in the first and second row and column.

```
bh_ary = BhArray.arange(25).reshape([5, 5])
p bh_ary             # => [[ 0,  1,  2,  3,  4],
                     #     [ 5,  6,  7,  8,  9],
                     #     [10, 11, 12, 13, 14],
                     #     [15, 16, 17, 18, 19],
                     #     [20, 21, 22, 23, 24]]
p bh_ary[0..1, 0..1] # => [[ 0,  1],
                     #     [ 5,  6]]
```

Listing 7: Indexing with two Ruby STL ranges.

```
bh_ary = BhArray.arange(25).reshape([5, 5])
p bh_ary[0..1, true] # => [[ 0,  1, 2, 3, 4],
                     #     [ 5,  6, 7, 8, 9]]
```

Listing 8: Indexing with a Ruby STL range and `true`.

```
bh_ary = BhArray.arange(9).reshape([3, 3]) # => [[0, 1, 2],
                                           #     [3, 4, 5],
                                           #     [6, 7, 8]]
bh_ary[0..1, 0..1] = 2
p bh_ary # => [[2, 2, 2],
         #     [2, 2, 5],
         #     [6, 7, 8]]
```

Listing 9: Setting a view to a constant.

```
bh_ary = BhArray.arange(9).reshape([3, 3])
bh_ary[0..1, 0..1] = BhArray.new([0, -1, -3, -4]).reshape([2, 2])
p bh_ary # => [[ 0, -1,  2],
         #     [-3, -4,  5],
         #     [ 6,  7,  8]]
```

Listing 10: Setting a view to another view of same shape.

Instead of asking the array for its size, if you want everything from one of its dimensions, you can simply add `true` as seen in Listing 8. This says to take the first and second row, but all the columns in each of these. Had we reversed the order, we would have gotten all rows for the first two columns instead.

Since we can now index into views, we should also be able to set views. This is possible through the assignment operator `[]=` on a view as seen in Listing 9. Here we again get the first two columns and rows, and then we set them to the value 2 for all entries.

Another example of this is where we assign a different view to a view indexed in an array as seen in Listing 10. All that is required is that the new view is the same shape as the view that is being overwritten.

3. Results

The performance from Bohrium.rb comes from Bohrium's ability to translate into the newest, as of the writing of this paper (August 2018), OpenMP, OpenCL, or CUDA kernels. In this

```
ary1 = [2] * n
ary2 = [3] * n
ary1.zip(ary2).map { |i| i.reduce(:+) }
```

Listing 11: ADD benchmark – Vanilla Ruby (`zip`).

```
ary1 = [2] * n
ary2 = [3] * n
ary1.zip(ary2).map(&:sum)
```

Listing 12: ADD benchmark – Vanilla Ruby (`zip_sum`).

```
ary1 = [2] * n
ary2 = [3] * n
ary3 = Array.new(n)
ary1.each_with_index { |elem, idx| ary3[idx] = elem + ary2[idx] }
# ary3 has the result
```

Listing 13: ADD benchmark – Vanilla Ruby (`idx`).

section, we will be performing two benchmarks, to see how Bohrium.rb's performance is compared to the STL and to a similar gem already existing for Ruby: Numo/Narray. Numo/-Narray has a similar API to NumPy and Bohrium.rb. The two benchmarks are fairly similar, both add numbers; however they work on different datasets, the first on a one-dimensional array, the second on a large matrix.

The speed-up numbers were gathered on a MacBook Pro with a 3.1 GHz Intel Core i7 processor and 16 GB 1867 MHZ DDR3 RAM. All benchmarks were run with the default OpenMP setting in Bohrium, and with the default Numo/Narray settings. The numbers are all gathered as an average across 10 runs.

3.1. Add

The first benchmark is simply to add a large set of numbers together. We create two arrays of the same size and then add them together element-wise, like we saw in Listing 3.

As with everything with Ruby, there are multiple ways of achieving this with the STL. In Listings 11 to 18 we see the different implementations of adding all numbers in the array. For the `matrix`, Numo/Narray, and both Bohrium benchmarks, we have a matrix with a zero-dimension instead of an array.

The first four vanilla Ruby implementations are different ways of implementing the same thing. The fifth method uses the Ruby STL matrix implementation, which adds two matrices in the way we would expect, namely element-wise.

The first `zip` method is the most idiomatic way of writing this adding scheme in Ruby[5].

The results on running these 8 different benchmarks can be seen in Figure 3. This graph shows speed-up compared to the vanilla Ruby `zip` implementation, since this is both the most idiomatic, but also slowest, solution. For the very small cases, 10^1 and 10^2 (not shown here), both Numo/Narray and Bohrium.rb are slower, because of the inherent overhead in each. For the largest case here, 10^8, we see that Numo/Narray is a factor $2.2\times$ faster than `zip`, but it gets beaten by the more *clever* `map!` solution. Bohrium.rb seems to get the fastest, here

[5]As asked in a local Ruby user group.

```
ary1 = [2] * n
ary2 = [3] * n
ary1.map!.with_index { |elem, idx| elem + ary2[idx] }
# ary1 is overwritten with the result
```

Listing 14: ADD benchmark – Vanilla Ruby (`map!`).

```
require "matrix"
m1 = Matrix[[[2] * n]]
m2 = Matrix[[[3] * n]]
m1 + m2
```

Listing 15: ADD benchmark – Vanilla Ruby (`matrix`).

```
require "numo/narray"
m1 = Numo::Int64.new(n, 1).fill(2)
m2 = Numo::Int64.new(n, 1).fill(3)
m1 + m2
```

Listing 16: ADD benchmark – Numo/Narray.

```
require "bohrium"
ary1 = [2] * n
ary2 = [3] * n
a = BhArray.new(ary1)
b = BhArray.new(ary2)
a + b
```

Listing 17: ADD benchmark – Bohrium.rb (init).

```
require "bohrium"
a = BhArray.ones(1, n, 2)
b = BhArray.ones(1, n, 3)
a + b
```

Listing 18: ADD benchmark – Bohrium.rb (ones).

2.8× faster. The *init* solution is of course slower, since we have to create the STL arrays and transfer than to Bohrium.rb afterwards, before computing the sum and returning the result.

Listing 19 contains the Bohrium IR for the program shown in Listing 18. Here we see that all 1×10000 elements in `a0` is set to 2. Likewise `a1` is set to 3. Finally we add the two arrays, and put the result into `a2`.

3.2. Sum

Summing columns poses a bit more challenging task. To begin with, we no longer have a one-dimensional array – or a matrix with one dimension being equal to one. Instead we have a matrix with a fixed amount of columns – here 1000. The number of rows are then increased as the experiment goes on. The number of columns are the size of our result array, and are

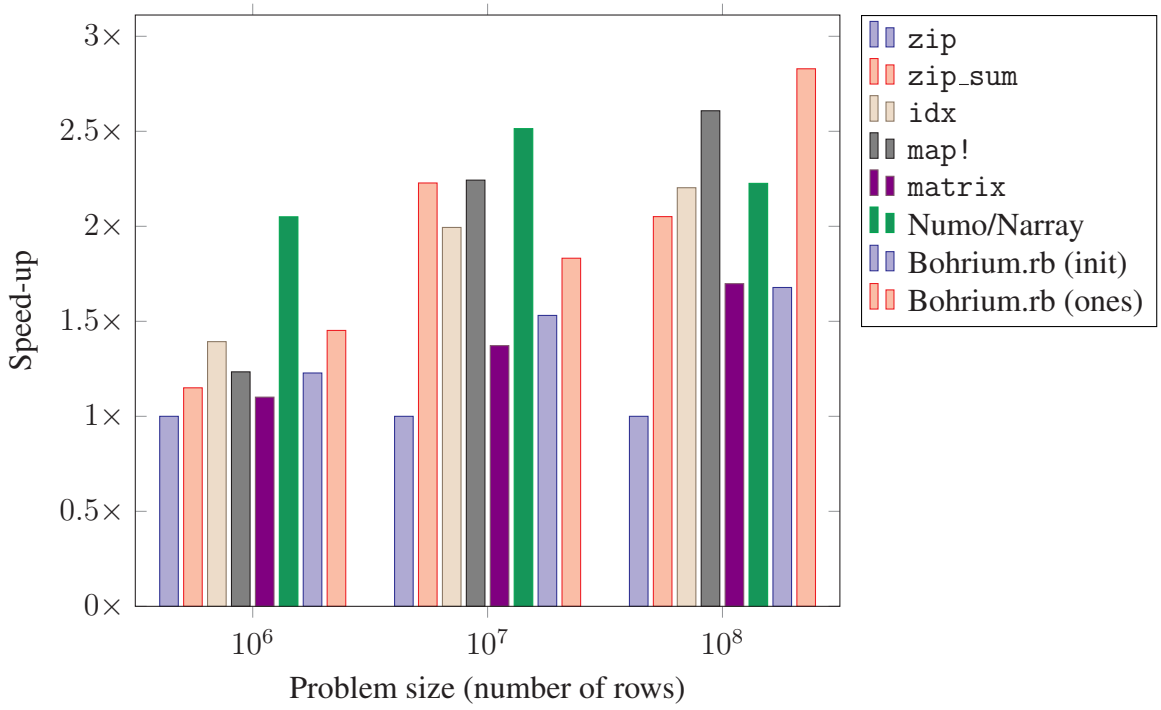


Figure 3. ADD benchmark results.

```
1 BH_IDENTITY a0[0:1:10000,0:10000:1] 2
2 BH_IDENTITY a1[0:1:10000,0:10000:1] 3
3 BH_ADD a2[0:1:10000,0:10000:1] a0[0:1:10000,0:10000:1] a1[0:1:10000,0:10000:1]
4 BH_FREE a0[0:1:10000,0:10000:1]
5 BH_FREE a1[0:1:10000,0:10000:1]
6 BH_FREE a2[0:1:10000,0:10000:1]
```

Listing 19: Bohrium IR for Listing 18.

thus, in terms of measuring the performance, not important, so we simply choose a random number.

Again, as with the adding benchmark, we time both the creation of the matrix and the work done, here summing each column to a single number, yielding 1000 numbers at the end. The numbers in the matrix are consecutive numbers starting from 0. Thus this benchmark tests both the generation of the sequential numbers as well as the ability to sum memory, that lives in row-major order, efficiently.

The various implementations can be seen in Listings 20, 21, and 22. From this, it should be easy to see, that the standard library is not suited for this kind of computations. The Bohrium.rb and Numo/Narray implementations are fairly similar, because they both agree that it should be easy to sum over an axis in a matrix. We could easily add `sum` as an alias for `add_reduce`, however `add_reduce` gives a better sense of what is going on beneath.

Looking at the results in Figure 4 we see that Numo/Narray is fast, even for small matrices, however it quickly start to decline and gets beaten by Bohrium.rb for 10^5 number of rows, where Bohrium.rb is approximately 43 times faster than vanilla Ruby. The reason 10^6 sees a decrease in speed-up for Bohrium.rb might stem from the fact that 10^6 rows times 1000 columns yields matrices larger than 16 GB of data, which cannot be stored in memory on the MacBook Pro that the experiments were conducted on.

Listing 23 contains the Bohrium IR for Listing 22. Here we see `BH_RANGE` in action. This is the `seq` method in the Ruby code. The `BH_ADD_REDUCE` is the `add_reduce` and the

```ruby
i = -1
ary = Array.new(r) { Array.new(c) { i += 1 } }
result = Array.new(c) { 0 }
c.times do |col_idx|
  r.times do |row_idx|
    result[col_idx] += ary[row_idx][col_idx]
  end
end
```

Listing 20: SUM benchmark – Vanilla Ruby.

```ruby
require "numo/narray"
result = Numo::Int32.new(r * c).seq.reshape(r, c).sum(axis: 0)
```

Listing 21: SUM benchmark – Numo/Narray.

```ruby
require "bohrium"
result = BhArray.seq(r * c).reshape([r, c]).add_reduce(0)
```

Listing 22: SUM benchmark – Bohrium.rb.

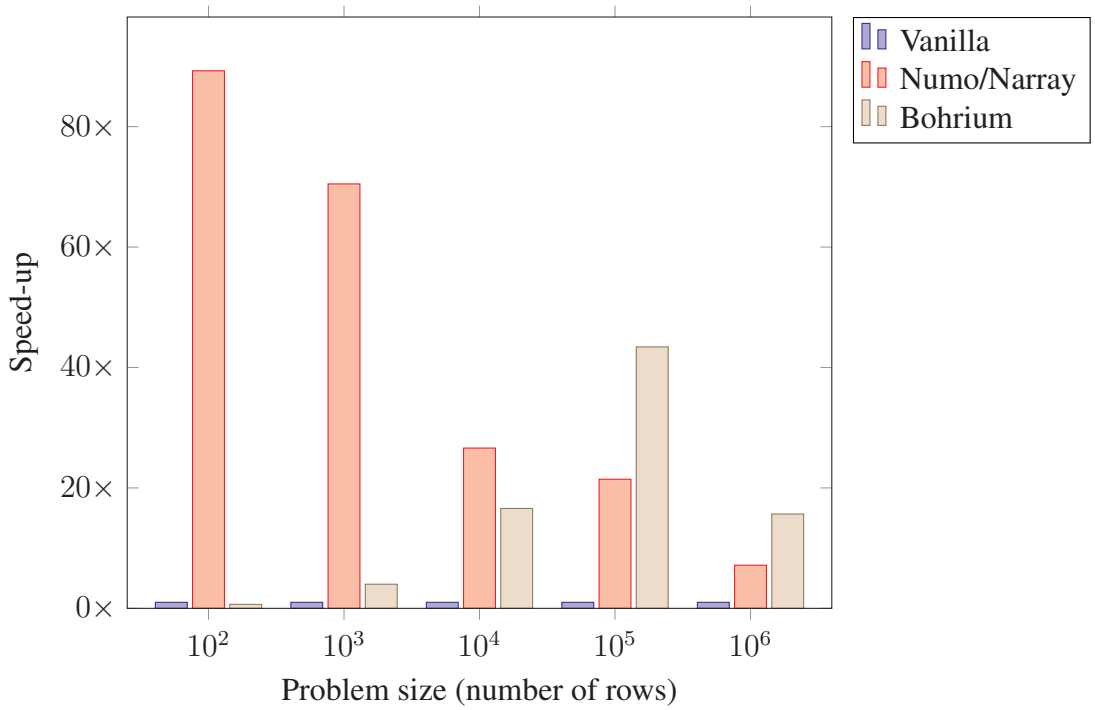


Figure 4. SUM benchmark results.

last parameter, the 0 here, is the axis on which we make the reduction. We notice that `a0` is a smaller array than `a1` and `a2`, which makes sense, since we store the result here and we are summing the 10^6 rows into 1000 columns.

```
1 BH_RANGE a1[0:100000:1]
2 BH_IDENTITY a2[0:100000:1] a1[0:100000:1]
3 BH_ADD_REDUCE a0[0:1000:1] a2[0:100:1000,0:1000:1] 0
4 BH_FREE a1[0:100000:1]
5 BH_FREE a2[0:100000:1]
6 BH_FREE a0[0:1000:1]
```

Listing 23: Bohrium IR for Listing 22.

4. Future Work

Some work has been put into making Bohrium.rb user-friendly. We have different ways of initializing new arrays as well as many ways of using the various math operators on them, for example with both named methods and symbols, as well as overwriting (bang) methods already popular in the Ruby community. We did however have to specify a new array interface. This new interface has to be maintained. Instead we could try to "monkey patch"[6] the `BhArray` interface onto the current `Array` or maybe `Matrix` class in the STL. The `Matrix` class already does several of the things we are interested in with Bohrium.rb, just internally in Ruby instead of generating OpenMP or OpenCL kernels.

Most of the methods used in Bohrium.rb are defined in the Bohrium framework in a JSON-file. On compile time this JSON-file is read and the entire Bohrium.rb gem is generated from that. In the future we might have some intermediate steps, where we clean the JSON-file or add/remove methods from it.

The utility methods for Bohrium.rb are all written in C++. These could be optimized better. Bohrium.rb does not help the Ruby garbage collector to mark unused arrays yet, which would be of huge benefit, for it to clean up, when an array is no longer referenced in future Bohrium instruction sets.

5. Conclusions

We have shown a new Ruby front end for the automatic parallelization framework Bohrium. This front end has the same basic features as NumPy for Python does. This means that programmers can now achieve automatic parallelization with the Ruby programming language by requiring and using `bohrium`.

We have also shown that Bohrium.rb is faster than the already established library Numo/-Narray when considering large matrices. Where Numo/Narray only beats the STL by a factor of $7\times$, Bohrium.rb is $15.5\times$ faster, double the speed of Numo/Narray. This is for the largest problem size, however for the second largest tested, Bohrium.rb was actually $43.4\times$ faster than the STL and Numo/Narray only $16.6\times$ faster. At the moment however, Bohrium.rb is still fairly new and not at all feature-complete compared to Numo/Narray. Bohrium.rb does support all the expected methods that you could need in a linear algebra setting, but Numo/-Narray has a lot of utility methods that Bohrium.rb would do well to copy.

Acknowledgments

This work is part of the CINEMA project and financial support from **CINEMA**: The Alliance for Imaging of Energy Applications, DSF-grant no. 1305-00032B under The Danish Council for Strategic Research is gratefully acknowledged.

[6]The term used in the Ruby community when opening an already closed class to extend it and overwrite certain methods.

References

[1] Travis E Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.
[2] Ruby/Numo – Numo/Narray. `https://ruby-numo.github.io/narray/`. [Online; accessed May 2018].
[3] The Bohrium Team. The Bohrium Project. `http://www.bh107.org/`. [Online; accessed May 2018].
[4] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.
[5] Mads R. B. Kristensen, Simon A. F. Lund, T. Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.
[6] Mads Ohm Larsen, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter. Current Status and Directions for the Bohrium Runtime System. In *Compilers for Parallel Computing 2016*, 2016.
[7] Kenneth Skovhede and Simon A. F. Lund. NumCIL and Bohrium: High productivity and high performance. In *Proceedings of the 11th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, LNCS. Springer, 2015.

Communicating Process Architectures 2018
K. Chalmers, J.B. Pedersen et al. (Eds.)
IOS Press, 2018

Towards Automatic Program Specification Using SME Models

Alberte THEGLER [1], Mads Ohm LARSEN, Kenneth SKOVHEDE, and Brian VINTER

Niels Bohr Institute, University of Copenhagen, Denmark

Abstract. This paper introduces a method to simplify hardware modeling and verification thereof in order for software programmers to, more easily, meet the demands of the growing embedded device industry. We describe a simple method for transpiling from the new SME Implementation Language into CSP$_M$ and using formal verification to verify properties within the generated program. We present a small example consisting of a seven segment display clock network and introduce how to verify the widths of the channels in the network.

Keywords. CSP$_M$, SME, transpiling

Introduction

The Internet of Things, computerized medical implants, and the omnipresent growth in robotics, brings with them an increased demand for programmers to develop software for those devices. While this observation may not in itself appear to present a new challenge, many other areas have previously presented a need for more programmers. The new challenge is that these new growth areas are all focused on small size, low power consumption, and high reliability. This means that traditional software engineering methods, and thus traditionally trained programmers, are often not sufficiently qualified to develop these technologies. In previous decades such systems have been developed by electronic engineers that apply far more rigid development approaches. Especially for hardware solutions like VLSI[2] and FPGA[3], correctness has always been favored over productivity. While tools have obviously improved and methods refined, the VLSI process is still mostly the same as presented in [1]. The primary workflow from [1] is shown in Figure 1; note the focus on verification in each step.

While the VLSI community is fundamentally following this 1980's design approach, more high-level tools and abstractions have been introduced. Philippe et al. [2] show a workflow (reproduced in Figure 2) where the important part is the verification that has been partly automated by basing the development on a formal specification of the solution.

There is no denying that the subjectively slow and rigid development process in the VLSI world [3] is highly successful in producing correct and reliable circuits. At the same time, conventional software development is highly focused on productivity and time-to-market, for example, smartphone applications are often developed for continuous release, where bug patches and new features are rolled out daily. This is of course not possible with hardware.

[1]Corresponding Author: *Alberte Thegler, Blegdamsvej 17, 2100 Copenhagen OE*. E-mail: tpq587@alumni.ku.dk.

[2]Very-large-scale integration.

[3]Field-Programmable Gate Array.

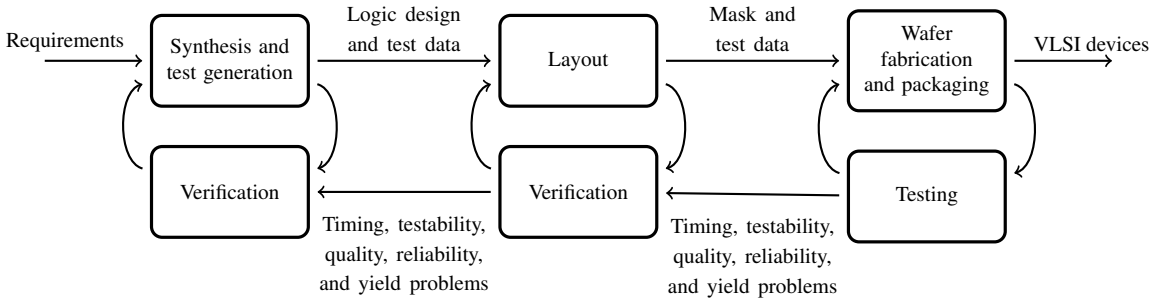


Figure 1. VLSI process workflow.

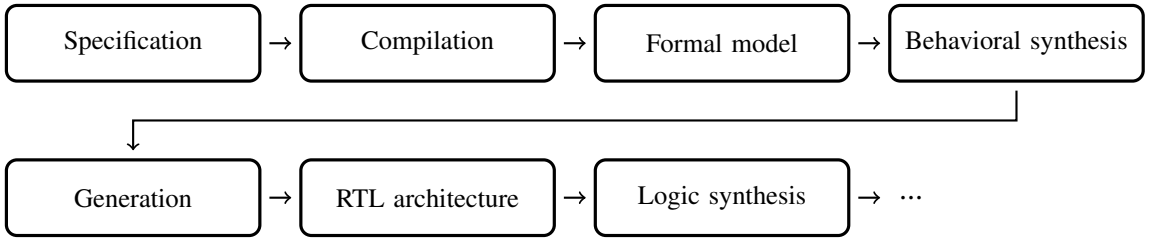


Figure 2. Reproduced workflow from Philippe et al. [2].

Thus, the authors argue that there is a growing chasm between the way most programmers are trained and the competencies that are needed to support the growth in mission critical embedded devices.

In this work, we propose a tool to help bridge the gap between available programmer profiles and the required competencies for embedded devices. Our approach is based on building a specification from a software implementation and test-suite observations. The overarching goal is to reach a level where a conventional software programmer can write a solution in Synchronous Message Exchange (SME) [4,5], and develop a conventional test suite in the software engineering tradition. By combining the implementation with the *observed* values of internal states in an SME based system implementation, we can produce a formal specification of the system. This specification can be fed into a formal verification tool and thus improve the correctness guarantees from only what is covered by the individual test vectors to the entire space that is spawned by the set of test vectors. We approach the task by transpiling[4] the new SME Implementation Language (SMEIL) [6] for SME into CSP$_M$ [7] and verify the formal properties of this version with a tool like FDR4 [8].

This paper builds on the SME model, which have been covered in papers [4,5,9]. In this paper we only include a brief description of the elements required to understand the setup we have developed, and encourage readers to seek out more information in the mentioned papers.

[4]Source-to-source compile.

1. Background

1.1. Synchronous Message Exchange

SMEIL is based on the SME model and therefore we give a brief introduction to SME.

SME was first introduced in 2014 and after several iterations [4,5,9] now presents as a programming model, a simulation library, and VHDL code generators [10]. The original idea was conceived following an attempt to create hardware descriptions from a vector processor model, modeled in PyCSP [11], a Communicating Sequential Processes (CSP) [12] library for Python. After this attempt, it became clear that the structure of CSP was poorly suited for modeling clocked systems, and therefore it was decided to create the SME model, based on the CSP algebra. The idea was to only use the subset of the CSP algebra that provided beneficial functionality to hardware modeling which, most importantly, meant that external choice was omitted. However, the shared-nothing property of CSP showed to be very useful, since the network state could only be changed by process communication.

In SME, a network is a combination of processes that are connected through buses. The processes communicate through a collection of signals in a bus, instead of CSP's synchronous rendezvous model, but retains the shared-nothing trait of CSP. SME uses the term `bus` instead of `channel` to enforce the semantic correlation between the SME bus and a physical hardware signal bus. The process communication is handled by a hidden clock which eliminates the complexity that arose from adding synchronicity to a CSP network. The combination of the hidden clock and the synchronous message passing between processes means that the SME model provides hardware-like signal propagation.

An SME clock cycle consists of three phases: it reads, executes, and writes as can be seen in Figure 3. The process is activated on the rising clock edge where it reads from the bus and it reads, executes and writes to the bus in one clock cycle. Just before the rising edge of the clock, all signals are propagated on all buses which means, that all communication happens simultaneously. Because of this structure, if a value is written by a process in cycle i, it is read by the receiving process in cycle $i + 1$.

SME is able to detect read/write conflicts where multiple writes are performed to a single bus within the same clock cycle as well as reads from a signal that has not been written to in the previous clock-cycle.

Since SME is based on CSP, all SME models have a corresponding CSP model, and because

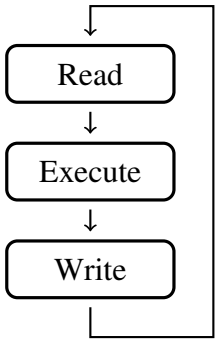


Figure 3. SME process flow for one clock cycle.

of this property, we are able to create a transpiler translating SME models to CSP_M. The SME model is currently implemented as libraries for the general-purpose languages C# [9], C++ [13], and Python [14]. The Python and C# libraries both have code generators for VHDL as well.

```
proc addone (in inbus)
    bus outbus {
        val: int;
    };
{
    outbus.val = inbus.val + 1;
}

    ⋮

network net() {
    instance a of addone(b.outbus);
    instance b of ..
    ⋮
}
```

Listing 1. Small example of process and network syntax in SMEIL.

1.2. SMEIL

With the different SME implementations, a need arose for a common intermediate language. SMEIL was developed as a Domain Specific Language (DSL) for SME, usable both as an IL and as an independent implementation language. It has a C-like syntax with a type system that makes hardware modeling simple. In spite of its simplicity, SMEIL still provides hardware-specific functionality that is more difficult to create with general-purpose languages. Often when modeling hardware in Hardware Description Languages (HDLs) like VHDL or Verilog, code for testing and verifying are often written in the same language as the design itself. Unfortunately, the HDLs often does not have the functionality for generating proper simulation input. Using general-purpose languages for testing hardware models are useful since the range of available libraries are much larger. Therefore the SMEIL simulator provides a simple language-independent API which enables SME implementations written for general-purpose languages to communicate with SME networks written in SMEIL, so-called co-simulation.

The two fundamental components of an SMEIL program is `process` and `network`. The process consists of variable and bus definitions, as well as the statements that are evaluated once for each clock cycle. The purpose of the `network` declaration is to define the relations between each entity in the program. A small example of process and network syntax can be seen in Listing 1.

There are several different ways to use SMEIL, one being co-simulation as described above. However, in this work, we focus on the independent SMEIL representation and thus we only present examples in pure SMEIL. These pure SMEIL programs must contain a process which generates input for the network since the network cannot receive input elsewhere. The program is simulated using the command line tool. Simulation is done in order to test the design of the system.

During the simulation, ranges for all observed values are captured so the observed values and types can be used to constrain the original defined types and ranges. This property is of great value when translating into CSP_M, and when creating assertions, since we can use these values to actually assert the network. The number of clock cycles, that the simulation is run for, is specified by the programmer via the command line tool. If the simulation is not passing through enough clock cycles, the verification might be inadequate. Since the verification builds on the observed values, the simulation needs to be long enough such that the whole possible range of input values is exhausted.

In Figure 4 the SMEIL transpiler structure can be seen.

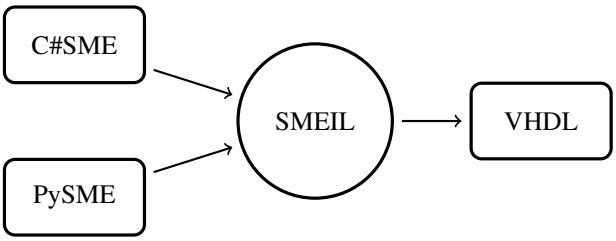


Figure 4. SMEIL transpiler structure.

2. Seven Segment Display Clock in SMEIL

In order to explain how we can transpile programs from SMEIL to CSP$_M$, we have designed an example using a seven segment display clock. In this section, the seven segment display example will be explained as well as the SMEIL implementation of the network.

A seven segment display is an electronic display device which is used in displays such as digital clocks or other types of devices that display numerals. An example of a typical digital clock display can be seen in Figure 5. When a digit has been determined for a seven segment display, it is encoded to a bitstream that represents the digit in the correctly activated display segments. In this example, we wish to model a typical digital clock that is able to calculate

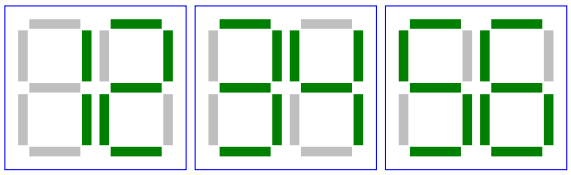


Figure 5. Digital clock with six seven segment displays, displaying 12:34:56.

and display the current time in hours, minutes, and seconds. Listing 2 shows this example written in Python. When creating this model in SMEIL some input must be added to the network, just like `time_since_midnight` in Listing 2. The input value represents seconds since midnight, and in order to calculate hours, minutes, and seconds we model three different processes, called the `time` processes in this example.

When writing hardware models in pure SMEIL, the only way to generate input for the network is to create a data generator process. This process, called the `clock` process in our example, is instantiated with the start time and is incremented by 1 for each simulation cycle, representing a one second increase. The result is communicated on the process output bus, where the three `time` processes are listening. These `time` processes receive the number and by the use of simple integer arithmetic, calculate the hours, minutes, and seconds since midnight respectively. It is obvious that at some point in time, each `time` process will calculate a two-digit result, for example at 12 hours or 42 seconds. However, a single seven segment display can only show one digit between 0 and 9. Therefore we need two seven segment displays for each `time` process in order to show the correct time in a 24-hour interval. Each `time` process has an output bus with two individual channels that represent the communication to each different display. The number representing either hours, minutes, or seconds are separated into first and second digit, by $\lfloor \frac{x}{10} \rfloor$ and $(x \bmod 10)$. These six different results are then communicated onto the six different channels which represent the six different seven segment displays. The outline of this network can be seen in Figure 6.

In Figure 6 the network consists of four processes, the data generator process, *I*, which creates the input that is broadcasted out on the network. The three `time` processes, hours (*H*),

```python
from math import floor

def time(time_since_midnight):
    hours   = floor(time_since_midnight / 3600)
    minutes = floor((time_since_midnight - hours * 3600) / 60)
    seconds = time_since_midnight - hours * 3600 - minutes * 60
    return [hours, minutes, seconds]

print(time( 57100)) # =>  15:51:40
print(time(  3601)) # =>  01:00:01
print(time( 66666)) # =>  18:31:06
```

Listing 2. A Python implementation of the seven segment display example.

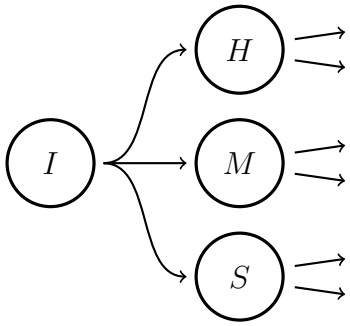


Figure 6. SMEIL network for a seven segment display clock. Each SMEIL process is represented by a cicle with a letter corresponding to the processes Input, Hours, Minutes and Seconds respectively.

minutes (M), and seconds (S) are the processes described above, which calculate each part of the current time. The outputs are communicated on the six outgoing channels.

The full SMEIL code for this example can be seen in Listing 7 in the appendix.

3. Supporting Technologies

3.1. FDR4

We not only want to transpile SMEIL to CSP$_M$, we also want to be able to verify different properties in CSP$_M$ in order to prove correctness. Today, there exists several tools for formal verification, both in academia and in the industry. One of the currently most favored tools is the Failures-Divergences Refinement tool (FDR4). This tool is a CSP refinement checker that can analyze programs written in the machine-readable version of CSP; CSP$_M$. It provides a parallel refinement-checking engine that can scale up linearly with the number of cores. This means that it can handle processes with a large number of states in a reasonable time. FDR4 can handle several different types of assertions, deadlocks being the most used. However, due to the structure of SMEIL, we use FDR4 in a different way than is typical. Since the SME model cannot have cyclic-wait we have no need to verify the system in this manner.

For our current implementation of the transpiler, we can assert the ranges of the channel inputs, for example, we can automatically assert that the observed ranges, provided by the SMEIL simulation, and the possible input on the CSP$_M$ channels are not conflicting. In hardware, we would typically want to verify that the communication on a bus does not exceed a certain range or that the sum of multiple signals does not exceed a specific value. A bus might be able to carry other data than needed, and being able to model a circuit that can assert that the bus never carries other data than expected, is of great value.

CSP was not initially developed for hardware modeling, and therefore it is not evident how to handle the clock cycle, which is an essential part of hardware modeling. When we transpile the SME network into CSP_M the SMEIL simulation have provided the ranges of all values from the simulation and therefore all clock cycles. This means that when FDR4 asserts a property it asserts on all possible communication combinations for all the simulated clock cycles. Therefore, even though we are transpiling from an SME model, where the clock is crucial, we can simply translate "one-to-one" from the SMEIL program and still get an accurate assertion on the properties.

3.2. Transpiling SMEIL to CSP_M

When transpiling from SMEIL to CSP_M one of the difficult components was to find a generalized method for transpiling, that could be generalized to most problems. We have worked on separation of concerns in order to simplify, but also have a greater chance of being able to match more SMEIL programs.

An SMEIL process consists of bus and variable declarations, the statements to be run per clock cycle as well as the outgoing communication from the process. Channels within an SMEIL bus can be translated directly to CSP_M channels. It is, however, important to give channel names that will be unique since a CSP_M channel is global as opposed to the local channel within each SMEIL bus. An example of an SMEIL process, where the process structure is evident, can be seen in Listing 3 and the corresponding CSP_M code in Listing 4.

In order to keep the outwards communication and the arithmetic statements together within each process in CSP_M, we generate CSP_M processes with a `let within` statement. The arithmetic statements go into the `let` section and the communications go into the `within` section. This gives us the possibility of separating the outwards communication and arithmetic statements while still keeping them within the same CSP_M process. In Listing 4, an example of the `let within` statement can be seen in lines 7-14. This structure will work as a general translation structure from SMEIL processes to CSP_M processes.

The network in an SMEIL program is the crucial part which ties all the processes and communication together. We can standardize the network generation by creating a two-step communication part. Instead of having the actual processes receive the incoming data, they receive the data by their process parameter. The process parameter is then set by the network process which receives the communication from the channels and provides the process with the communicated value. This ensures that we can generate the processes easily without having to traverse the network in the SMEIL program beforehand to find out which channel provides input for which process. An example of this is shown in Listing 8 in the appendix on lines 61 to 66.

4. Seven Segment Display Clock Transpiling

In the following we use a classic hardware design to illustrate each of the steps in the transpiling, and how the types, constraints, and assertions are carried from the original SMEIL program into the CSP_M program.

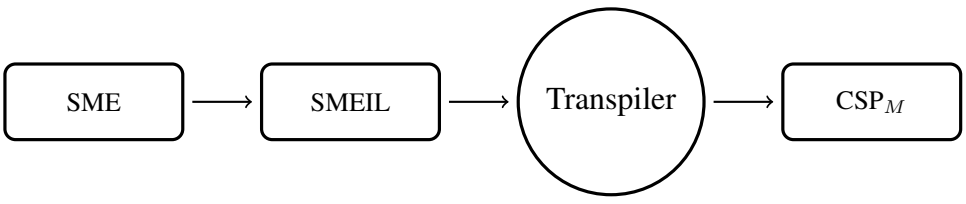


Figure 7. SME to CSP_M transpiler.

We wish to model the network presented in Section 2 in SMEIL in order to transpile it to CSP_M so that we may verify properties in FDR4. In Figure 7 the workflow of this system can be seen.

Even though SME buses can contain a series of channels, every single channel is translated into a CSP_M channel. The properties we will assert with FDR4, are the width of the CSP_M channels. That is, we want to prove that certain values will never be communicated on certain channels. It is easy to imagine that 4 bits can be communicated between the `time` processes and the seven segment displays. But 4 bits can represent the numbers 0 through 15, and our seven segment displays can only display the numbers 0 through 9. Therefore we wish to assert that even though the channels can carry 4 bits, the actual communication on the six output channels does not exceed 9. In general, the displays will be able to display 0 through 9, but since the example is a clock showing a 24-hour interval, the displays will of course not be able to show minutes and seconds above 59 and hours above 23.

We know that a program in pure SMEIL must have a data generation process, but this is not the case in a CSP network. Since we are only transpiling from pure SMEIL networks, we can be certain that there will always be a process which just contributes an initial value to the rest of the network. We also know that a process must either have communication in or out or both. Therefore, we can assume that all SMEIL processes with no input bus will be a data generator process of some kind, and therefore must have some outwards communication. So when transpiling to CSP_M, we do not translate the SMEIL process to a CSP_M process, but simply create a CSP_M channel that represents the values communicated out of this SMEIL process.

We assume that the SMEIL programs we transpile only contains channels with types and range annotations. During the simulation, the type will be restricted to the lowest representation possible. For example, if a channel was originally set to be `int` (unbounded), but the observed values from the simulation show that it could be changed to an `i8` (signed 8-bit integer with a range of -128 to 127), then the simulated output would be `i8`.

When creating channels in CSP_M, we need to define its range of possible values. If a channel is only defined by having the integer type, FDR4 would try to verify for all possible integers, which results in a seemingly unbounded runtime. As explained in Section 1.2, all simulated SMEIL programs will include the observed range and restricted types for all channels and variables. The types represent the observed width of the channels in bits, and by calculating the possible range from these types, we can create the corresponding channels in CSP_M, and thereby avoid having a seemingly endless runtime in FDR4.

Since the assertion we wish to make is to verify the widths of the channels, it might seem redundant to create CSP_M channels with a limited range. FDR4 would always only check the values in the defined channel range and therefore there is no point in asserting if the values go beyond this range. After simulating the SME network, SMEIL provides us with both a type and a range of observed values. The type is used to create the CSP_M channel range and the observed values are used for the assertion. The type will always represent equal or more values than the range of observed values, and by using these values the assertions becomes valuable.

When it comes to transpiling the data generator process into a CSP_M channel, we also use the types of the SMEIL simulation to define it. We use this instead of the observed values because we cannot guarantee the precise input values of the system. If we used the observed values, the assertions will pass every time, since it will test the values already used to generate the rest of the observed values.

An example of simulated SMEIL code can be seen in Listing 3. Notice on lines 2 and 3 that the two channels are defined both with a type `u3` and `u4` and with a range 0 to 5 and

```
1  proc seconds (in seconds_in)
2      bus seconds_out {first_digit: u3 range 0 to 5;
3                       second_digit: u4 range 0 to 9;};
4      var seconds: u6 range 1 to 59;
5      var seconds_first_temp: u3 range 0 to 5;
6      var seconds_second_temp: u4 range 0 to 9;
7  {
8      seconds = seconds_in.val % 60;
9      seconds_first_temp = seconds / 10;
10     seconds_second_temp = seconds % 10;
11     seconds_out.first_digit = seconds_first_temp;
12     seconds_out.second_digit = seconds_second_temp;
13 }
```

Listing 3. Example of the `seconds` process from the SMEIL seven segment display example. See full example in Listing 7 in the appendix.

```
1  channel seconds_out_first_digit : {0..7}
2  channel seconds_out_second_digit : {0..15}
3
4      ⋮
5
6  Seconds(seconds_in) =
7  let
8      seconds = seconds_in % 60
9      seconds_first_temp = seconds / 10
10     seconds_second_temp = seconds % 10
11 within
12     seconds_out_first_digit ! seconds_first_temp ->
13     seconds_out_second_digit ! seconds_second_temp ->
14     SKIP
```

Listing 4. Example of the `Seconds` process from the generated CSP_M code in the seven segment display example. See full example in Listing 8 in the appendix.

0 to 9. These are the observed types and value ranges the simulation tracked for the specific channel. In order to create the CSP_M channels based on the types, we need to convert `u3` and `u4` into its corresponding range, which for `u3` is 0 through 7 and for `u4` is 0 through 15. In Listing 4 on lines 1 and 2, the calculated ranges are used to define the CSP_M channels.

When creating the assertions, we decided to create separate assert functions to keep the code structure clean. We know that for each CSP_M channel there must be an assertion, except for the input channel. Consequently, we create a *monitor* process for each channel and its only job is to listen in on the channel communication and assert the values communicated there. The monitor process is a process that we add specifically for asserting legal communication values in FDR4 and it does not affect the original SME network. In Figure 8 the outline of this kind of structure can be seen and we expect that this structure can be used for several different types of problems and thereby ensure a cleaner code structure.

The monitor process asserts the observed values of the CSP_M channels and in Listing 5 the two monitor processes for the Seconds `time` process can be seen. The values used for these statements are the observed values from the SMEIL simulation, as can be seen at the end of lines 2 and 3 in Listing 3. In Listing 5 the ranges are used to assert that the only values communicated on the channels are within 0 and 5, and 0 and 9 respectively.

After translating the SMEIL processes and creating the monitor processes, we need to

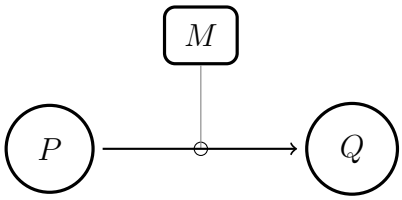


Figure 8. The monitor process M listens in on the communication between P and Q in order to assert the communicated values.

```
1 Seconds_out_first_digit_monitor(c) =
2     c ? x -> if 0 <= x and x <= 5 then SKIP else STOP
3 Seconds_out_second_digit_monitor(c) =
4     c ? x -> if 0 <= x and x <= 9 then SKIP else STOP
```

Listing 5. Example of the `Seconds` monitor processes from the generated CSP$_M$ code in the seven segment display example. See full example in Listing 8 in the appendix.

create the network described in the last part of the SMEIL program, see lines 53 to 59 in Listing 7 in the appendix. We wish only to assert the values the `time` processes are communicating to the monitor processes, and therefore we have to synchronize these processes into a single network in CSP$_M$. We create three network processes, one for each part of the network, and we create a nested synchronization, in order to have all monitor processes synchronized with the `time` process. An example of this network can be seen in on lines 61 to 66 in Listing 8 in the appendix. This network process is also the process that receives the input from the input channel. By not adding the receiving communication in the `time` processes, we avoid having to specify the name of the input channels before creating the network which simplifies the translation, as described in Section 3.2. In SMEIL, this information is part of the `network` section, and therefore it fits well within this part of the CSP$_M$ code.

After creating the network we add the actual assert function calls. For these kinds of assertions, where we want to check a range, the best solution is to assert that the network processes behave as the `SKIP` process. This is done by having the monitor process running the `SKIP` process if the value is within the range and the `STOP` process if not. Two examples can be seen in lines 2 and 4 in Listing 5. We assert this by using the FDR4 failures model on the the `SKIP` process along with hiding communication events, which can be seen in lines 68, 78 and 88 in Listing 8 in the appendix.

The different parts of transpiling the seven segment display example have been presented and in Figure 9 the corresponding network of the CSP$_M$ system is presented. The corresponding network in CSP$_M$ consists of 12 different processes, all created so that not only the network is simulated correctly, but also so the assertions we wish to make, are in place. The input is represented by a triangle, since it transpiles from an SME process to a CSP$_M$ channel. Each of the dotted squares represents the network of synchronizations for each `time` processes, which in itself is a process in CSP$_M$. For each network, we have the `time` processes and two monitor processes, for example, H, M_{H_1} and M_{H_2}.

In order to show that the verification is accurate, the example in Listing 6 contains an error that results in FDR4 failing the verification. In Listing 6 the example is only able to handle an input that is below 24 hours. This is because the calculation in the `Hours` process does not handle the wrap around at the 24$^{\text{th}}$ hour. This means that if the input represents more than 24 hours, the assertions will fail in FDR4 because one seven segment display suddenly has to display two digits instead of one. An example of such could be the input `131071`, which

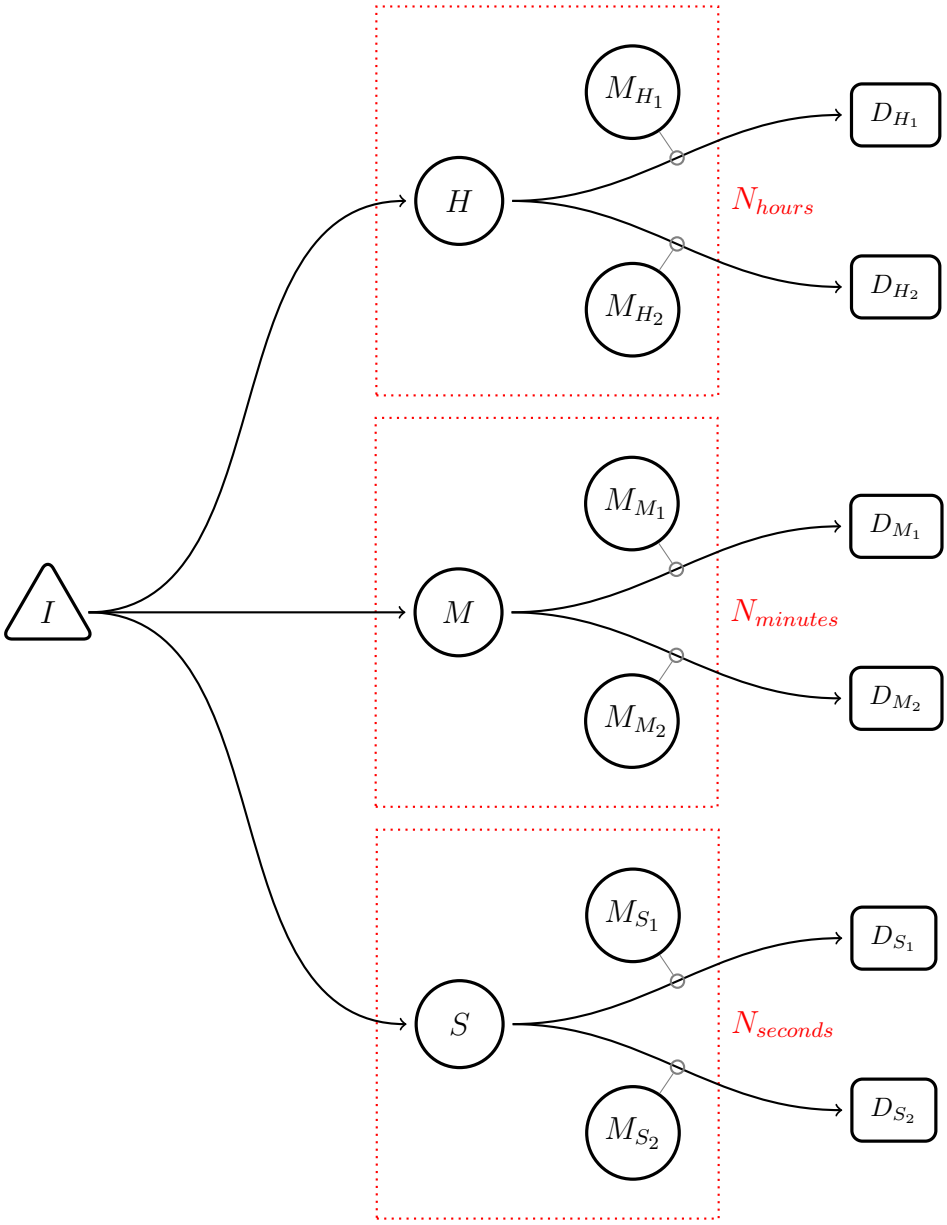


Figure 9. A seven segment display clock network in CSP$_M$. I represents the input channel. N_{hours}, $N_{minutes}$ and $N_{seconds}$ represent the network processes with H, M and S as the `time` processes. The results from the `time` processes are communicated to the displays. The displays are represented by a square since they are not actual CSP$_M$ processes. Each display communication also has a monitor process which assert the legal communication values.

represents 36 hours, 24 minutes and 31 seconds, or 1 day, 12 hours, 24 minutes and 31 seconds. When trying to assert the code from Listing 6 in FDR4, the assertion fails. The counterexample shows that the number 3 is communicated on `hours_out_first_digit`, which is not allowed according to the monitor process on lines 12 and 13 in Listing 6.

This example of failure shows how verifying the solution with a tool like FDR4 actually catches errors that the programmer might have overseen. In this case, the error is simply corrected by adding `% 24` on the end of line 9 in Listing 6 and can be seen corrected in Listing 8 in the appendix at line 15. Now when we try to assert the example in FDR4, it passes. By using modulo on the result, we ensure that we still get the accurate time of day, no matter how many full days the input represents.

The full SMEIL and CSP$_M$ code for the seven segment display example can be seen in Listing 7 and in Listing 8 in the appendix.

```
channel clock_out_val : {0..131071}

channel hours_out_first_digit : {0..3}
channel hours_out_second_digit : {0..15}
    ⋮

Hours(hours_in) =
let
    hours = hours_in / 3600
    ⋮

Hours_out_first_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 2 then SKIP else STOP
Hours_out_second_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 9 then SKIP else STOP
```

Listing 6. Example of an erroneous version of the `Hours` process from the CSP_M seven segment display example seen in Listing 7 and in Listing 8 in the appendix.

5. Future Work

With this work, we have taken a small step towards creating a simpler method for software developers to model hardware as well as verify properties within this model. In future work, we would like to extend this to software-hardware co-design, with which we would be able to assert deadlocks.

It would be desirable to be able to automatically create a human-readable report on the ranges and communications that are used within the system. This could become a standard addition to the documentation of the system, which would give a programmer an easy overview of a complicated system and would also allow for easier contemplation over the system.

Another, more complex idea for future work, is to implement support for multi-channel invariants. This is not something that can easily be simulated and therefore it would require some work, but it would provide the ability to express more complex assertions.

6. Conclusions

We have presented a transpiler that transpiles SME intermediate language (SMEIL) into CSP_M for then to use the Failure-Divergences Refinement tool (FDR4) to assert properties in a CSP_M network. We provide a simple approach that makes it more accessible for software programmers to program hardware and thereby bridging a gap between software programmers and the needs of the industry. Instead of having to create advanced test-benches, our tool provides a simple way to verify the hardware model via FDR4s assertion functionalities. We can assert that the observed values of a channel, in a simulated SMEIL program, are in fact the only possible values communicated on that specific channel. We have also shown this to work in an example case of a seven segment display.

Acknowledgements

Thanks to Uwe Zimmermann who made the seven segment example in TikZ on `http://www.texample.net/tikz/examples/segment-display/`.

References

[1] Vishwani Agrawal and Samuel H. C. Poon. VLSI Design Process. In *Proceedings of the 1985 ACM Thirteenth Annual Conference on Computer Science*, CSC '85, pages 74–78, New York, NY, USA, 1985. ACM.

[2] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[3] Jeremy Kepner. Hpc productivity: An overarching view. *The International Journal of High Performance Computing Applications*, 18(4):393–397, 2004.

[4] Brian Vinter and Kenneth Skovhede. Synchronous message exchange for hardware designs. *Communicating Process Architectures*, pages 201–212, 2014.

[5] Brian Vinter and Kenneth Skovhede. Bus centric synchronous message exchange for hardware designs. *Communicating Process Architectures*, pages 245–257, 2015.

[6] Truls Asheim. SMEIL: A Domain-Specific Language for Synchronous Message Exchange Networks. To be published, 2018.

[7] Bryan Scattergood. *The semantics and implementation of machine-readable CSP*. PhD thesis, 1998.

[8] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[9] Kenneth Skovhede and Brian Vinter. Building hardware from C# models. In *FSP 2016; Third International Workshop on FPGAs for Software Programmers; Proceedings of*, pages 1–9. VDE, 2016.

[10] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-1987*, 1998.

[11] John Markus Bjørndalen, Brian Vinter, and Otto J Anshus. PyCSP-Communicating Sequential Processes for Python. In *Cpa*, pages 229–248, 2007.

[12] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[13] Truls Asheim. Implementing high performance synchronous message exchange, 2015. Bachelor's Thesis.

[14] Truls Asheim, Kenneth Skovhede, and Brian Vinter. VHDL Generation From Python Synchronous Message Exchange Networks. *Proceedings of Communicating Process Architectures 2016*, 2016.

Full SMEIL and CSP$_M$ code

```
proc clock ()
    bus clock_out {val: u17 range 1 to 86401;};
    var i: u17 = 0 range 0 to 86401;
{
    i = i + 1;
    clock_out.val = i;
}

proc hours (in hours_in)
    bus hours_out {first_digit: u2 range 0 to 2;
                   second_digit: u4 range 0 to 9;};
    var hours: u5 range 0 to 23;
    var hours_first_temp: u2 range 0 to 2;
    var hours_second_temp: u4 range 0 to 9;
{
    hours = hours_in.val / 3600 % 24;
    hours_first_temp = hours / 10;
    hours_second_temp = hours % 10;
    hours_out.first_digit = hours_first_temp;
    hours_out.second_digit = hours_second_temp;
}

proc minutes (in minutes_in)
    bus minutes_out {first_digit: u3 range 0 to 5;
                     second_digit: u4 range 0 to 9;};
    var minutes: u6 range 0 to 59;
    var minutes_first_temp: u3 range 0 to 5;
    var minutes_second_temp: u4 range 0 to 9;

{
    minutes = minutes_in.val / 60 % 60;
    minutes_first_temp = minutes / 10;
    minutes_second_temp = minutes % 10;
    minutes_out.first_digit = minutes_first_temp;
    minutes_out.second_digit = minutes_second_temp;
}


proc seconds (in seconds_in)
    bus seconds_out {first_digit: u3 range 0 to 5;
                     second_digit: u4 range 0 to 9;};
    var seconds: u6 range 0 to 59;
    var seconds_first_temp: u3 range 0 to 5;
    var seconds_second_temp: u4 range 0 to 9;
{
    seconds = seconds_in.val % 60;
    seconds_first_temp = seconds / 10;
    seconds_second_temp = seconds % 10;
    seconds_out.first_digit = seconds_first_temp;
    seconds_out.second_digit = seconds_second_temp;
}

network clock_network ()
{
    instance g of clock();
    instance h of hours(g.clock_out);
```

```
    instance m of minutes(g.clock_out);
    instance s of seconds(g.clock_out);
}
```

Listing 7. The full SMEIL code used for transpiling in the seven segment display example.

```
channel clock_out_val : {0..131071}

channel hours_out_first_digit : {0..3}
channel hours_out_second_digit : {0..15}

channel minutes_out_first_digit : {0..7}
channel minutes_out_second_digit : {0..15}

channel seconds_out_first_digit : {0..7}
channel seconds_out_second_digit : {0..15}


Hours(hours_in) =
let
    hours = hours_in / 3600  % 24
    hours_first_temp = hours / 10
    hours_second_temp = hours % 10
within
    hours_out_first_digit ! hours_first_temp ->
    hours_out_second_digit ! hours_second_temp ->
    SKIP

Hours_out_first_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 2 then SKIP else STOP
Hours_out_second_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 9 then SKIP else STOP


Minutes(minutes_in) =
let
    minutes = minutes_in / 60  % 60
    minutes_first_temp = minutes / 10
    minutes_second_temp = minutes % 10
within
    minutes_out_first_digit ! minutes_first_temp ->
    minutes_out_second_digit ! minutes_second_temp ->
    SKIP

Minutes_out_first_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 5 then SKIP else STOP
Minutes_out_second_digit_monitor(c) =
    c ? x -> if 0 <= x and x <= 9 then SKIP else STOP


Seconds(seconds_in) =
let
    seconds = seconds_in % 60
    seconds_first_temp = seconds / 10
    seconds_second_temp = seconds % 10
within
    seconds_out_first_digit ! seconds_first_temp ->
```

```
52      seconds_out_second_digit ! seconds_second_temp ->
53      SKIP
54
55  Seconds_out_first_digit_monitor(c) =
56      c ? x -> if 0 <= x and x <= 5 then SKIP else STOP
57  Seconds_out_second_digit_monitor(c) =
58      c ? x -> if 0 <= x and x <= 9 then SKIP else STOP
59
60
61  N_hours = clock_out_val ? variable ->
62            (Hours(variable)
63            [| {| hours_out_first_digit|} |]
64            Hours_out_first_digit_monitor(hours_out_first_digit))
65            [| {| hours_out_second_digit|} |]
66            Hours_out_second_digit_monitor(hours_out_second_digit)
67
68  assert SKIP [F= N_hours \ Events
69
70
71  N_minutes = clock_out_val ? variable ->
72              (Minutes(variable)
73              [| {| minutes_out_first_digit|} |]
74              Minutes_out_first_digit_monitor(minutes_out_first_digit))
75              [| {| minutes_out_second_digit|} |]
76              Minutes_out_second_digit_monitor(minutes_out_second_digit)
77
78  assert SKIP [F= N_minutes \ Events
79
80
81  N_seconds = clock_out_val ? variable ->
82              (Seconds(variable)
83              [| {| seconds_out_first_digit|} |]
84              Seconds_out_first_digit_monitor(seconds_out_first_digit))
85              [| {| seconds_out_second_digit|} |]
86              Seconds_out_second_digit_monitor(seconds_out_second_digit)
87
88  assert SKIP [F= N_seconds \ Events
```

Listing 8. The full CSP$_M$ code after transpiling the seven segment display example, as seen in Listing 7 in the appendix.