Unifying Framework For Development of Message-Passing Applications

Stanislav Böhm

Ph.D. Thesis

Faculty of Electrical Engineering and Computer Science
VŠB – Technical University of Ostrava

2013

Acknowledgements

First and foremost, I would like to thank my advisor, professor Petr Jančar, for his support and advices and the opportunity he provided to see science from both the theoretical and applied sides. Thank you, I owe you so much.

I would also like to thank Marek Běhálek, Ondřej Meca, and Martin Šurkovský, the coauthors of the project that is the subject of this thesis. Without their work, this dissertation thesis would not reach its current state. I am also very grateful to Martin Šurkovský for various comments on this text.

I would also like to thank all those who helped me during my Ph.D. studies, especially Martin Kot and Zdeněk Sawa. They were always ready to provide useful advice.

My thanks goes to Ondřej Jakl who believed in our project and allows us to use hardware at Institute of Geonics in Ostrava. This access was very helpful for the practical verification of our our ideas.

The tool Kaira, which is the central topic of this thesis, was supported by grants: GAČR P202/11/0340, IT4Innovations Center of Excellence (project CZ.1.05/1.1.00/02.0070), and student grant SP2011/25 prolonged as SP2012/127 and SP2013/145.

Last but not least, I thank Jana for her love and constant support.

Abstract

This thesis focuses on the development of applications for distributed memory systems in the area of scientific and engineering computations. The work addresses the problems inherent to rapid development of parallel applications and the complexity of tools used during their implementation.

Herein, an abstract computation model for describing MPI (Message Passing Interface) applications is proposed. In the described topic of interest, MPI is the de facto API. The proposed approach is based on the usage of visual programming for expressing parallel aspects and communication in a developed application. The intention is not to create a complete program visually; rather, a user writes sequential parts of the application in C++ and puts them into the visual program. This allows simple modification of the communication structure and provides immediate visual feedback of the program's behavior through a visual debugger. An abstract model is also designed in a way that provides a unified view of activities that are used during development, including debugging, performance analyses, and verifications. While tools exist in all of these areas, they are usually single-purpose tools. In the proposed approach, all these activities are roofed by a single conceptual frame; everything from configurations to providing results of analyses uses one set of terms.

The thesis makes the following contributions: It introduces the syntax and semantics of the visual language for describing parallel aspects and communication in applications for distributed memory systems. The semantics of the language is formalized and the usage of the language is demonstrated for various problems. These examples show that the sizes of diagrams describing the visual programs remain reasonably small for many practical parallel applications. Measurements show that the performance of the resulting programs is close to manually created C++ applications. Also, the features of the proposed approach may help in various ways in common activities that occur during development.

The ideas presented in this thesis are implemented as a tool named Kaira (`http://verif.cs.vsb.cz/kaira/`); it is an open source tool released under GPL (GNU General Public License).

Abstrakt

Tato práce se zabývá vývojem aplikací pro systémy s distribuovanou pamětí v oblasti vědecko-technických výpočtů. Práce se zaměřuje na rychlý vývoj paralelních aplikací a integraci nástrojů používaných při vývoji.

Navržený přístup je založen na abstraktním výpočetním modelu a vizuálním programování. Tyto prostředky slouží k vyjádření paralelismu a komunikaci ve vyvíjené aplikaci. Sekvenční části aplikace jsou psány přímo v C++ a tento kód je kombinován s vizuálním modelem. Navržený přístup dovoluje jednoduchou manipulaci s programem a umožňuje poskytnout přímočarou vizualizaci chování aplikace. Abstraktní model je také navržen za účelem sjednocení aktivit, které se objevují při vývoji aplikací: debugování, analýza výkonu, predikce výkonu a verifikace. Pro tyto aktivity existuje mnoho různých etablovaných nástrojů, které jsou mnohem vyzrálejší a nabízejí více funkcí než zde předkládaná implementace. Z pohledu této práce se ale většinou jedná o programy zaměřené pouze na jednu z oblastí. Navrhovaný přístup spojuje všechny tyto aktivity do jednoho myšlenkového rámce s jednotnou terminologií, konfigurací a prezentací výsledků.

Hlavní přínosy této práce spočívají ve vytvoření syntaxe a sémantiky vizuálního jazyka pro popis paralelních částí a komunikace v aplikacích pro systémy s distribuovanou pamětí. Sémantika jazyka je plně formalizována a využití jazyka je demonstrováno na různých příkladech. Tyto příklady ukazují, že diagramy reprezentující program zůstávají relativně malé a přehledné při řešení různých praktických problémů. V této práci je dále ukázáno, že navržený přístup může sjednotit a zjednodušit mnoho různých činností objevujících se v souvislosti s vývojem softwaru v oblasti zájmu této práce. Prezentované myšlenky jsou implementovány v podobě nástroje Kaira (`http://verif.cs.vsb.cz/kaira/`). Kaira je uvolněna jako open-source nástroj pod GPL (GNU General Public License).

Declaration

I declare that I composed this thesis, and I am the original author and leader of the group that works on this project. I have also done most of the work related to technical matters and programming. Some of these results have been previously published in [1, 2, 3, 4, 5, 6, 7, 8].

Contents

List of Algorithms

List of Figures

List of Listings

Chapter 1

Introduction

Parallel computing, or the simultaneous use of multiple processing units, reduces the computational time for solving many scientific and engineering problems. Nowadays, parallel computers are more available and more people can use and develop software for them. However, implementing efficient cooperation among computing units brings many challenges, including synchronization, data placement, and communication. Experimenting with parallel algorithms is also more complicated, as more time may be needed to develop a working prototype. But the complexity lies also in activities that are used during the implementation, such as debugging, performance analysis, or verification; these activities that are referred to as *supportive activities* in this thesis. Even an experienced programmer of sequential applications may spend some time learning how to use tools like parallel debuggers or parallel profilers. They are usually more complicated in comparison to their sequential counterparts.

An important aspect of a parallel computer is its memory model. The two basic memory models are *shared memory* and *distributed memory*. In the former, all computing units may access a global (shared) memory. In the latter, each computing node has its own private memory and they communicate with each other through some kind of network.

The overall topic of this thesis is the reduction of complexity of the development of parallel applications for distributed memory systems in the area of scientific and engineering computations. This thesis proposes a unifying prototyping framework for creating, debugging, performance analyzing, and verifying parallel applications. It proposes an environment in which a user can implement and experiment with his or her ideas in a short time; create a real running program; and verify its performance, scalability, and correctness.

There are many approaches aimed at simplifying the parallel programming: automatic paralleling compilers, programming languages with special constructs, or different paradigms to express parallelism. From the perspective of these tools, the

proposed approach is on a lower level of abstraction. The goal is to provide an environment for simple experimentation with various algorithms; for this reason, enabling the user to have good control over the results is important. At the same time, it is also important to maintain a program in a form that is easy to modify; therefore, some low-level parts of the implementation are hidden and auto-generated.

One important aspect of the proposed approach is that it also provides a platform that integrates supportive activities. Tools exist for many different varieties of these supportive activities; they are mostly on a higher level of maturity and provide more functions in their areas of interest in comparison to the implementation of the tool proposed in this work. These tools, while being very diverse, usually focus only on a single area. In the framework introduced with this work, all of the activities are presented in one way. Once the user knows the basics, all these features may be easily used without learning a completely new tool for each supportive activity. A second important aspect of the proposed approach is the interoperability among all the activities.

Let us consider the following scenario: We are developing an implementation of a load-balancing algorithm and want to inspect how this application behaves during real runs. But we also want to verify the performance and the correctness in general; hence, we want to observe a simulated run of the application on a virtual computer with some very special performance characteristics to verify corner cases of the algorithm. A natural wish is to able to compare and analyze all these data together. If an anomaly is discovered in these analyses, we would want to see the situation in a debugger and observe the program's run in step-by-step detail.

The user usually needs at least three tools to accomplish goals in this scenario. Also notable is that it can be hard to reuse results from one tool in another tool if the used tools do not understand each other. In particular, getting the developed application under control of a debugger into an exact state obtained from analysis of another tool can be hard. In the proposed approach, this scenario is achievable and all activities are presented in a unified way.

The contributions of this thesis can be summarized as follows: It introduces the syntax and semantics of the visual language for describing parallel aspects and communication in applications for distributed memory systems. The semantics of the language is formalized and the language is used with various problems. These examples show that sizes of diagrams describing the visual programs remain reasonably small for many practical parallel applications. The performance of resulting applications is close to manually created C++ applications. The language allows easy modifications of programs and also (together with the visual simulator) immediate visual feedback about the behavior of the developed program. The proposed model is also designed in a way that unifies and simplifies many supportive tasks that occur during development.

The proposed ideas are implemented in Kaira[1]. It is an open source project released under *GNU General Public License*[2]. The author of the thesis is the original author, the leader of the team developing Kaira, and also its main programmer.

The content of this thesis is organized in the following way: In Section 2, the state of the art is discussed. In Chapter 3, Kaira is introduced; Chapter 4 provides examples of programs developed in Kaira. The formal semantics of Kaira is given in Chapter 5. Chapter 6 describes features of Kaira together with demonstration through examples. In Chapter 7, the basic internal structure of Kaira is introduced. The thesis concludes with Chapter 8. In all chapters, it is assumed that the reader is familiar with C++ and basic principles of parallel programming. The formal semantics given in Chapter 5 may be too technical for some readers; it can be skipped, and most of Chapter 6 should remain understandable.

[1] `http://verif.cs.vsb.cz/kaira/`

[2] `http://www.gnu.org/licenses/gpl.html`

Chapter 2

State of the art

This chapter presents an overview of tools and technologies used in the area of programming applications for distributed memory systems. Some of these tools will be compared with KAIRA in Section 3.5. This chapter starts by introducing *Message Passing Interface* (MPI). It has become a de facto standard Application Programming Interface (API) for computations on distributed memory systems. The following text describes the state of the art of programming, debugging, analyzing performance, predicting performance, and verifying applications that use MPI. Some other tools related to programming of distributed memory systems are introduced in Section 2.2. Next, some works in the area of visual programming and Petri nets are mentioned because visual programming plays an important role in KAIRA and Petri nets were the original source of inspiration when KAIRA was designed.

2.1 Message Passing Interface

MPI[9] is a specification of a portable message-passing system designed to work on a wide variety of parallel computers. It is a specification of a library that offers message-passing related functions for C and FORTRAN. There are several implementations of MPI, for example OPENMPI[1], MPICH[2], and LAM[3]. MPI is standardized by MPI Forum[4]; it is an open group of all interested parties (computer vendors, authors of implementations, researchers, etc.). The MPI standard is available for free.

Roughly speaking, MPI is an interface for sending and receiving messages between processes. Listing 2.1 demonstrates a basic usage of MPI. When this program is started, several processes start to simultaneously perform the `main` function in

[1] `http://www.open-mpi.org/`
[2] `http://www.mcs.anl.gov/research/projects/mpich2/`
[3] `http://www.lam-mpi.org/`
[4] `http://www.mpi-forum.org/`

Listing 2.1: A simple MPI program

```
#include <mpi.h>
int main(int argc, char **argv) {
    MPI_Init (&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if(myrank == 0){
        int data[10];
        fill_data_with_something(data);
        // send 10 bytes to process 1 with tag 1
        MPI_Send(data, 10, MPI_BYTE, 1, 1, MPI_COMM_WORLD)
    }
    if(myrank == 1){
        int data[10];
        // receive 10 bytes from anyone with any tag
        MPI_Recv(data, MPI_BYTE, 10,
            MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COM_WORLD)
        do_something(data);
    }
    MPI_Finalize();
    return 0;
}
```

independent memory spaces. The only way to communicate between processes is through sending messages via MPI interface.

In this example, it is assumed that the program is started with at least two processes. Each process at the beginning determines its *rank*. The rank is the identification number of the process; ranks are numbered from zero. The first process (process 0) fills the memory for a message and sends the message to process 1. Process 1 waits for any message and then processes the received message. MPI provides not only `MPI_Send` and `MPI_Recv`, but a variety of functions in areas related to message passing, including: nonblocking operations (it allows communication and computations to overlap); collective communication (more processes are involved to efficiently perform operations like broadcast or scatter/gather).

The brief history of MPI (more details in [10]):

- 1992 – The preliminary draft of MPI was proposed as the first step to create a portable message passing library specification that adopts features from existing, but vendor-specific solutions. MPI Forum was founded.
- 1993 – The draft of MPI-1 was presented.

- 1994 – MPI-1 was released.
- 1995 – MPI-1.1 corrected some minor errors. The work on MPI-2 started; the main topics were: parallel I/O, dynamic processes, and extensions to collective operations.
- 1997 – MPI-2 was published.
- 2008 – MPI-1 and MPI-2 were consolidated into a single document as MPI-2.1.
- 2009 – MPI-2.2 was published. It solved some scalability issues and added some missing functionalities.
- 2012 – MPI-3 was published. Major new functionalities: non-blocking collectives, sparse collectives, new one-sided communication.

2.1.1 Debugging

Finding bugs is a common activity during the implementation of an application; therefore *debuggers* are vital supportive tools. These tools allow inspection of the behavior of an application while tracking down a bug. The main task of a classic interactive debugger is to show the inner state of the application and to allow control of the application's flow. Besides interactive approaches, automatic debuggers also exist that monitor a run of the application and detect misuse of resources without any user intervention. An overview of some debugging techniques in the MPI environment is presented in [11, 12]. Documentations of MPI implementations are another useful source of information[5].

The most direct way of debugging an MPI application is to use standard tools for sequential programming, because an MPI application runs on each computing node like a sequential program. Tools like GDB[6] (an interactive debugger) or Valgrind[7] (an automatic debugger) can be named as examples. This approach is sufficient to find many types of bugs, but the major disadvantage is that instances of the tool are completely separated for each MPI process. It is not easy to work with and understand more simultaneously running instances of these debuggers.

To overcome this issue, there are specialized debuggers. They provide the same functionality like ordinary debuggers (showing stack traces, placing breakpoints, adding memory watches), but they allow debugging of a distributed application as a single piece. The user can debug an MPI application in the standard way even if

[5] `http://www.open-mpi.org/faq/?category=debugging`
[6] `http://www.gnu.org/software/gdb/`
[7] `http://valgrind.org/`

the application runs on several independent computers. Distributed Debugging Tool[8] or TotalView[9] are a couple of the most well-known tools in this category.

Besides these tools, there are also non-interactive tools like MPE[10] or Padb[11]. MPE provides additional features over MPI, including the display of traces of MPI calls or real-time animations of communication. Padb helps with gathering stack traces and job monitoring.

There are also automatic debugging tools that are specialized for use with MPI: Marmot [13], MPI-CHECK [14], or Umpire [15]. They monitor a running application and detect issues like incorrect use of MPI calls, reaching a deadlock, or mismatched collective operations.

2.1.2 Performance analysis

The fundamental goal of parallel computing is to provide computation power for problems that are not solvable on a single computer in a reasonable time. Therefore, performance analysis plays a key role in the development of parallel applications. The primary question answered by such analysis is: *Where is a bottleneck of the program, and why is it the bottleneck?* The analysis should reveal to the programmer what needs to be improved to obtain better application performance. The term "performance analysis" will be used for the rest of the thesis for analyses of real runs of applications on real computers. It is the most common way of measuring performance. The other kinds of analyses will be covered in the next section.

Tools for performance analyses can be categorized into two groups by the method of gathering results: *instrumentation* and *statistical* tools. The results from both types of tools are usually provided as overall characteristics of a run, e.g. computation times of program's parts or communication costs. Some more sophisticated tools can also point to suspicious places of a program's run.

As with debugging, sequential tools can be used to analyze performance of MPI applications. However, the use of sequential tools brings similar problems; namely, measurements are performed separately for each MPI process. Tools Gprof[12] [16] and Callgrind[13] [17] can be named as examples of sequential instrumentation tools. The instrumentation means adding an extra measuring code into the measured application. These tools instrument an application with the assistance of the compiler (Gprof) or directly through its machine code (Callgrind). The aforementioned tools provide statistical summaries (*profiles*) of the application's run in the form of call times and frequencies for each function. The approach of generating

[8] `http://www.allinea.com/products/ddt/`
[9] `http://www.roguewave.com/`
[10] `http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/`
[11] `http://padb.pittman.org.uk/`
[12] `http://sourceware.org/binutils/docs/gprof/`
[13] `http://valgrind.org/docs/manual/cl-manual.html`

profiles can be extended into the environment of MPI. It is implemented by PGPROF[14] or mpiP[15]. These tools are prepared to work in the distributed environment; hence, they analyze the application as one piece, not as separate MPI processes. Moreover these tools understand MPI; they provide statistics of MPI calls and communication.

The profile is not always as useful for parallel applications as it is for sequential applications because of communications costs, waiting times, synchronization, and so forth. Therefore many analytical tools for parallel programs record a *trace* of an application's run where important events are stored with time stamps. The trace allows more precise reconstruction and analysis of an application's behavior. Scalasca [18, 19] and TAU [20] can be named as examples of tracing tools. The drawback of this approach is the introduction of greater overhead in comparison with gathering a profile. Moreover, a trace grows with the length of a program's run and its size can be a major issue. For viewing traces, there are specialized tools for trace visualizations: Vampir [21], Paraver [22], Pajé [23], or Jumpshot[16].

In the case of the statistical approach, a tool does not operate during the whole run of an application. It probes the application in regular time intervals. This approach is much less intrusive than the instrumentation approach. On the other hand, the results are less accurate and only a profile can be obtained. HPCToolkit[17] [24] can be named as a tool using this approach.

The performance analysis techniques are reviewed in broader detail in [25].

2.1.3 Performance prediction

The performance analysis in the previous section allows us to inspect performance of already implemented programs on existing computers. The performance prediction is used when at least one of these components is missing or not fully operational, or if a generic description of an algorithm's performance is needed. The prediction helps to answer questions like: *How will an algorithm scale before its actual implementation? Is it worth it to implement some optimizations?* Or, *how will an application behave on a platform that is not yet available?*

There are two major approaches to performance prediction. The *analytical approach* (for example [26]) consists of a handmade formal analysis of an algorithm. The result is often given as a formula describing how the computational time depends on the characteristics of a given computer. The analytical approach is out of the scope of this thesis and the more automatic approach will be considered – predictions by *simulations*. Tools offering simulations fall into two categories: *online*

[14] `http://www.pgroup.com/products/pgprof.htm`
[15] `http://mpip.sourceforge.net/`
[16] `http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/`
[17] `http://hpctoolkit.org/`

simulators and *offline simulators*.

An online simulator directly executes the application and mimics the behavior of the target platform. It is implemented in tools BigSim [27] and SimGrid [28]. Because of the direct execution, the major challenge is to reduce the demands on the CPU and memory. These tools allow skipping of some computations and can simulate delays that would be caused by executing the computations. It works only for an application with a data-independent behavior, because some parts of the data are not really computed. Many practical applications satisfy this condition and this approach can provide good predictions for them in a short time.

Another important aspect is the complexity of the network simulation. The most precise method for network simulation is a packet-level simulation (MPI-NetSim [29]) but it can be very resource-consuming. The other way, used in most simulators, is to use a simple analytical model, for example SimGrid or BigNetSim (the network simulator for BigSim). Results of a simulation are often provided in the form of a trace; therefore, existing tools for displaying traces can be used.

Offline simulators ([30, 31, 32]) use a trace of an application's run as the input instead of the application itself. The tracelog is replayed in conditions of the simulated platform to obtain predictions. The structure of communication is replayed as was recorded in the trace; computing and waiting times are modified according to the target platform. This way provides predictions while using fewer computations than online simulators, but the problems occur for applications where the structure of the communication is not fixed. In such cases, because of the different order of message arrivals, the program can get to different states and send different messages than recorded. The reader can find more detailed surveys about performance prediction tools in [33, 34].

2.1.4 Verification

The *formal software verification* is the process of proving the correctness of a software system with respect to its specification. All possible behaviors of an application are considered, allowing the exclusion of some kinds of bugs. This can be contrasted with software testing, in which the presence of a bug can be shown, but the converse cannot be proven.

It is well known that even the halting problem for sequential programs is undecidable and many decidable verification problems involve exponential blow-ups or even worse complexities. Therefore there are many limitations to the questions that can be answered in practice. In the case of a verification of an MPI application, the typical questions are the presence of a deadlock or the existence of a run that violates some assertions. These questions are answered with assumptions that input is fixed and all runs are finite.

One option for the verification of MPI applications is MPI-Spin [35], which is

built upon the well-known verification framework Spin [36]. Spin verifies models created in the special programming language. Even this language is based on C; MPI applications cannot be verified directly and the user has to create a model of the application.

To overcome difficulties connected with creating and maintaining a model, tool ISP [37] operates directly on the MPI application. But working directly on C++ codes brings new issues, because it is nontrivial to fully describe a state of the application. Hence ISP (in contrast to Spin) offers only basic analyses.

Both tools are based on the state-space analysis, i.e. a systematic exhaustive exploration of all execution paths that a system might take in order to verify desired properties. Because the state-space grows exponentially, there are state spaces reduction techniques exploiting symmetries in state spaces that are introduced by independent behaviors of parallel processes. Both mentioned tools involve *partial-order reduction* techniques [38]. For example, it allows ISP to search just a single representative path to exclude a deadlock for an application that uses only `MPI_Send` and `MPI_Recv` with explicit ranks (i.e. no receive with `MPI_ANY_SOURCE`).

The community around formal verification methods for MPI applications is much smaller than communities developing other supportive tools. The reasons for the low level of use of formal verification in MPI (and proposals for how to change this) are discussed in [39].

2.2 High-level tools

In the previous section, we have assumed development of applications in C++ (or Fortran) that directly use MPI. But MPI itself is a quite low-level interface from the view of application programmers. Sometimes it is called the "assembler" for distributed memory computations [40].

The simplest way to obtain a higher-level MPI programming environment is to use a higher-level language. The standard of MPI defines API only for C and Fortran, but there are MPI libraries for every major programming language. These libraries work like a bridge that makes C functions accessible from an implementation of MPI to the given language. The following tools can be considered as possible candidates of high-level environments for developing applications that use MPI: Python[18] – a high-level programming language with existing libraries for many areas. Octave[19] – a high-level language for numerical computations. Octave will be discussed more in Section 6.6.2. R[20] – a tool for statistical computing and graphics.

[18] `http://www.python.org/` (MPI for Python `http://mpi4py.scipy.org/`)

[19] `http://www.gnu.org/software/octave/`

[20] `http://www.r-project.org/` (MPI for R: `http://www.stats.uwo.ca/faculty/yu/Rmpi/`)

Each of these tools provides a good high-level prototyping environment where the user does not have to solve many low-level issues. On the other hand, this approach has two basic issues: the performance of applications written in these high-level languages is not comparable with C or FORTRAN for many problems. The second issue lies in supportive tools. Almost all tools named in the previous sections are not easily applicable to this kind of usage, and even basic debugging can be nontrivial.

A different approach would be to use a domain-specific library that uses MPI as its back end. For example, there are PETSc [41] or TRILINOS [42] in the area of problems modeled by partial differential equations. When a user wants to use standard algorithms from this area, these libraries provide tuned algorithms without a need of touching MPI interface. These tuned algorithms are often the best choice when the application can be composed of standard components and it is clear how to implement it. These libraries, however usually do not serve well as generic experimentation and prototyping tools. It can also be difficult to interpret results from generic supportive MPI tools; the user possibilities of debugging and performance analyses therefore strongly depend on the library.

Leaving the paradigm of message passing is another option for obtaining a higher-level approach. MAPREDUCE [43] is one potential option. It is a simple concept of data processing that can be seen as some kind of MPI collective operations with user-defined operations. The paper [44] compares MAPREDUCE with MPI as follows:

> "In general, MapReduce is suitable for non-iterative algorithms where nodes require little data exchange to proceed (non-iterative and independent); MPI is appropriate for iterative algorithms where nodes require data exchange to proceed (iterative and dependent)."

This paradigm was popularized by Google; they state that 80% of their data processing is done by MAPREDUCE [45]. Besides Google's implementation, other implementations have emerged, for example HADOOP[21].

Another approach is *stream processing* [46]. The computation can be seen as a set of components acyclically connected via unidirectional "streams". Each component reads from its input streams and puts data to its output streams – there is no other way of communication. The user describes the computation by components and their connection; the management of components, streams, and their mapping to hardware is left on the tool.

Programming languages based on *Partitioned Global Address Space* (UPC [47], CHAPEL [48]) provide another group of tools. These languages primarily address the problem of combining the shared memory and the distributed memory model. They usually introduce high-level constructions to the programming language and they are able to generate applications that use various communication libraries and

[21] `http://hadoop.apache.org/`

one of them is also MPI. The languages then can also be seen as part of a high-level approach for creating MPI applications.

2.3 Visual parallel programming

Visual programming is another high-level programming approach. In 1990's, there were many tools that combined visual programming with parallel programming. In the paper [49], twelve such tools are listed. They were mainly based on message passing library PVM, one of the predecessors of MPI. The differences between PVM and MPI are summarized in [10].

To the best knowledge of the author, these tools have not been developed in recent years, and the tools themselves are no longer available or run on hardware or operating systems that are no longer available.

As two representatives, tools GRADE [50] and CODE [51, 52] will be briefly described. They are designed as stand-alone development environments. GRADE and CODE are both systems where a user can create a visual model that describes the behavior of a developed application. Such a model could be combined with sequential codes written in C.

Two types of diagrams exist in GRADE. The first diagram describes communication between processes; the second one describes an internal (sequential) behavior of a process in the form of flow-charts enriched by message passing constructs. The tool offered an integrated development environment for developing, executing, and monitoring developed applications.

The semantics of CODE is different; it is based on creating computations units that are executed when some data arrives to its input channels. CODE's semantics shares some basic aspects as Petri nets (Petri nets are defined in the next section). Computing units in CODE are described in a textual form, and a visual language is used for connecting units together. It appears as if the visual language was not designed to catch a state of a program's parallel execution; therefore CODE did not support visual simulations of modeled programs.

Some of these tools were not only programming tools but also supported some supportive activities, such as debugging [53] or performance analysis features [54].

2.4 Petri nets

Kaira is heavily inspired by *Colored Petri nets* (CPNs); therefore *Petri nets* [55] are introduced here as a related work. A Petri net (PN) is a mathematical modeling tool for distributed systems. A basic form of Petri nets is a *Place/Transition Petri net* (PTN).

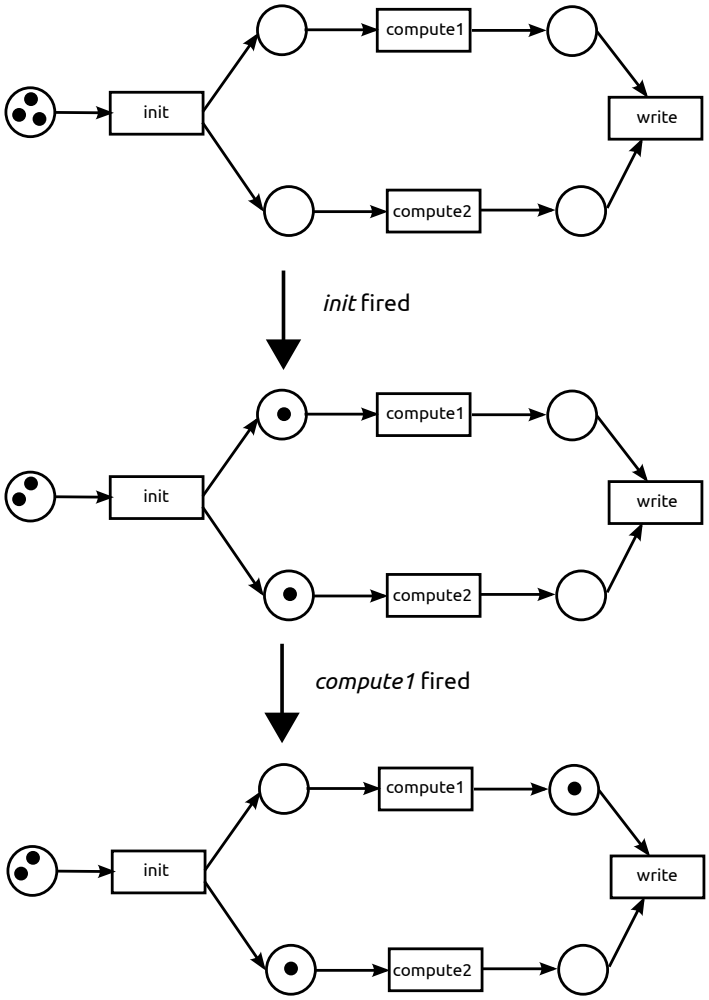


Figure 2.1: An example of Place/Transition Petri net together with states after firing two transitions

A PTN consists of *places*, *transitions*, and *arcs*. Figure 2.1 shows a graphical notation of a PTN and two steps of its execution. Places are denoted as circles, transitions as boxes. Arcs can run from a place to a transition (*input arc*) or from a transition to a place (*output arc*). Arcs never run between two places or two transitions. If we fix a transition then *input (output) places* are places connected with the transition by input (output) arcs.

Places can be seen as memory spaces and transitions as actions. *Tokens* (in PTN) are anonymous entities stored in places. A state of a PTN (called *marking*) is caught by a number of tokens in each place. Tokens are usually depicted as small black circles in places. An evolution of a net is a process of removing and adding tokens from/into places. Manipulations with tokens are realized through transitions.

When a transition is *fired* then it consumes tokens from input places and it produces tokens in output places. A single token is consumed per each input arc and a single token is created per each output arc. A transition can be fired only if there are enough tokens in its input places. An initial distribution of tokens in places is called the *initial marking*.

Figure 2.1 contains a model of a simple situation where a computation is split into two separated parts (*compute1*, *compute2*) and when they are both finished then results are written at once.

Formally $PTN = (P, T, I, O, m_0)$ where P is a finite set of *places*, T is a finite set of *transitions*, $P \cap T = \emptyset$, $I : T \times P \to \mathbb{N}$ defines a multiplicity of input arcs, and $O : T \times P \to \mathbb{N}$ defines a multiplicity of output arcs where $\mathbb{N} = \{0, 1, 2, \dots\}$. A *marking* is a mapping $P \to \mathbb{N}$; $m_0 : P \to \mathbb{N}$ is *initial marking*.

A transition t is *enabled* in a marking m if $\forall p \in P : m(p) \geq I(t, p)$. If a transition t is enabled in m, then it can be *fired* and the system gets to a new state m' such that $\forall p \in P : m'(p) = m(p) - I(t, p) + O(t, p)$.

PTNs are a useful formalism in theoretical research. As a modeling language, however, they are too low-level for use with practical problems because models become too large. To solve this problem, different extensions known as *high-level* Petri nets were developed. Colored Petri Nets, one type of high-level Petri nets, are formally defined in [56]. The idea behind CPNs is not to consider tokens only as anonymous "black dots", but rather to consider each token in a CPN as carrying a value. Therefore in CPNs, the content of a place is not just a number of tokens but it is a multiset of values. There are also more complex conditions when a transition is enabled and which tokens are consumed/produced when a transition is fired. One of the most well-known tools based on CPNs is CPN Tools[57]. It is a general-purpose modeling tool where models can be created and analyzed.

CPNs are not the only high-level Petri net formalism. *Reference nets* can be named as another example; this formalism is implemented in the tool Renew [58]. It is a modeling tool with a tight integration of Petri nets and Java.

Chapter 3

Kaira

This chapter introduces Kaira; the basic principles and ideas are presented together with an informal description of the visual language and semantics. Kaira was already reported in [1, 2, 3, 4, 5, 6, 7, 8]. The content of this chapter starts with the description of design goals and key features in Section 3.1. The next sections cover a simple example (Section 3.2), syntax and semantics of programs in Kaira (Section 3.3), and the history of Kaira (Section 3.4). The chapter ends with the comparison of existing tools in Section 3.5.

3.1 Design goals and decisions

The main motivation of the presented tool is to simplify and make more accessible parallel programming of distributed memory applications in the area of scientific and engineering computations. The four design goals of Kaira are the following:

1. *Prototyping* – The user is able to create a working version of a developed application fast and in a form that allows experimenting, easy modifications, and performance evaluations.

2. *Unification* – All activities during the development are controlled and presented in the same conceptual frame. Results from different analyses are easily comparable and reusable.

3. *Real applications* – The primary output is a real application that can be executed on a real parallel computer. The performance of the resulting application must be as close as possible to a manually created solution.

4. *Integration* – Existing sequential codes are easily reusable in the developed program. The integration should also work in the other way, i.e. the tool is able to create parallel libraries that can be called in any sequential application.

To achieve these goals, the tool is designed as a complete integrated development environment (IDE), where an application can be designed and analyzed. For practical reasons, C++ and MPI were chosen as target platforms. Both are natural choices, C++ as a general purpose programming language widely used in the area of scientific and engineering computations and MPI as a portable wide-spread communication layer. But in comparison to a classic development of MPI applications, the user does not have direct access to MPI in Kaira. Parallel aspects and communication in a developed application are expressed in an abstract way with the following two features:

- Semantics of Kaira is based on a simple *abstract computation model* with natural parallelisms. It should provide a mental model for thinking about parallel algorithms without dealing with unimportant technical details. On the other hand, the intention is to keep the model quite low-level with a simple connection to MPI, to preserve the user's control over the developed algorithm. The layer between the model and MPI is also thin to achieve the performance goal of resulting applications.

- *Visual programming* is used as the way of creating and manipulating with programs. Visualizations are often used in the area of parallel programming, but in most cases, they are used as a way of presenting results of analyses. In Kaira, the visualization is one of the integral parts of the whole development process. The intention is to use the visual representation not only for programming but also as a unifying element through different tasks, as will be shown in Chapter 6. A visual program is used to show an inner state of a running parallel application directly in a visual representation drawn by its programmer and provides a platform for configuring analyses and displaying their results. On the other hand, visual programming in Kaira is not intended for creating complete applications, but only for expressing parallel aspects and communication, i.e. designing parts that are not present in a sequential application. The sequential parts are written directly in C++ in the textual way.

The abstract computation model in Kaira is based on CPNs. They provide a natural way of describing distributed computations and also a natural visual representation, including display of a distributed state of the computation. Implementation of model visualization in Kaira is heavily inspired by CPN Tools. Because of the specific needs of Kaira, CPNs semantics was extended and modified to be more suitable for parallel programming with MPI. In short, the most distinctive element is the usage of queues to store tokens instead of multisets.

The features offered by Kaira can be summarized as follows:

- Creating and editing visual programs. A screenshot of Kaira during program editing is shown in Figure 3.1.

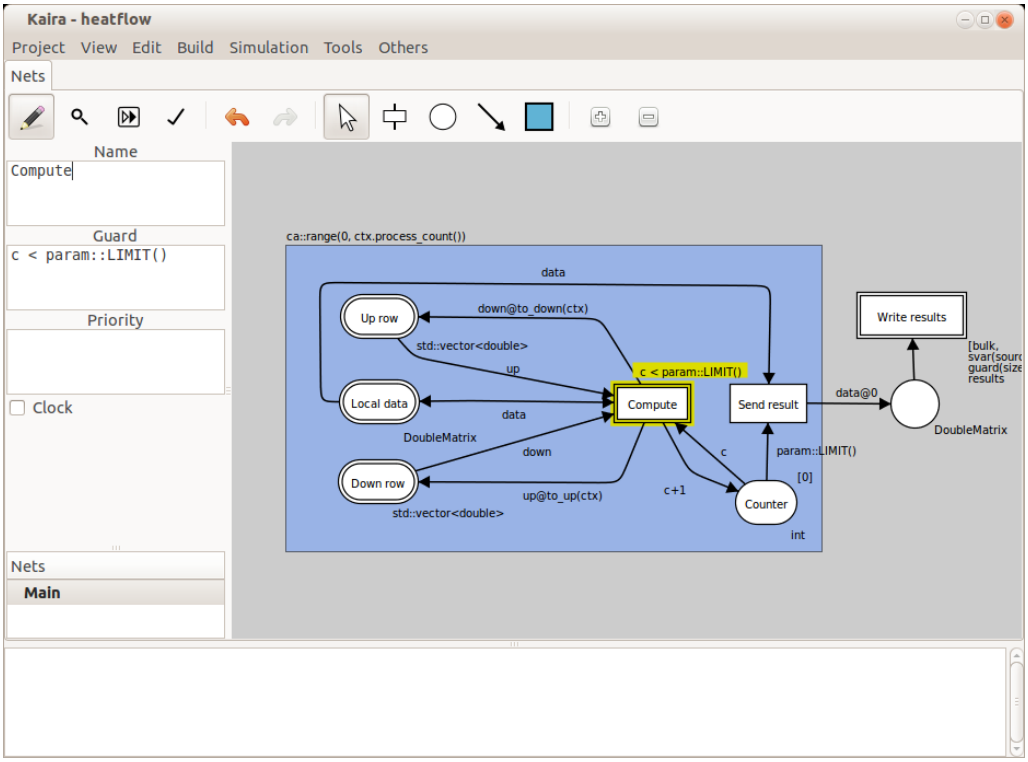


Figure 3.1: The editor with a visual model in KAIRA

- Inserting and editing C++ codes in a visual model.
- Translating visual models into C++ MPI programs and libraries.
- Showing and controlling a developed program in the visual debugger.
- Configuring and generating a tracing version of the application that records its own run. A record obtained in this fashion can be loaded back into KAIRA for further processing including visual replay.
- Providing functionality for the performance prediction through online simulations.
- Providing functionality for the verification through state-space analysis.
- Exporting results of analyses; they can be loaded into the visual debugger or prepared for the usage with external tools (e.g. R).

3.2 "Hello world" example

This section demonstrates the development of a simple application in KAIRA. The goal is to present basic ideas and principles in KAIRA. More details and a more precise description are provided in the next section. This section uses a very simple example; advanced examples can be found in Chapter 4.

Let us consider Figure 3.2, which shows a visual program in KAIRA. The visual programs are called *nets*. This net contains three *places* (circles) and a single *transition* (box). Places represent memory spaces where data are stored in queues. Places a and c store values of type `int` and place b has type `std::string`. Values stored in places are called *tokens*. The initial content of a place is written in the upper right corner, e.g. place a starts with tokens `1`, `2`, and `3` and place c is empty. The transition represents the behavior of the program. It takes tokens from the input places (a and b in the example) and puts tokens into output places (place c in the example).

When the program is started in the simulator (visual debugger) then the user will see Figure 3.3.A. The simulator shows the situation immediately after net initialization. Tokens are depicted in the green box, suffix `@0` informs about the process where tokens are placed. They are all on process 0, because in this simple example, only a single process is used. The transition is highlighted by the green rectangle; it means that the transition is *enabled* (there are enough tokens in the input places). Because it is enabled, it may be *fired*.

In a generated program, enabled transitions are automatically fired, but in the simulator the program is paused and waits for a user action. When the user clicks on the transition then it is fired. In this example, the first token from each input place is consumed and values of variables are set according to these tokens. When the execution of the transition is finished then a new token is produced in the output place. There are two variables: `x` and `str`. During this particular transition execution, the value of `x` is `1` and the value of `str` is `"Hello"`.

The resulting state is shown in Figure 3.3.B. The transition is still enabled (and therefore highlighted) because there are still enough tokens. Figure 3.3.C shows the resulting state after firing the transition again. In this last state, the transition is not enabled anymore. It needs a token in all input places and place b is empty.

Technically, it is possible to describe a complete algorithm in this way, but the visual language is primarily designed to describe communication and data flows. Classic textual C++ codes should be used to describe sequential parts of the application. They are inserted into transitions as depicted in Figure 3.4. KAIRA opens an editor with a template that cannot be changed and the user can fill any C++ code as a body of the function. The variable `var` makes access to variables around the transition. Every time the transition is fired, this code is executed. If the code is written as in the figure, we obtain a program that prints on the standard output:

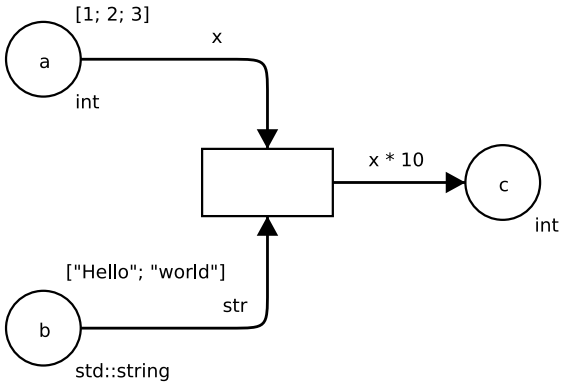


Figure 3.2: Example "Hello world"

```
1 Hello
2 World
```

So far, the example has described a behavior of a single process. When a program is run on more processes, an independent copy of the net called *net-instance* is created for each process. Transitions always consume tokens only from their own net-instance, but they can create tokens in net-instances of other processes. It is the only way of communication between net-instances. An output arc creates a token locally (i.e. in the same process) as default, but it can be changed by symbol `@`. For example, transition *divide* in Figure 4.1 sends a token to a process defined by variable `worker`.

3.3 Programming in Kaira

In the previous section, creating programs in Kaira was showed in a very simplified way. This section explains programming in a way that the reader should be able to create any common program. Only some technical details and some corner cases of semantics are omitted. The semantics is fully described in Chapter 5. The technical details of API are given in User Guide[1].

A program in Kaira consists of a *visual program* (net), *sequential codes* embedded into the net and *global configurations*. This text starts with the most important part, the visual language for describing nets. The goal of the visual language is to describe data-flows and communication in the developed program. It consists of four main elements: *Places, Transitions, Arcs*, and *Init areas*. Graphical elements drawn in the surrounding figures are usually presented in their default sizes, but each element can be resized. The overview of visual elements is depicted in Figure 3.5. Some attributes of an element are depicted as text around the element's

[1] `http://verif.cs.vsb.cz/kaira/docs/userguide.html`

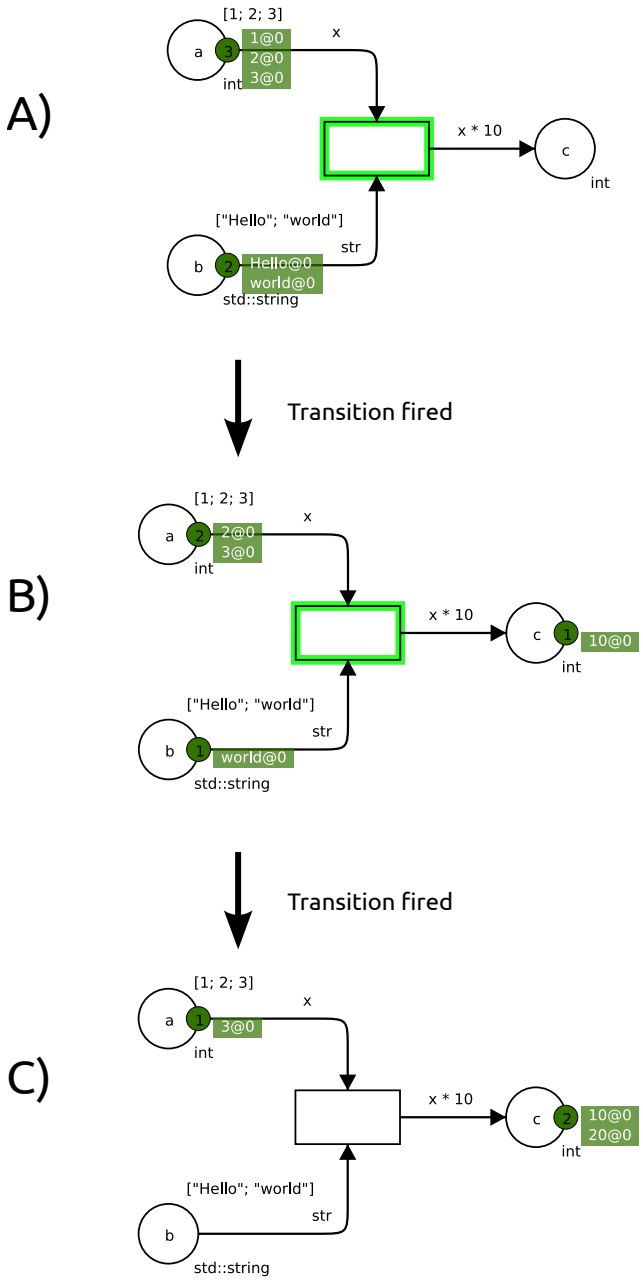


Figure 3.3: Three steps of “Hello world” simulation

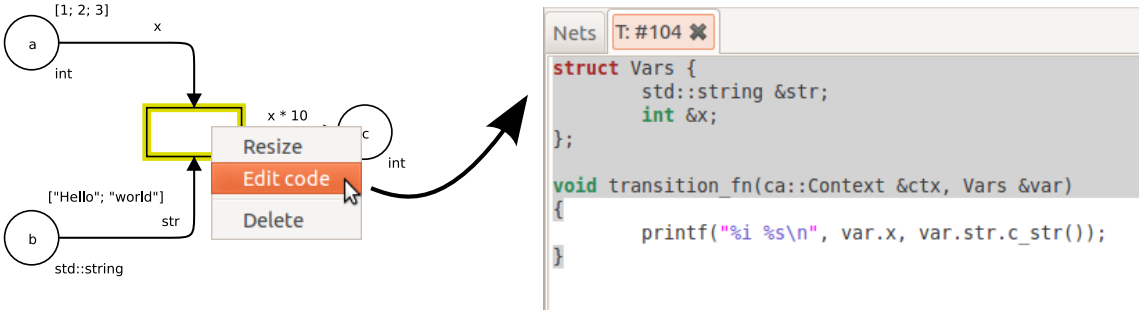


Figure 3.4: Inserting a C++ code into a transition

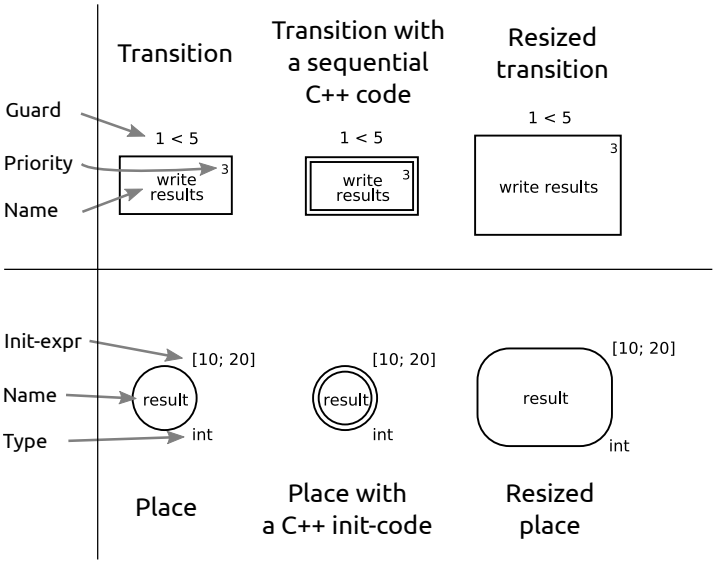


Figure 3.5: Visualizations of transitions and places

main visual representation. They can be freely moved by the user in the editor. In the following text, default positions of attributes are described and the convention is to keep their relative position, even though they can be slightly moved to achieve a clearer diagram.

Some attributes of an element contain C++ expressions. Each expression should not have any side effect and a returned value should depend only on the values of its free variables. The user cannot make any assumption about when and how many times any one of the expressions is evaluated. In each expression, the special variable `ctx` can be used. It holds an instance of `ca::Context`, this class is described in 3.3.8. The instance provides basic information about the current process rank and the total number of processes. Namespace `ca` contains functions and classes provided by KAIRA.

3.3.1 Places

Places serve as memory spaces for the program and they can be seen as queues of tokens. Tokens are usually manipulated by First-In-First-Out policy, but some other ways are also allowed. Each place has the following properties:

- *Name* – An arbitrary string for describing the place with no actual meaning. It is depicted as a text in the center of the place's circle.
- *Type* – An arbitrary C++ type that determines what values can be stored in the place. This attribute is placed at the bottom-right side of the place. Standard types like `int` or `std::vector` can be used without any additional action. In the case of user defined types, several special functions have to be defined, this topic is covered in Section 3.3.10.
- *Init-expression* – An expression that is used for the place initialization. It puts tokens into the place at the beginning of the program run. It may be empty (i.e. no initialization by init-expression) or have one of the following two forms:
  - A C++ expression of type `std::vector<`*t*`>` where *t* is the type of the place. For example, it is used in place *ready* in Figure 4.1 (The function `ca::range` returns `std::vector<int>`)
  - A list of C++ expressions of the place's type. It has to start with character `[` and end with `]`. C++ expressions have to be separated by semicolons. Example: when the type of a place is `double`, expression `[1.0; 1.5; 2.8]` may be used to initialize the place by three values. It is used for example in place *counter* in Figure 4.1.
- *Init-code* – A C++ code that allows a generic place initialization. A place containing an init-code is depicted with the double border. For example, place *Local data* in Figure 4.3 contains an init-code. More about init-codes will be said in Section 3.3.8.

3.3.2 Transitions

Transitions define the behavior of the program; the program evolves by firing them. A transition takes tokens from its input places, executes a computation and then puts new tokens into its output places. These token manipulations are defined by arcs connected with the transition.

For the following definitions, let us fix a transition. There are two types of arcs: *input arc* (from a place to the transition) and *output arc* (from the transition to a place). *Input (output) places* are places that are connected with the transition by

input (output) arcs. Arcs are described in the next three sections. Here follows a description of the transition properties independent of them.

- *Name* – An arbitrary string for describing the transition with no actual meaning. It is depicted as the text in the center of the transition.
- *Guard* – A boolean C++ expression that may contain input variables (variables that occur on input arcs). The guard is evaluated as a part of the test if the transition is *enabled* after fixing values of the input variables; it will be explained in Section 3.3.4. If the resulting value is not `true` then the transition is not enabled. If the guard expression is empty then it is assumed that the guard is always `true`. The guard is depicted at the top of the transition. A guard can be seen over transition *divide* in Figure 4.1.
- *Priority* – An integer value that specifies the priority of the transition. When a transition is enabled then all transitions with lower priorities in the same process cannot be fired. If the priority is unspecified then priority 0 is assumed. If there are more enabled transitions with the same priority in the same process simultaneously, no fairness guarantees are provided. The priority value is depicted as a small number in the right part of the transition's rectangle. The example can be seen in transition *Send result* in Figure 4.5.
- *Fire-code* – A C++ code that is processed each time the transition is fired. More about fire-codes is explained in Section 3.3.8. The transition with a fire-code is depicted with the double border.

3.3.3 The syntax of expressions on arcs

Input arcs and output arcs have different semantics, but both types have associated expressions with the same syntax. An arc expression consists of subexpressions called *inscriptions*. Inscriptions are separated by semicolons and each arc expression has at least one inscription. The scheme of the inscription syntax is the following:

[*configuration*] *main-expression*`@`*target*

- *Configuration* consists of *configuration items* separated by commas. Each configuration item has the form `keyword` or `keyword(`x`)` where x is a C++ expression. If there is no configuration item, square brackets can be omitted.
- *Main-expression* is a C++ expression. The actual meaning depends on the type of arc and configuration items. Generally speaking, it defines value(s) of token(s) that are taken from / put into the arc's place.

- *Target* is a C++ expression defining the process(es) where created tokens are sent. It can be used only on output arcs. If the target is not defined then `@` has to be omitted.

Here are four examples of valid arc expressions below this paragraph, each per line. The second and the fourth example are composed of more inscriptions:

```
x + 1
10; 30; 40@x*2
[bulk] results
[if(x>2), multicast] x1@where; x2
```

A set of *inscription variables* is the set of variables that occur in any C++ expression of a given inscription. A set of arc variables is a union of inscription variables of a given arc.

3.3.4 Input arcs

Input arcs define inputs of the transition. The transition is *enabled* if there are "right" tokens in its input places. What tokens are expected depend on the transition's input arcs and the guard expression. When a transition is enabled it can be *fired*. When the transition is fired, tokens are removed from input places according to input arcs and new tokens are produced into output places according to output arcs. For input arcs, the basic syntax rules are the following:

- There is at most one input arc between each place and the transition (but there can be more inscriptions on an input arc).
- Inscriptions of an input arc cannot contain a target.

First, inscriptions with empty configurations will be considered. For such inscriptions, the following two general rules apply:

- When the transition is tested for whether it is enabled, each inscription selects a first token from the arc's place, not taken by previous inscriptions. Inscriptions are evaluated from the left. The transition is enabled only if the main-expression of each inscription "matches" the selected token.
- When the transition is fired, each inscription removes the selected token from the place.

The examples that will be referred in the following text are from Figure 3.6. The most common input arc is the arc with a single inscription containing no configuration part and its main-expression is a variable. In this case, the transition is enabled only if there is at least one token in the arc's place. Because a variable is used, the inscription matches any token, hence it takes the first one (example 1).

Variables are always related to a particular transition. Therefore types of variables are unique across all expressions connected with the transition (guard expressions, input and output arcs expressions). But variables for different transitions are completely separated sets, hence two transitions may have a variable of the same name but with different types. Variables also exist only during checking if the transition is enabled and when the transition is fired; therefore they carry no values between each transition execution.

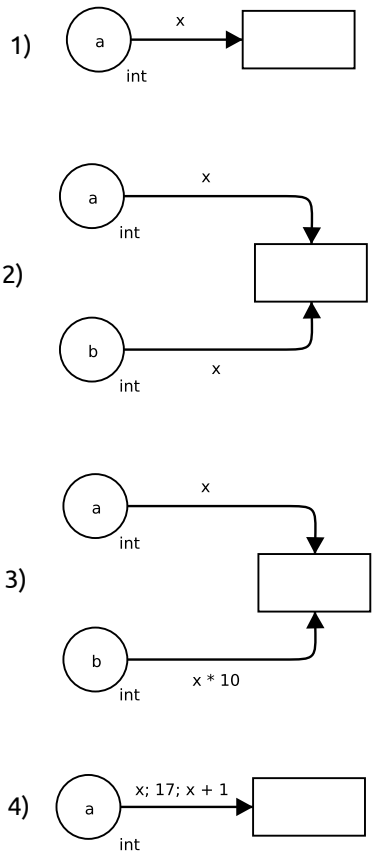
If a variable has already assigned value from another inscription then the token has to have the same value as the variable (example 2). If the main-expression is not a variable then the inscription matches the selected token only if the token value is equal to the value obtained by evaluating the main-expression (example 3). If there are n inscriptions on an arc then n first tokens are checked and taken when the transition is fired (example 4).

This behavior of an inscription can be modified by following configurations items. Examples are in Figure 3.7.

- `bulk` – When an inscription contains `bulk` then it has to be the only inscription on the arc and its main-expression has to be a variable. An arc with such an inscription is called a *bulk arc*. The bulk arc itself does not make any condition when the transition is enabled, but when it is fired then it takes all tokens from the place. The list of taken tokens is set to the variable in the main-expression. The type of the variable is `ca::TokenList<`t`>` where t is the type of the place. Class `ca::TokenList` serves as a container for tokens. The API is described in User Guide. The configuration item `bulk` cannot be combined with `filter` and `from`. A usage of a bulk arc is demonstrated in example 5.

- `filter(`x`)` – The expression x has to be a boolean C++ expression. The inscription ignores tokens that do not satisfy the condition defined by x (example 6). It means that the inscription selects a first token that satisfies the condition and that is not selected by any previous inscription. An inscription without `filter` can be seen as an inscription with `filter(true)`. This configuration item cannot be combined with `bulk`.

- `if(`x`)` – The expression x has to be a boolean C++ expression. If `if(`x`)` is in the inscription configuration, then x is evaluated before any effect caused by this inscription is taken. If the result is `false`, then the inscription is completely ignored otherwise the inscription is processed normally.



The transition is enabled if there is at least one token in place *a*. When it is fired then it takes the first token from the place and sets its value to variable *x*.

The transition is enabled if the first tokens in both places have the same value. When it is fired then it takes these first tokens and sets the value of tokens to variable *x*.

The transition is enabled if both places are nonempty and the first token in place *b* is ten times bigger than the first token in place *a*. When it is fired then it takes these first tokens and sets the value of the first token from place *a* to variable *x*.

The transition is enabled if there are at least three tokens in the place such that the second token equals to 17 and the third token is greater by one than the first token.

Figure 3.6: Basic examples of input arcs

- `guard(`x`)` – The expression x has to be a boolean C++ expression. It defines an additional guard condition for the transition. In comparison to a transition's guard expression, variable `size` (type `int`) in x may be used. It contains the number of tokens in the place. Other properties are the same as for the transition's guard, it is evaluated in the same time and the transition is not enabled if the result is not `true`. The difference between `guard` and `filter` is shown in example 7.

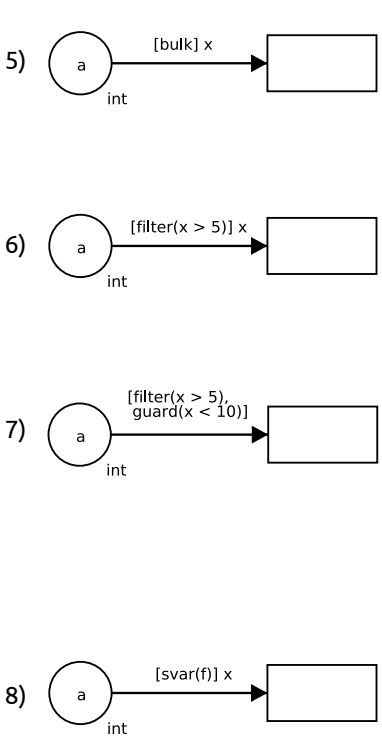
- `svar(`x`)` – The expression x has to be a variable. It declares `int` variable x that stores the *source* of the token. The *source* is the rank of the process where the token has been created. The usage is demonstrated in example 8. In the case of a bulk arc, x has type `std::vector<int>` and the variable contains sources for all tokens that were in the place.

- `from(`x`)` – The expression x has to be an integer C++ expression. It is an abbreviation for `[svar(`α`), filter(`α`==`x`)]` where α is a fresh variable; in other words, the inscription considers only tokens from process x. It cannot be combined with `bulk`.

- `sort_by_source` – This configuration item has to be used together with `bulk`. Tokens taken from the place are sorted by their sources.

Let us note that during checking whether a transition is enabled, expressions in `guard` and `if` are evaluated at most once. But a `filter`'s expression may be evaluated more than once (in the worst case, it is evaluated for each token in the place).

The next rule may look slightly complicated, but it only forces a natural condition to allow a direct derivation of types and an efficient implementation of token matching. For practical examples, it is no problem to satisfy this condition; even the user is not aware of it. Let us fixed a transition. An *input variable* is the variable that occurs in any expression on any input arc of the transition; except special variable `ctx`. For each input variable v, there has to be a *variable defining inscription* (VDI). VDI is an inscription on an input arc of the transition, where the main-expression is directly v or v is used in `svar`. VDI for a variable v determines a type of v. When v occurs in the main-expression of VDI, then the type of v is the type of arc's place. When v is used in `svar` then the type is `int` (or `std::vector<int>` in the case of a bulk arc). If there are more VDIs for v, then all VDIs have to derive the same type, otherwise an error is reported. Moreover there cannot be cyclic dependency between VDIs. It is formalized in Chapter 5, but the idea is shown in examples 9 and 10 in Figure 3.8 that contain two invalid cases.



The transition is always enabled even if the place is empty. Therefore in real examples, there is always another input arc when a bulk arc is used. When the transition is fired, then all tokens are removed from the place and set to variable *x* as a list of tokens.

The transition is enabled if there is at least one token bigger than five in the place. If the transition is fired then the first such token is taken and the value is set to variable x.

The same as the previous example, but the first token that passed through the filter (i.e. bigger than five) has to also be less than ten to enable the transition. Only the first token that passes through the filter is checked by the guard. If the guard fails, no other tokens are checked and the transition is definitely not enabled. For example, if tokens in place *a* are [3, 6, 12] then the transition is enabled, if tokens are [3, 12, 6] then the transition is not enabled.

The same as example 1 in the previous figure, but when the transition is fired, then besides setting up the value of *x*, also the value of integer variable *f* is set to the rank of the process that created the taken token.

Figure 3.7: Examples of input arcs with configuration items

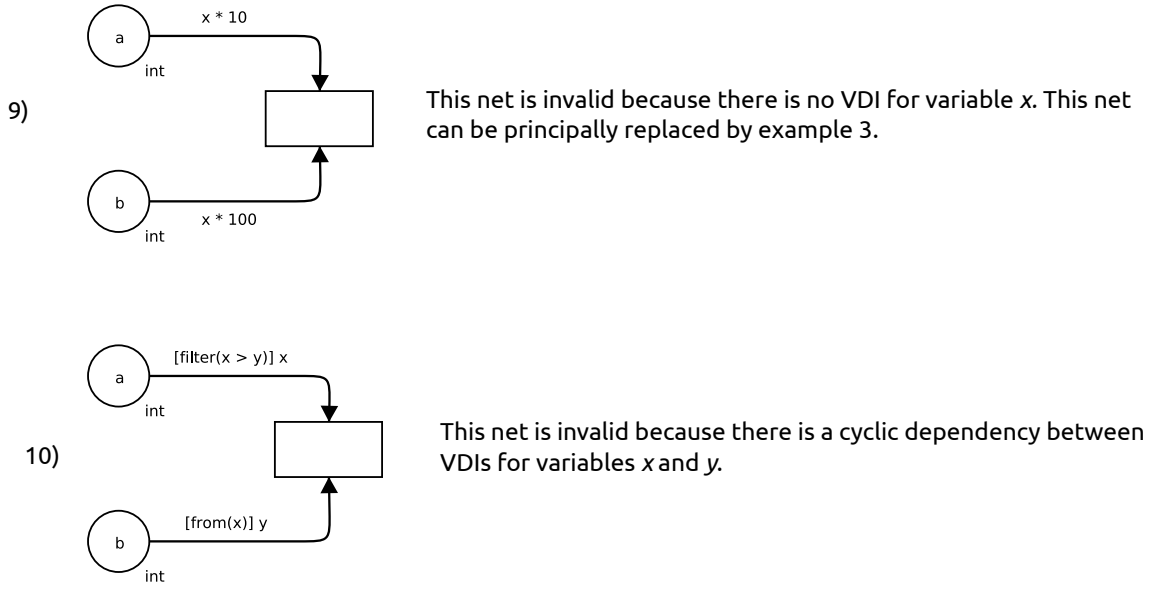


Figure 3.8: Examples of invalid configuration of input arcs

3.3.5 Output arcs

When the transition is fired, then output arcs define which tokens are created and where they are created. In comparison with input arcs, there can be more arcs between a transition and a place and targets in inscriptions are allowed. The examples used in this section are depicted in Figure 3.9. Let us start again with the case of inscriptions without configuration items. In this case, the main-expression of the inscription is evaluated and the obtained value is used to create a new token. If the target of the inscription is not specified, the token is created in the same net-instance where the transition was fired. Otherwise the new token is sent to the net-instance in the process specified by the target (example 11). For each pair of processes a and b, tokens sent from a to b are always received in the order in which they were sent. Receives are performed automatically between firing transitions.

The default behavior of output inscriptions can be changed by the following configurations:

- `bulk` – The inscription produces an arbitrary number of tokens. The main-expression must have type `ca::TokenList<`t`>` where t is the type of the place. It produces a new token from each element of the token list (example 12). There are no restrictions like in the case of bulk input arcs; therefore the main-expression does not have to be only a variable and there can be more than one inscription on the arc.

- `multicast` – The target of the inscription must have type `std::vector<int>` and token(s) created by this inscription are sent to all processes defined by the target expression (example 13).
- `if(`x`)` – This configuration item works in the same way as for input arcs. The expression x has to be a boolean expression. If x is evaluated to `false`, then the inscription is ignored.
- `seq(`x`)` – x has to be a constant integer. If `seq` is not defined then `seq(0)` is assumed. This configuration controls the order in which inscriptions on all output arcs are evaluated; an inscription with a lower number is evaluated before an expression with a higher number. If two inscriptions have the same `seq` number then `bulk` inscriptions are prioritized. If none of these rules are able to be used to determine the order, then Kaira evaluates output inscriptions in the order in which they are written on arcs from left to right and inscriptions on different arcs are evaluated in an unspecified order. The configuration item `seq` is used only in cases where sending tokens in specific order to different places is important.

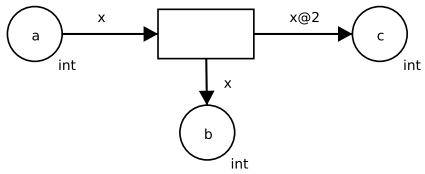
All input variables can be used in expressions on output arcs. New variables can also be introduced, but VDI must be present among output inscriptions if the variable was not used on input arcs. VDI for output arcs is an inscription where the variable is used as the main-expression or the target. However, as with input inscriptions, in practice this presents no actual restriction.

3.3.6 Net-instances

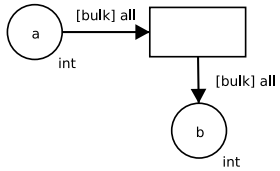
When the application is executed on n processes then n independent copies of the program are created; they are called *net-instances*. When an MPI version of the program is generated then a net-instance exists for each MPI process. The number of processes is fixed at the beginning of the execution of the program and cannot be changed. The current version of Kaira does not support a dynamic creation of processes.

From the user's view, the behavior of the application can be described as repeated searching of enabled transitions and firing them. Each process works only with transitions and places in the assigned net-instance. Between transition executions, each process tries to receive tokens that were sent to it. Sends of tokens are performed immediately at the end of the transition execution that creates non-local tokens (output arcs containing symbol `@`). All sends and receives are processed by MPI non-blocking calls. Resources used for sends are automatically freed when the send is completed.

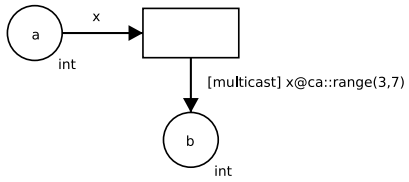
There is an experimental feature in Kaira, a hybrid mode where more threads operate over a single net-instance. In other words, there can be more threads for

11) 

When the transition is fired, it takes the first token from place *a,* puts a new token of the same value into place *b* in the same process where transition is executed, and sends token into place *c* in process 2.

12) 

When the transition is fired, it takes all tokens from place *a* and puts them into place *b* at once.

13) 

When the transition is fired, it takes the first token from place *a* and sends the value to place *b* in processes 3,4,5,6. Function *ca::range(x,y)* returns a vector of integers in a range [x, y).

Figure 3.9: Examples of output arcs

each MPI process and more enabled transitions in one process can be processed at once. This feature is implemented and works, but it is not optimized and not fully supported in all parts of Kaira. Accordingly, this thesis does not consider this features; it is always assumed that only one thread operates in each MPI process. The hybrid mode is mentioned again in Section 8.1.3 as a potential subject for future work.

3.3.7 Init-areas

At the beginning of a program run, places are initialized by init-expressions and init-codes. They are evaluated only in process 0 by default; places in other processes remain empty. This behavior can be changed by *init-areas*. An init-area is depicted as a rectangle with a blue background with a single attribute in the top-left corner. The attribute has one of the following forms: C++ expression of type `std::vector<int>` or `int` expressions separated by semicolons and surrounded by square brackets. It is analogous to syntax of init-expressions. The expression in the attribute is evaluated at the beginning of a program's run and it determines processes where the places inside the area are initialized (i.e. where init-expressions and init-codes are evaluated). Places that stay outside of all init-areas are initialized only for process 0. The example of an init-area can be seen in Figure 4.3.

3.3.8 Sequential codes

The integration of the visual language with C++ codes has been almost fully introduced in previous sections. C++ expressions can be freely used on arcs, transition guards, and place-initializing codes. As was shown in Section 3.2, a C++ code can be placed into a transition. In this case, Kaira generates a template in the form showed in Listing 3.1. This code is then executed every time the transition is fired. In comparison to expressions used on arcs, it is ensured that this code is executed exactly once per each transition execution. It is executed after removal of tokens from input places and before the evaluation of inscriptions on output arcs. Therefore, when this code changes the value of a variable, it will have an effect on newly created tokens.

The second parameter `var` enables access to variables used on arcs of the transition. The structure `Vars` is generated separately for each transition. The first parameter enables access to an instance of the class `ca::Context`; it provides the following methods:

- `int process_id()` – returns the rank of the current process.
- `int process_count()` – returns a total number of processes.

Listing 3.1: An empty template for a sequential code in a transition

```
struct Vars {
    // ... Variables on the arcs of the transition
};

void transition_fn(ca::Context &ctx, Vars &var)
{

}
```

- `void quit()` – terminates the whole program.

A similar template is used for editing an init-code of a place (Listing 3.2). This code is called at the beginning of the program execution to initialize the place.

Listing 3.2: An empty template for an init-code of a place where t is the type of the place

```
void place_fn(ca::Context &ctx, ca::TokenList<t> &place)
{

}
```

3.3.9 Global configurations

The global configurations of a net are composed of:

- *Parameters* – Constant values that are initialized at the beginning of a program run. In the case of a generated MPI program, they can be set through command line arguments. In the program, values of parameters can be accessed via static members of class `param`. The example can be seen in the guard of transition *divide* in Figure 4.1.

- *Head-code* – A C++ code that is inserted at the beginning of the generated source code, it usually contains `#include` directives with header files of used libraries or declarations of used types.

- *Build options* – A list of external C++ source code files and flags for the C++ complier and linker; that is, information necessary to build an executable form of the program.

- *Configurations of tracing, simulated runs, and verifications* – These configurations are not necessary to build a resulting program, but they are important for supportive analyses. They are usually depicted as labels in the net. They are described in more detail in sections of Chapter 6. Examples can be seen in Figures 6.5, 6.16, and 6.22.

3.3.10 Integration of C++ types

As was already said, any C++ type can be used as the type of a place. Standard C++ types like `int` or `std::string` can be used without any additional action; the full list of such supported types are listed in User guide[2]. In the case of other types, the user has to define the following three functions:

- `token_name` – The function returns a textual representation of tokens of the given type. This representation is used during visual simulations.
- `pack` – The function serializes a token.
- `unpack` – The function deserializes a token.

If tokens of the given type are not sent between net-instances, then the last two functions are not necessary to implement. The example of binding a user-defined type is given in Listing 3.3.

3.4 History

Project Kaira was initiated by the author of this thesis. The tool was first conceptualized in 2008 during the author's master's studies. In 2010, the idea was revived as part of the grant proposal "Modeling and verification of parallel systems" (GAČR P202/11/0340) led by professor Petr Jančar, the advisor of the author. The grant was accepted and the development of Kaira was supported by it from 2011.

In the end of summer 2010, the first working prototype was finished and presented to several people, one of these was Marek Běhálek, who joined to the project. At the beginning of 2011, version 0.1 was finished and the first paper was published [1]. From the perspective of this thesis, version 0.1 was designed as a more high-level tool and mapping to MPI processes was more indirect. In that time, Ondřej Meca and Martin Šurkovský joined the project. The next version (0.2) was released few months later and made the semantics of Kaira even more abstract. After finishing this version, we realized that the proposed abstraction was not well-supported by practical examples and it was hard to translate nets into efficient MPI programs.

[2] `http://verif.cs.vsb.cz/kaira/docs/userguide.html#Types`

Listing 3.3: Example of binding a simple type

```
struct SimpleType {
    std::string name;
    double x;
};

namespace ca {

    std::string token_name(const SimpleType &v) {
        std::stringstream s;
        s << "Name=" << v.name << " x=" << v.x;
        return s.str();
    }

    void pack(ca::Packer &packer, const SimpleType &v) {
        pack(packer, v.name);
        pack(packer, v.x);
    }

    template<> SimpleType unpack(ca::Unpacker &unpacker) {
        SimpleType v;
        v.name = unpack<std::string>(unpacker);
        v.x = unpack<double>(unpacker);
        return v;
    }

}
```

Table 3.1: Contributions to the source code of KAIRA 1.0 based on statistics from GIT repository.

Name	Commits (%)	Added lines	Removed lines
Stanislav Böhm	861 (76.40%)	75671	52771
Martin Šurkovský	174 (15.44%)	13611	7196
Ondřej Meca	78 (6.92%)	4003	2035
Ondřej Garncarz	4 (0.35%)	295	67
Marek Běhálek	3 (0.27%)	267	70
Lukáš Tomaszek	2 (0.18%)	293	4
Martin Kozubek	1 (0.09%)	16	1

These reasons led to rethinking of some of the basic ideas of the project, and many things were simplified. KAIRA was largely rewritten in version 0.3. Additionally, the back end of KAIRA originally written in HASKELL was rewritten into PYTHON. Version 0.3 was released in the end of 2011. From 2012, the project was also supported by IT4Innovations Center of Excellence (project CZ.1.05/1.1.00/02.0070). Versions 0.4 (summer 2012) and 0.5 (beginning of 2013) represent further evolutionary steps from version 0.3. Features including tracing, the state-space analysis, and integration with OCTAVE were introduced. The papers [2, 3, 4, 5, 6, 7] were based on this generation of KAIRA.

Version 0.6 (May 2013) brought many new changes. The most visible one reflects the decision to focus on C++. The domain-specific language used for expressions in the visual language was removed and C++ is directly used from this version (C++ codes in transitions are used from version 0.1). The paper [8] is based on version 0.6. Parallel with the completion of this thesis, version 1.0 was released in November 2013 as a continuation of 0.6 where all features were finished in the form as they are presented in this thesis.

In addition to the work performed by the four members of the core team, some minor features were also implemented by Ondřej Garncarz, Martin Kozubek, and Lukáš Tomaszek. A rough overview about contributions to the source code is provided by statistics from the version control system showed in Table 3.1.

3.5 Comparison with selected tools

A natural next step is to compare the development of applications in KAIRA and classic programming in C++ with MPI. Everything that can be done with KAIRA can be also achieved by the direct use of C++ and MPI. The opposite direction does not hold. The current version of KAIRA does not use collective communication, i.e. all communication is realized through point-to-point sends. This problem is

discussed in Section 6.6.3. Another limitation is that programs translated from nets introduces an overhead in comparison to manually written programs. But it will be shown in Section 6.1.1 that the introduced overhead is minimal because the semantics of Kaira can be efficiently translated into MPI programs. On the other hand, Kaira offers an environment where a working application can be rapidly implemented in the form where the communication structure can be easily modified and the behavior of the application can be instantly observed.

The concept of Kaira is close to visual parallel programming tools from Section 2.3. These tools had the same goal as Kaira, providing a developing environment for message passing applications. However, as was mentioned earlier these tools are no longer developed. Unfortunately, the author did not find any general explanation concerning the termination of development of all these tools, even as some of the original authors of the tools were contacted. We believe that the broader scope of Kaira makes adoption of our tool by practitioners more likely. As far as the author knows, the aforementioned tools did not support any performance predictions or verifications. Integration with other tools was not the major goal of these tools; none of these tools were able to generate libraries. Parallel computer have become more available in the time between the development of the discontinued tools and of Kaira; for this reason, more non-experts have the opportunity to create programs for them. Tools that simplify the creation of these programs have therefore more opportunities to be used.

The usage of queues connects Kaira with stream programming. Kaira is a more low-level tool than stream programming environments; Kaira uses simple explicit mapping computations to MPI processes, in contrast to the sophisticated algorithms for scheduling and mapping of computations in stream environments. The approach that Kaira offers is less automatic and gives the user more control of resulting programs. It is important to allow experimentation with different kinds of algorithms; this is related to our first design goal.

When comparing Kaira and high-level languages like Python or Octave with their MPI libraries, Kaira offers the performance of C++ and the unified environment for supporting activities. But in some sense, Kaira may be considered not as an alternative to these high-level languages, but rather as a complement. An integration of Kaira with Octave will be demonstrated in Section 6.6.2.

As was already said, CPN Tools was a great inspiration when Kaira was designed and the visualization of models used in Kaira is mostly based on the visualization from CPN Tools. The fundamental differences that distinguish these two tools emerge from their different main goals. CPN Tools is a generic modeling tool, hence a large collection of problems that exceeds the scope of Kaira can be modeled and analyzed. But it would be difficult to use CPN Tools as a developing environment. It cannot create a stand-alone application. Furthermore where Kaira uses C++, CPN Tools uses Standard ML [59], hence an integration with many

existing libraries would be more complicated.

Chapter 2 has shown many tools for various supportive activities that can be used during development of an application that uses MPI. They are generic tools that can usually be used with any program. The supportive infrastructure in Kaira works only for programs developed in it. Tools mentioned in Chapter 2 are usually more mature and optimized in comparison to our implementation. But they are usually single-purpose tools with different terms, configurations, and ways of displaying results. In Kaira, all activities are roofed by the semantics of Kaira and their usage is unified under a single concept through the visual language. Chapter 6 presents each feature separately together with comparisons to some of these other tools.

Chapter 4

Examples

This chapter contains six examples of programs developed in Kaira. The examples are inspired by patters that commonly appear in parallel computations. All examples are presented with nets and important pieces of textual parts of programs. The full source codes of all examples including all variants can be found at the homepage of Kaira (Example 4.6) or in the distribution package (all other examples).

4.1 Example: Workers

The first example solves a classic problem. A master node (process 0) divides jobs to working nodes (other processes). When a working node finishes an assigned job then it asks the master node for a new job. This process is repeated until the computation is not over. Figure 4.1 contains a net solving this problem. In this example, a job is an interval of numbers that can be easily changed to any other structure. The net has two parameters: `LIMIT` and `SIZE`. The master node assigns intervals from the range $\{0, \ldots,$ `LIMIT` $- 1\}$ and `SIZE` defines the size of intervals assigned to workers. For the sake of simplicity, it is assumed that `SIZE` divides `LIMIT`. The job is a simple structure defined in the head-code, the code is listed in Listing 4.1.

The place *ready* holds ranks of idling workers; it is initialized by the ranks of all workers. Function `ca::range(`x`,`y`)` returns a vector of integers in the range $\{x, \ldots, y - 1\}$. Place *counter* keeps a start of a next assigned interval. Transition *divide* takes a value from place *counter* (variable `start`) and a rank of an idling process (variable `worker`) and it sends a new interval to the worker. The value in *counter* is increased to assign a different interval in the next step.

Transition *compute* performs the computation on an assigned interval. In this implementation, it just naïvely searches prime numbers in the interval. When an execution of a job is finished, results and a token with the worker's rank are sent back to process 0. Transition *write result* takes all results and writes them on the standard output. It has to wait until the token in *counter* reaches `LIMIT` and all

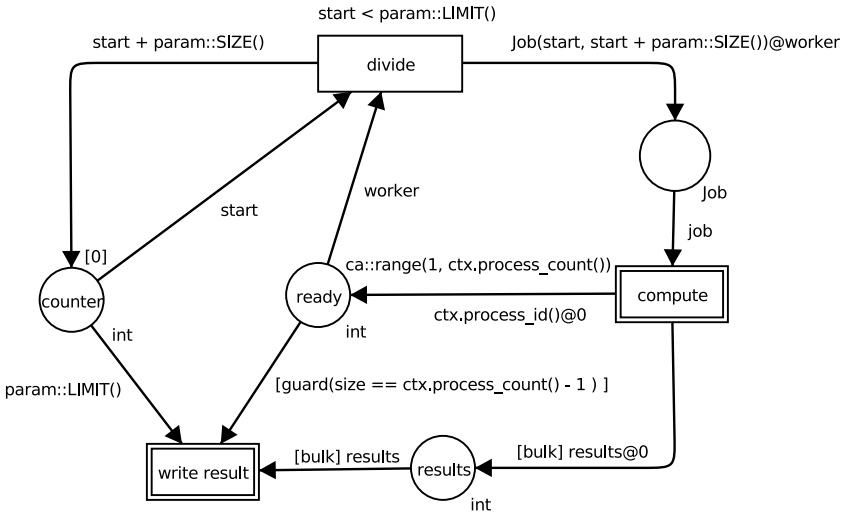


Figure 4.1: The net of the workers example

workers are ready, hence the absence of pending computations.

In the default configuration, process 0 only divides jobs and is not involved in their processing. It can be easily changed by modifying the init-expression for place *ready* to `ca::range(0, ctx.process_count())` and setting a larger priority to *divide*. This change causes process 0 to also assign jobs to itself. In this case, it should be noted that when process 0 is executing transition *compute*, transition *divide* cannot be fired and other processes have to wait for new jobs. This effect can be minimized by another simple modification. Process 0 will assign itself smaller intervals and therefore it can check if *divide* is enabled more often.

4.1.1 Usage of GMP

To demonstrate how to use an external library in a program created in Kaira, let us assume that we need to work with arbitrary large numbers in example Workers. GMP[1] is an open source library for arbitrary precision arithmetic. Among other features, it provides class `mpz_class` for arbitrary precision integers.

Only a few modifications are necessary in example Workers to use this library. Because class `mpz_class` is an external type (Section 3.3.10), the user needs to define several functions. The code is shown in Listing 4.2. It allows us to obtain a version that works with arbitrary large integers by replacing `int` to `mpz_class` in the net and C++ codes where jobs are involved. Namely, the types of places *counter* and *results* and members of struct `Job` have to be changed together with some minor alterations of codes in transitions *compute* and *write result*.

[1] `http://gmplib.org/`

Listing 4.1: The head-code for example Workers

```
struct Job {
    Job(int start, int end) : start(start), end(end) {}
    int start;
    int end;
};

namespace ca {
    std::string token_name(const Job &job) {
        std::stringstream s;
        s << "Job [" << job.start << "," << job.end << ")";
        return s.str();
    }

    void pack(Packer &packer, const Job &job) {
        /* direct_pack serializes trivially copyable types */
        direct_pack(packer, job);
    }

    template<> Job unpack(Unpacker &unpacker) {
        /* deserializes trivially copyable types */
        return direct_unpack<Job>(unpacker);
    }
}
```

Listing 4.2: The head-code for the GMP variant of Workers example

```
#include <gmpxx.h>

struct Job {
    Job(const mpz_class &start, const mpz_class &end)
        : start(start), end(end) {}
    mpz_class start;
    mpz_class end;
};

namespace ca {

    std::string token_name(const mpz_class &obj) {
        return obj.get_str(10);
    }

    void pack(Packer &packer, const mpz_class &obj) {
        ... serialization of mpz_class ...
    }

    template<> mpz_class unpack(Unpacker &unpacker)  {
        ... deserialization of mpz_class ...
    }

    std::string token_name(const Job &job) {
        std::stringstream s;
        s << "Job start=" << job.start.get_str(10)
            << " end=" << job.end.get_str(10);
        return s.str();
    }

    void pack(Packer &packer, const Job &job) {
        pack(packer, job.start);
        pack(packer, job.end);
    }

    template<> Job unpack(Unpacker &unpacker) {
        mpz_class start = unpack<mpz_class>(unpacker);
        mpz_class end = unpack<mpz_class>(unpacker);
        return Job(start, end);
    }

}
```

4.2 Example: Heat flow

The example in this section solves a simple heat flow problem on a surface of a cylinder. The borders of its lateral area have a fixed temperature and one fixed point in the area is heated. The goal is to compute a distribution of temperatures on the lateral area. In the presented solution, the surface is divided into discrete points in a grid as it is depicted in Figure 4.2. Temperatures are computed by an iterative method; a new temperature of a point is computed as an average temperature of its surrounding four points.

This approach can be easily parallelized by splitting the grid into parts; each part is assigned to one process. In this example, we assume that the grid is split by vertical cuts. No communication is needed to compute new temperatures of inner points of the assigned area. To compute temperatures in the top and bottom row, the process needs to know rows directly above and below this area. Therefore each process exchanges its border rows with neighbors in each iteration.

The implementation in Kaira is depicted in Figure 4.3. Transition *Compute* executes a single iteration of the algorithm. It takes its own assigned part of the grid (from place *Local data*) and two rows, one from the neighbor above (place *Up row*) and one from below (place *Down row*). The transition computes new temperatures and sends top and bottom rows to neighbors. Function `to_down` is a simple function defined in the head-code as:

```
int to_down(ca::Context &ctx) {
    return (ctx.process_id() + 1) % ctx.process_count()
}
```

Function `to_up` is defined analogously. When the desired number of iterations is reached (parameter `LIMIT`) then results are sent to process 0. When all parts arrive, process 0 assembles the complete grid and writes it into a file.

The class `DoubleMatrix` is a simple implementation of a grid where two values are remembered for each point at the same time (an original value and a newly computed value) and these values can be switched efficiently.

4.2.1 Rectangle variant

In heat flow example, communication is symmetric for all processes. Now we will assume a rectangular 2D area, where points on all four edges have fixed temperatures. In this case, symmetry of communication does not hold. The top and bottom part of the grid are not connected, and therefore the first and the last process do not need exchanges of rows.

The net solving this problem is depicted in Figure 4.4. The asymmetry is expressed by conditional arcs (keyword `if` in inscription configurations). Functions `is_first` and `is_last` are simple functions defined in the head-code; they simply

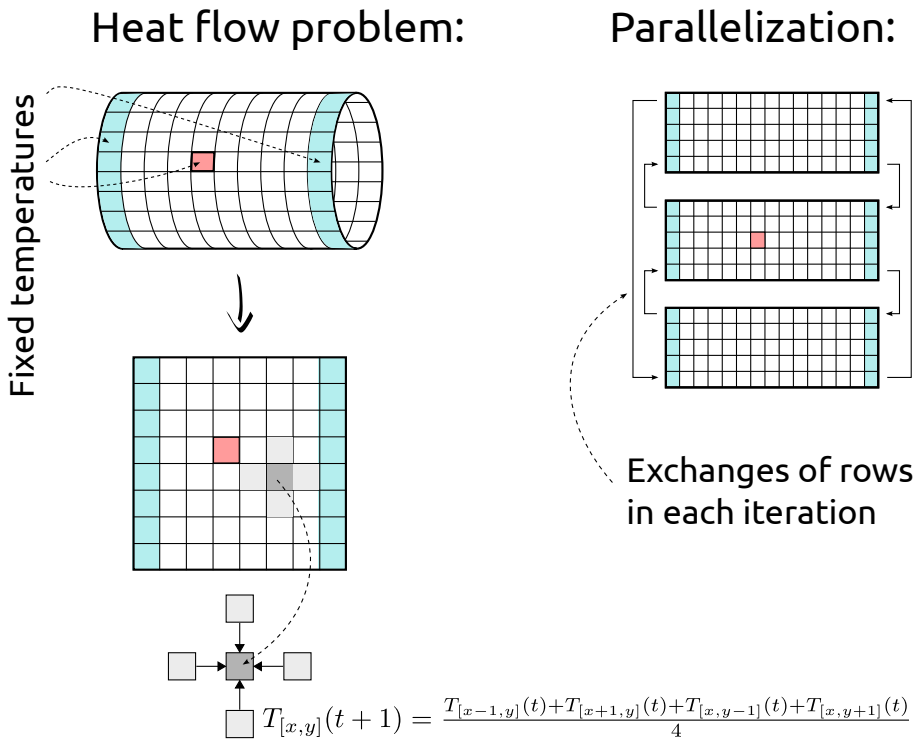


Figure 4.2: Computing a heat distribution on a cylinder

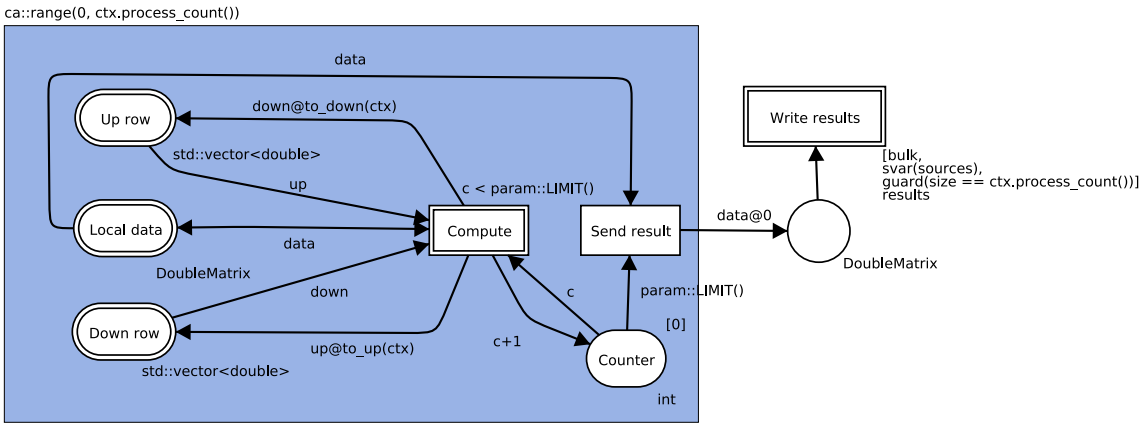


Figure 4.3: The net of the heat flow example

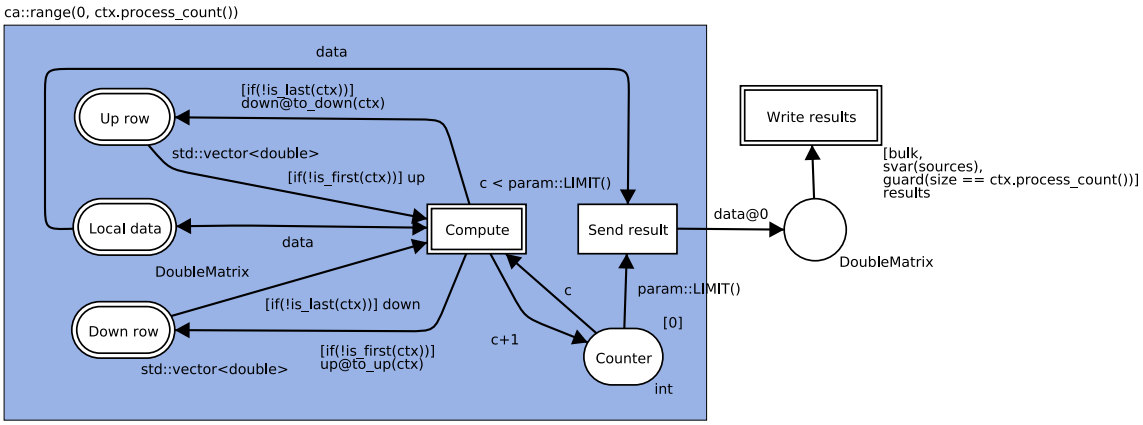


Figure 4.4: The net for rectangle variant of the heat flow example

check the process rank. The first (last) process does not expect a token in place *Up row* (*Down row*) and does not send the token into place *Down row* (*Up row*). The other processes communicate with two neighbors as in the original variant. The C++ code in transition *Compute* is slightly modified to set fixed temperatures in the top (bottom) row in the first (last) process. Additionally, the token is not initialized for the first (last) process in place *Up row* (*Down row*).

4.3 Example: Heat flow & load balancing

The following example is based on the heat flow example. The basic idea remains the same: a parallel computation of a temperature distribution by an iterative method. But in this variant, load balancing of the computation is implemented. Rows of the grid are not distributed to processes statically; rather, the distribution is changed in time according current process performances. This example implements a decentralized variant of load balancing, i.e. no central arbiter is involved. Every process balances itself only in cooperation with its neighbors. Each process periodically compares its own performance with that of its neighbors, and if an imbalance is detected then some rows are transferred to a faster neighbor.

Figure 4.5 shows the implementation of the algorithm in Kaira and Listing 4.3 contains declarations of used data types. Besides parameters used in the original heat flow example, a new parameter `LB_PERIOD` has been added. It determines how often (in the number of iterations) balancing is performed. When balancing occurs, *Compute* does not send border rows but sends its own performance information to its neighbors; that is, the time spent in the computing phase and the number of rows in its own part of the grid. The transition *Balance* determines how many rows are needed to exchange for balancing computational times. The formula is

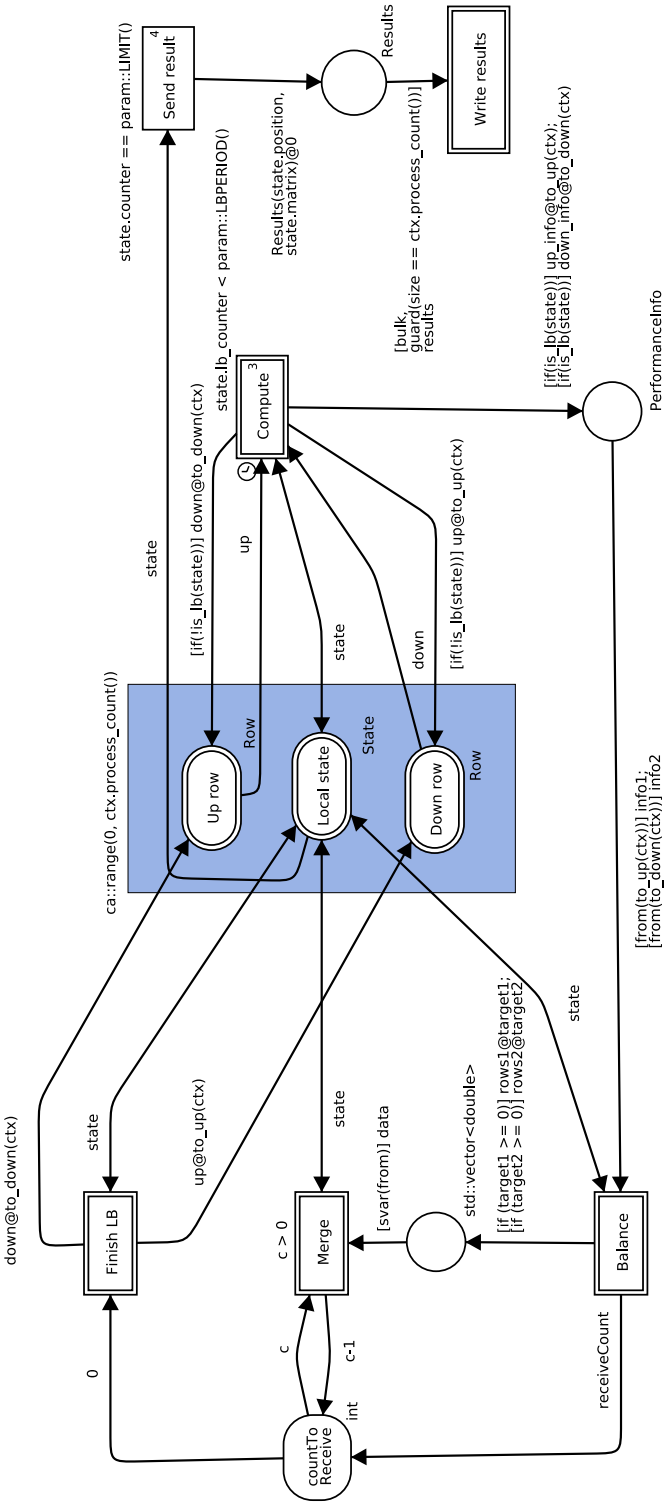


Figure 4.5: The net for the heat flow with load balancing example

based on solving the equation: $\frac{l_m-\Delta}{s_m} = \frac{l_n+\Delta}{s_n}$, where Δ is the number of rows that should be sent from the process to the neighbor process, l_m (l_n) is the number of own (neighbor's) rows, and s_m (s_n) is own (neighbor's) performance – a number of rows computed per second. If $\lfloor\Delta\rfloor > 0$ then *Balance* sends rows to the neighbor. In each process, place *countToReceive* indicates how many neighbors will send their rows to this process. This value is monitored because it is important not to resume the computation in a process until balancing with both neighbors is resolved; otherwise, the process would work with an invalid part of a grid and send wrong border rows. Transition *Merge* adds received rows into the local part of the grid. Transition *EndOfLB* finishes the balancing, it can be fired when all local balancing row exchanges are processed (i.e. *countToReceive* contains zero). The transition resets variable `lb_counter` and sends border rows to its neighbors; therefore the normal computation is resumed.

The rest of the program runs almost like in the original heat flow example. Each process has to just remember its own current position in the grid to heat the fixed point, and for the final assembling of the complete grid at the end. The clock symbol in the left side of transition *Compute* means that the transition gains access to a clock. It is used to measure how much time was spend on computations in transition *Compute*. This clock is explained later in Section 6.4.

4.4 Example: Matrix multiplication

In this section, the matrix multiplication problem is solved; parallelization is achieved by a simple Cannon's algorithm [60]. In this problem we are assuming that processes are arranged into a square $r \times r$ and each process can be uniquely addressed by a pair $(i, j) \in \{1, \ldots, r\}^2$. Let $\mathbf{A}$ and $\mathbf{B}$ be input matrices and $\mathbf{AB}$ is computed. For the sake of simplicity, it is assumed that $\mathbf{A}$ and $\mathbf{B}$ are square matrices and r divides dimensions of $\mathbf{A}$ and $\mathbf{B}$. In the algorithm, matrices $\mathbf{A}$ and $\mathbf{B}$ are partitioned into $r \times r$ sub-blocks. Let $a_{i,j}, b_{i,j}, c_{i,j}$ be local variables of process (i, j). The algorithm proceeds as follows. At the beginning, each process (i, j) obtains a block i, j of both input matrices. A sub-block of $\mathbf{A}$ ($\mathbf{B}$) is saved into $a_{i,j}$ ($b_{i,j}$) and $c_{i,j}$ is set to a zero matrix. The algorithm now works in r iterations. In each iteration, each process (i, j) computes $c_{i,j} = c_{i,j} + a_{i,j}b_{i,j}$ and then the value of $a_{i,j}$ is sent to process $(i + 1 \bmod r, j)$ as a new value of $a_{i+1 \bmod r,j}$ and $b_{i,j}$ is sent to $(i, j + 1 \bmod r)$ as a new content for $b_{i,j+1 \bmod r}$. The communication scheme is depicted in Figure 4.6 for the case $r = 4$. When a process receives new values of $a_{i,j}$ and $b_{i,j}$, then a new iteration starts. At the end of the algorithm, the result is obtained by composing blocks $c_{i,j}$.

The net solving this problem is depicted in Figure 4.7. All examples except this one are created as stand-alone programs; this example serves to demonstrate the creation of libraries. Libraries will be covered in Section 6.6 in more detail. This net

Listing 4.3: Head-code for the example of heat flow with load balancing

```
struct PerformanceInfo {
    long time; /* Duration of computations in
                  the last balancing period (ms) */
    int rows;  /* Number of rows processed in
                  the last balancing period */
};

typedef std::vector<double> Row;
typedef std::vector<Row> Rows;

struct State {
    State(int size_x, int size_y, int position) :
        matrix(size_x, size_y),
        position(position),
        counter(0),
        lb_counter(0),
        time_sum(0) {}
    DoubleMatrix matrix;
    int position;   /* The position of the local grid part */
    int counter;    /* The counter of iterations */
    int lb_counter; /* Number of iterations
                       from last load balancing */
    int time_sum;   /* Time spent in the computational phase
                       from last load balancing */
};

struct Results {
    Results(int position, const DoubleMatrix &matrix)
    : position(position), matrix(matrix) {}

    int position;
    DoubleMatrix matrix;
};
```

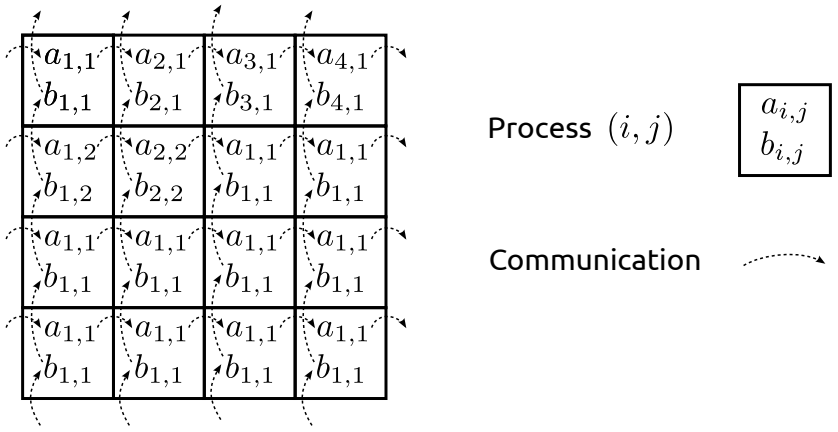


Figure 4.6: The communication scheme of Cannon's algorithm

represents a function with two inputs (`m1` and `m2`) and one output (`output`). When the function is called then values provided as the input are placed into input places. The normal initialization of these places is ignored. When the net is terminated, the resulting value of the function call is taken from the output place.

In this example, OCTAVE's C++ API is used. Integration with OCTAVE is covered in Section 6.6.2. This API provides class `Matrix` as a data type that represents a dense matrix. KAIRA offers basic integration with OCTAVE out of box; therefore explicit definitions of functions `token_name`, `pack`, and `unpack` for this class are not needed.

Places named *input* serve to store the input. The transition *distribute* sends blocks of both matrices to processes. The code in the transition is shown in Listing 4.4. When a process receives both initially assigned blocks, transition *prepare* is fired. It prepares a resulting matrix of the appropriate size into place *result*. After this step, transition *compute* is repeatedly fired once per iteration. The C++ code inside the transition is simple: `var.result += var.m1 * var.m2;`. The rest of the net serves to gather the results and put them into place *output* from which the resulting value is taken when the computation is terminated. Transition *compose output* assembles the final matrix; it takes one of the input matrix to reuse its memory.

4.5 Example: Sieve

This example contains a parallel implementation of *Sieve of Eratosthenes*, a well-known algorithm for finding prime numbers up to a given limit. The net solving this problem is shown in Figure 4.8. Class `Segment` represents a bit array that contains marks (prime/composite number) for a range of numbers. Each process holds its

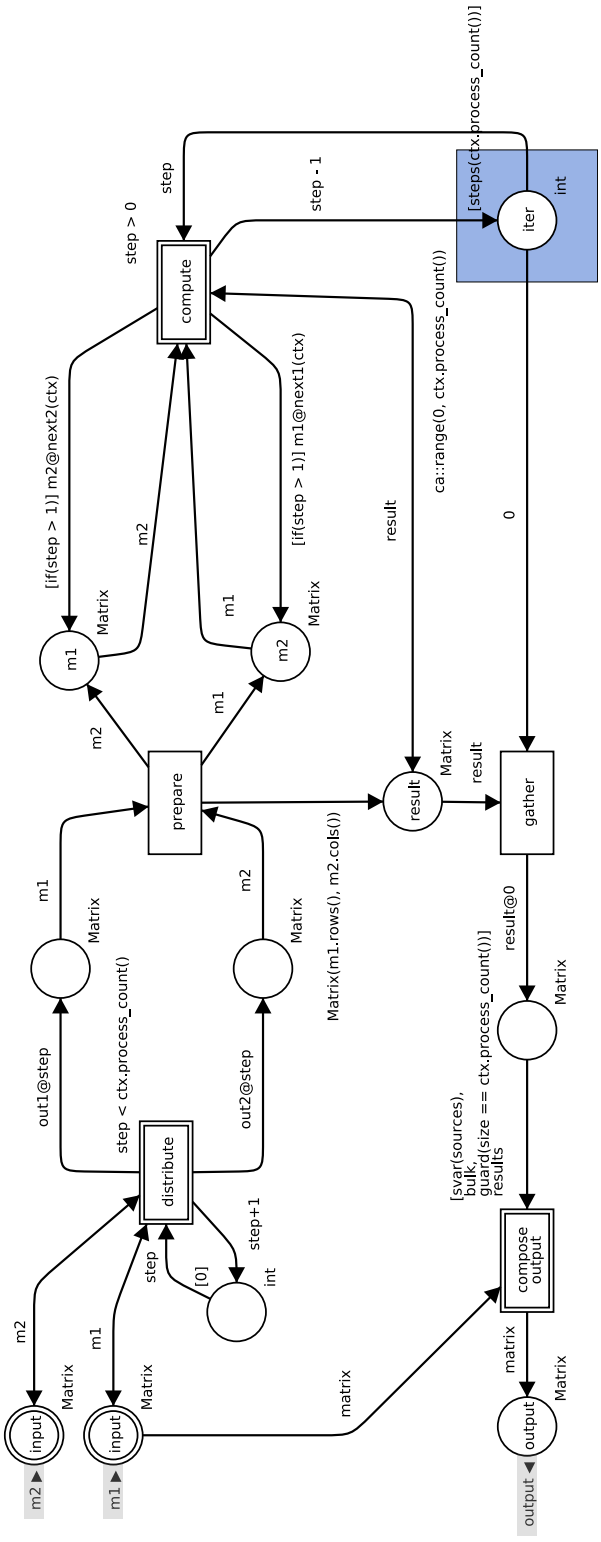


Figure 4.7: The net for the example of matrix multiplication

Listing 4.4: The code inside transition *distribute* in the matrix multiplication example

```
struct Vars {
    Matrix &m1;
    Matrix &m2;
    Matrix &out1;
    Matrix &out2;
    int &step;
};

void transition_fn(ca::Context &ctx, Vars &var)
{
    // It is assumed that process_count is a square number
    int n = sqrt(ctx.process_count());
    // Size of block
    int rs = var.m1.rows() / n;
    int cs = var.m1.cols() / n;
    // Position of block
    int r = (var.step / n) * rs;
    int c = (var.step % n) * cs;
    // Take sublocks from matrices m1 and m2
    var.out1 = var.m1.extract_n(r, c, rs, cs);
    var.out2 = var.m2.extract_n(r, c, rs, cs);
}
```

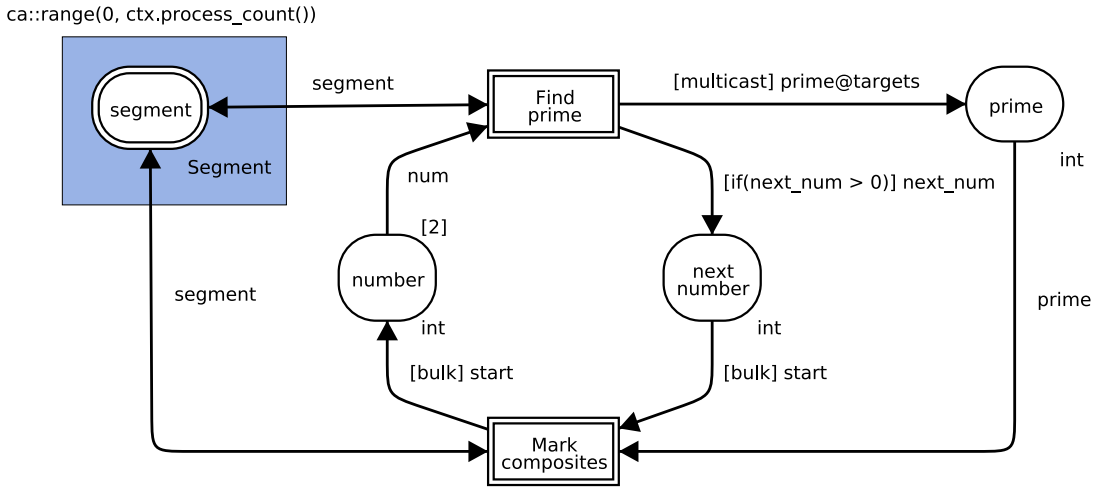


Figure 4.8: The net for the sieve example

own instance of `Segment` in place *Segment*. The process that has a token in place *number* finds prime numbers in own segment. When a prime number is generated, it is written on the standard output and the process puts it to own place *prime* and sends it to all processes with a bigger rank. Transition *Mark composites* takes the prime number and marks all its multiples as composites in the range of the process's segment. Places *number* and *next number* control how far the computation is and if the prime number is generated or composites are marked. When the process generating prime numbers reaches the end of its own interval, then the control is passed to the next process. When the processed number reaches the square root of the target number, then prime numbers are just written and not propagated to other processes. The control is passed from one process to another until the segments are searched.

4.6 Example: Ant colony optimization

The example in this section implements a variant of *ant colony optimization* (ACO) [61]. ACO is a nature-inspired meta-heuristic algorithm with widespread use. It is a successful tool with practical applications in the areas such as optimizations and scheduling. This example uses ACO to obtain a solution for *Traveling Salesman Problem* (TSP) [62]. The ACO is used in the island variant, i.e. ants are separated into colonies – one per each process. Each colony evolves independently on the others. When a colony finds a better solution than the previous best-known, it sends this information to others.

The implementation in Kaira is shown in Figure 4.9. A colony is stored in the place in the top-left corner. The transition *Compute* takes a colony and computes

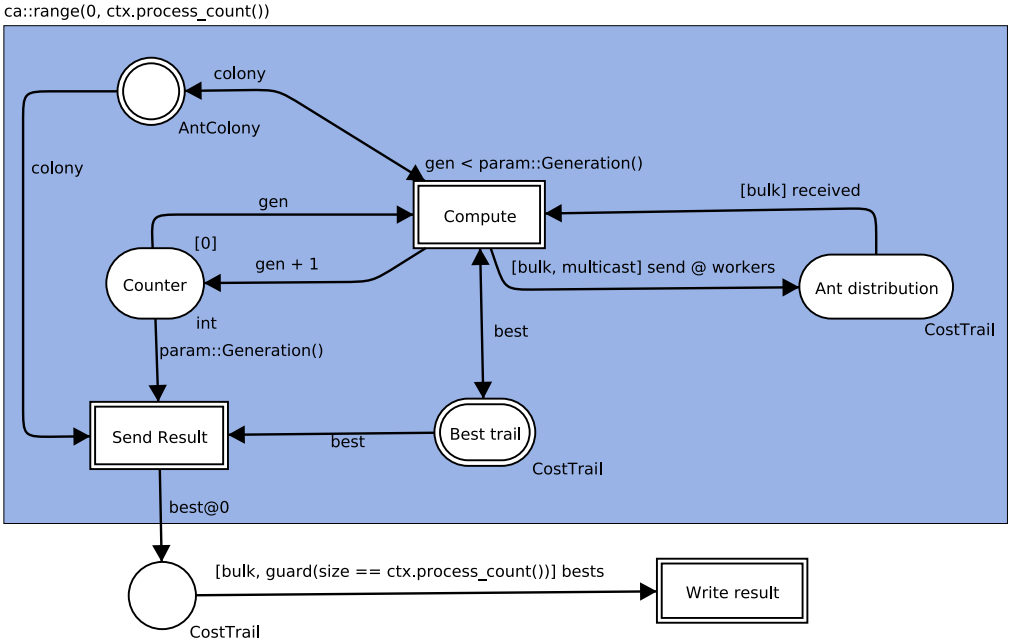


Figure 4.9: The net for the ant colony optimization example

a new generation of ants. In each iteration, every process saves the best solution to place *Best trail*. When this solution is better than the previous one, it is distributed to others through place *Ant distribution*. Transition *Compute* takes all tokens in this place; if it is nonempty then it embodies new ants into the colony assigned to the process. If it is empty or not, the transition also computes a new generation of ants. When the last generation is computed, *Send results* takes the best solution in the process and sends it to process 0, where the overall best solution is chosen.

Chapter 5

Formal semantics

This part provides the formal semantics of Kaira. In other words, the behavior of programs created in Kaira is described in a rigorous way. The user does not need formal semantics for everyday programming, and many commonly used systems do not provide any formal description; MPI is also one of them. But formal semantics generally allows deeper reasoning about the system, and in this work it allows us to provide an exact description of analyses, especially in the case of verification.

This section starts with basic definitions followed by the definition of *basic transition system* (BTS) (Section 5.2). It will serve as an underlying formalism for *Kaira program* (KP) (Section 5.3). KP is a straightforward formalization of the visual language as was introduced in Chapter 3. BTS is inspired by *Variable Transition System* from [63]. It can be seen as an abstraction of Coloured Petri nets, where tokens are stored in lists instead of multisets. The behavior of a KP for a specific number of processes is given by the translation of KP into BTS described in Section 5.4.

5.1 Basic definitions

The set of natural numbers is denoted as $\mathbb{N} = \{0, 1, \dots\}$ and $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$. If $n \in \mathbb{N}$ then $[n] = \{1, 2, \dots, n\}$. If A is a set then $|A|$ denotes the cardinality of A and A^* denotes the set of all finite sequences of elements of A. An element of A^* can be written as $\langle a_1, a_2, \dots, a_n \rangle$ where $\forall i \in [n] : a_i \in A$. The empty sequence is denoted as $\langle\rangle$. The functions for working with sequences are defined as follows: Let $u, v \in A^*$ and $u = \langle a_1, a_2, \dots, a_n \rangle, v = \langle b_1, b_2, \dots, b_m \rangle$ then $|u| = n$, $a \in u \Leftrightarrow \exists i \in [n] : a_i = a$, $u \cdot v = \langle a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m \rangle$. If $i \in [n]$ then $at(u, i) = a_i$. The function $remove : A^* \times 2^{\mathbb{N}} \to A^*$ removes elements from a sequence at given indices, formally $remove(u, \{i_1, i_2, \dots, i_k\}) =$

$$\langle a_1, a_2, \dots, a_{i_1-1}, a_{i_1+1}, \dots, a_{i_2-1}, a_{i_2+1}, \dots, a_{i_k-1}, a_{i_k+1}, \dots a_{n-1}, a_n \rangle$$

(without loss of generality it is assumed that x_i are sorted in ascending order). The set of all sequences of elements of A where each element is used exactly once is denoted as $sq(A) = \{u \in A^* \mid (\forall a \in A : a \in u) \wedge |u| = |A|\}$. A domain of a function f is denoted as $dom(f)$. The empty mapping is denoted as $\varnothing$, i.e. $dom(\varnothing) = \emptyset$.

In contrast to the previous informal description of semantics, no particular programming language is considered. Only basic properties and sets related to a programming language are defined in order to obtain a more abstract definition and also to overcome the difficulties of dealing with the full semantics of C++. The reader may safely imagine C++ or a similar language behind the following definitions. The approach of describing semantics independently on any particular language is inspired by the definition of CPNs in [56].

Let Exprs be a set of all expressions of a programming language, Vars be a set of all variables ($\mathsf{Vars} \subset \mathsf{Exprs}$), and $\mathcal{V}$ be a set of all possible values. The type system of the used programming language is ignored in the rest of this chapter, because the semantics is given independently on it. Expanding the formalism to include a type system is possible and straightforward when needed, but does involve technical work that is not necessary for understanding the executional semantics of programs in Kaira.

Let $B = \mathsf{Vars} \to \mathcal{V}$ be the set of all *bindings*. The partial function $eval : \mathsf{Exprs} \times B \to \mathcal{V}$ evaluates a given expression under a given binding. The function $var : \mathsf{Exprs} \to 2^{\mathsf{Vars}}$ returns free variables in a given expression.

For technical reasons, the following is assumed:

$$\texttt{ctx} \in \mathsf{Vars}, \mathbb{N} \subset \mathcal{V} \text{ and } \{\texttt{true}, \texttt{false}, \bullet, \square\} \subset \mathcal{V}$$

where `true` and `false` are standard boolean values, $\bullet$ will be used as a unit value and $\square$ means a default value (if we consider a type system, there will be more such values for each type. In the context of C++, values created by default constructors are meant). We also assume that $\mathcal{V}$ is closed under "sequentization"; hence $\langle\rangle \in \mathcal{V}$ and if $a_1, \ldots, a_n \in \mathcal{V}$ then $\langle a_1, \ldots, a_n \rangle \in \mathcal{V}$.

5.2 Basic transition system

A *basic transition system* (BTS) is a tuple $\mathcal{B} = (P, T, Take, Put, PR, PG, m_0)$, where P is a finite set of places, T is a finite set of transitions and $T \cap P \neq \emptyset$. Functions $Take$ and Put define the behavior of transitions. $Take : T \times (P \to \mathcal{V}^*) \to (P \to 2^{\mathbb{N}}) \cup \{\bot\}$ where $\bot$ is a special symbol that is used in the meaning "not enabled". $Put : T \times (P \to \mathcal{V}^*) \to (P \to \mathcal{V}^*)$. $PR : T \to \mathbb{N}$ assigns priorities to transitions, $PG \subseteq T \times T$ is an equivalence relation (reflexive, symmetric, transitive) decomposing transitions into *priority groups*. A *marking* represents a state of the system, it is a mapping $P \to \mathcal{V}^*$; m_0 is the *initial marking*. A transition t is *enabled* in a marking

m (denoted as $m \xrightarrow{t}$) if

$$Take(t,m) \neq \bot \text{ and } \forall t' \in T : (t,t') \in PG \wedge PR(t) < PR(t') \implies Take(t',m) = \bot$$

The behavior of BTS is defined by a set of relations $(\xrightarrow{t})_{t \in T}$ such that $\forall t \in T : \xrightarrow{t} \subseteq (P \to \mathcal{V}^*)^2$ and $(m,m') \in \xrightarrow{t}$ iff t is enabled in m and $\forall p \in P$ holds:

$$m'(p) = remove(m(p), Take(t,m)(p)) \cdot Put(t,m)(p)$$

Let $\to = \{(m,m') \mid \exists t \in T : (m,m') \in \xrightarrow{t}\}$ and let $\to^*$ be the reflexive and transitive closure of $\to$. Instead of writing $(m,m') \in \xrightarrow{t}$, $(m,m') \in \to$, $(m,m') \in \to^*$, the notations $m \xrightarrow{t} m', m \to m', m \to^* m'$ will be used. The notation $m \xrightarrow{t} m'$ is read as "t is fired in m and system ends in marking m'".

5.3 Kaira program

A *Kaira program* (KP) is a tuple $\mathcal{K} = (P, T, A, E, G, PR, C, I, R)$ where P is a finite set of places, T is a finite set of transitions and $T \cap P \neq \emptyset$, $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $G : T \to \mathsf{Exprs}$ assigns guard expressions to transitions, $PR : T \to \mathbb{N}$ assigns priorities to transitions, $C : T \to (B \to B \times \{running, quit\})$ represents a sequential computation in a transition (fire-code), i.e. it assigns a function to a transition that takes binding and returns new binding and an indication if the program continues or if it was stopped (i.e. `ctx.quit()` was called). If a transition t has no fire-code than $C(t) = \lambda x.(x, running)$). Mapping $I : P \to \mathsf{Exprs}$ assigns initial expressions such that $eval(I(p), \varnothing)$ is a sequence for each $p \in P$. The set $R \subseteq \mathsf{Exprs} \times 2^P$ defines init-areas and has to satisfy $\forall (e,X) \in R : eval(e,\varnothing) \in \mathbb{N}^*$. Mapping $E : A \to \mathsf{Ins}^*$ assigns expressions on arcs, where Ins is a set of all inscriptions as defined in Section 3.3.3. The function $expr : \mathsf{Ins} \to \mathsf{Exprs}$ returns the main-expression of a given inscription, $target : \mathsf{Ins} \to \mathsf{Exprs} \cup \{\bot\}$ returns the target, symbol $\bot$ is a special fresh symbol such that $\bot \notin \mathsf{Exprs}$ and it is used in the case that the target is not defined. For all $(p,t) \in A \cap (P \times T) : target(E(p,t)) = \bot$. The predicates $bulk(\iota)$ and $multicast(\iota)$ where $\iota \in \mathsf{Ins}$ are true iff ι contains a keyword of the same name in configuration items. Functions *filter*, *guard*, *if* return expressions in configuration items `filter`,`guard`,`if`; if a given inscription does not contain a configuration item of that name then the corresponding function returns expression `true`. The function $svar : \mathsf{Ins} \to \mathsf{Vars} \cup \{\bot\}$ returns an expression from a configuration item `svar` or $\bot$ if the configuration item is not used. Function $seq : \mathsf{Ins} \to \mathbb{N}$ returns an integer from configuration item `seq` or 0 when `seq` is not used. Let $vars(\iota) \stackrel{\text{def}}{=} var(expr(\iota)) \cup var(filter(\iota)) \cup var(if(\iota)) \cup svar(\iota) \setminus \{\bot\}$ and $dvars(\iota) \stackrel{\text{def}}{=} (expr(\iota) \cup svar(\iota)) \cap \mathsf{Vars}$.

A sequence a is a *well-ordered input for* $t \in T$ if

$$a = \langle (p_1, x_1), (p_2, x_2), \ldots, (p_n, x_n) \rangle \in sq(A')$$

where $A' = \{(p, x) \mid \exists (p, t) \in A \wedge x \in [|E((p, t))|]\}$ such that $\forall i \in [|a|] : (\forall j \in \{i + 1, \ldots, |a|\} : p_i = p_j \implies x_i < x_j) \wedge (\forall v \in vars(at(E(p_i, t), x_i)), \exists j \in [i] : v \in dvars(at(E(p_j, t), x_j)))$.

For each KP, there has to be a function $InputInscriptions : T \to (P \times \mathbb{N})^*$ such that for each $t \in T$: $InputInscriptions(t)$ is a well-ordered input for t. There may exist more valid well-ordered input sequences for a transition, but later in Proposition 1, it will be shown that the semantics is independent for a particular choice.

In the same sense, function *OutputInscriptions* has to exist. It assigns a sequence of output inscriptions to a transition. Formally

$$OutputInscriptions(t) = \{(p_1, x_1), \ldots, (p_n, x_n)\} \in sq(A')$$

where $A' = \{(p, x) \mid \exists (t, p) \in A \wedge x \in [|E((t, p))|]\}$ and $\forall i \in [|OutputInscriptions(t)|] : (\forall j \in \{i, \ldots, |OutputInscriptions(t)|\} : seq(at(E(p_i, t), x_i)) \leq seq(at(E(p_j, t), x_j)) \wedge (p_i = p_j \implies x_i < x_j))$.

For technical reasons we also assume that the name of each place can be expressed in our programming language, i.e. we assume that $P \subset \mathcal{V}$.

5.4 Instantiation of Kaira program

For the rest of this section, a Kaira program $\mathcal{K} = (P, T, A, E, G, PR, C, I, R)$ and an integer $n \in \mathbb{N}_+$ is fixed. Now, we define a BTS that will represent a run of $\mathcal{K}$ on n processes. An *n-instantiation* of $\mathcal{K}$ is

$$\mathcal{B}_n^{\mathcal{K}} = (P', T', Take', Put', PR', PG', m_0')$$

such that

$$P' = \{p_i \mid p \in P; i \in [n]\} \cup \{\mathbf{p}_{j \to i} \mid i, j \in [n]\} \cup \{\tau_i \mid i \in [n]\} \cup \{\mathbf{p}_{run}\}$$

where p_i are places from KP instantiated for each process i; $\mathbf{p}_{j \to i}$ represents a unidirectional communication channel from process j to process i; τ_i stores what a process i is doing; and $\mathbf{p}_{run}$ is a global flag if the program is still running.

$$T' = \{t_{i,x} \mid t \in T, i \in [n], x \in \{+, -\}\} \cup \{\mathbf{r}_{i,j,+} \mid i, j \in [n], x \in \{+, -\}\} \cup \{\mathbf{r}_{i,-} \mid i \in [n]\}$$

First the behavior of $t_{i,+}$ ($t \in T, i \in [n]$) will be defined. Let m be a marking for the rest of this section. The meaning of $t_{i,+}$ is "A transition t is started in a process i".

$Take'(t_{i,+}, m) = k$ where $(k, b) = FindBinding(t, i, m)$. The function *FindBinding* is defined in Algorithm 5.1. $Put(t_{i,+}, m) = \lambda p.\text{if } p = \tau_i \text{ then } \langle (t, b) \rangle \text{ else } \langle\rangle$, where $(k, b) = FindBinding(t, i, m)$.

The meaning of $t_{i,-}$ ($t \in T, i \in [n]$) is "Transition t has been finished in process i". $Take(t_{i,-}, m) = \lambda p.(\text{if } p = \tau_{i,+} \vee p = \mathbf{p}_{run} \text{ then } \{1\} \text{ else } \emptyset)$ if $\exists b \in B : m(\tau_i) = \langle (t, b) \rangle$ and $m(\mathbf{p}_{run}) = \langle \bullet \rangle$ otherwise $Take(t_{i,-}, m) = \bot$. $Put(t_{i,-}, m) = PutTokens(t, i, m)$ where function *PutTokens* is defined in Algorithm 5.3.

The meaning of $\mathbf{r}_{i,j,+}$ ($i, j \in [n]$) is "Process i starts to receive tokens from process j". If $m(\tau_i) = \langle \bullet \rangle$ and $|m(\mathbf{p}_{j \to i})| > 0$ then $Take(\mathbf{r}_{i,j,+}, m) = \lambda p.(\text{if } p = \tau_i \vee p = \mathbf{p}_{j \to i} \text{ then } \{1\} \text{ else } \emptyset)$ otherwise $Take(\mathbf{r}_{i,j,+}, m) = \bot$. $Put(\mathbf{r}_{i,j,+}, m) = \lambda p.\text{if } p = \tau_i \text{ then } \langle m(\mathbf{p}_{j \to i}) \rangle \text{ else } \langle\rangle$.

The meaning of $\mathbf{r}_{i,-}$ ($i, j \in [n]$) is "Process i has finished receiving". If $\exists x \in P \times (\mathcal{V} \times \mathbb{N})^* : m(\tau_i) = \langle x \rangle$ then $Take(\mathbf{r}_{i,-}, m) = \lambda p.\text{if } p = \tau_i \text{ then } \{1\} \text{ else } \emptyset$ otherwise $Take(\mathbf{r}_{i,-}, m) = \bot$. $Put(\mathbf{r}_{i,-}, m) = \lambda p.\text{if } p = p' \text{ then } k \text{ else } \langle\rangle$ where $\langle (p, k) \rangle = m(\tau_i)$ and $(p', \_) = x$.

The initial marking of BTS is defined as:

$$m'_0(p) = \begin{cases} eval(I(p'), \varnothing) & \text{if } p = p'_i \wedge \exists (e, X) \in R : p' \in X \wedge i \in eval(e, \varnothing) \\ \langle \bullet \rangle & \text{if } (p = \tau_i \wedge i \in [n]) \vee p = \mathbf{p}_{run} \\ \langle\rangle & \text{otherwise} \end{cases}$$

Priorities are assigned as follows. Let $t_1, t_2 \in T'$ then $(t_1, t_2) \in PG'$ iff

$$t_1 = t_2 \vee (t_1 = t'_{i,+} \wedge t_2 = t''_{i',+} \wedge i = i')$$

and $PR'(t') = PR(t)$ if $t' = t_{i,+}$ otherwise $PR'(t') = 0$.

The following proposition says that the semantics is independent on a particular evaluation order of input inscriptions. Hence we do not need an equivalent of configuration item `seq` for input inscriptions.

Proposition 1. *Assume that there are InputInscriptions′ and InputInscriptions″, both assigning well-ordered input sequences, and there are FindBinding′ that uses InputInscriptions′ and FindBinding″ that uses InputInscriptions″. For all $t \in T, i \in [n], m \in P \to \mathcal{V}^*$, it holds that $FindBinding'(t, i, m) = FindBinding''(t, i, m)$.*

Proof. Assume that for some $t \in T$:

$$InputInscriptions'(t) = \langle a_1, \ldots, a_m \rangle$$

$$InputInscriptions''(t) = \langle a_1, \ldots, a_{f-1}, a_g, a_{f+1}, \ldots, a_{g-1}, a_f, a_{g+1}, \ldots, a_m \rangle$$

where $1 \leq f < g \leq m$, $a_f \neq a_g$ and $(\overline{p}_i, x_i) = a_i$. In other words, both functions return almost the same sequence; only two elements (a_f and a_g) are swapped. From the definition of well-ordered input sequence follows that $\overline{p}_f \neq \overline{p}_g$ and $\forall i \in \{f + 1, \ldots, g - 1\} : \overline{p}_f \neq \overline{p}_i \wedge \overline{p}_g \neq \overline{p}_i$. Therefore $k'_{\overline{p}_i} = k''_{\overline{p}_i}$ for $i \in \{f + 1, \ldots, g - 1\}$

has the same content before processing a_i element in the "for cycle" where k'_p (k''_p) is variable k_p from function *FindBinding*$'$ (*FindBinding*$''$). Moreover $k'_{\overline{p}_f}$ before iteration f has the same value as $k''_{\overline{p}_g}$ before processing iteration g (and it also holds in the case when f and g are exchanged). Function *AddVarsAndCheck* ensures that in both cases binding b is always created with same values; otherwise, both cases of *FindBinding* will return $(\perp, \perp)$. Therefore *FindBinding*$' =$ *FindBinding*$''$ for the case if two elements are swapped in well-ordered input sequences.

It remains to be shown that if a and a' are well-ordered input sequences for $t \in T$ then a can be rearranged to a' by series of swapping two elements and all such created sequences are also well-ordered for t. Assume a and a' such that they cannot be rearranged and $a = \langle a_1, \ldots, a_m \rangle$ and $a' = \langle a'_1, \ldots, a'_m \rangle$. Let $f = \min\{i \in [m] \mid a_i \neq a'_i\}$. There has to be $g, g' \in [f+1, m]$ such that $a_g = a'_f$ and $a''_g = a_f$. It is easy to see that a can be rearranged to a' by swapping two elements through intermediate sequences

$$\langle a_1, \ldots, a_f, a_g, a_{f+1}, \ldots, a_{g-1}, a_{g+1}, \ldots, a_m \rangle$$

$$\langle a_1, \ldots, a_g, a_f, a_{f+1}, \ldots, a_{g-1}, a_{g+1}, \ldots, a_m \rangle$$

$$\langle a_1, \ldots, a'_f, a'_{g'}, a'_{f+1}, \ldots, a'_{g'-1}, a'_{g'+1}, \ldots, a'_m \rangle$$

and all these sequences are well-ordered for t. □

5.4.1 Run of a program

The behavior of a program in KAIRA described by $\mathcal{K} = (P, T, A, E, G, PR, C, I, R)$ on n processes is defined by the behavior of $\mathcal{B}^{\mathcal{K}}_n = (P', T', Take', Put', PR', PG', m'_0)$, i.e as n-instantiation of $\mathcal{K}$. A run of the program is as a sequence

$$\langle m_0, t_1, m_1, t_2, \ldots, m_{z-1}, t_z, m_z \rangle$$

such that $m_0 = m'_0$, $\forall t \in T' : m_z :\not\xrightarrow{t}$, and for all $i \in [z]$ holds: $t_i \in T'$, $m_{i-1} \xrightarrow{t_i} m_i$.

Algorithm 5.1 The Definition of function *FindBinding*. Internal functions defined in Algorithm 5.2.

function FINDBINDING(t, i, m)
  if $m(\tau_i) \neq \langle\bullet\rangle \vee m(\mathbf{p}_{run}) \neq \langle\bullet\rangle$ **then**
    return $(\bot, \bot)$ ▷ Thread is not ready
  end if
  $b \leftarrow \{(\texttt{ctx}, (i, n))\}$
  $k_p \leftarrow \emptyset$ for all $p \in P'$
  $k_{\tau_i} \leftarrow \{1\}$ ▷ Take token from τ_i
  for (p, x) in $\mathit{InputInscriptions}(t)$ **do**
    $\iota \leftarrow \mathit{at}(E(p, t), x)$ ▷ Take inscription
    if $\mathit{eval}(\mathit{if}(\iota), b) \neq \texttt{true}$ **then** ▷ Check `if` item
      $b \leftarrow \mathit{AddVars}(b, \iota, (\square, i))$
      continue
    end if
    if $\mathit{bulk}(\iota)$ **then**
      $\langle(v_1, s_1), \ldots, (v_k, s_k)\rangle = m(p)$
      $b \leftarrow \mathit{AddVarsAndCheck}(b, \iota, (\langle v_1, \ldots, v_k\rangle, \langle s_1, \ldots, s_k\rangle))$
      $k_p \leftarrow [|m(p)|]$
    else
      $f \leftarrow \mathit{filter}(\iota)$
      $\alpha \leftarrow \{j \in [|m(p)|] \setminus k_p \mid \mathit{eval}(f, \mathit{AddVars}(b, \iota, \mathit{at}(m(p), j)) = \texttt{true}\}$
      if $\alpha = \emptyset$ **then** ▷ No token passed through filter
        return $(\bot, \bot)$
      end if
      $b \leftarrow \mathit{AddVarsAndCheck}(b, \iota, \mathit{at}(m(p), \min \alpha))$
      $k_p \leftarrow k_p \cup \{\min \alpha\}$ ▷ Works only with the first token from α
    end if
    if $b = \bot$ **then**
      return $(\bot, \bot)$ ▷ A collision in assigning variables
    end if
    if $\mathit{eval}(\mathit{guard}(\iota), b \cup \{(\texttt{size}, |m(p)|)\}) \neq \texttt{true}$ **then**
      return $(\bot, \bot)$ ▷ The guard of the inscription failed
    end if
  end for
  if $\mathit{eval}(G(t), b) \neq \texttt{true}$ **then**
    return $(\bot, \bot)$ ▷ The guard of the transition failed
  end if
  return $(\lambda p.k_p, b)$
end function

Algorithm 5.2 The definitions of functions *AddVars* and *AddVarsAndCheck*

function AddVars(b, ι, (value, source))
  if $expr(\iota) \in \mathsf{Vars} \wedge expr(\iota) \notin dom(b)$ **then**
    $b \leftarrow b \cup \{(expr(\iota), value)\}$
  end if
  if $svar(\iota) \neq \bot \wedge svar(\iota) \notin dom(b)$ **then**
    $b \leftarrow b \cup \{(svar(\iota), source)\}$
  end if
  return b
end function
function AddVarsAndCheck(b, ι, (value, source))
  $b \leftarrow AddVars(b, \iota, (value, source))$
  if $eval(expr(\iota), b) = value \wedge (svar(\iota) \neq \bot \vee eval(svar(\iota), b) = source)$ **then**
    return b
  end if
  return $\bot$
end function

Algorithm 5.3 The definition of function *PutTokens*

function PUTTOKENS(t, i, m)
  $(t, b) \leftarrow at(m(\tau_i), 1)$
  $(b', r) \leftarrow C(p)(b)$ ▷ Call the C++ code in the transition
  $k_p \leftarrow \langle\rangle$ for all $p \in P'$
  if $r = running$ **then**
    $k_{\mathbf{p}_{run}} \leftarrow \langle \bullet \rangle$ ▷ Return $\bullet$ to $\mathbf{p}_{run}$; the application is still running
  end if
  $k_{\tau_i} \leftarrow \langle \bullet \rangle$
  for (p, x) in $OutputInscriptions(t)$ **do**
    $\iota \leftarrow at(E(t, p), x)$
    if $eval(if(\iota), b') \neq \texttt{true}$ **then**
      continue
    end if
    if $bulk(\iota)$ **then**
      $\langle v_1, v_2, \ldots, v_z \rangle = eval(expr(\iota), b')$
      $k = \langle (v_1, i), \ldots, (v_z, i) \rangle$
    else
      $k = \langle (eval(expr(\iota), b'), i) \rangle$
    end if
    if $target(\iota) = \bot \vee eval(target(\iota), b') = i$ **then**
      $k_p \leftarrow k_p \cdot k$
    else
      $t = eval(target(\iota), b')$
      if $multicast(\iota)$ **then**
        for x in t **do**
          $k_{\mathbf{p}_{i \to x}} = k_{\mathbf{p}_{i \to x}} \cdot \langle (p, k) \rangle$
        end for
      else
        $k_{\mathbf{p}_{i \to t}} = k_{\mathbf{p}_{i \to t}} \cdot \langle (p, k) \rangle$
      end if
    end if
  end for
  return $\lambda p.k_p$
end function

Chapter 6

Features of Kaira

This chapter presents features available to the programmer in Kaira. The chapter begins by building a program from a net and evaluating its performance in comparison to a manually created C++ program. This is followed by a description of features for supporting activities that are introduced together with demonstrations on the examples from Chapter 4. The chapter ends by building libraries and integrating Kaira with Octave.

Any usage of Kaira features is preceded by creation of a visual program. Kaira offers an editor that allows the creation, editing, and removal of transitions, places, arcs, and init-areas. All elements can be freely moved and resized. The method of editing the visual program does not deviate from any common diagram editor. The screenshot of the editor was already shown in Figure 3.1. The diagram editor integrates a source code editor; it provides basic features like syntax highlighting or jumping to a specific line where an error is detected, but more advanced features like auto-completion are missing in the current version. The code editor was shown in Figure 3.4.

To formalize some of the presented features, let us fix for the rest of this chapter a Kaira program $\mathcal{K} = (P, T, A, E, G, PR, C, I, R)$, a number of processes $n \in N_+$, and $\mathcal{B}_n^{\mathcal{K}} = (P', T', Take', Put', PR', PG', m'_0)$.

6.1 Generating applications

The basic feature of Kaira as a development environment is to produce executable applications. The resulting applications can be generated in three modes: MPI, threading, and the sequential mode. The first one is the main mode. As the name suggests, it produces an application that uses MPI as its communication back end. The other two modes are designated to utilize external supportive tools that work not well or not at all in the distributed environment of MPI. Both modes emulate the behavior of MPI. The threading mode emulates the MPI layer by Pthreads

instead of stand-alone processes. In the sequential mode, the application is executed sequentially, so that even tools that is designed for sequential applications can be used.

This feature allows easy use of tools like GDB or VALGRIND to debug sequential parts of the application. KAIRA is focused on debugging and analyzing parallelism and communication, and it is assumed that these existing tools are used to analyze sequential codes in transitions. Any KAIRA application can be generated in all three modes without changing the net. The process of generation is fully automatic; details are described in Chapter 7.

6.1.1 Performance of applications

The goal of this section is show that the abstract model used in KAIRA can be effectively translated into applications while introducing only a small overhead. The reason is to allow creation of applications directly usable as productive versions. But even if KAIRA is used "only" as a tool for experiments, it is important to achieve good performance with a small overhead because of the precision of analyses.

All measurements described in this thesis were executed on the following two computers:

- *Anselm* – A cluster where each node is composed of: two Intel Sandy Bridge E5-2665, 8-core, 2.4GHz processors; 64 GB of physical memory[1]. Only a single core in each node was used in experiments to observe the behavior in the distributed-memory environment. The use of Anselm was carried under IT4Innovations Center of Excellence (project CZ.1.05/1.1.00/02.0070).

- *Hubert* – 8 processors AMD Opteron/2500 (total 32 cores), shared memory 128 GB. This computer is owned by the Institute of Geonics[2] and was used with the kind permission of Ondřej Jakl.

In the following text, the implementation of the heat flow example (Section 4.2) is compared with a manual implementation of the same problem. Both implementations share the same computation code. It is about 380 LOC (lines of code without comments). The manually implemented solution contains about 100 LOC not shared with the solution in KAIRA. The following experiments were executed in the two settings:

- Grid size 2600×8200 and 5000 iterations.

- Grid size 2600×2600 and 3000 iterations.

[1] `http://support.it4i.cz/docs/anselm-cluster-documentation/hardware-overview`
[2] `http://www.ugn.cas.cz/`

All experiments were performed without writing results to a file to exclude effects of I/O operations. Measured times of computations for each experiment are depicted in Figures 6.1 and 6.2. Data used to plot these charts are presented in Appendix A. As the data show for this example, the overall scalability of the application and absolute times follow the trend of the manually created solution.

6.2 Simulator

The *simulator* is an integral part of Kaira. It executes the application in a simulated environment with an arbitrary number of processes in a form that is fully controllable by the user. The inner state and the control of the application is presented through the visualization of the net.

One purpose of the simulator is to serve as a debugging tool that allows to observe the application as a single piece with the possibility to go step by step through a run of the application. The simulator is also tightly connected with prototyping. It provides an immediate feedback of the application behavior through the visualization of the application's inner state, without any additional debugging infrastructure. Therefore, it may be used from very early stages of the development with an incomplete application. For example, it can be used to see what data are sent to another process even no actual implementation of the receiving part exists.

Figure 6.3 depicts how a running application is shown to the user. The two types of information are depicted:

1. Tokens in place (The state of the memory)
2. Packets transported between nodes (The state of the communication environment)

It completely describes a distributed state of the application. The user can control the behavior of the application by the two basic actions:

1. Start an enabled transition
2. Start receiving a packet

These two actions provide a complete well-formed control of non-determinism caused by parallel execution. This high-level control produces a smaller number of observable states than a classic debugger that operates on lines of a source code. In spite of this, all possible computational paths remain reachable, because states that are not captured by Kaira are states induced by inner sequential computations. Moreover, all such sequential codes (mostly codes in transitions) can be tested or debugged separately or a sequential application can be generated and debugged by a classic debugger.

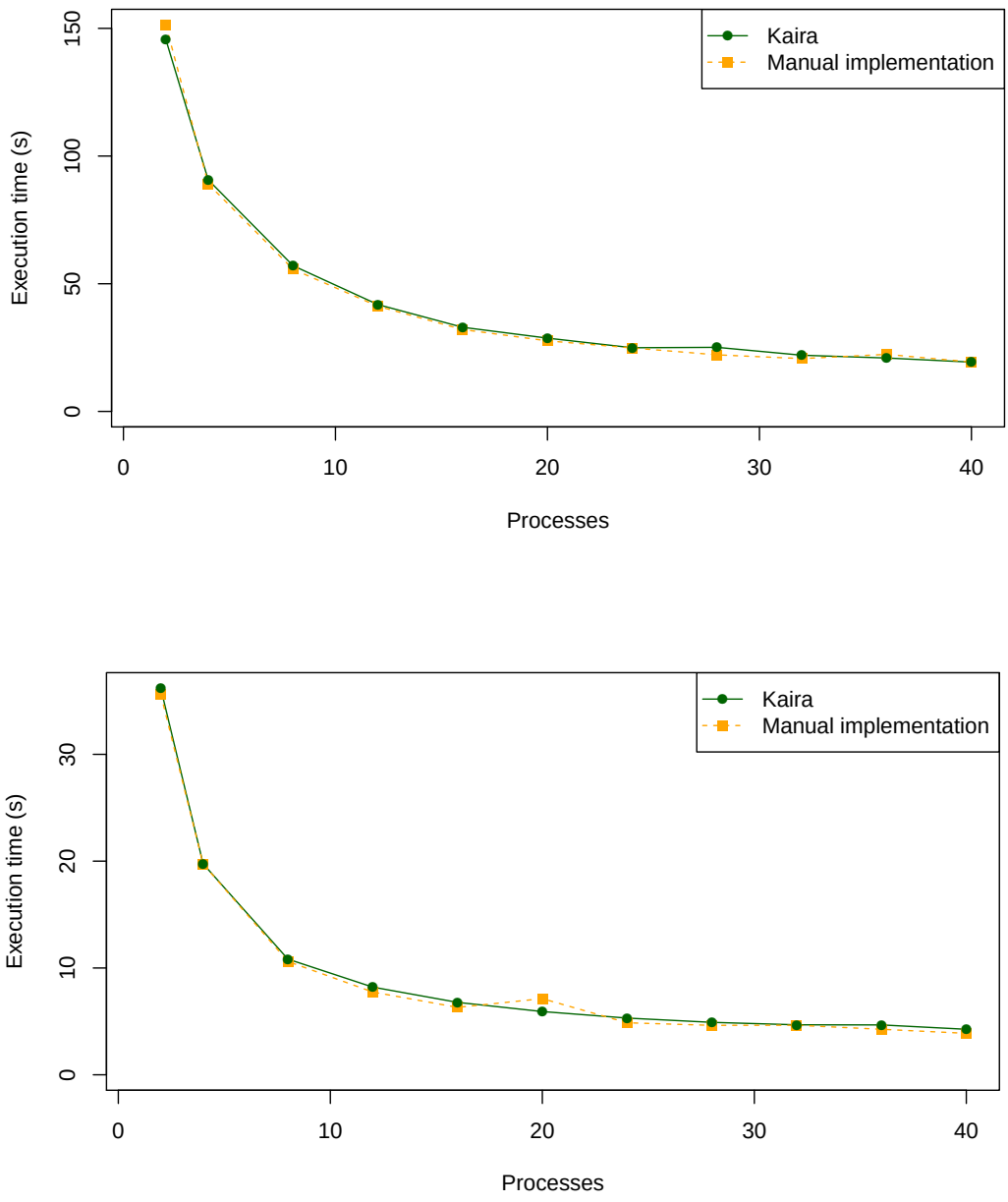


Figure 6.1: Execution times of the heat flow example on Anselm in configurations 2600×8200 (top) and 2600×2600 (bottom)

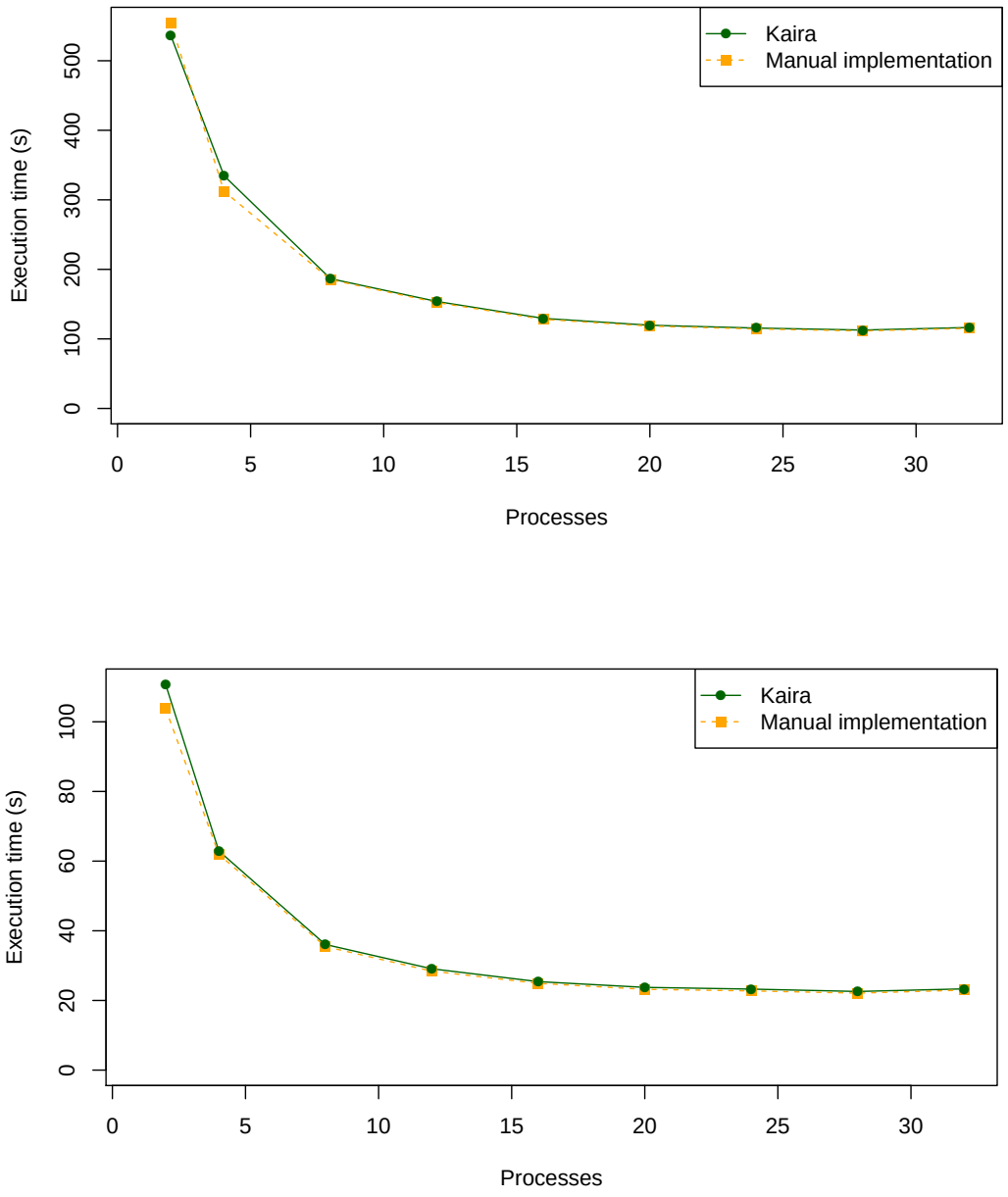


Figure 6.2: Execution times of the heat flow example on Hubert in configurations 2600×8200 (top) and 2600×2600 (bottom)

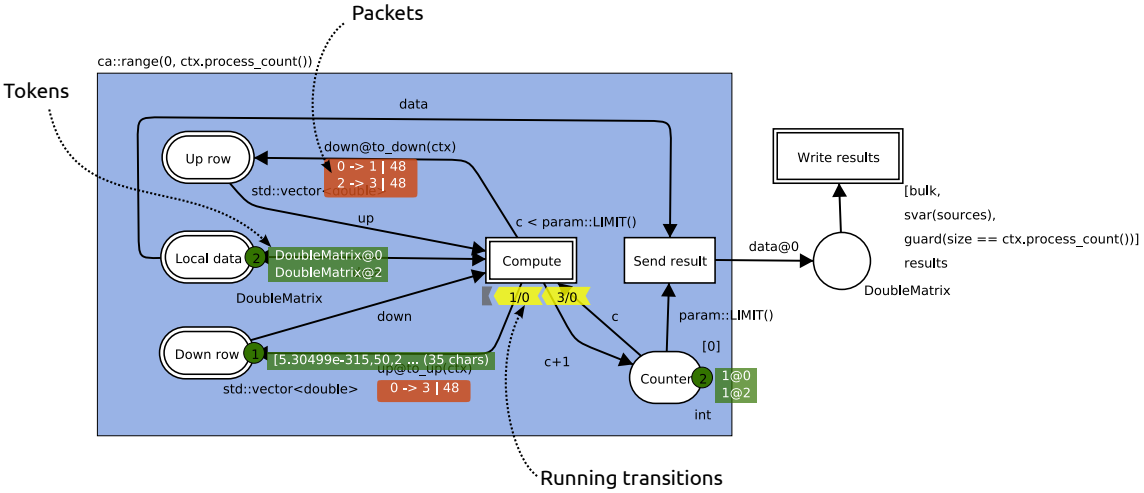


Figure 6.3: The simulator during an execution of the heat flow example.

Displaying only important states with respect to the global application behavior and hiding sequential behavior produces a smaller number of observable states. It allows a more simplistic view of the program. This aspect makes possible to store all states showed during a simulation with a reasonable memory consumption, and the user may browse in the history of the execution. Control through two high-level actions allows easy manipulation with a run of the application. The simulator allows to get the application into any state and the user can observe consequences. Moreover it allows to store a *control sequence* – a sequence of these two actions. Figure 6.4 shows a screenshot of the control sequence viewer. Such a sequence can be loaded into the simulator and gets the program into a desired state. Because recorded actions describe the behavior in a high-level way, control sequences remain relevant even if some selected changes are made to the program. For example, the user can add a debugging code into a transition and rerun the program with the same control sequence to obtain more information about a problematic run. Control sequences can be created by saving a run in the simulator; they also can be extracted from traces (Section 6.3) or obtained as a result of the state-space analysis (Section 6.5).

The simulator infrastructure also offers a feature often provided by debuggers – dynamic connecting into a running application. In the case of Kaira, the current state of the program is translated into terms of the net after the connection. Therefore the user can observe and control the inner state of the application exactly as in the simulator. When the connection is closed, the application continues in the normal run. The current implementation has still some restrictions. It can be used only for applications generated in the thread or sequential modes; the MPI back end

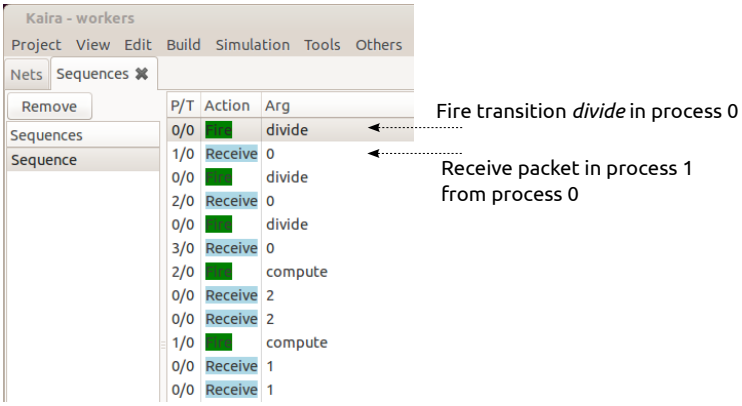


Figure 6.4: A screenshot of the control sequence viewer with a sequence from the workers example

is not yet supported.

Many Petri Nets tools contain a similar type of a visual view with similar control over the models; it is often called *token game*[3]. Almost all of the tools just display a simple abstract computation of Place/Transition Petri nets. Only a few of them work with a full-featured programming language, for example the tools CPN Tools and Renew can be named. But in comparison to Kaira, they use more high-level languages.

To formalize this section, let i and j range over $[n]$, t over T, p over P where n, T, P are defined at the beginning of this chapter. Because the behavior of $\mathcal{K}$ on n processes is described by $\mathcal{B}_n^{\mathcal{K}}$, hence the state of the application is described as a marking of $\mathcal{B}_n^{\mathcal{K}}$. A memory state is described by tokens in places $p_i \in P'$ and packets in the network environment are tokens of places $\mathbf{p}_{i \to j}$. The two actions offered to the user are firing $t_{i,+} \in T'$ (a transition t in a process i is started) and $\mathbf{r}_{i,j,+} \in T'$ (a process i receives packet from a process j). Transitions $t_{i,-}$ and $\mathbf{r}_{i,-}$ are fired automatically when it is possible[4], because when a process $k \in [n]$ starts executing a transition or receives a packet, no other transition can change places controlled by the process k and hence there is no reason to wait with finishing the activity. The control sequence of $\mathcal{K}$ is defined as a sequence $\langle a_1, a_2, \ldots, a_m \rangle \in T'^*$.

[3]Petri Net Tool Database `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html` contains the list of 84 tools with "Token Game Animation" to date 08/20/2013.

[4]In fact, the simulator has a mode where $t_{i,-}$ is controlled by the user. It is designed to observe effects of more threads when the hybrid run (mentioned in Section 3.3.6) is enabled, but as was noted earlier, this feature is not considered in this thesis. Moreover it needs a slightly extended definition of n-instantiation to catch more threads.

6.3 Tracing

Performance analytical features in KAIRA are based on tracing generated applications. The tool allows the application to be generated in the form that records its run into *tracelogs*. When the run is finished, the tracelog can be loaded back into KAIRA and used for *visual replay* or for generating various charts and statistical summaries.

Generally speaking, tracing of applications consists of three basic areas: *measurement specifications*, *instrumentations*, and *presentations of results*. A specification of a measurement is highly dependent on the purpose of the measurement, specifically on whether an overview of the application's run is required or just a specific issue is pursued. It also depends on whether tracing is used for a performance analysis or for reconstructing the run for debugging purposes. But in all cases it is crucial to store only the necessary information. Tracing creates an overhead that may deform the run and may have an impact on event timings. Furthermore, the sizes of tracelogs may become a significant factor when too many events are stored. These are the main reasons that measurement specifications play important roles.

Specifications in common tracing tools are usually implemented by specifying a filter or by a manual instrumentation. The manual instrumentation means manually placed tracing calls into the source code. It allows the tracing of a program on an arbitrary granularity level and storage of arbitrary values. However, maintaining this instrumentation can be laborious. Filters are used when the tool traces function calls. In a common program, tracing all functions would cause tracelogs of enormous sizes, therefore the user has to set up a list of what to exclude from the tracelog. The assembly of such a list usually requires some experience and knowledge concerning what can safely be excluded. In KAIRA, the user specifies what is measured in terms of places and transitions. It is implemented as placing labels in a visual model (Figure 6.5). The user may enable tracing for each place and transition individually. An arbitrary function can additionally be specified for each place to store more information about tokens. This approach provides the user with a simple and an easily understandable way of control what to measure. Furthermore, the approach makes obvious the information that will be gained or lost after enabling or disabling each setting.

The second task is the *instrumentation*, i.e. putting the measuring code inside the application. KAIRA can automatically place the measuring codes while translating a net into C++ sources. Parallel and communication parts are generated by KAIRA; therefore the tool knows the location of the interesting places to put measuring codes according to the specification, and the measuring codes are automatically generated with the rest of the application. For these reasons, KAIRA does not use any instrumentation techniques like an instrumentation with the assistance of a compiler or an instrumentation of machine codes. The approach used herein permits

a tracing version of the application to be obtained in a way that does not depend on a compiler or a computer architecture. From a different point of view, it could be said that Kaira automatically generates a kind of manual instrumentation.

The third task is the presentation of results. Because of specific needs, Kaira does not use any existing trace format and provides its own visualization facilities, but data from the tracelog can be exported for post-processing in some external tools. Additionally, tracelogs can be presented to the user in the form of a visual replay. The run recorded in the tracelog is shown in the same way as in the simulator from the previous section; that is, as the net with tokens in places, running transitions, and packets on the way (Figure 6.6). The user can jump to any state in the recorded application run and examine the effects of fired transitions and stored tokens.

For any state observed in the replay, a control sequence can be exported. Such a control sequence brings the simulator into the state from the replay. In other words, it allows combination of the debugger and the profiler as is mentioned in the scenario given in the Introduction. In the case of use of other existing tools, getting a debugger exactly into a state caught by a trace is usually a hard task.

Kaira provides statistical summaries and standard charts like common trace visualizers (the example in Figure 6.7). But additionally, some information is presented using the terms of the abstract model. Some examples of this include the utilization of transitions, transition execution times (Figure 6.8), the numbers of tokens in places, and so forth. Besides the visualization, the gathered data can be also exported to subsequent processing. It is demonstrated later in Section 6.3.2.

Formally a tracelog can be seen as a control sequence enriched by time-stamps, i.e. tracelog is a sequence $\langle (s_1, t_1), (s_1, t_1), \ldots, (s_m, t_m) \rangle \in (\mathbb{N} \times T')^*$ where $s_i < s_j$ for $i, j \in [m], i < j$. The number s_i is a discrete time-step for an action t_i.

6.3.1 Tracing of heat flow

This example serves as a demonstration of basic tracing facilities. For this purpose, the net from Section 4.2 is used. Tracing was performed in three settings as follows:

- A – All transitions are traced.
- B – All transitions and `token_name` for all places except *Up row* and *Down row* are traced. This configuration is shown in Figure 6.5. (In the used implementation, the `token_name` for type `DoubleMatrix` returns just a short string with the name of the class, not values of the matrix).
- C – All transitions and `token_name` for all places are traced. Moreover, the average temperature in the grid in place *Local data* is also traced for each process.

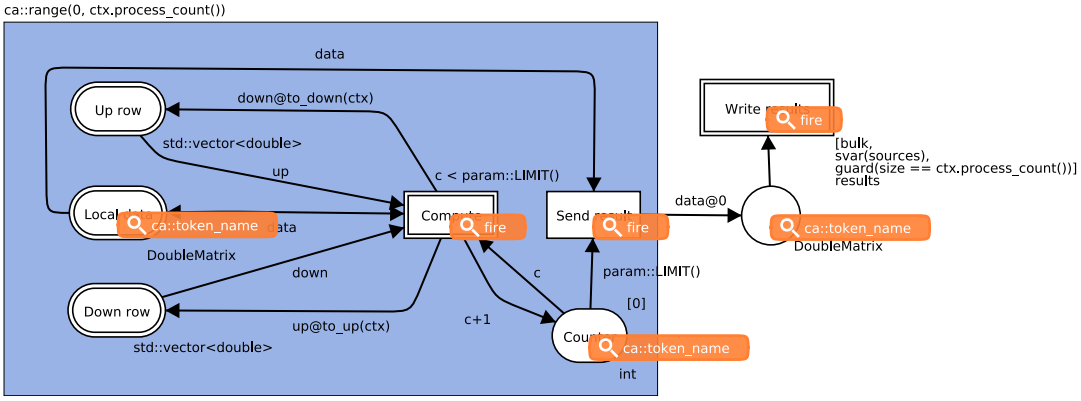


Figure 6.5: The heat flow example with the tracing configuration for variant B

The first variant serves to obtain a general overview of the application's performance with a minimal overhead. Variant B provides a picture of data flows in the application. In this variant two places are not traced, because of `token_name` function for `std::vector` returns all elements. It would produce unnecessarily big data for this goal. There are more options for how to solve it; `token_name` function can be changed to return something smaller, a different function could be traced, or tracing of this place can be switched off. Tracing other functions is shown in the next examples, so the last and the simplest one is used here.

The overhead introduced by variants A and B is negligible and the run is fully comparable with the original run. Even variant A traces only the basic information about executing transitions; charts shown a utilization of each process (Figure 6.7) or a histogram of transition execution times (Figure 6.8) can be obtained. Variant B also provides the visual replay (Figure 6.6) and statistics related to tokens. The computation times for all three variants are shown in Figure 6.9.

6.3.2 Tracing of ACO

This section shows how the user can perform a simple measurement and export traced data. Example ACO (Section 4.6) is used in this section, file *eil51.tsp* from TSPLIB[5] was chosen as the instance for the TSP problem.

Let us assume that we want to observe the fitness value in time during a computation and compare it with the scenario in which communication is disabled. The communication between colonies can be simply disabled by removing the arc with expression `[bulk, multicast] send@workers`. The measurement is enabled by attaching a simple function returning a fitness value of an ant in place *Best trail* (Figure 6.10). When a token arrives at this place, its value is stored into the

[5] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

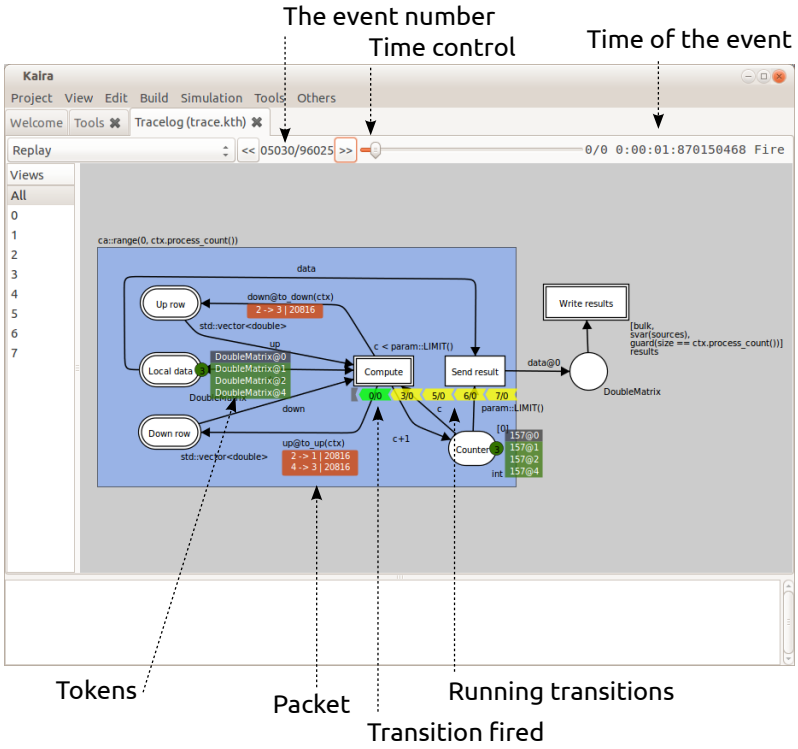


Figure 6.6: A replay of a tracelog obtained by tracing the heat flow example with the tracing configuration of variant B

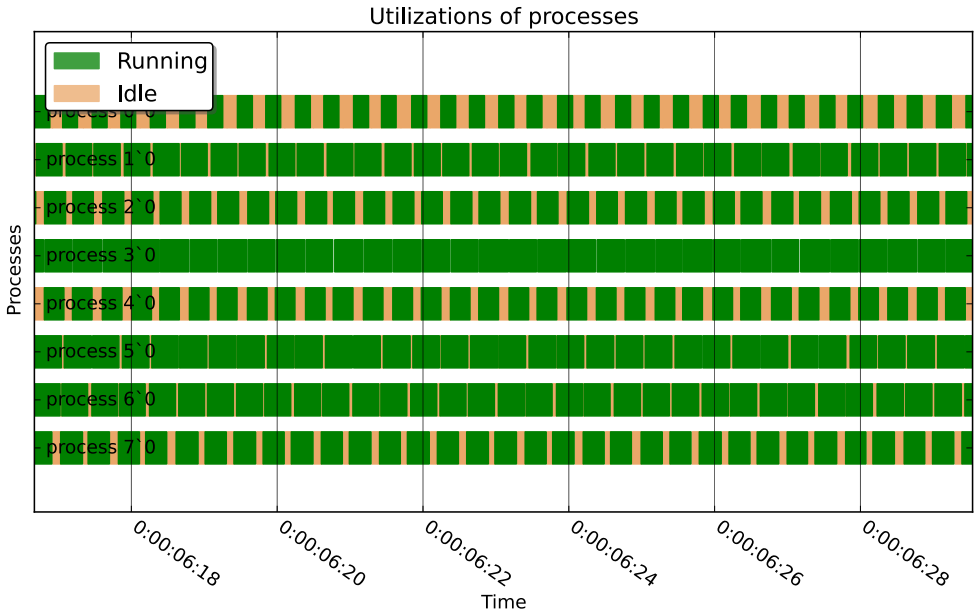


Figure 6.7: A magnified part of a chart showing a process utilization obtained from a tracelog

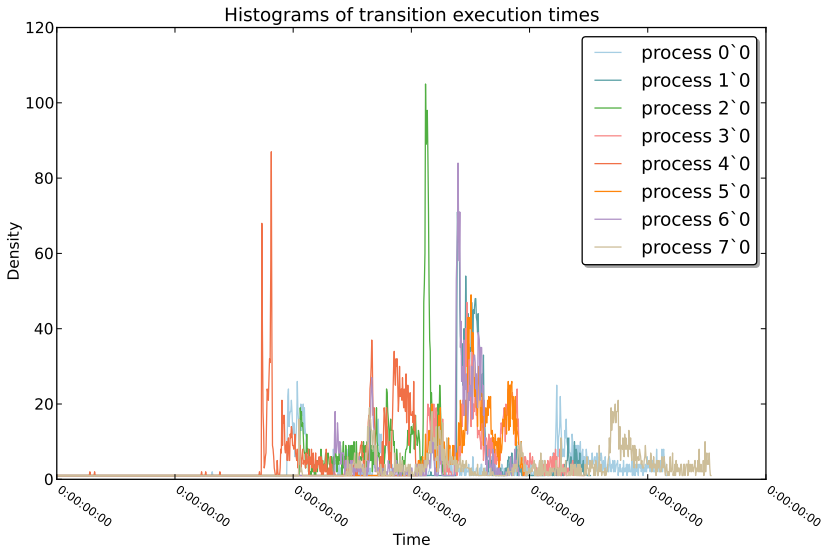


Figure 6.8: The histogram of transition execution times for each process

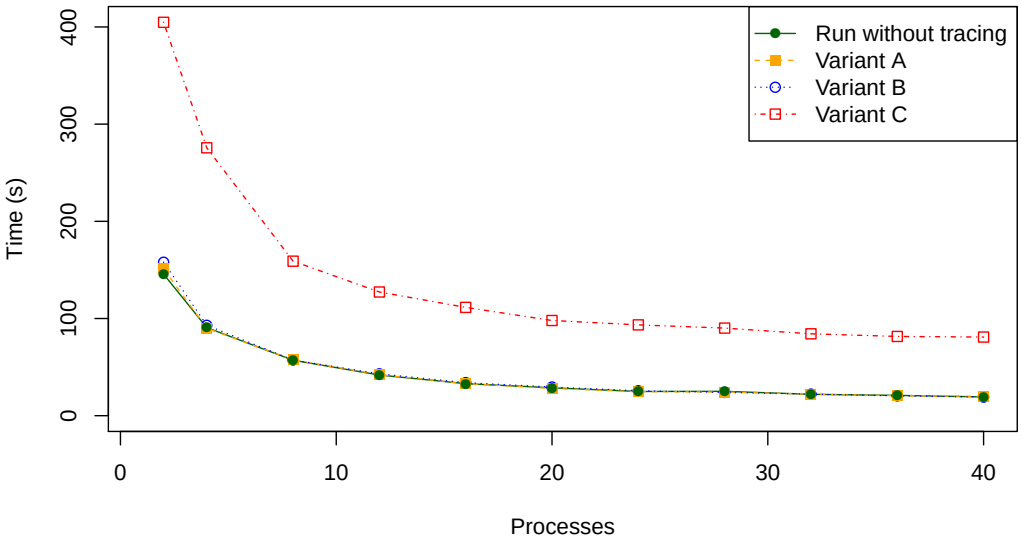


Figure 6.9: Execution times for tracing of the heat flow example in variants A, B, C from Section 6.3.1

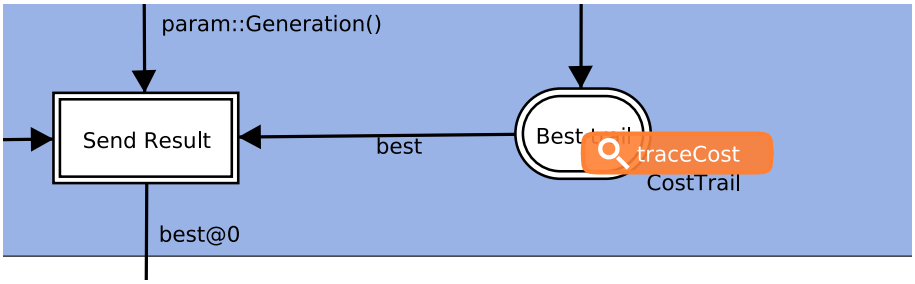


Figure 6.10: The tracing configuration for the example from Section 6.3.2

tracelog. Figure 6.11 shows how a table can be exported from the tracelog. The resulting table can be saved as simple plain text data and reused in an external tool. In this case, tool R is used to post-process data and obtain the charts in Figure 6.12. As expected, the convergence of the fitness value is almost the same when processes communicate, and it differs in the non-communicating version.

As the last two demonstrations show, KAIRA offers an infrastructure where the user can easily set up an experiment, perform measuring, and obtain the resulting data while still working in terms of the model where the application is created.

6.4 Performance prediction

The performance prediction in KAIRA is implemented as online simulations, i.e. a full computation of the program is performed in a simulated environment. The communication layer is simulated through an analytical model. In KAIRA there is not a fixed number of models, but the user may specify any model as a C++ function in a similar way as C++ sequential codes are edited in transitions or places. This function is called for each packet and returns a time needed to transfer the packet. The basic information like the size of the packet and the rank of the sender and the receiver is passed to this function. A simple linear model is shown in Listing 6.1. Additionally, `casr::Context` (the subclass of `ca::Context`) makes the access to information about the value of the global clock and the current workload of the network between each pair of processes. Therefore more sophisticated models can be defined; models that reflect an overall situation in the network or models with dynamic changes of the bandwidth in time.

The model of communication is not the only configurable setting. In KAIRA, execution times of each transition and size of data transfered through each arc can be arbitrary modified. It is designed to answering questions like "what will be the overall effect when a code in a transition is optimized and is 20% faster than before the optimization", but it can be also used for reducing a computation time of predictions. The former is demonstrated in Section 6.4.2, the latter in Section 6.4.1.

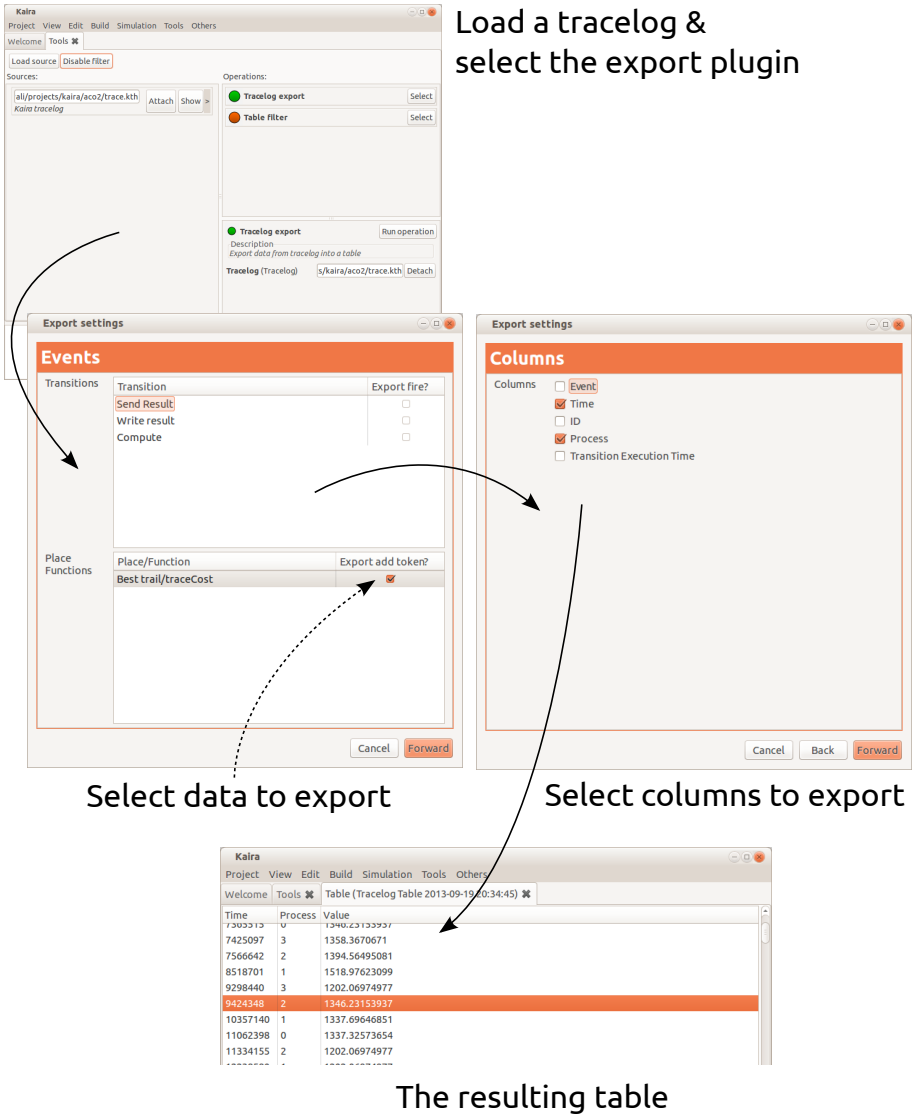


Figure 6.11: The process of exporting data from a tracelog

Listing 6.1: A simple linear model of communication

```
ca::IntTime packet_time(casr::Context &ctx,
                        int source_id, int target_id, size_t size)
{
    const ca::IntTime latency = 5847;   // [ns]
    double bandwidth = 1.98059;         // [byte/ns]
    return latency + size / bandwidth;
}
```

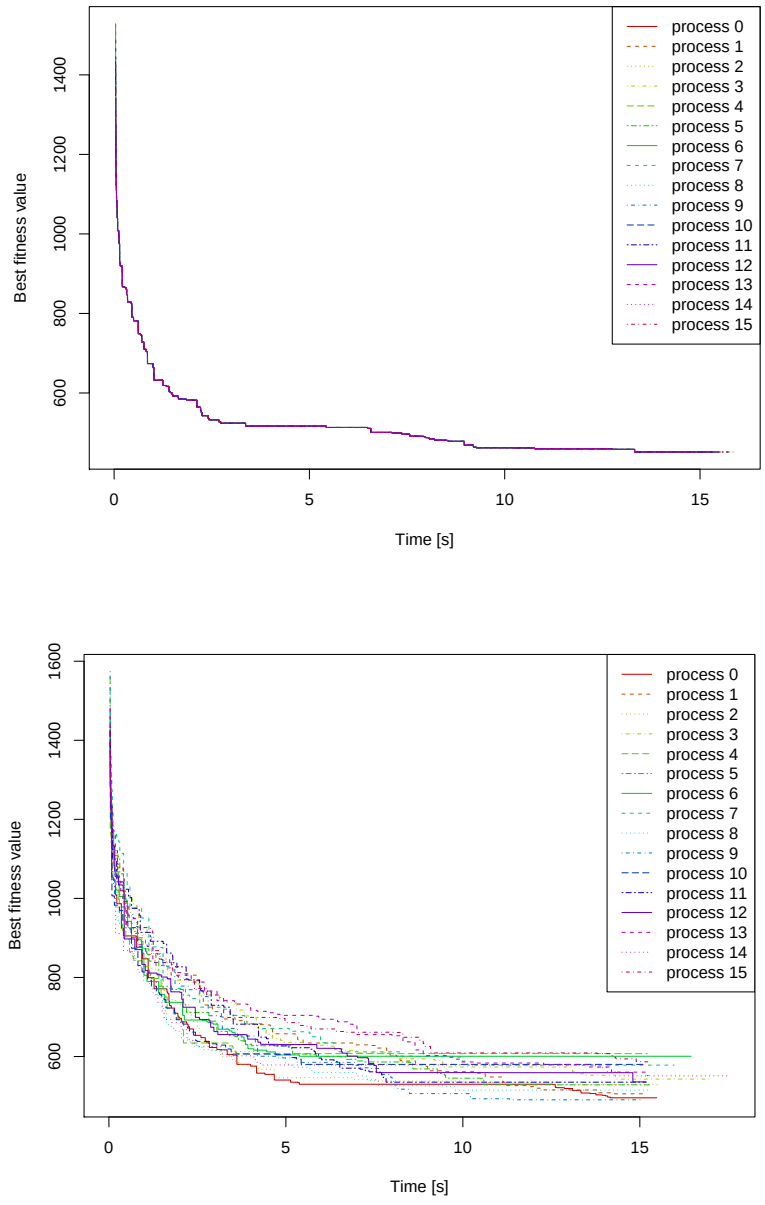


Figure 6.12: The fitness value of the token in place *Best trail* in time. The top chart is for the original net, the bottom chart is for the variant without communication.

The configuration of this feature is specified in the same way as for tracing, i.e. as labels placed into the net (examples in Figures 6.16 and 6.20). In the case of a transition, the expression in the label specifies how the running time is modified. The transition is computed as usual, but the program in the simulator behaves as if the computational time of the transition is the time obtained from the expression in the label. In the expression, an instance of `casr::Context` is accessible through the variable `ctx` and variable `transitionTime` provides the access to the original computation time. For example, if the expression in the label of a transition is `transitionTime / 2`, then the simulated program behaves like the program where this transition is two times faster. Additionally, any variable from expressions on input arcs of the transition can be used in the label, hence the simulated computational times may depend on computed data.

The configuration for arcs works in a similar way by modifying the sizes of tokens produced by an arc. This value is used when a token is transferred through the network; the receiver obtains the data as they are sent, but the network simulation considers modified sizes of packets. The variable `size` can be used in the label and it enables access to the original size of the data.

Kaira additionally offers a special clock. It runs like an ordinary clock in a normal run and can be arbitrarily modified in a simulation in the same way as running times of transitions. A transition using this clock is depicted with a small clock symbol on the left side. The clock provides methods `tic()` and `toc()` where `toc()` returns the time elapsed from the last call of `tic()`. In the simulated run, the user may provide an expression that is called after with each `toc()` and modifies returned time. This feature is used in the experiment with load balancing in Section 6.4.2.

The simulated program produces a tracelog where the simulated run is recorded. It uses the same infrastructure as was described in the previous section, including the way in which a measurement is specified and the results are post-processed. It provides richer possibilities for tracing than do existing prediction tools, they can usually just switch tracing on or off. Standard tracing tools cannot be used with simulators, because of the simulated network environment and the time control.

6.4.1 Performance prediction of the heat flow example

In this section, performance prediction will be demonstrated using the heat flow example from Section 4.2 and its manual implementation used in Section 6.1.1. The goal is to predict running times of this example on Anselm in settings: 2600×8200, 5000 iterations while using only a single core of Anselm. The running times of the application are predicted by Kaira and SimGrid.

The prediction in Kaira was executed with a linear model of the network; parameters were obtained by a simple measurement of transport times between two nodes. SimGrid was configured according the computer specification and the

latency of the network was configured to the same value as in the case of Kaira. The results are shown in Figure 6.13 and Figure 6.15. The former shows predicted times and the latter shows logarithmic errors as defined in [64] ($Err_{log} = |\log t - \log p|$ where t is the real time of the execution and p is the predicted time). As results show, Kaira gives a worse prediction in absolute numbers than specialized tool SimGrid. But the overall trend of the program behavior with an increasing number of processors is reflected also in the case of Kaira prediction.

To reduce the computing times of predictions in this particular example, it is possible to exploit the fact that the structure of communication is independent on computed data. A smaller instance of the problem can be computed, but the computing times and sizes of transferred data are appropriately resized. The prediction named as XY was obtained by computing an instance with a smaller grid; both dimensions of the grid are divided by two. The computation time of transitions *Compute* and *Heat flow* linearly depends on the size of the grid; therefore each process obtains a four times smaller part of the grid. Thus the computing times for the transition are multiplied by four, to simulate the original time. The configuration is shown in Figure 6.16. The size of exchanged rows depends on the width of the grid, hence it is only halved and the factor two is used on edges that transfer rows, to simulate the original size. In this sense, predictions X and Y are also configured. In prediction X, the number of columns is halved; in Y the number of rows is halved. In the case of SimGrid only the global power of nodes was multiplied by four (XY) or two (X and Y), because SimGrid does not allow the more precise adjustments that are possible with Kaira.

The results are shown in Figures 6.14 and 6.15. The predictions obtained by reducing the size of the grid still show the overall trend but times to obtain the predictions were reduced proportionally to the size of the grid. Hence XY prediction needs about a quarter of the original time; X and Y need a half of the origin time. The main goal of this example is to demonstrate configuration options for performance prediction in Kaira. The user can simply skip a part of the computation and configure the simulation in a way that the original run is simulated.

6.4.2 The experiment with load balancing

In this experiment, the same infrastructure as in the previous section is used. But now, the goal is not to predict some real times of computations but test the correctness of the load balancing algorithm presented in Section 4.3. For the demonstration, the following configuration has been chosen: 2600×2600 with 5000 iterations, with load balancing performed every 100 iterations.

Let us start with observing the normal run of the program by the tracing infrastructure. The settings of tracing are shown in Figure 6.17. Two values are traced: the number of rows is monitored by function `rows_count` in place *Local state* and the

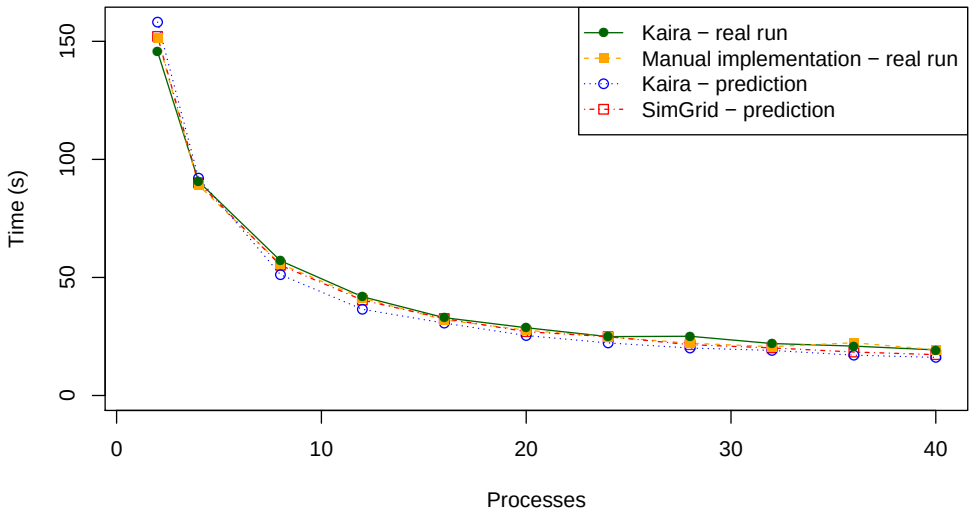


Figure 6.13: Prediction of execution times for the heat flow problem (2600 × 8200; 5000 iteration)

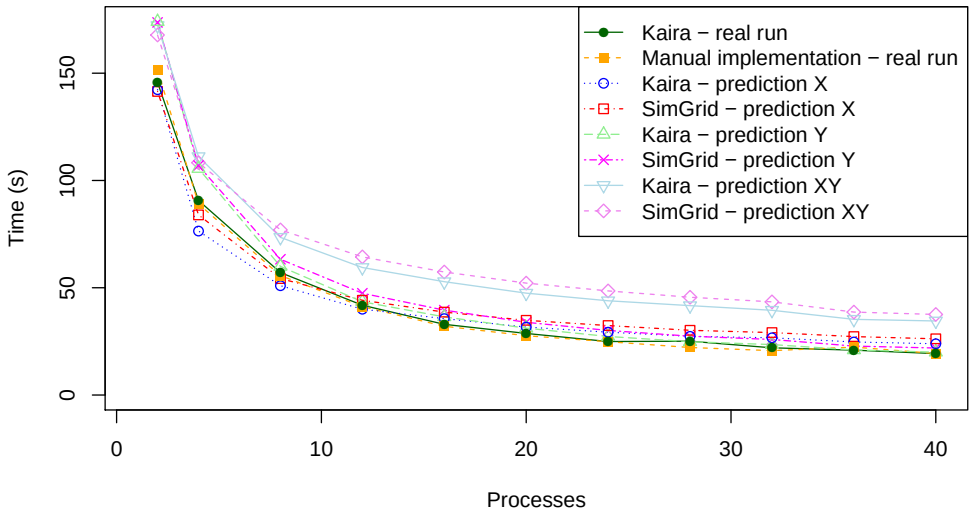


Figure 6.14: Prediction of execution times for the heat flow problem (2600 × 8200; 5000 iteration)

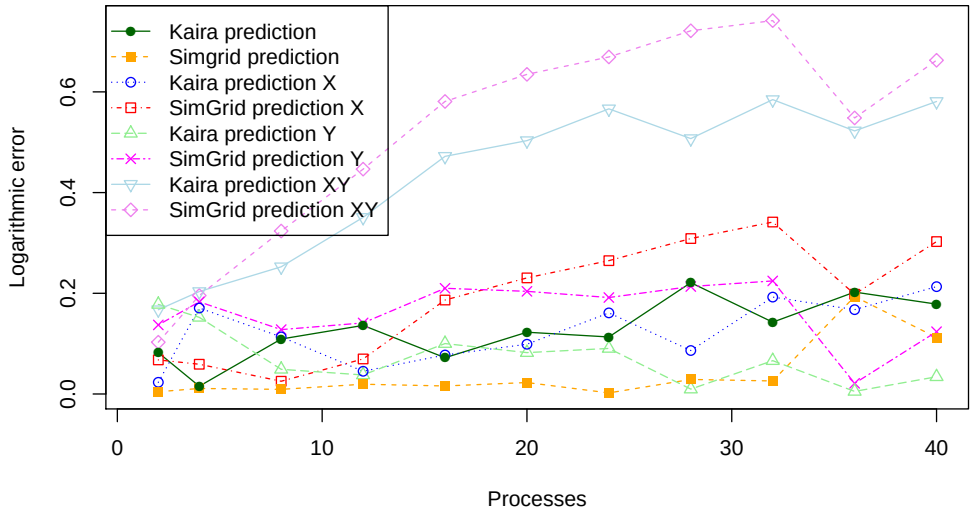


Figure 6.15: Prediction errors for execution times in Figure 6.13 and Figure 6.14

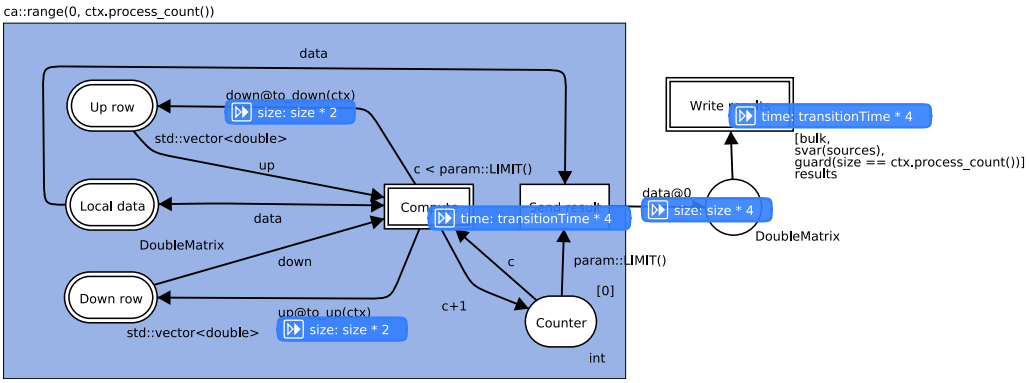


Figure 6.16: The configuration of the simulated run for experiment “XY”

average time of the computation is recorded through a new extra place connected to transition *Balance*. Data can be exported from the tracelog in the same way as in Section 4.6. The results from Anselm are shown in Figure 6.18. The load balancing is actively involved, because the spreading non-zero values through the grid cause a slow down of processes. Figure 6.19 shows the average computation times of a single iteration when load balancing is disabled. The results indicate that load balancing works in the expected way; slower processes are disposing rows and the average times for a single iteration are more balanced than without load balancing.

The process described above only allows us to see the behavior that is reproducible on our currently available hardware. The prediction environment can be used to observe the behavior of the algorithm under more extreme conditions. Let us assume, that we want to see the behavior when a process suddenly becomes much slower and then slowly returns to its original speed. More precisely, after seven seconds of computing, process 4 becomes twelve times slower for another seven seconds and then it uniformly returns back to the original speed for another ten seconds.

The settings in Figure 6.20 and function `experiment_time` shown in Listing 6.2 are used to perform this experiment. The function changes the computation time of process 4 in the desired way. Besides changing the transition execution time, the clock that is used by the load balancing algorithm has to be also modified. In this example, the clock is started at the beginning of a transition execution and stopped at the end of the computation; therefore the measured time almost exactly matches the full time of the transition execution. Hence the same function can be safely used for both settings. The result of the experiment is shown in Figure 6.21. After seven seconds of computations, process 4 suddenly slows down, but it is balanced in three seconds by disposing almost all rows to neighbors. When the execution time returns to normal, the rows are gradually returned back.

This demonstration shows that the user can test developed programs in various situations just by changing a simple C++ expression and easily obtain results due to the tracing framework.

6.5 Verification

The verification in Kaira is based on constructing and exhaustively exploring the state space of a verified program. Therefore analyses are not performed on a particular run but all possible program behaviors are considered. By analyzing the state space, the tool provides the following analyses: The detection of deadlocks and cyclic computations, checking uniqueness of results and uniqueness of characteristic vectors of computation paths. The deadlock and cyclic computations are classic problems. Violating uniqueness of both kinds does not necessary indicate a bug in a program. But for many computational programs this property holds and its violation indicates an error. All analyses will be more precisely described below.

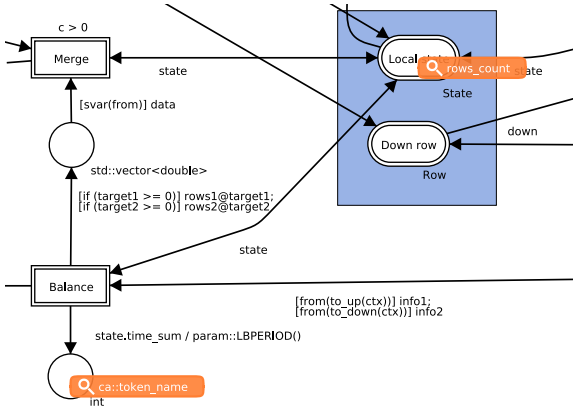


Figure 6.17: The tracing configuration for the net of the heat flow with load balancing example. The full picture of the net is in Figure 4.5.

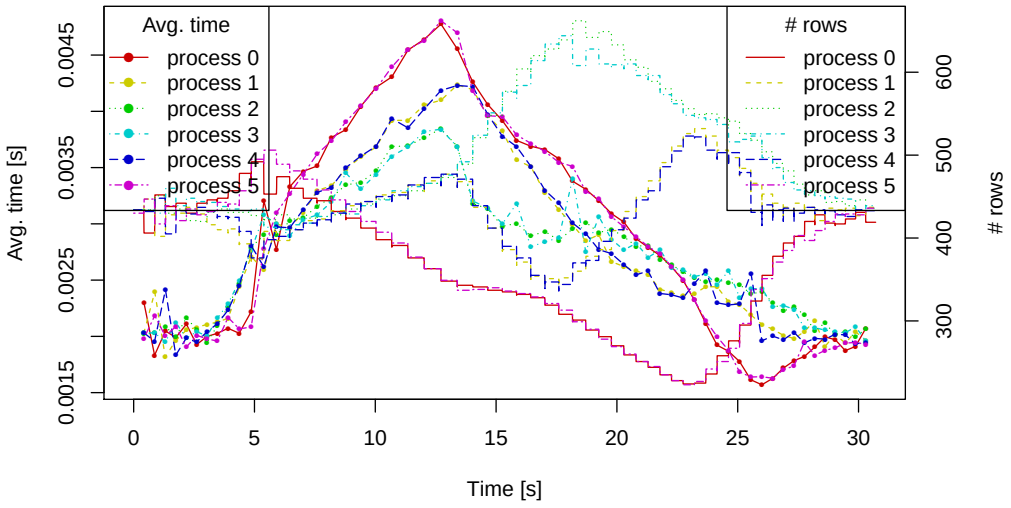


Figure 6.18: The average computation times of iterations and row counts in the heat flow example with load balancing (6 processes; 2600×2600; 6000 iterations)

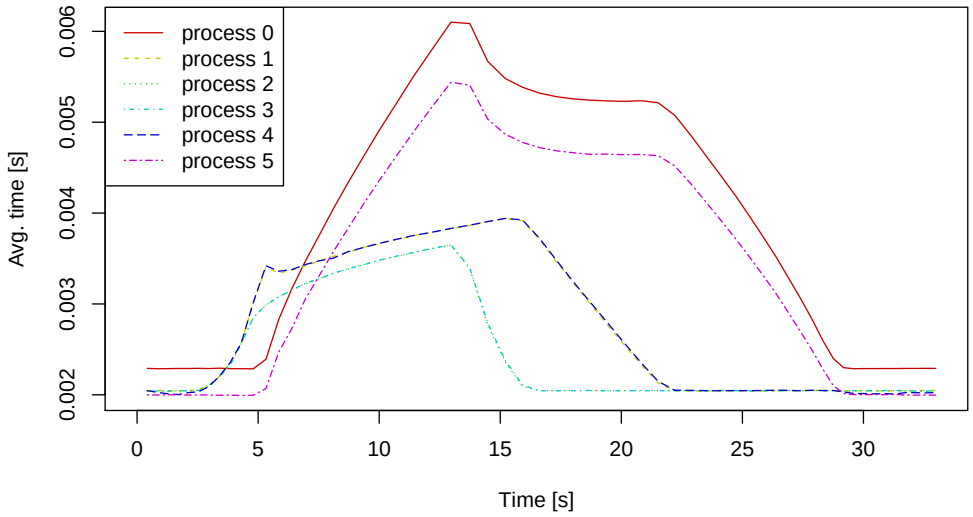


Figure 6.19: The average computation times of iterations in the heat flow example without load balancing (6 processes; 2600×2600; 6000 iterations).

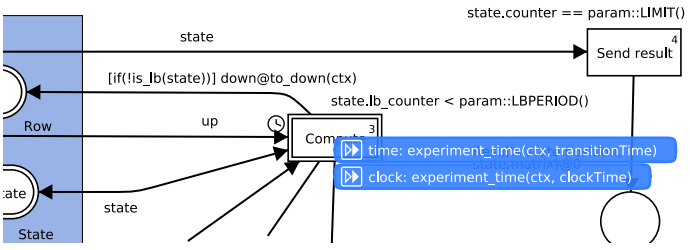


Figure 6.20: The configuration of the simulated run from Section 6.4.2. The full picture of the net is in Figure 4.5.

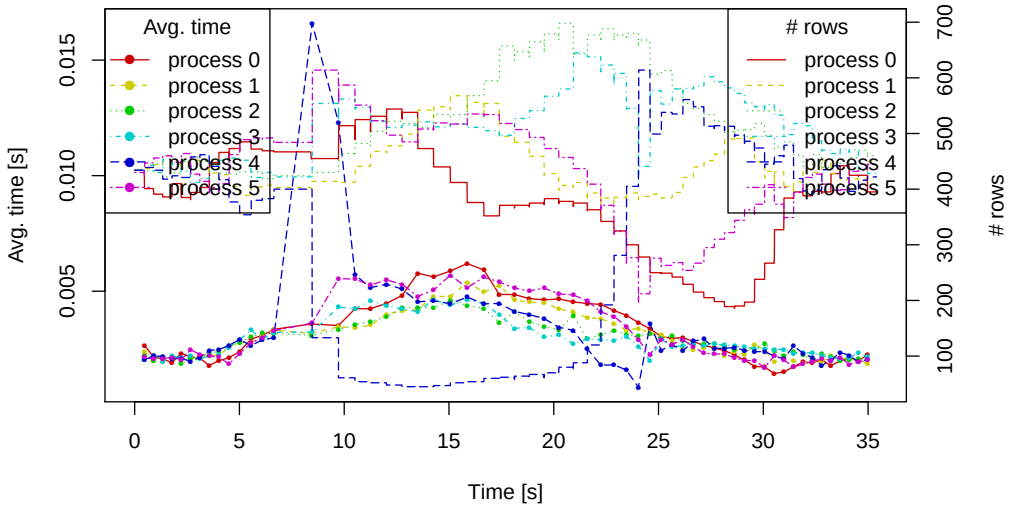


Figure 6.21: The average computation times of iterations and row counts in the heat flow example with load balancing (6 processes; 2600 × 2600; 6000 iterations) in the experiment where process 4 is slowed down

Listing 6.2: The function used in configurations of the time and clock substitutions for the experiment with load balancing of heat flow

```
ca::IntTime experiment_time(casr::Context &ctx,
                            ca::IntTime time)
{
    if (ctx.process_id() == 4) {
        if (ctx.time() > 7e9 && ctx.time() < 14e9)  {
            return time * 12;
        }
        if (ctx.time() >= 14e9 && ctx.time() < 24e9)  {
            return (time * (24e9 - ctx.time()) * 12.0) / 10e9;
        }
    }
    return time;
}
```

The verification in Kaira has the following limitations to make it feasible. The program is verified for a fixed input and it is assumed that each computation reaches only a finite number of states. These limitations are fundamental because they follows from the limits of what is decidable in the verification of generic programs. Kaira also verifies only programs where the effect of each transition depends only on its input variables, the current process rank and the number of running processes. In practice, it means that a verified application cannot use random numbers or time measurements. The workers, heat flow, sieve, and matrix multiplication examples satisfy this condition and can be verified, but ACO (random numbers) and heat flow with load balancing (time measurements) not.

Because all behaviors of the application are explored, the memory and CPU time demands may grow more than exponentially. For this reason, only small instances of a program can be verified. But many bugs that occur in practice, like race conditions, may be often observed even on very small instances. With these type of bugs, the problem is not usually the size of the instance but the rarity of the error behavior among runs that does not expose the error. But the rarity is not a problem in the state-space analysis because all behaviors are examined.

The current implementation of verification in Kaira serves mainly as a proof of the concept and does not contain any advance method for reducing the state space, so in the current version, Kaira cannot compete with tools like ISP in the size of instances that can be verified. But verification in Kaira benefits from more abstract description of programs and the rest of the Kaira framework. ISP offers a deadlock detection; when it is detected, then the sequence of MPI calls that goes to the deadlock is shown as a graph. In the case of Kaira, results are provided in the form of control sequences. Hence the result of the analysis can be loaded into the visual debugger and the user can observe step by step the behavior of the application.

Formally let *state space* be a set $SP = \{m \mid m'_0 \rightarrow^* m\}$ and the set of *end states* be $ES = \{s \in SP \mid \neg\exists s' \in SP : s \rightarrow s'\}$. In these and the following definitions, we are again considering $\mathcal{K}, n, \mathcal{B}_n^{\mathcal{K}}$ defined in the beginning of this chapter.

Now the description of analyses follows:

Deadlock In this analysis, a situation where the application cannot progress but `quit` was not called is detected. The application contains a deadlock if there is a state $m \in ES$ such that $m(\mathbf{p}_{run}) \neq \langle\rangle$.

Cyclic computation In this analysis, it is checked if a computation cannot arrive again to the same state, this would cause a potential endless computation. It can be formalized as follows: the application contains a cyclic computation if there are states $s, s' \in SP$ such that $s \neq s'$, $s \rightarrow^* s'$ and $s' \rightarrow^* s$.

Uniqueness of final markings This analysis is based on the fact that a large number of computational programs should return the unique result independently on a used computation path. In KAIRA, the user can specify a set $F \subseteq P$. It is a set of places that are checked in each state from ES. In other words, the cardinality of the set $X = \{(m(p_i))_{(p,i)\in F\times[n]} \mid m \in ES\}$ is checked. If $|X| > 1$ then there are more outcomes of the application with respect to F. It is up to the user to choose correct F, but is usually a simple task to choose places that should have a unique result in the end.

Uniqueness of the characteristic vector This analysis is based on the observation that for many computational applications hold that an application executes the same set of actions in all its executions, only the order of actions differs. In workers example, a fixed number of jobs is assigned. Therefore transition *divide* has the same number of executions in all possible runs when an input is fixed. Moreover, for all runs each interval is assigned exactly once, therefore transition *Compute* is executed for each job exactly once. In this analysis, the user specifies the following mapping $cfg : T \to X$ where $X = \{\mathbf{n}, \mathbf{c}, \mathbf{b}, \mathbf{p}, \mathbf{bp}\}$. Elements of X are symbols with the following meaning:

- $cfg(t) = \mathbf{n}$ – The transition t is ignored in the analysis.
- $cfg(t) = \mathbf{c}$ – Occurrences of t in computation paths are counted.
- $cfg(t) = \mathbf{p}$ – Occurrences of t for each process in computation paths are counted.
- $cfg(t) = \mathbf{b}$ – Occurrences of variable bindings of t are counted.
- $cfg(t) = \mathbf{bp}$ – Occurrences of variable bindings of t for each process are counted.

Let $CS = \{\langle t_1, t_2, \ldots, t_z\rangle \in T'^* \mid \exists m_0, \ldots, m_z : m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} \ldots \xrightarrow{t_z} m_z \wedge m_0 = m'_0 \wedge m_z \in ES\}$ is a set of *computation sequences*. Let a *characteristic vector of* $u = \langle t'_1, t'_2, \ldots, t'_z\rangle \in CS$ *with respect to cfg* denoted as $Ch_{cfg}(u)$ be $(Ch'_{cfg}(u,t))_{t\in T}$ where

$$Ch'_{cfg}(u,t) = \begin{cases} 0 & \text{if } cfg(t) = \mathbf{n} \\ |\{j \in [z] \mid \exists i \in [n] : t'_j = t_{i,+}\}| & \text{if } cfg(t) = \mathbf{c} \\ (|\{j \in [z] \mid t'_j = t_{i,+}\}|)_{i\in[n]} & \text{if } cfg(t) = \mathbf{p} \\ (|\{j \in [z] \mid \exists i \in [n] : t'_j = t_{i,+} \wedge b = b_{t,i,m_{j-1}}\}|)_{b\in B} & \text{if } cfg(t) = \mathbf{b} \\ (|\{j \in [z] \mid t'_j = t_{i,+} \wedge b = b_{t,i,m_{j-1}}\}|)_{(i,b)\in[n]\times B} & \text{if } cfg(t) = \mathbf{bp} \end{cases}$$

where $(\_, b_{t,i,m}) = \mathit{FindBinding}(t, i, m)$. Checking of the uniqueness of characteristic vectors is checking if $|\{Ch_{cfg}(u) \mid u \in CS\}| = 1$.

6.5.1 Verification of the workers example

Let us demonstrate verification features on the workers example (Section 4.1). The configuration of the verification is again done through labels; for the example used here this is shown in Figure 6.22. The deadlock and cycle detections do not need any settings therefore all configurations are for the uniqueness analyses. For checking of uniqueness of final markings, the new place *Final* was added and transition *write results* puts sorted prime numbers there. This place is explored in the analysis of the final marking uniqueness (i.e. $F = \{final\}$). The setting for the uniqueness of the characteristic vector is configured as

$$cfg(compute) = \mathbf{b}; cfg(divide) = cfg(write\ results) = \mathbf{p}$$

This configuration naturally follows from the way how the program works.

Transition *Divide* is fired for a fixed number of times, and is fired only at process 0. Configuration **bp** cannot be used because different runs can assign jobs to different processes. *Write results* is fired only once for process 0. But prime numbers in variable `results` can be ordered in many ways; therefore the binding cannot be checked. Each job (interval) is computed exactly once, hence each interval also occurs in variable `job` of transition *Compute* exactly once, but in different processes for different runs, therefore option **b** is used and not **bp**.

To demonstrate the error detection, an intentional bug was introduced by removing the arc with inscription `[guard(size == ctx.process_count() - 1 )]` between place *ready* and transition *write results*. Figure 6.22 shows the variant in which the arc is already removed. The modified program may return the correct result like the original one, but may also write results prematurely without waiting to all workers. When analyses are run, the error is detected by both uniqueness analyses. The execution of the program where the error occurs has a different result than the correct run, hence there are two different markings of the place *Final*. Analysis of characteristic vectors discovers the problem because *Compute* is fired fewer times when the error occurs. The report with results is shown in Figure 6.23. From this report, two minimal control sequences that prove non-uniqueness can be exported. One shows a correct run; the second one shows premature writing of results.

All these analyses are implemented by simple graph-searching algorithms over the graph of the state space. The main problem is the construction of this graph. As was already said, many optimizations are still missing in the Kaira implementation; therefore sizes of instances are very limited now. The following instances of workers can be verified on a computer with 8GB RAM: about ten thousand of the assigned intervals for two processes (1 master, 1 worker), about eleven intervals for three processes, and eight intervals for four processes. Similar analysis for Heat flow is able to verify about thousands of iterations of the algorithm for two processes, about 200 iterations for three processes, and two iterations for four processes.

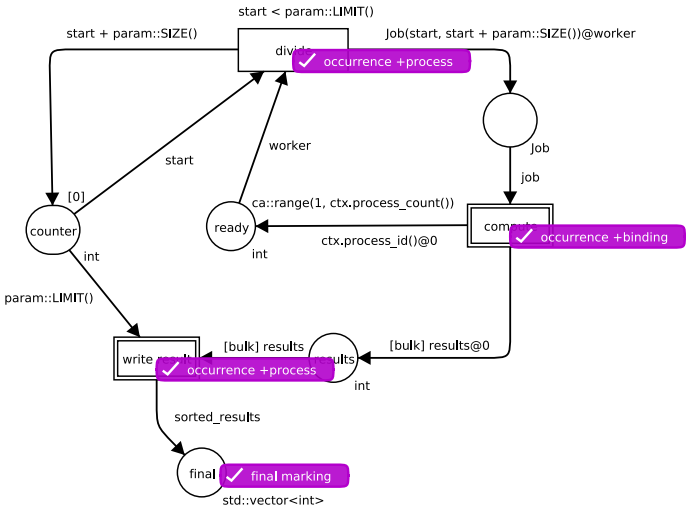


Figure 6.22: The configuration of workers for the state-space analysis. The arc between *ready* and *write result* is intentionally removed to introduce an error.

To conclude this section, Kaira offers the verification infrastructure with analyses that are more than detecting deadlocks or assertion violations. These analyses can be explained in terms of the used model without many technicalities, they are easily configurable and the results are provided in the form of control sequences, i.e. the states that show the problem can be directly loaded into a debugger. The implementation is still young and there are many potential places where the implementation can be improved to manage much bigger instances than the presented ones. The author believes that the proposed approach can be one way to make formal verification techniques a natural part of the development of MPI applications.

6.6 Libraries

This section covers generating libraries. The main goal of this feature is to provide a simple way of introducing parallel computations into existing sequential applications. Kaira is able to generate a C++ library with functions that run internally in parallel and that can be called from any sequential C++ code. It allows to parallelize only computationally demanding parts without changing the rest of the application.

The first part of this section covers C++ libraries. This feature is further used to create Octave modules, serving as a demonstration of a use of a high-level tool together with Kaira.

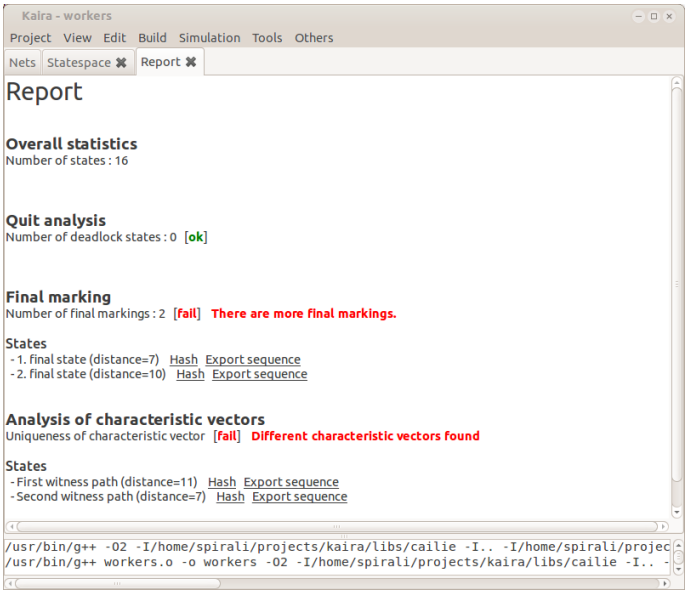


Figure 6.23: The report of the state-space analysis for the net in Figure 6.22

6.6.1 C++ libraries

So far, a program in KAIRA was described by a single net. In the case of a library, there can be one or more nets; each net represents a single function in the resulting library. The syntax of the visual language is extended to express inputs and outputs of a function. A place can be marked as an input (output) interface by *input (output) marks*. The example is shown in Figure 4.7. The net has two input marks `m1` and `m2` and a single output mark `output`. The C++ function generated from this net has the following declaration:

```
void matmult(const Matrix &m1, const Matrix &m2, Matrix &output);
```

Types of arguments are taken from the places with input/output marks and `matmult` is the name of the net. Functions in the library created in this way can be called in any sequential C++ application. The code of an application that uses the library will be referred to as the *main code* in the following text. When a function from the library is called, then the run of the main code is blocked and the computation described by the net is performed. When the computation is finished (i.e. a transition calls `ctx.quit()`), then the run of the main code is resumed.

Libraries can be generated in two modes: the direct usage of MPI and RPC (*Remote Procedure Call*) mode. In the former, the main code runs only at process 0 and all other processes wait until the parallel function is not called. The generated library contains initialization function `calib_init`. The user is obligated to call this

function as the first function in the main code. It makes all necessary initialization and makes sure that the main code runs only in process 0 and other processes wait.

In RPC mode, KAIRA creates both the server and the client side. The client part is a light C++ library that is used in the main code; it sends all calls to the server where actual computations are performed. The sequential code of the application and the MPI part run as completely independent applications and they can run on different computers.

The complete feature set of KAIRA presented in this whole chapter is available for libraries; therefore functions in a generated library can be traced, verified, etc. The simulator is also available in a mode in which the main code runs normally, but control is overtaken by the simulator when a generated function is called. After that, the user can control the application in the same way as for stand-alone applications.

One net can be used to build a stand-alone program or a library function. When it is built as a library, then init-expressions and init-codes of places with input marks are ignored. When the net is built as a stand-alone program, then input and output marks are ignored and initializing codes are evaluated normally. It allows the user to simply create a function from a program by adding interface marks or to create a simple test program from a net that originally described a function.

6.6.2 Integration with Octave

OCTAVE is described according its own webpage[6] as:

> "GNU Octave is a high-level interpreted language, primarily intended for numerical computations. It provides capabilities for the numerical solution of linear and nonlinear problems, and for performing other numerical experiments. It also provides extensive graphics capabilities for data visualization and manipulation. Octave is normally used through its interactive command line interface, but it can also be used to write non-interactive programs. The Octave language is quite similar to Matlab so that most programs are easily portable."

OCTAVE was chosen as a proof of the concept of combining KAIRA with a high-level tool. Such a connection can improve the prototyping capabilities of both tools in the area of MPI applications. For OCTAVE, there already exists package OPENMPI_EXT that provides bindings of basic MPI functions. But OCTAVE itself is not very suitable for working with distributed memory applications; therefore, it could be very hard to debug an application or perform other supportive activities. Generic tools for C/C++ applications can be also hard to use, because they usually do not count with such use; starting such an application correctly or understanding results

[6] `http://www.gnu.org/software/octave/`

could be a problem without knowledge of the internal C++ implementation of Octave. With Kaira, the user gains the complete infrastructure presented in this thesis. Including the possibility to create a library in the RPC mode; therefore, Octave may run on a local computer and demanding parts are sent to a cluster.

Kaira automatically creates all necessary codes to connect Octave and the C++ library; therefore parallel codes generated by Kaira can be directly called from Octave. With basic types (`int`, `std::string`, etc.) or Octave's C++ types (like `Matrix` in the example from Section 4.4), no additional action has to be taken. To transfer other C++ types between a net and Octave, conversion functions have to be written; this is done in a similar way as for the definition of packing and unpacking functions for a type.

Let us assume that the matrix multiplication example from Section 4.4 is generated as an Octave library named *kaira-mult*. Then the following code can be used in Octave:

```
source kaira-mult.m # Load library
A = [ 1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16; ];
B = [ 10 20 30 40; 50 60 70 80; 90 10 11 12; 13 14 15 16 ];
C = matmult(A, B); # Call function
```

6.6.3 Drawbacks

In this section, the main issues of Kaira are summarized. They can be divided into two categories, issues of the current implementation and ones that fundamentally follow from the used approach. The former category includes the absence of collective communication in Kaira. MPI contains a set of functions that implements communication in which more nodes are involved in the same time. It allows efficient implementation of operations like a broadcast or scatter-gather over all processes. The current implementation of Kaira does not use collective communication in generated codes; all operations are implemented through the point-to-point communication. Therefore applications that heavily use collective communication can be implemented in a more efficient way by a direct usage of MPI than in Kaira. The implementation of collective communication into Kaira is proposed as the first task for future works (Section 8.1.1).

The second flaw is connected with the performance of some analyses. We are focused on the ability to generate applications running with a minimal overhead when they are simulated or traced. On the other hand, some post-process analyzing codes are not optimized in the current version. In the case of tracing, it works well for the scale of problems demonstrated in this thesis, but Kaira is not suitable now to analyze or display a tracelog obtained by a longer computation with hundreds of processes. Also sizes of instances for verification is limited, as was said in Section 6.5.

The second group of issues (the fundamental issues) mostly stem from the decision to use visual programming in Kaira. Many tools used by programmers like version control systems or documentation generators assume that source codes are provided in a form of plain text files. Therefore the usage of such tools may bring issues; for example, merging changes made in a single net. These problems generally occurs in visual programming and solving them is out of the scope of this thesis. Specialized tools that understand a particular system is usually used instead of generic ones. But such specialized tools does not exists yet for Kaira.

These problems are common not only for visual programming tools, but also for other tools that do not store source codes of developed applications as plain text files; Smalltalk implementations can be named as an example. On the other hand, Kaira is focused on applications developed by individuals or small teams; therefore problems with these external tools are usually not critical.

Chapter 7

Implementation

This section describes the internal structure of Kaira. At the beginning the architecture of Kaira is introduced. The next section covers the behavior of generated applications. At the end of this chapter, the way to obtain error messages from a C++ compiler is described.

7.1 Architecture

The Kaira consists of three main parts:

- *Gui* – The most visible part from the user's perspective. It allows editing of nets and source codes, control of simulations, and showing of analyses results. All screenshots of Kaira in this thesis are screenshots of this component.
- *PTP* (Project-To-Program) – It is the compiler that translates nets into a C++ code. It is a main part of Kaira.
- *Libraries* – Kaira consists of six C++ libraries that are linked together with resulting programs:
  - *Cailie* is the base library that is always linked with a generated program.
  - *CaVerif* is linked if a state-space analysis is performed.
  - *CaSimrun* is linked if a performance prediction is performed.
  - *CaOctave* is linked if the Octave integration is used.
  - *CaClient* and *CaServer* are used if a RPC library is built.

Figure 7.1 shows the process of building an executable application, a library, or an RPC server/client. A project file contains nets, user-defined sequential source codes, and global configurations. When the project is built, the first step is to

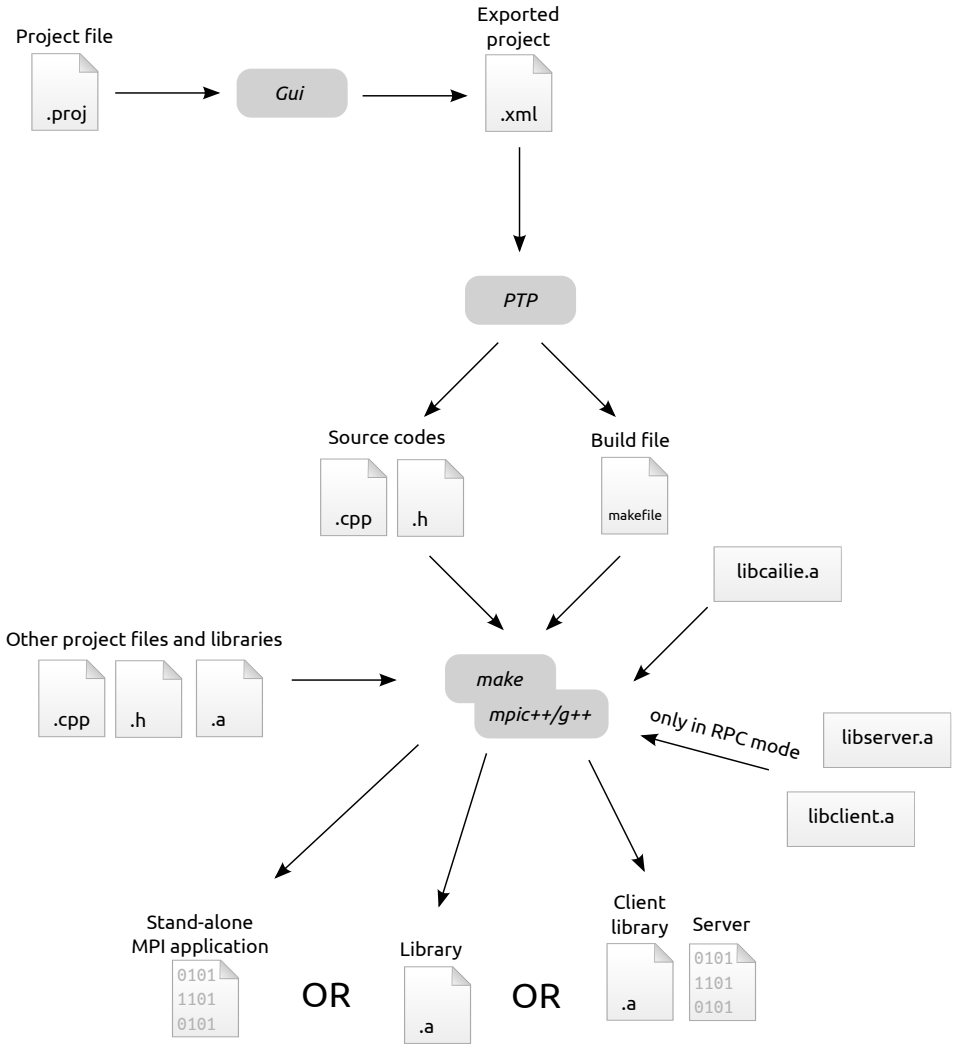


Figure 7.1: Building a program or a library in Kaira

generate an internal description of the project for PTP by Gui. Gui just simply preprocesses net descriptions. From this description, PTP generates C++ source files and a makefile. After that, MAKE builds a stand-alone application, a library, or an RPC server-client pair depending on the setting.

Gui and PTP are implemented in PYTHON. Gui moreover uses GTK+[1] as the library for the graphical user interface, MATPLOTLIB[2] for drawing charts and GTKSOURCEVIEW[3] for the text-editing widget with syntax highlighting. Library CaVerif internally uses SPARSEHASH[4] and MHASH[5].

7.2 Generated programs

The topic of this section is internal parts of generated programs. The behavior of a generated application is quite simple. Originally, most of the implementation was created as a prototype, and the plan was to replace the prototype with a more sophisticated code in the future. But the analysis of the performance showed that it works well for real examples and we focused on other topics. The only problem that was solved in a larger scale is avoiding unnecessary data copying. It is covered later in this section.

A program generated from a net is managed by a simple scheduler whose implementation can be described as the pseudo-code in Algorithm 7.4. Each MPI process has its own instance of the scheduler that manages only transitions and places that belong to the net-instance assigned to the process. The main task of the scheduler is to fire transitions. It picks a transition from set *ready* with a highest priority, i.e. a potentially enabled transition. The function `fire_transition` tries to fire it and returns `true` if the transition was fired. If `false` is returned, then the transition was not enabled and the scheduler removes the transition from set *ready*. The transition is returned into this set only if at least one of its input places is changed, i.e. another transition puts there a token or a token is received from the network. This functionality is implemented in `fire_transition` and receiving functions. Every iteration, the scheduler tries to receive MPI messages and finish MPI sends. All messages are sent by non-blocking MPI calls, hence active sends are periodically checked if they are finished and resources can be freed.

Function `fire_transition` executes a specific code for each transition that depends on transition's arcs, the guard expression and the assigned C++ code. If C++ code in transition calls `ctx.quit()` then *quit* is set to `true`. Assume example 11 in

[1] `http://www.gtk.org/`
[2] `http://matplotlib.org/`
[3] `https://projects.gnome.org/gtksourceview/`
[4] `http://code.google.com/p/sparsehash/`
[5] `http://mhash.sourceforge.net/`

Algorithm 7.4 The pseudo-code of the main cycle in a generated application

```
quit ← false
ready ← all transitions in the net
while ¬quit do
    try_receive_mpi_messages()                ▷ Can modify set ready
    try_finish_sends()
    if ready = ∅ then
        wait_for_mpi_message()                ▷ Can modify set ready
    end if
    t ← a transition with the highest priority from ready
    if ¬fire_transition(t) then               ▷ Can modify set ready or flag quit
        ready ← ready \ {t}
    end if
end while
```

Algorithm 7.5 The pseudo-code generated for the transition from example 11 in Figure 3.9

```
if place a is empty then
    return false
end if
t ← the first token in a
remove the first token from a
send t.value to place c in process 2
put t into b
return true
```

Figure 3.9, then the code like in Algorithm 7.5 is generated for the transition in the net.

The execution of this simple scheduler introduces a minimal overhead. The only real performance issue is the cost of token manipulations. Moving tokens of types like `int` has a negligible overhead, but it would be a significant overhead when a large matrix would be copied in a case where a token is just moved from one place to another. By a simple analysis it can be detected when a token can be moved from one place to another, as in the case of Algorithm 7.5. Places are implemented as double-linked lists of tokens, where a token is directly a node in the linked list containing the value and pointers to the previous and the next node (token) in the list. A token can be directly moved between an input and output place when there is an input inscription and an output inscription with the same main-expression that is a variable and both inscriptions are not bulk. It covers most of the cases where this optimization can be used. There are also some small tweaks in pairing input/output inscriptions (e.g. preferring inscriptions without `if` keyword), but they are rarely used.

The described scheduler is used when the final stand-alone application is generated. When the user starts a visual debugger, then PTP generates the application where the scheduler is replaced by a controller that listens commands from Gui via a TCP/IP connection. The same infrastructure is used for dynamic connecting into running applications (the feature mentioned in Section 6.2). In the case of simulated runs for performance predictions, the scheduler fires transitions sequentially and maintains a virtual global clock to compute the effects of delays and modified transition execution times. In the case of the state-space analysis, the scheduler explores all possible behaviors and constructs the state-space graph.

7.3 Error messages

Showing correct and precise error messages is a crucial feature to make a practical tool. If a tool gives no or misleading error messages, then it quickly becomes very frustrating and time consuming to use.

In Kaira, it is important to ensure good error messages for expressions used in nets. Originally, a simple domain-specific language was used for these expressions (up to version 0.5). Hence, syntax and type checking were quite easy. But it became more complicated when general C++ expressions started to be used. C++ is a complex programming language; the draft of the latest standard [65] has more than 1300 pages. It is not possible to create and maintain our own front-end of a C++ compiler in our small group. Therefore it is necessary to reuse an existing C++ compiler. There are several libraries that provide analysis of C++ codes; one of the

most advanced is LIBCLANG[6]. But these tools are usually designed for usages like automatic code completing in programming editors; therefore they are prepared to work with complete source code files, and checking a single expression under certain conditions is problematic.

Given this background, the author has developed a different and surprisingly simple idea. It consists of two parts. The first part is the implementation of the parser that accepts a very simple superset of C++ syntax. Basically it only checks if the expression roughly "looks like" a C++ expression; that is, it checks some trivial errors, mainly that all parentheses, string literals, and comments are correctly closed. When an expression is accepted by this simple parser, the second step is to create a specially crafted C++ code where each line may contain at most one more error than the previous line. When such a code is compiled by a C++ compiler and it returns some errors, the smaller error line is taken to determine the actual error. Because the simple parser has already checked issues like unclosed parenthesis or comments, the error cannot be detected on a later line than it really occurs.

This approach can be demonstrated on the following example. Assume that we need to check expression `f(a.x + 10)` where variable `a` has type `MyStruct` and the expression has to return type `MyType`. First, the simple parser checks that the expression looks "ok". The code generated in the next step is shown in Listing 7.1; it does not contain intentionally any indentation. If an error occurs at line 1 then `MyStruct` is not defined; if the first error occurs at line 2 then `MyType` is not defined. If the first error occurs at line 4 then the expression is invalid. For example, `f` does not exist, takes different argument(s), or `MyStruct` does not have attribute `a`. It is possible to decompose the expression and determine the error more precisely by the same method. But in this case, an error message from the C++ compiler can be directly taken and presented to the user. Compilers return errors in the format "`filename:linenumber:error message`", hence the prefix can be cut off and the error message for the expression obtained, including information about the column if it is provided. If the first error occurs in line 7 then the type returned by the expression is wrong. This error has to be checked separately because the error message obtained from the compiler may refer to the returning type of the crafted function and it would make no sense to the user. But if any error does not occur earlier, it is the only possible error on this line, hence the error is known and an error message can be provided.

Such crafted codes can be created for all expressions that need checking and all these codes can be put in a single file, allowing a single run of the compiler to check all expressions. This technique works across compilers; the quality of error messages evolves together with C++ compilers; it should also work for future standards of C++; the only thing that has to be maintained is a very simple parser.

[6] `http://clang.llvm.org/doxygen/group__CINDEX.html`

Listing 7.1: The example code generated for checking the correctness of an expression

```
1 void __test1(MyStruct &__arg1) {}
2 void __test2(MyType &__arg1) {}
3 void __test3(MyStruct &a) {
4 f(a.x + 10);
5 }
6 MyType __test4(MyStruct &a) {
7 return f(a.x + 10);
8 }
```

Chapter 8

Conclusions

This dissertation thesis defines and demonstrates ideas related to an abstract programming environment for the development of parallel applications in the area of distributed memory. Through this work, it was shown that various problems can be expressed in the proposed abstract model in a way that provides a simplifying and unifying background for rapid development and various supportive activities. The most important findings from this work can be summarized as follows:

- The user creates a model of communication and parallel aspects in the visual way. It was demonstrated that the diagrams of various parallel algorithms stay small. The structure of the program can be easily modified and the user can experiment with the developed program.
- Semantics of the computation model was formally defined.
- A thin layer between model and MPI allows generation of effective C++ programs from the visual language. The generated programs have a comparable performance to manually created C++ applications. This approach also allows use of existing tools for MPI applications together with programs generated by Kaira.
- Developed programs can be run in the visual debugger where their inner state can be observed and controlled. This allow us to observe the behavior of the developed program from its very early stages. The debugger is controlled in a simple well-formed way. It allows to work with sequences of high-level events that can get the program into required states. These sequences can be created by the simulator or can be obtained as a result of analyses. Therefore, when a state of a developed application is obtained from any analysis, the application can be put into this state at any time and debugged by our debugger.
- Tracing of programs is possible in a way that allows the configuration and presentation of results through abstract terms of the model. Collected data

can be simply exported for further post-processing and the traced run can be replayed in the same visual way as for the visual debugger.

- The proposed tool offers predictions of application behaviors through online simulations with an analytical model of the network. The used model allows simple configuration of the predictions and observation of the results. During predictions, the complete tracing infrastructure is available. It can be used to check various "what if ..." scenarios.

- The approach used allows implementation of the state-space analysis of the developed program together with easily configurable analyses. The presented approach of implementing formal methods as a natural component of the development environment can increase the utilization of this important part of computer science among MPI users.

- It was shown that the presented model can serve also for development of parallel libraries usable in any sequential C++ applications. This infrastructure is also used for combining Kaira and Octave to obtain a domain-specific prototyping environment for developing parallel applications.

All ideas in this thesis and their implementation are presented with the hope that they will serve well as a useful and practical tool for programmers of parallel applications. As always, there is space for improvements and new ideas. The author plans and hopes that after finishing this thesis, he will have the opportunity to continue working on Kaira and to expand the tool in many different ways. Possible avenues for future development are outlined in the next chapter.

8.1 Ideas for future work

Work on Kaira does not end with the completion of this thesis. There are still many ways in which Kaira can be improved. This section contains three directions that the author has identified as the most interesting ideas that should also have a practical impact on the use of Kaira.

8.1.1 Collective communication

As was noted in Section 6.6.3, Kaira in the current version is not able to utilize collective communication. Making this important part of MPI accessible increases the number of problems that can be solved by Kaira. The solution has to include two parts. The first part is to enrich the programming language by this construct. The plan is to introduce a new transition with a special semantics that will capture a collective behavior. The second part is to update analytical parts of Kaira. Collective

communication has a more complex behavior that depends on an implementation of MPI. To offer performance analyses and predictions, it will be necessary to create a tighter connection between Kaira and an existing implementation of MPI or to use a more sophisticated simulator.

8.1.2 Advanced libraries

Kaira generates libraries where each net represents a function that blocks sequential computation until the function does not finish its computation. It is limiting for some usages and it can also cause an issue in which the same data are repeatedly transferred between the main program and computing nodes for each call. It can be provisionally solved by saving data into files, but it is not a very elegant way. For these reasons, there is a space to introduce more advanced libraries in which generated functions can run concurrently together with the main code. Moreover the main program will be able to dynamically influence the parallel parts, like fill new data, and take partial results.

The preliminary ideas are based on special transitions that are not fired by the scheduler but for each such transition a function in the library will be generated. Calling this function in the program will cause firing this transition. Hence, semantics of Kaira will be used to introduce a simple and well-formed control of distributed applications.

8.1.3 Hybrid computation

Kaira was originally designed as the tool for distributed memory computations. On the other hand, these days many parallel systems are composed of multi-core shared memory nodes; from a global perspective the system has distributed memory but inside each node, a shared-memory parallel computation can be performed. To fully utilize such a computer, systems like MPI and OpenMP[1] are used together. Because these technologies were not designed for such usage, using them in combination brings new issues [66]. PGAS programming languages already mentioned in Section 2.2, were designed to simplify the programming of such systems.

Between versions 0.3 and 0.6 of Kaira, we tried to address issues related to combining shared memory and distributed memory programming. The result is that with Kaira it is possible to run applications in the mode where the behavior can be roughly described as follows: if there are more enabled transitions in a process, then they are performed simultaneously by threads. This feature still remains in version 1.0, but it is considered to be experimental; it is not optimized and some analyses do not support this mode. The work on this aspect was suspended, because the

[1] "The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran." – `http://openmp.org`

proposed semantics was not sufficiently supported by practical examples and some implementation problems surfaced.

The author still believes that the potential exists for the use of Kaira in the design of applications where shared memory and distributed memory approaches are combined. However, as it stands now more thinking is needed on the topic. One way is to extend the current approach. The runtime will use transparently more threads to execute more enabled transitions but without visible effects, hence the overall result will be the same as a run with a single thread. It solves some problems of the current implementation and makes it also compatible with a large part of the current implementation. On the other hand, the user will lose some control over the application, which is counter to the idea of a general prototyping tool. Another direction is to change the visual language and introduce some shared memory constructs. But it is not clear what constructions should be proposed to achieve sufficiently universal, efficiently implementable semantics with good support by practical examples.

8.1.4 Use of Kaira in education

Although it is not the main objective, Kaira could naturally be used to teach the message-passing paradigm. A student can reuse knowledge of sequential programming in C++ and communication constructs can be explained on examples with immediate feedback available through the visual simulator. Kaira can also serve for a quick demonstration of supportive activities used in programming without the need to explain many technical details. The main obstacle for using the tool in this way is the lack of documentation that introduces the message-passing paradigm and parallel programming through the use of Kaira. The existing documentation is focused on the reader that is already familiar with these topics.

Závěr

Tato disertační práce definovala a demonstrovala vývojové prostředí pro vývoj paralelních aplikací v prostředí systémů s distribuovanou pamětí. Práce je založena na abstraktním výpočetním modelu v kombinaci s vizuálním jazykem určeným pro prototypování a implementaci programů ve zmíněné oblasti. Dále tento model slouží jako sjednocující základ pro rychlý vývoj aplikací a další podpůrné aktivity, které se objevují v průběhu vývoje. Základní aspekty prezentovaného nástroje mohou být shrnuty následovně:

- Uživatel vytváří model komunikace a paralelních částí ve vizuálním jazyce. Práce demonstruje, že takto vytvořené diagramy mají relativně malou velikost a zůstávají přehledné i v případě řešení praktických problémů. Vizuální jazyk usnadňuje změnu programu a uživatel může jednoduše experimentovat s vyvíjenými programy.

- Sémantika výpočetního modelu je plně formalizována.

- Tenká vrstva mezi modelem a MPI umožňuje generovat efektivní C++ aplikace z vizuálního jazyka. Generované programy mají srovnatelný výkon s manuálně vytvořenými C++ programy. Tento přístup také zpřístupňuje využití mnoha existujících nástrojů pro analýzu programů vygenerovaných nástrojem Kaira.

- Vygenerované programy lze spustit ve vizuálním debuggeru, ve kterém je možné zobrazit a řídit jejich vnitřní stav. Tímto způsobem může uživatel pozorovat chování aplikace od raných fází vývoje. Debugger je ovládán pomocí dvou vysokoúrovňových operací, což nabízí jeho řízení pomocí sekvencí příkazů, které umožňují směřovat program do požadovaného stavu. Tyto sekvence mohou být vytvořeny simulátorem nebo získány jako výsledek analýz.

- Tracování programů je poskytováno ve formě, která využívá abstraktní model ke konfiguraci a prezentaci výsledků. Takto získaná data mohou být jednoduše exportována a dále zpracovávána nebo může být zaznamenaný běh zobrazen stejným způsobem jako v případě vizuálního debuggeru.

- Kaira poskytuje možnost online simulací s analytickým modelem sítě. Skrze vizuální model může být simulátor konfigurován pro specifické nastavení experimentů. Zároveň je k dispozici kompletní tracovací infrastruktura.
- Navržený model umožnil jednoduchou implementaci verifikace pomocí analýzy stavového prostoru.
- Bylo demonstrováno použití nástroje Kaira pro generování paralelních C++ knihoven. Tato infrastruktura byla dále využita při propojení Kairy s Octave.

Další vývoj

Dokončením této práce není vývoj nástroje Kaira ukončen, je zde stále mnoho oblastí, ve kterých může být rozšiřován. Tři hlavní směry vývoje jsou: implementace kolektivní komunikace, rozšířených knihoven a hybridního módu. Kolektivní komunikace je důležitá součást MPI, která není v současné verzi Kairy využita. Přidání této funkcionality umožní uživateli vytvářet efektivní implementace širšího spektra programů.

Generování rozšířených knihoven má za cíl souběžné zpracování hlavního kódu spolu s knihovními voláními, do kterých může hlavní kód dynamicky zasahovat. Hlavním cílem je odstranit nutnost znovu rozesílat data při každém volání knihovní funkce. Implementace této vlastnosti může být založena na rozšíření vizuálního jazyka o speciální přechody, které nejsou prováděny plánovačem, ale které lze ovládat ze sekvenčního kódu.

Hybridní mód je zde myšlen ve smyslu plného využití systémů, ve kterých je kombinována sdílená a distribuovaná paměť. Velké množství paralelních počítačů je dnes vytvořeno jako cluster s multiprocesorovými uzly se sdílenou pamětí. Autor práce věří, že navržený výpočetní model bude možné rozšířit o prvky, které zpřístupní vývoj aplikací i v této oblasti.

Author's publications

Publications related to the thesis

- Böhm, S., Běhálek, M.: Generating parallel applications from models based on Petri nets. Advances in Electrical and Electronic Engineering 10 (1) (2012) [SCOPUS]

- Meca, O., Böhm, S., Běhálek, M., Šurkovský, M.: Prototyping framework for parallel numerical computations. In Proceedings of: The 10th International Conference on Parallel Processing and Applied Mathematics (PPAM) (2013)

- Böhm, S., Běhálek, M., Meca, O., Šurkovský, M.: Visual programming of mpi applications: Debugging and performance analysis. In Proceedings of: The 4th Workshop on Advances in Programming Language (WAPL) (2013)

- Böhm, S., Běhálek, M., Meca, O.: Kaira: Generating parallel libraries and their usage with octave. In Proceedings of: Languages and Compilers for Parallel Computing (LCPC). Volume 7760 of Lecture Notes in Computer Science. Springer (2013). 268–269

- Böhm, S., Běhálek, M.: Usage of Petri nets for high performance computing. In Proceedings of: 1st ACM SIGPLAN workshop on Functional high-performance computing. (FHPC), New York, NY, USA, ACM (2012). 37–48 [SCOPUS]

- Böhm, S., Běhálek, M., Garncarz, O.: Developing parallel applications using Kaira. In Proceedings of: Digital Information Processing and Communications. Volume 188 of Communications in Computer and Information Science., Springer (2011). 237–251 [WoS]

- Běhálek, M., Böhm, S., Krömer, P., Šurkovský, M., Meca, O.: Parallelization of ant colony optimization algorithm using Kaira. In Proceedings of: 11th International Conference on Intelligent Systems Design and Applications (ISDA), Cordoba, Spain (2011). 510-515 [SCOPUS]

- Böhm, S., Běhálek, M.: Kaira: Modelling and generation tool based on Petri nets for parallel applications. In Proceedings of: UkSim 13th International Conference on Computer Modelling and Simulation (2011). 403–408 [SCOPUS]

Publications related to one-counter automata

- Böhm, S., Göller S., Jančar, P.: Equivalence and regularity for real-time one-counter automata. Journal of Computer and System Sciences (JCSS), Elsevier, http://dx.doi.org/10.1016/j.jcss.2013.11.003 [IF 1.000]
- Böhm, S., Göller S., Jančar, P.: Equivalence of deterministic one-counter automata is NL-complete. In Proceedings of: 45th ACM Symposium on the Theory of Computing (STOC), Palo Alto, CA, USA, June 1-4 (2013). 131–140 [SCOPUS]
- Böhm, S., Göller S.: Language equivalence of deterministic real-time one-counter automata is NL-complete. In Proceedings of: 36th International Symposium on Mathematical Foundations of Computer Science (MFCS), Lecture Notes in Computer Science, Springer (2011). 194–205 [WoS]
- Böhm, S., Göller S., Jančar, P.: Bisimilarity of one-counter processes is PSPACE-complete. In Proceedings of: 21st international conference on Concurrency theory (CONCUR), Lecture Notes in Computer Science 6269, Springer (2010). 177–191 [WoS]

Bibliography

[1] Böhm, S., Běhálek, M.: Kaira: Modelling and generation tool based on Petri nets for parallel applications. In: UkSim 13th International Conference on Computer Modelling and Simulation. (2011) 403 –408

[2] Böhm, S., Běhálek, M., Garncarz, O.: Developing parallel applications using Kaira. In: Digital Information Processing and Communications. Volume 188 of Communications in Computer and Information Science., Springer (2011) 237–251

[3] Běhálek, M., Böhm, S., Krömer, P., Šurkovský, M., Meca, O.: Parallelization of ant colony optimization algorithm using Kaira. In: 11th International Conference on Intelligent Systems Design and Applications (ISDA 2011), Cordoba, Spain (2011)

[4] Böhm, S., Běhálek, M.: Generating parallel applications from models based on Petri nets. Advances in Electrical and Electronic Engineering **10**(1) (2012)

[5] Böhm, S., Běhálek, M.: Usage of Petri nets for high performance computing. In: Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing. FHPC '12, New York, NY, USA, ACM (2012) 37–48

[6] Böhm, S., Běhálek, M., Meca, O.: Kaira: Generating parallel libraries and their usage with octave. In: Languages and Compilers for Parallel Computing (LCPC). Volume 7760 of Lecture Notes in Computer Science. Springer (2013) 268–269

[7] Meca, O., Böhm, S., Běhálek, M., Šurkovský, M.: Prototyping framework for parallel numerical computations. In: The 10th International Conference on Parallel Processing and Applied Mathematics (PPAM). (2013)

[8] Böhm, S., Běhálek, M., Meca, O., Šurkovský, M.: Visual programming of mpi applications: Debugging and performance analysis. In: The 4th Workshop on Advances in Programming Language (WAPL). (2013)

[9] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI-The Complete Reference, Volume 1: The MPI Core. 2nd. (revised) edn. MIT Press, Cambridge, MA, USA (1998)

[10] Hempel, R., Walker, D.W.: The emergence of the mpi message passing standard for parallel computing. Comput. Stand. Interfaces **21**(1) (1999) 51–62

[11] Squyres, J.M.: Mpi debugging – can you hear me now? ClusterWorld Magazine, MPI Mechanic Column **2**(12) (2004) 32–35

[12] Squyres, J.M.: Debugging in parallel (in parallel). ClusterWorld Magazine, MPI Mechanic Column **3**(1) (2005) 34–37

[13] Krammer, B., Bidmon, K., Müller, M., Resch, M.: Marmot: An {MPI} analysis and checking tool. In G.R. Joubert, W.E. Nagel, F.P., Walter, W., eds.: Parallel Computing Software Technology, Algorithms, Architectures and Applications. Volume 13 of Advances in Parallel Computing. North-Holland (2004) 493 – 500

[14] Luecke, G.R., Chen, H., Coyle, J., Hoekstra, J., Kraeva, M., Zou, Y.: Mpi-check: a tool for checking fortran 90 mpi programs. Concurrency - Practice and Experience **15**(2) (2003) 93–100

[15] Vetter, J.S., de Supinski, B.R.: Dynamic software testing of mpi applications with Umpire. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM). Supercomputing '00, Washington, DC, USA, IEEE Computer Society (2000)

[16] Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. **17**(6) (1982) 120–126

[17] Weidendorfer, J.: Sequential performance analysis with callgrind and kcachegrind. In: Parallel Tools Workshop. (2008) 93–113

[18] Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: A scalable tool architecture for diagnosing wait states in massively parallel applications. Parallel Comput. **35**(7) (2009) 375–388

[19] Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience **22**(6) (2010) 702–719

[20] Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2) (2006) 287–311

[21] Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M., Nagel, W.: The Vampir performance analysis tool-set. In Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A., eds.: Tools for High Performance Computing. Springer Berlin Heidelberg (2008) 139–155

[22] Pillet, V., Pillet, V., Labarta, J., Cortes, T., Cortes, T., Girona, S., Girona, S., Computadors, D.D.D.: Paraver: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18 (1995)

[23] J. Chassin de Kergommeauxa, B. Steinb, P.B.: Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. Parallel Comput. **26**(10) (2000) 1253–1274

[24] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. Concurr. Comput. : Pract. Exper. **22**(6) (2010) 685–701

[25] Bordner, J.: MPI performance tools. `http://lca.ucsd.edu/projects/phys244/raw-attachment/wiki/LectureNotes/bordner-perf.pdf` (2012)

[26] Kerbyson, D.J., Alme, H.J., Hoisie, A., Petrini, F., Wasserman, H.J., Gittings, M.: Predictive performance and scalability modeling of a large-scale application. In: SC. (2001) 37

[27] Zheng, G., Wilmarth, T., Jagadishprasad, P., Kalé, L.V.: Simulation-based performance prediction for large parallel machines. Int. J. Parallel Program. **33**(2) (2005) 183–207

[28] Casanova, H., Legrand, A., Quinson, M.: Simgrid: A generic framework for large-scale distributed experiments. In: Proceedings of the Tenth International Conference on Computer Modeling and Simulation. UKSIM '08, Washington, DC, USA, IEEE Computer Society (2008) 126–131

[29] Penoff, B., Wagner, A., Tüxen, M., Rüngeler, I.: MPI-NetSim: A network simulation module for MPI. In: Proc. of the 15th International Conference on Parallel and Distributed Systems. (2009)

[30] Tikir, M., Laurenzano, M., Carrington, L., Snavely, A.: Psins: An open source event tracer and execution simulator for mpi applications. In Sips, H., Epema, D., Lin, H.X., eds.: Euro-Par 2009 Parallel Processing. Volume 5704 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 135–148

[31] Zhai, J., Chen, W., Zheng, W.: Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. SIGPLAN Not. **45**(5) (2010) 305–314

[32] Hermanns, M.A., Geimer, M., Wolf, F., Wylie, B.J.N.: Verifying causality between distant performance phenomena in large-scale mpi applications. In: Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on. (2009) 78–84

[33] Allan, R., Science, Britain), T.F.C.G.: Survey of HPC Performance Modelling and Prediction Tools. Technical report (Science and Technology Facilities Council (Great Britain))). Science and Technology Facilities Council (2010)

[34] Pllana, S., Brandic, I., Benkner, S.: Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art. In: CISIS. (2007) 279–284

[35] Siegel, S., Avrunin, G.: Verification of halting properties for mpi programs using nonblocking operations. In Cappello, F., Herault, T., Dongarra, J., eds.: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 4757 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 326–334

[36] Holzmann, G.: Spin model checker, the: primer and reference manual. First edn. Addison-Wesley Professional (2003)

[37] Vakkalanka, S.S., Sharma, S., Gopalakrishnan, G., Kirby, R.M.: Isp: a tool for model checking mpi programs. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. PPoPP '08, New York, NY, USA, ACM (2008) 285–286

[38] Godefroid, P.: Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem (1995)

[39] Gopalakrishnan, G.L., Kirby, R.M.: Top ten ways to make formal methods for hpc practical. In: Proceedings of the FSE/SDP workshop on Future of software engineering research. FoSER '10, New York, NY, USA, ACM (2010) 137–142

[40] Träf, J.L.: History and development of the MPI standard. slides (2013)

[41] Balay, S., Brown, J., , Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory (2013)

[42] Heroux, M., Bartlett, R., Hoekstra, V.H.R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., Williams, A.: An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories (2003)

[43] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1) (2008) 107–113

[44] Chen, W.Y., Song, Y., Bai, H., Lin, C.J., Chang, E.: Parallel spectral clustering in distributed systems. Pattern Analysis and Machine Intelligence, IEEE Transactions on **33**(3) (2011) 568–586

[45] Zhao, J., Pjesivac-Grbovic, J.: Mapreduce: The programming model and practice (2009) Tutorial.

[46] Stephens, R.: A survey of stream processing. Acta Informatica **34**(7) (1997) 491–541

[47] Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. CCS-TR-99-157 (1999)

[48] Inc, C.: Chapel Language Specification. (2011)

[49] Browne, J.C., Dongarra, J., Hyder, S.I., Moore, K., Newton, P.: Visual programming and parallel computing. Technical report, Knoxville, TN, USA (1994)

[50] Kacsuk, P., Cunha, J.C., Dózsa, G., Lourenço, J.a., Fadgyas, T., Antão, T.: A graphical development and debugging environment for parallel programs. Parallel Comput. **22**(13) (1997) 1747–1770

[51] Newton, P., Khedekar, S.Y.: (CODE 2.0 User and Reference Manual) `http://www.cs.utexas.edu/~code/download/docs/CODE_2.0_Manual.ps`.

[52] Newton, P., Browne, J.C.: The code 2.0 graphical parallel programming language. In: Proceedings of the 6th international conference on Supercomputing. ICS '92, New York, NY, USA, ACM (1992) 167–177

[53] Hyder, S., Werth, J., Browne, J.: A unified model for concurrent debugging. In: Parallel Processing, 1993. ICPP 1993. International Conference on. Volume 2. (1993) 58–67

[54] Kacsuk, P.: Performance visualization in the grade parallel programming environment. In: High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on. Volume 1. (2000) 446–450 vol.1

[55] Reisig, W., Rozenberg, G., eds.: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996. In Reisig, W., Rozenberg, G., eds.: Petri Nets. Volume 1491 of Lecture Notes in Computer Science., Springer (1998)

[56] Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)

[57] Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and cpn tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**(3) (2007) 213–254

[58] Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for petri nets: Renew. In Cortadella, J., Reisig, W., eds.: ICATPN. Volume 3099 of Lecture Notes in Computer Science., Springer (2004) 484–493

[59] Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)

[60] Cannon, L.E.: A cellular computer to implement the kalman filter algorithm. PhD thesis, Bozeman, MT, USA (1969) AAI7010025.

[61] Dorigo, M., Stutzle, T.: The ant colony optimization metaheuristic: Algorithms, applications, and advances. In: Handbook of Metaheuristics. Volume 57 of International Series in Operations Research; Management Science. Springer New York (2003) 250–285

[62] Manfrin, M., Birattari, M., Stützle, T., Dorigo, M.: Parallel ant colony optimization for the traveling salesman problem. In: Proceedings of the 5th International Conference on Ant Colony Optimization and Swarm Intelligence. ANTS'06, Berlin, Heidelberg, Springer-Verlag (2006) 224–234

[63] Valmari, A.: Stubborn sets for reduced state space generation. In Rozenberg, G., ed.: Advances in Petri Nets 1990. Volume 483 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1991) 491–515

[64] Velho, P., Legrand, A.: Accuracy study and improvement of network simulation in the simgrid framework. In: Proceedings of the 2nd International Conference on Simulation Tools and Techniques. Simutools '09, ICST, Brussels, Belgium, Belgium, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2009) 13:1–13:10

[65] Working draft, standard for programming language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf` (2012)

[66] Rabenseifner, R., Hager, G., Jost, G.: Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. Parallel, Distributed, and Network-Based Processing, Euromicro Conference on **0** (2009) 427–436

Appendix A

Performance measurements

Execution times of the heaf flow example on Anselm (2600 × 8200, 5000 iterations)

	Kaira		Manual implementation		Difference	
# Processes	Mean	SD	Mean	SD	Absolute	Relative
2	145.66	0.04	151.40	0.42	-5.75	-0.04
4	90.66	3.22	89.07	0.41	1.60	0.02
8	57.08	1.07	55.68	0.69	1.39	0.03
12	41.85	0.31	41.20	0.42	0.66	0.02
16	32.96	0.13	32.07	0.17	0.88	0.03
20	28.71	0.17	27.65	0.03	1.06	0.04
24	24.93	0.04	24.85	0.08	0.09	0.00
28	25.09	0.20	22.16	0.07	2.93	0.13
32	22.01	0.18	20.69	0.08	1.32	0.06
36	20.92	0.17	22.34	0.59	-1.42	-0.06
40	19.31	0.14	19.36	0.18	-0.05	0.00

Execution times of the heat flow example on Anselm (2600 × 2600, 3000 iterations)

	Kaira		Manual implementation		Difference	
# Processes	Mean	SD	Mean	SD	Absolute	Relative
2	36.20	0.14	35.64	0.19	0.56	0.02
4	19.75	0.18	19.73	0.20	0.03	0.00
8	10.83	0.02	10.61	0.06	0.21	0.02
12	8.21	0.02	7.75	0.02	0.46	0.06
16	6.78	0.02	6.31	0.01	0.48	0.08
20	5.92	0.01	7.13	1.46	-1.21	-0.17
24	5.31	0.01	4.87	0.01	0.44	0.09
28	4.91	0.02	4.63	0.05	0.28	0.06
32	4.68	0.02	4.64	0.09	0.04	0.01
36	4.67	0.19	4.26	0.12	0.40	0.09
40	4.26	0.12	3.88	0.09	0.38	0.10

Execution times of the heat flow example on Hubert (2600×8200, 5000 iterations)

	Kaira		Manual implementation		Difference	
# Processes	Mean	SD	Mean	SD	Absolute	Relative
2	536.55	0.04	554.65	0.42	-18.10	-0.03
4	334.67	3.22	312.36	0.41	22.31	0.07
8	186.57	1.07	185.19	0.69	1.38	0.01
12	153.78	0.31	152.31	0.42	1.47	0.01
16	129.52	0.13	128.09	0.17	1.43	0.01
20	119.65	0.17	118.38	0.03	1.27	0.01
24	115.60	0.04	114.28	0.08	1.33	0.01
28	112.73	0.20	111.44	0.07	1.29	0.01
32	116.41	0.18	115.17	0.08	1.24	0.01

Execution times of the heat flow example on Hubert (2600×2600, 3000 iterations)

	Kaira		Manual implementation		Difference	
# Processes	Mean	SD	Mean	SD	Absolute	Relative
2	110.73	0.04	103.76	0.42	6.98	0.07
4	62.91	3.22	61.93	0.41	0.98	0.02
8	36.07	1.07	35.47	0.69	0.60	0.02
12	29.09	0.31	28.41	0.42	0.68	0.02
16	25.40	0.13	24.90	0.17	0.50	0.02
20	23.76	0.17	23.20	0.03	0.56	0.02
24	23.26	0.04	22.74	0.08	0.52	0.02
28	22.58	0.20	22.10	0.07	0.48	0.02
32	23.33	0.18	22.97	0.08	0.36	0.02