Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1483–1490

Visual Programming of MPI Applications: Debugging and Performance Analysis

Stanislav Böhm, Marek Běhálek, Ondřej Meca, Martin Šurkovský
Department of Computer Science
FEI VŠB Technical University of Ostrava
Ostrava, Czech Republic
stanislav.bohm@vsb.cz, marek.behalek@vsb.cz, ondrej.meca@vsb.cz, martin.surkovsky@vsb.cz

***Abstract*—Our research is focused on the simplification of parallel programming for distributed memory systems. Our overall goal is to build a unifying framework for creating, debugging, profiling and verifying parallel applications. The key aspect is a visual model inspired by Colored Petri Nets. In this paper, we will present how to use the visual model for debugging and profiling as well. The presented ideas are integrated into our open source tool Kaira.**

I. Introduction

PARALLEL computers with distributed memory have recently become more and more available. A lot of people can participate in developing software for them, but there are well-known difficulties of parallel programming. Therefore for many non-experts in the area of parallel computing (even if they are experienced sequential programmers), it can be difficult to make their programs run in parallel on a cluster computer. The industrial standard for programming applications in the area of distributed memory systems is *Message Passing Interface* (MPI)[1]. It represents a quite low-level interface. There are tools like *Unified Parallel C*[2] that simplify creating parallel applications, but the complexity of their development lies also in other supportive activities. Therefore, even an experienced sequential programmer can spend a lot of time learning a new set of tools for debugging, profiling, etc.

The overall goal of our research is to reduce complexity in parallel programming. We want to build a unified prototyping framework for creating, debugging, profiling and formally verifying parallel applications, where a user can implement and experiment with his/her ideas in a short time, create a real running program and verify its performance and scalability. The central role in our approach is a visual programming language (based on Petri nets) that we use for modeling developed applications. In this paper, we present how to use the same model for debugging and profiling. The presented ideas are implemented in Kaira[3], a tool that we are developing.

The work is partially supported by: GAČR P202/11/0340, the European Regional Development Fund in the IT4Innovations Center of Excellence project (CZ.1.05/1.1.00/02.0070)

[1]http://www.mpi-forum.org/
[2]http://upc.lbl.gov/
[3]http://verif.cs.vsb.cz/kaira

II. Tool Kaira

This section serves as an overview for our tool Kaira; for more details see [1], [2]. Our goal is to simplify the development of MPI parallel applications and create an environment where all activities are unified under one concept.

The key aspect of our tool is the usage of a visual model. In the first place, we have chosen the visual model to obtain an easy and clear way how to describe and expose parallel behavior of applications. The other reason is that a distributed state of the application can be shown through such visual model. The representation of an inner-state of distributed applications by a proper visual model can be more convenient than traditional ways like stack-traces of processes and memory watches. With this feature, we can provide visual simulations where the user can observe a behavior of developed applications. It can be used for incomplete applications from an early stage of the development. In a common way of developing MPI programs, it often takes a long time to get the developed application into a state where its behavior can be observed. In context of this paper, the visual model is also useful for debugging and a performance analysis as will be demonstrated later.

On the other hand, we do *not* want to create applications completely through the visual programming. Sequential parts of the developed application are written in the standard programming language (C++) and combined with the visual model that catches *parallel aspects* and *communication*. We want to avoid huge unclear visual diagrams; therefore, we visually represent only what is considered as "hard" in parallel programming. Ordinary sequential codes are written in a textual language. Moreover, this design allows for easy integration of existing C++ codes and libraries.

It is important to mention that our tool is *not* an automatic parallelization tool. Kaira does not discover parallelisms in applications. The user has to explicitly define them, but they are defined in a high-level way and the tool derives implementation details.

Semantics of our visual language is based on Coloured Petri nets (CPNs)[3]. Petri nets is a formalism for the description of parallel processes. They also provide well-established terminology, a natural visual representation for visual editing

978-1-4673-4471-5/$25.00

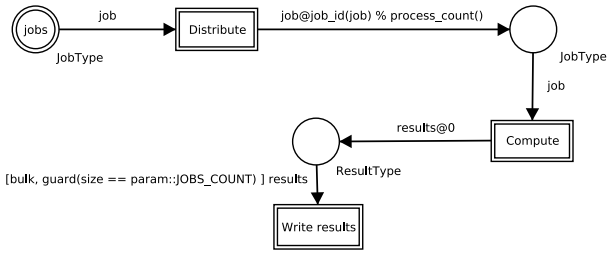

Fig. 1. The example model

of models and their simulations. Modeling tool *CPN Tools*[4] was also the great inspiration for us (especially how to visualize the model).

To demonstrate how our model works, let us consider the model in Figure 1. It presents a problem where some jobs are distributed across computing nodes and results are sent back to process 0. When all the results arrive, they are written into a file. Circles (*places* in terminology of Petri nets) represent memory spaces. Boxes (*transitions*) represent actions. Arcs run from places to transition (*input arcs*) or from transition to places (*output arcs*). The places contain values (*tokens*). Input arcs specify what tokens a transition needs to be *enabled*. An enabled transition can be executed. When a transition is executed, it takes tokens from places according to input arcs. After finishing the computation of the transition, new tokens are placed into places according to output arcs. In CPNs places store tokens as multisets, in our approach we use queues.

A double border around of a transition means that there is a C++ function inside and it is executed whenever the transition is fired. A double border of a place indicates an associated C++ function creating the place's initial content. Arcs' inscriptions use C++ enriched by several simple constructions. A computation described by this model runs on every process. Tokens can be transferred between processes by expressions after "@" symbol on output arcs.

As a more advance example, we use the heat flow problem on a cylinder. We will use a version of this problem where the body is discretized by a grid depicted in Figure 2. The implementation of this problem in Kaira is depicted in Figure 3. The transition *Compute* executes single iteration of the algorithm. It takes a process' part of the grid and two rows, one from neighbor above and one from below. It updates the grid and sends top and bottom rows to neighbors. When the limit of iterations is reached then the results are sent to process 0 where they are written. The init area (depicted as the blue rectangle) is used to set up initial values of places not only on process 0 but over specified processes (all processes in our case). The measurements of this program and a comparison to the sequential version are part of Section V.

[4]http://cpntools.org/

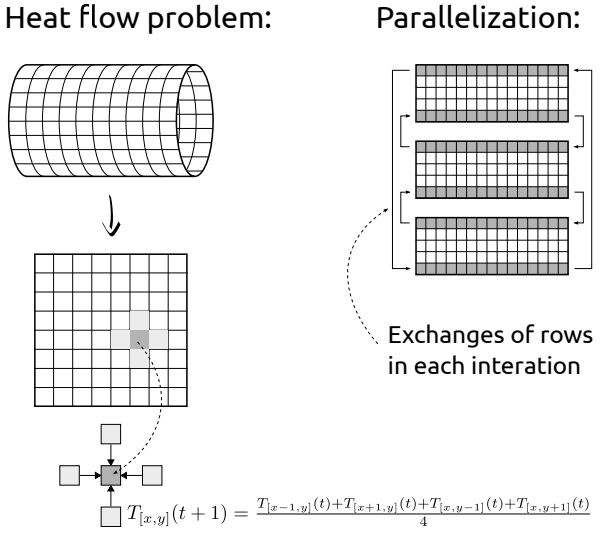

Fig. 2. The heat flow problem on a cylinder and the used method of parallelization

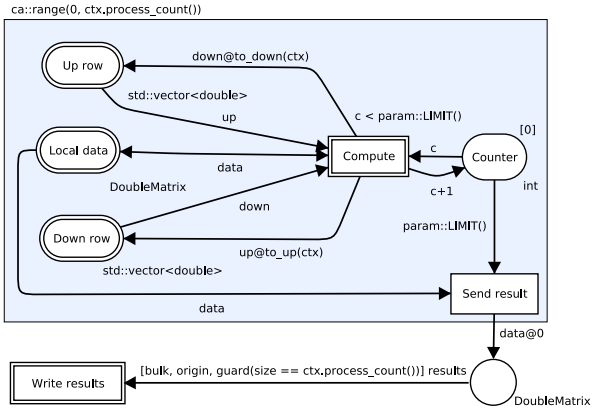

Fig. 3. The implementation of the heat flow problem in Kaira

III. Simulations and records of generated applications

In this section, we will introduce two crucial features: *simulations* and *tracing* of generated applications. Both can be used for debugging and the latter for profiling. Later we will describe two other features and we will also discuss the drawbacks of our approach.

A. Simulations

Besides generating standalone parallel applications from the model, the user can also run the developed application in the simulator. The main task of the simulator is to expose an inner state and it allows for controlling a run of the generated application. The inner state is shown in the form of labels over the original model (see Figure 4). The three types of information are depicted:

- Tokens in place (The state of memory)
- Running transitions (The state of execution)

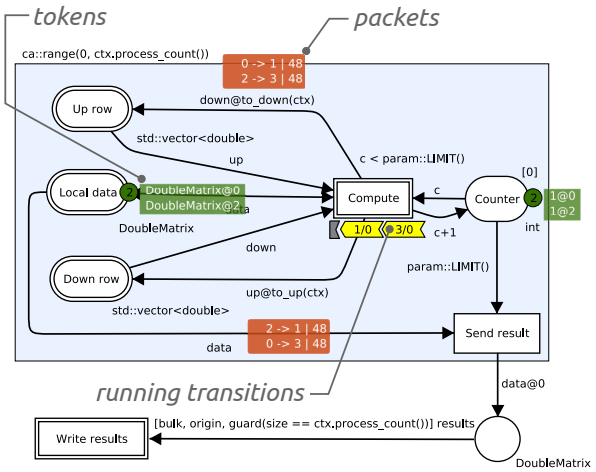

Fig. 4. The model in the simulator

- 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 three basic actions:

- Start an enabled transition
- Finish a running transition.
- Receive a packet from a network.

By executing these three types of actions, the application can be brought to any reachable state. The model naturally hides irrelevant states during sequential computations and only aspects important to parallel execution are visible and controllable.

This approach also gives us the possibility to observe the behavior of the application in a very early state of the development without any additional debugging infrastructure. For example, we can see which data are sent to another process even if there is no implementation of the receiving part.

The user has complete well-formed control of the application in the simulator; therefore, the application can be put into an interesting state (and the user can observe the consequences) even if the application rarely reaches such state.

B. Tracing

An application developed in Kaira can be generated in the tracing mode, where activities of a run of the application are recorded into a *tracelog*. When the application finishes its run, the tracelog can be loaded back into Kaira and used for the *visual replay* or for statistical summaries. Generally, issues with such post-mortem analysis can be categorized into these basic groups: *selection what to measure*, *instrumentation* and *presentation of results*. Such tracelogs can be useful both for profiling and debugging.

In the case of debugging, we usually want to collect detailed information of the run for the reconstruction of the cause of the problem. In the case of profiling, we want to discover performance issues and therefore need to measure a run with time characteristics as close as possible to real runs of the application. But the measurement itself creates an overhead that devalues the gathered information about performance. Therefore, in both cases, it is important to specify what to store in the tracelog. In common profilers, specifications of measurements are usually implemented as a list of functions that we want to measure/filter out. But it can be a non-trivial task to assemble such a list, especially in the case when we use some third-party libraries with an unclear purpose to the user. It often needs some experience to recognize what can be safely thrown away.

In Kaira, the user specifies what is measured in terms of places and transitions. It is done just by placing labels in a model (Figure 6). The tracing of transitions enables the recording of information about their execution. The tracing of places enables the recording of information about tokens that go through them. The user can easily control what to measure and it is obvious what information will be gained or lost after switching on or off each setting. Moreover, our approach also allows for simply enriching the model by more detailed tracing. Places and transitions can trace additional data. It is implemented as connecting functions to places and transitions.

The usage of this feature is demonstrated in the experiment in Section V. The experiments also demonstrate tracelog sizes so even if we trace all transitions and names of all tokens in places (that is useful for debugging), sizes of tracelogs are usually manageable. The recording of high-level information from the perspective of our visual model is far from recording every function call in the program.

The second task is the *instrumentation*, i.e. putting the measuring code inside the application. In our case, Kaira can automatically place the measuring codes during the process of generation of the parallel application. Parallel and communication parts are generated from the model, therefore we know where are interesting places where to put measuring codes. By this approach, we can obtain a traced version of an application that does not depend on the complier or computer architecture. In contrast to a standard profiler or debugger for generic applications, we do not have to deal with a machine code or manual instrumentation.

As we already said, the results are presented to the user in the form of a visual replay or as statistical summaries. In replay, the data stored in tracelog are shown in the same way as in the simulator, thus as the original model with tokens in places, running transitions and packets on the way (Figure 5). The user can jump to any state in the recorded application. Our tool also provides statistical summaries and standard charts like a normal profiler, and additionally, information is presented using the terms of the model. For example, the utilization of transitions (Figure 10), the numbers of tokens in places, etc.

C. Combination of simulation and recording

The useful feature for debugging parallel applications is a technique usually called *deterministic replay* [4]. Existing

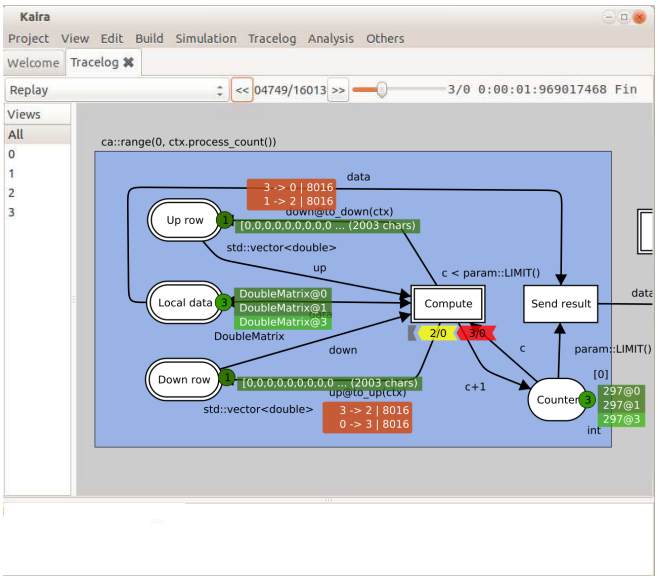

Fig. 5. The screenshot of a replay.

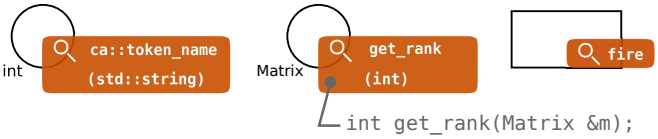

Fig. 6. Tracing labels, from left: Tracing names of tokens that arrive into the place; tracing values obtained by applying a function to each token arriving to this place; tracing transition firing.

tools use the *data-replay*, the *order-replay* or some combination of both approaches. In the data-replay approach, every communication message is recorded and a single process can be rerun with the same communication as was recorded. The advantage is that it is feasible even for instances with many processes. The disadvantage is huge tracelogs and it can be hard to discover errors that need an overall context. In the order-replay approach, we store the ordering of incoming messages. We get smaller tracelogs, but we must simulate all processes during the replay.

In Kaira we have implemented the order-replay approach in the form of *control sequences*. This feature naturally connects the infrastructure of our simulator with tracing abilities. A control sequence is a list containing actions. Each action is one of three basic types from Section III (starting and finishing transitions and receiving packets). Actions contain information about the process and the thread where the activity is executed, the transition's name (in the case of transition firing) and the source process of the message (in the case of receiving packets). When we store this information we are able to repeat the run of the application.

Sequences are generated in the simulator or they are extracted from tracelogs. The simulator can replay sequences and get the application into the desired state. Because the control sequence and the model are loosely connected, the

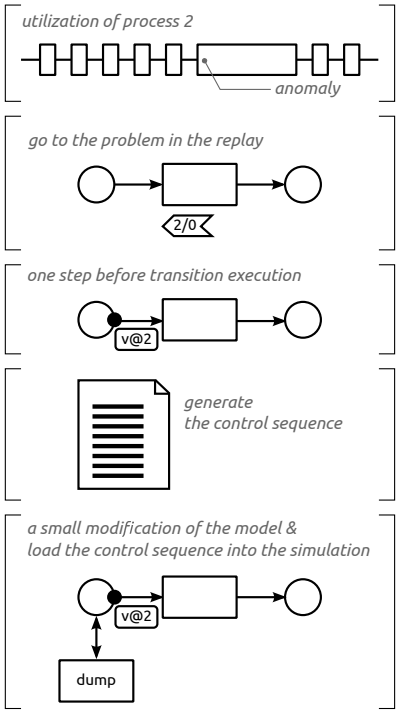

Fig. 7. The use case of control sequences

sequence remains relevant even if we make some changes into the model. The usefulness can be exposed by the following scenario: The user finds a problem by a visual replay or by summaries obtained from a tracelog. Then a sequence that brings the application exactly one step before the problem can be exported from the tracelog. Then the model can be enriched by more precise debugging outputs. For example, it can be a `printf` added into a transition's code or an extra debugging transition. Now we can get the application into the state before the problem by replaying the sequence in the simulator. In this situation, we have the possibility to obtain more information about the problem because of the modified version of the application. This scenario is captured in Figure 7.

D. Other features

Our model allows implementing two additional features that can be used for debugging or the performance analysis. Because we control how parallel aspects are generated, we can always generate the sequential application from the model. Such generated application works like the original one but it is performed exactly by one thread independently on how many processes are specified. This feature does not need any change in the model. It enables easy profiling and debugging of sequential parts of developed programs by the tools designed for sequential applications without problems caused by threads or MPI.

The other feature is the possibility to connect into a running application. We can start a generated application in a mode where the application listens on a TCP port. The application normally runs but when we connect to this port, the run is

paused and the inner state of the application is displayed in the simulator. The application can be also controlled in the same way. When the connection is closed, the application continues computing. This way we can easily debug situations when the application hangs up or we can just observe how far the computation is. But in the current implementation there are some limitations. This feature works only for applications generated with the thread backend (i.e. it does not work for MPI applications) and after the connection to the application, the control is passed to the user after finishing all current running transitions.

E. Drawbacks

Here, we want to discuss the drawbacks of our approach. The most obvious issue is that our approach does not give us any tool to debug or profile codes in places and transitions. We can say what data were on the input of the transition, what was the output. We can get the application into a state before or after execution of the transition or profile the transition as a whole. But we cannot observe, debug or profile the inner state of transition executions. This can be a serious problem and may force the user to use other tools in some situations. On the other hand the codes in transitions are sequential codes without any communication so they can be easily profiled or debugged separately. It can be further simplified by the fact that we can always generate the sequential version of the program.

Other issue is connected with our current implementation. We have focused on minimizing the performance impact of the debugging and profiling infrastructure on generated applications. On the other hand, our tool itself was not subject of optimizations, and therefore, processing a huge tracelog or a long control sequence can be time consuming and demanding on memory. Therefore, our infrastructure is not yet suitable for debugging or profiling long running applications. Some numbers to this topic are provided in Section V.

IV. Related Works

In this section, we want to compare Kaira with selected tools for profiling and debugging. For the comparison with other types of tools we refer to [2].

Different approaches have been proposed for debugging MPI applications. More about debugging in MPI environment can be found in [5], [6].

First, a MPI application runs on each computing node like a normal program; therefore, we can use standard tools like *GDB*[5] (for debugging) or *Callgrind*[6] (for profiling). This approach is sufficient to find some types of bugs or performance issues, but the major disadvantage is completely separated instances of the supportive tool for each process. It is not easy to control more debugger instances simultaneously or merge several profiler's outputs.

There are specialized debuggers and profilers to overcome this issue. For debugging there are tools: *Distributed Debugging Tool*[7] or *TotalView* [8]. They provide the same functionality like ordinary debuggers (stack traces, breakpoints, memory watches), but they allow to debug a distributed application as a single piece. Besides these tools, there are also non-interactive tools like *MPI Parallel Environment*[9]. It provides additional features over MPI, like displaying traces of MPI calls or real-time animations of communication. These tools are universal in the sense that they can debug any application. In our approach, we can only debug applications created in Kaira. On the other hand, we are able to provide the debugging infrastructure on a higher level of abstraction than source codes.

To deal with hundreds of processors, there are also automatic debugging tools. These tools usually use the static analysis of source codes to discover misuse of MPI calls (*MPI-Check*) or the analysis based on the state space exploration (*ISP* [7]).

As it was mentioned in previous sections, a potentially powerful technique for debugging of MPI applications is the deterministic replay. An example of a tool implementing this approach is MPIWiz [4].

In the case of profilers for parallel applications, one of the most successful freely available tools is *Scalasca* [8]. The big advantage is the ability to work in an environment of thousands of processors. Scalasca implements the direct instrumentation approach, it provides data summarizations at a runtime or traces for postmortem analyses. In the tracing mode, Scalasca records performance related events. Summarized performance profiles are based on functions call paths. In both cases, resulting reports can be interactively explored in the graphical browser.

The similar tool to Scalasca is *TAU* (Tuning and Analysis Utilities) [9]. It is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements. From this perspective, both Scalasca and TAU adopt similar strategies. The main difference is in the low level measuring systems.

There are also different tools that focus mainly on the visualization of traced data like Vampir [10] and Paraver [11]. These tools are able to import tracelogs produced by others and the user is able to browse traced data. Usually, a set of filters can be specified to remove unnecessary details.

V. Experiments

This section contains two example programs. Their purpose is to demonstrate features mentioned in Section III. All programs were executed on a machine with 8 processors AMD Opteron/2500 (32 cores in total) and compiled with Intel Compiler at the optimization level `-O2`.

A. Basic measurements

As the first example, we show results for the heat flow problem introduced at the end of Section II. In this example, we

[5]http://www.gnu.org/software/gdb/
[6]http://valgrind.org/docs/manual/cl-manual.html
[7]http://www.allinea.com/products/ddt/
[8]http://www.roguewave.com/
[9]http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/

show a comparison between the hand-made solution profiled by Scalasca and the version created and profiled in Kaira. Both implementations are distributed together with Kaira.

The implementations share the same computation code. It is about 380 LOC (lines of code without comments). The solution in Kaira contains 25 LOC in transitions and places and 10 LOC for binding the external types. The hand-made solution contains 100 LOC, which are not shared with the solution in Kaira. The following experiments were executed on the instance of the size 6400×32000 and 300 iterations.

All places and transitions are traced except for places *Up row* and *Down row*. The standard function writing token name for the type `std::vector<double>` stores all values from the vector into the tracelog. (6400 doubles for places *Up row* and *Down row*). In our example, we do not need such information, so we can change this writing function to store a smaller amount of data or we can just switch off the tracing (as we have done here). In the case of the hand-made solution profiled by Scalasca, 5 patterns in the filter file was used (23 functions were filtered out). These numbers are small, because of simplicity of the example. For illustration, the code generated by Kaira contains more internal functions and when it is profiled in Scalasca, we had to use 14 patterns in the filter file and 382 functions were filtered out. Without the filter file, Scalasca produces extremely huge logs (in the order of gigabytes) and it deforms runs of the application, because traced data are very often flushed on the disk.

Table I shows the comparison between the solutions generated by Kaira and the handmade solution. In both cases, the measurements were done without writing the resulting matrix into the file. Our problem scales well up to 16 processors, then it reaches limits of used computer.

In case of Scalasca we have instrumented all source files. This instrumentation adds some overhead. It can be improved by additional separation of computation code and communication, but it cannot be always possible. For example when we use an external library.

Figure 8 shows that solution produced by Kaira is comparable to handmade solution and our tracing introduces only a small overhead. For this small number of processors, the measured times are better than the handmade solution profiled by Scalasca. Scalasca is designed for thousands of processors; therefore it is not well suited for this experiment. But our goal was to show that Kaira tracing is comparable (in the scale of tens of processes) with existing mature parallel profilers and Scalasca is a well-established tool in this area.

Figure 9 shows the grow of tracelog sizes. Kaira tracelogs are bigger but still comparable. In Kaira case, they contain information for a replay, not only profile data.

B. Advanced measurement

In this section we will demonstrate how a tracelog can be enriched by custom data and how tool R[10] can be combined with Kaira to obtain various statistics. R is one of the most

[10]http://www.r-project.org

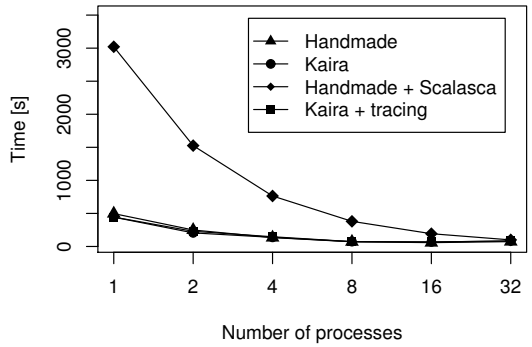

Fig. 8. The comparison of running times between the hand-made solution and the solution generated by Kaira for the heat flow example (based on Table I).

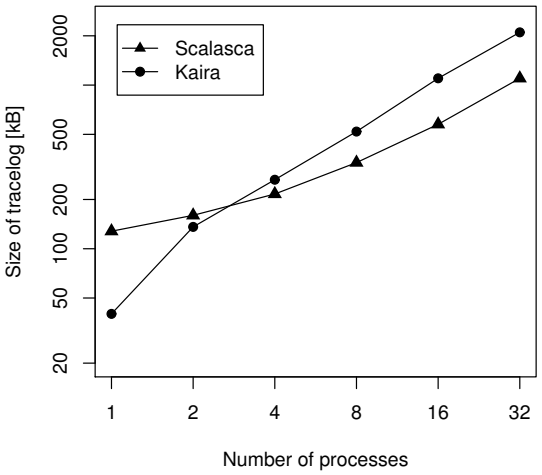

Fig. 9. The comparison of tracelog sizes between the handmade solution traced by Scalasca and Kaira's solution traced by Kaira (based on Table I).

popular statistical tools. Kaira can export collected data from a tracelog in a form of a table that can be loaded into R. Each row of this table corresponds to the three basic events (explained in Section III) and their subevents (token add, token removed, packet sent). In Kaira's distribution, there is a simple script for R that provides basic operations over such table. It is often easy to extract useful information about the performance from data in this form.

As the example, we have chosen the *Ant Colony Optimization* (ACO) algorithm that is used to solve *Traveling Salesman Problem* (TSP). There are many ways to parallelize this algorithm; the presented solution is described in more detail in the paper [12]. The visual model for the solution is depicted in Figure 11. We will show how to get specific data from the application's run and present them with the help of R.

TABLE I
MEASURED VALUES FOR THE HAND-MADE SOLUTION AND KAIRA'S SOLUTION OF THE HEAT FLOW EXAMPLE

Number of processes	**1**	**2**	**4**	**8**	**16**	**32**
Handmade solution [s]	497.39	249.58	134.39	70.98	57.88	73.38
Handmade solution + Scalasca [s]	3020.89	1525.62	763.63	380.23	193.08	99.33
Kaira solution [s]	443.5	205.57	137.75	72.95	68.04	83.09
Kaira solution with tracing [s]	444.78	229.67	147	72.98	68.14	83.06
Scalasca log size [kB]	128	160	216	336	576	1126
Kaira log size [kB]	40	136	264	520	1126	2150

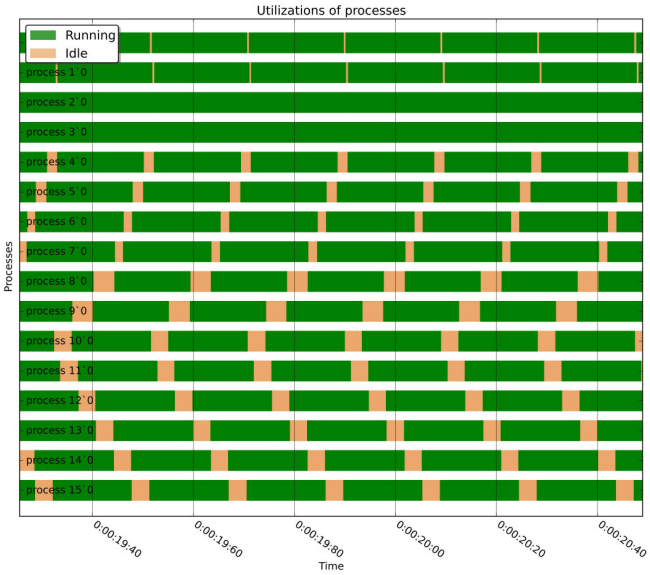

Fig. 10. The example of zoomed chart of process utilizations in the heat flow example with 16 processes.

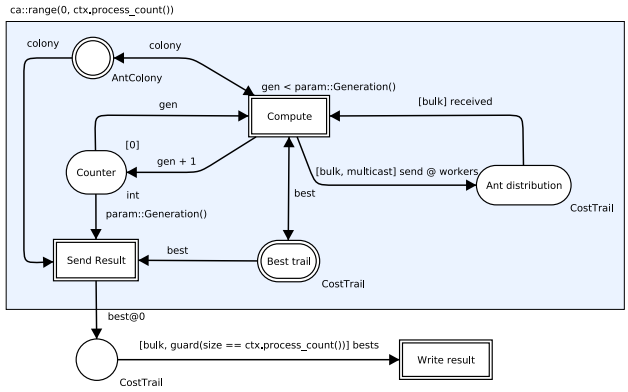

Fig. 11. The implementation of Ant Colony Optimization.

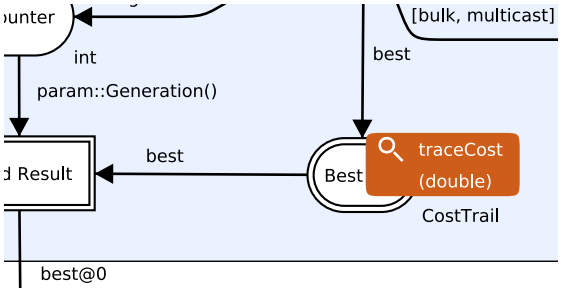

Fig. 12. Connecting a tracing function to the place *Best trail*.

For our experiment, we used the file *eil51.tsp* from TSPLIB[11].

In the used version of the ACO algorithm, ants are separated into colonies and the evolution of each colony is computed in parallel (each colony is assigned to a single MPI process). A colony is stored in the place in the top-left corner. The transition *Compute* takes a colony and computes the next generation of ants. In each iteration, every process saves the best solution to the place *Best trail*. It is distributed to other processes through the place *Ant distribution*. When the last generation is computed, *Send results* takes the best solution and it sends them to process 0, where the overall best solution is chosen.

To verify that the solution works properly, it is useful to inspect the fitness value (i.e. the quality of the solution) in time. We use the ability to connect a tracing function with a place. In our case, we connect a simple function returning a fitness value of an ant to place *Best trail* (Figure 12). When a token arrives to this place, its value is stored in the tracelog (in the scope of an event that creates this token). After exporting the tracelog table into *R*, we obtain the charts in Figures 13 and 14.

[11]It is available at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

When the best solution for each colony is sent to others, the convergence is the same for all processes (Figure 13), as we may expect. To check this assumption, we can disable communication, by removing the edge with the expression `[bulk, multicast] send@workers`. The fitness values for this case are shown in Figure 14.

VI. CONCLUSION

In previous papers, we have been focused on the development of MPI applications by usage of the visual model and visual programming. Our visual language is based on well-known formalism – Coloured Petri Nets. In this paper, we have presented how the same visual model and in fact the same approach was used for debugging and performance analyses. The presented ideas are implemented in our tool Kaira.

We introduced a simulator that allows the live introspection into developed programs. This simulator uses the original visual model. Thus the developer is able to inspect the de-

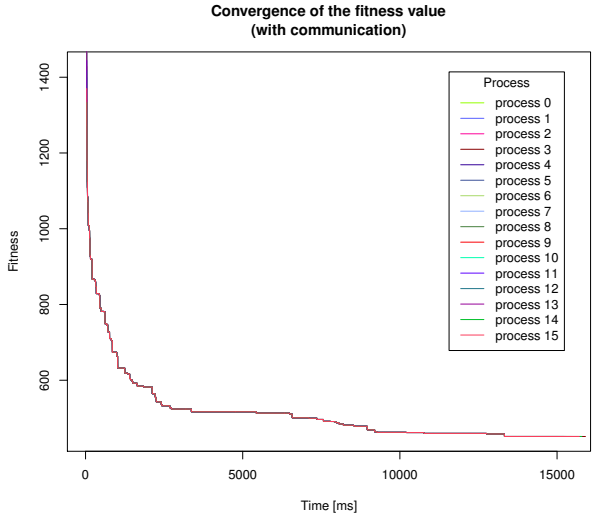

Fig. 13. Minimization of fitness values in time; colonies exchange the best solution.

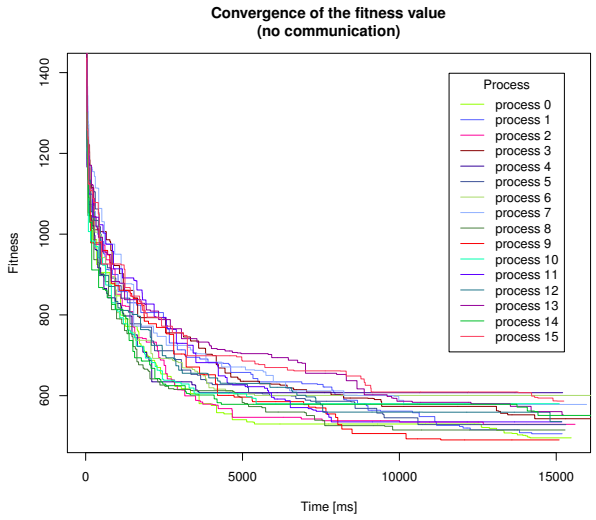

Fig. 14. Minimization of fitness values in time; without communication.

veloped application's behavior using the same visual model that he developed and that he understands. Using control sequences, we are able to capture a simulation and later it can be reproduced even on a modified visual model. They serve as basic infrastructure and they allowed us to implement deterministic replay and we want to implement more advanced features like a massive parallel replay.

Also for profiling we use a similar approach and we use the original model. We use it not only to present the obtained data (application's replay) but also to simplify the measurement specifications. This is crucial for profiling, because when we measure everything, the obtained data are usually useless and setup measurement filters in a standard tool can be hard.

We also demonstrate that presented features can be implemented with a performance that is comparable with existing mature tools. Practical experiments show that a performance of the handmade solution is comparable with the solution generated by Kaira. Measured times differences were up to 20%. The overhead introduced by tracings in Kaira is up to 3%. Our tracelogs are bigger than Scalasca's tracelogs, but their growths is similar.

We consider these features to be a successful step towards providing the unifying framework for prototyping and development of MPI applications. We are also working on more advanced features: performance prediction and verification. These parts are interconnected by our model and results from one analysis can be used in the rest of Kaira infrastructure. It can serve as another argument why to use Kaira.

REFERENCES

[1] S. Böhm and M. Běhálek, "Generating parallel applications from models based on petri nets," *Advances in Electrical and Electronic Engineering*, vol. 10, no. 1, 2012.

[2] ——, "Usage of Petri nets for high performance computing," in *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, ser. FHPC '12. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2364474.2364481

[3] K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[4] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker, "Mpiwiz: subgroup reproducible replay of mpi applications," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 251–260. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504213

[5] J. M. Squyres, "Mpi debugging – can you hear me now?" *ClusterWorld Magazine, MPI Mechanic Column*, vol. 2, no. 12, pp. 32–35, December 2004. [Online]. Available: http://cw.squyres.com/

[6] ——, "Debugging in parallel (in parallel)," *ClusterWorld Magazine, MPI Mechanic Column*, vol. 3, no. 1, pp. 34–37, January 2005. [Online]. Available: http://cw.squyres.com/

[7] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal verification of practical mpi programs," *SIGPLAN Not.*, vol. 44, no. 4, pp. 261–270, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504214

[8] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, Apr. 2010.

[9] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064482

[10] A. KnÃźpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. MÃźller, and W. Nagel, "The Vampir performance analysis tool-set," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer Berlin Heidelberg, 2008, pp. 139–155. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68564-7_9

[11] V. Pillet, V. Pillet, J. Labarta, T. Cortes, T. Cortes, S. Girona, S. Girona, and D. D. D. Computadors, "Paraver: A tool to visualize and analyze parallel code," In WoTUG-18, Tech. Rep., 1995.

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