GENERATING PARALLEL APPLICATIONS FROM MODELS BASED ON PETRI NETS

Stanislav BOHM[1], *Marek BEHALEK*[1]

[1]Department of Computer Science, Faculty of Electrical Engineering and Computer Science, VSB-Technical University of Ostrava, 17. listopadu 15, 708 33 Ostrava, Czech Republic

stanislav.bohm@vsb.cz, marek.behalek@vsb.cz

Abstract. *Parallel and distributed systems play an important role in the development of information technologies and their application. These systems are very useful but their development and usage is inherently more difficult. A solution can be a tool focused on systematic, well-arranged design, analysis and verifications these systems. This article briefly describes the tool Kaira intended for modelling, simulation and generation of parallel applications. A developer is able to model parallel programs and different aspects of communication using Kaira. Models are based on the variant of Coloured Petri nets. The important feature of our tool is automatic generation of standalone parallel applications from models. The final application can be generated with different parallel back-ends, currently it can be threads or MPI.*

Keywords

Coloured Petri nets, fast prototyping, modelling, parallel-distributed applications.

1. Introduction

In the world of scientific and technical computations, parallel computers are natural and common tools. They give us possibility to decrease computational time or run applications too large for a single machine. But these profits are paid by a more complex development of applications for such systems compared to their sequential counterparts. Difficulties with the development of parallel applications arise in each part of the development process from designing parallel algorithms, through their implementation to testing and debugging. It can be a problem especially for researchers with little or no experience in the field of a parallel programming. In these areas the real output is not often an implemented application but a product of its executions. In extreme cases we might run these applications only once. As a result, a sequential solution is often used even if we have access to supercomputers or computer clusters.

For such scientific computations we want to get a parallel implementation quickly and easily. As a solution we start to work on a programming tool Kaira. It allows creating a visual model of communication and parallelism of a program and a programmer can insert sequential codes into such model (thus reuse existing codes). Behaviour of a model can be observed in simulations and at the end a programmer can generate a final stand-alone application by "one click".

Visual models in Kaira are based on our variant of Coloured Petri nets (CPN) [1]. CPNs provide a theoretical background and we use their syntax and semantics.

In the current version, sequential codes inserted into a model are written in C/C++ and resulting applications can be generated for different parallel back-ends. Right now, *threads* or *Message Passing Interface* (MPI) [2] are supported. Therefore generated applications can run on parallel computers with a distributed memory architecture. Visual models are also used for debugging. The debug information can be shown to a user in a more high-level way compared to classic debugging solutions so a user gains a global overview of a state of computation more easily.

The tool is an open source project and can be obtained at http://verif.cs.vsb.cz/kaira.

2. State of the Art

There are many approaches to parallel programming and also many tools targeted to this area. A nice overview of tools for parallel and distributed applications development presents paper [3].

Today's CPUs usually contain more cores and we can for example use a system API for threads to create parallel applications, but this approach requires solving different low level issues. On a higher level of abstraction there are tools like *OpenMP*. If a main computation can

be expressed as more or less independent cycles then *OpenMP* can help a lot. But problem arises if a computation requires nontrivial communication and synchronization. Moreover these solutions are often closely tied to usage of computers with shared memory.

But when we move to the area of supercomputers then the memory is usually distributed and mentioned approaches cannot be straightforwardly applied. The industrial standard for computation without shared memory is MPI. But this standard represents rather a low level approach. Generally speaking, it is API for sending messages between nodes. There is no common approach how to parallelize sequential solutions in abstract way for environments without shared memory (compared to approach represented by *OpenMP*).

In our research, we are focused on the development of applications in high level way for computers without shared memory with tens or hundreds nodes. Especially we focus on applications with nontrivial communication.

Besides these well known and commonly used technologies like *OpenMP* and MPI there are also other approaches that are still subject of research. We want to mention particular technologies and approaches that we found interesting and that were an inspiration for us.

First we want to mention the functional style of programming. It can be very attractive for parallel applications [4]. However functional programming languages are not widely used and some algorithms are harder to express using them.

As we mentioned above our models are based on a variant of CPN. CPN are used in Kaira as the graphical language for modelling. So in some aspects the algorithm is also modelled in the declarative way.

Arcs in Kaira are annotated by a textual inscription language. This inscription language also adopts the functional style of programming. In the area of high level Petri nets, it is a common approach to combine Petri nets with a general programming language.

There are also approaches that extend widely used programming languages like Java to ensure ability of automatic parallel execution. For example Out-of-Order Java [5] and Deterministic Java [6] extends Java by constructs with well defined parallel behaviour. A parallel execution then came naturally and it is produced by a compiler. This kind of parallelization is mostly tied with computers with shared memory. Also a programmer has to rely on a compiler. For example, even if a bottleneck is found, it can be hard to change the resulting program and solve the problem.

In our tool, the parallelization has to be expressed explicitly in Petri nets but a programmer can chose different levels of abstraction. Models can be very specific but also they can be expressed in an abstract way and many things can be left on the tool. Translating models into executable forms is in all cases performed automatically. This explicitness gives a programmer more control (especially over communication) in a program but he can still choose to work with a high level model and specify some parts in more detail later if necessary. These options are crucial for obtaining prototypes in short time and also for possibility to create models with good performance.

CPN Tools [1], [7] and Renew [8], [9] are ones of the best known tools in the world of high level Petri nets. CPN Tools is one of the most famous tools for designing CPN. CPN ML (the variant of Standard ML) is used as the net inscription language. The strong points of this tool are simulations, the state space analysis and the performance analysis. CPN Tools is (for example) very good at modelling protocols. The main difference from Kaira is the absence of a code generation. So a user can model applications but when he wants to get executable codes he needs to write them by hand from scratch.

Renew is the tool for designing reference nets. These nets combine the object oriented programming with Petri nets. In Renew, Java is used as the inscription language for annotations of arcs and other elements. Any Java library can be used in models. Renew is also good at modelling parallelism. A user can have more parallel running instances of a net. Communication is ensured by synchronization of firing transitions, instead of flows of tokens like in Kaira. The simulation in Renew can run simultaneously on more processors but it is restricted to machines with shared memory. Any program generated by Kaira can also run on computers without shared memory due to MPI. Moreover Renew cannot generate stand-alone applications. The typical usage of Renew is designing multi agent systems. It is not designed for a high performance computing.

3. Introduction to Kaira

In this section we want to shortly describe Kaira basic features and properties. We show its usage on simple examples. More details can be found in papers [10] and [11].

The tool itself provides a standalone development environment so a whole application can be written in it. The most basic function of the tool is a creation of graphical models. The tool also assists a developer with inserting custom codes into models. A user is able to simulate these models and generate stand-alone parallel applications.

Kaira also provides other possibilities such as recording of generated application's runs. Created logs can be visualized in original models. We can use these logs for debugging and profiling. A replay of a log file is shown in Fig. 1. The net used in screenshot is the same as the net used later in the text (Fig. 2).

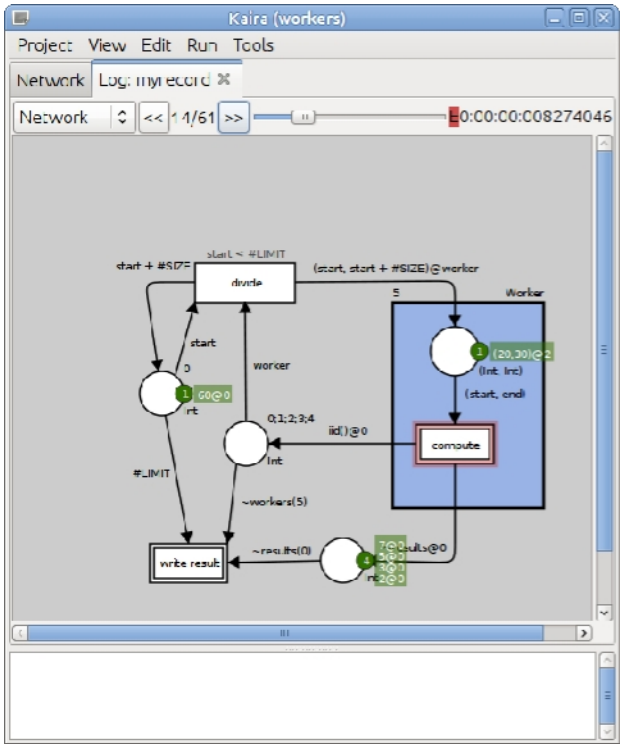

Fig. 1: The screenshot of Kaira during replaying of a log file.

The tool also offers an automatic agglomeration of parallel task to available computing nodes. So a user can create a high-level model with a large number of parallel activities and these tasks are then automatically assigned to processes.

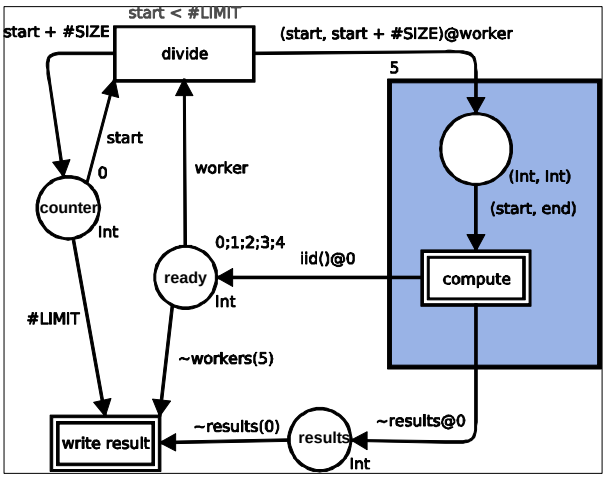

Fig. 2: The example of a model.

Our own variant of CPN is used as a graphical language for Kaira models. In CPN, tokens are not only *black dots* like in ordinary (Place/Transition) Petri nets but they have values like number *6* or string *Hello*. In Fig. 2 it can be seen an example of our CPN. In the example, each place has assigned a data type and only tokens with values of a corresponding type can be stored in a place. An assigned data type is displayed at the right bottom side of a place. A place's initial content is written at the upper right side.

Let us consider the following problem. We want to perform some computations for an interval of numbers. We can divide our task to separate subtasks but a computation time of each subinterval is notably different and we cannot guess it in advance. It is ineffective to simply divide all subtasks to working nodes at the beginning. As a result we introduce a master node that divides parts of the work to other nodes. When a working node finishes a computation of an assigned subtask, it sends results to the master node and waits for a new job. For the sake of simplicity we fix the number of working nodes to five. The net solving this problem is shown in Fig. 2.

The place *ready* in our example represents idling workers. At the beginning it contains an id (a number) of each worker. The place *counter* stores an integer representing the start of the next assigned interval. When the transition *divide* is fired then it takes an id of an idling worker from the place *ready* and it assigns a new subinterval and increases the number in the place *counter*.

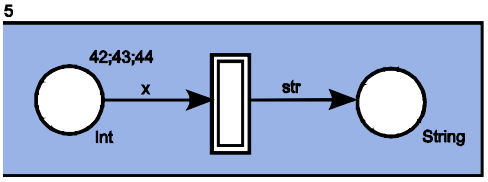

Fig. 3: The example of an area.

The significant feature of our modification of CPN is blue areas. The example can be seen in Fig. 3. There is a fragment of a net enclosed by blue area with inscription "5". It can be imagined in the way that we have separated copies of this fragment as it is shown in Fig. 4. These five replication's runs independently on each other with separated content of places. Blue areas allow us easily express a computation that is performed on distributed data.

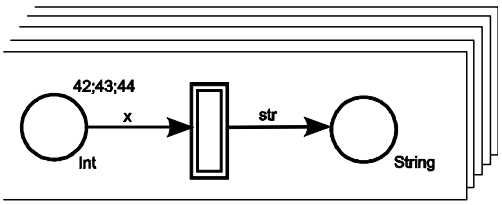

Fig. 4: The visualization of replications of the area in the Fig. 3.

Our example shows (Fig. 2) five replications of the right part of the net containing the transition *compute* and one place. Each replication represents a working node in our algorithm. The expression after "@" in the inscription of the edge from the transition *divide* specifies an identification number of replications where we want to send an interval. It allows us to establish communication between replications defined by blue areas. The double border of the transition *compute* means that there is a C++ function inside the transition. This function performs a computation on an interval of numbers. When this transition finishes its computation then it returns a token

into the place *ready* and it also sends results to the master node.

When we reach the limit in the place *counter* and all workers finish their computations then all results are written at once using the transition *write results*.

The integration with C++ code into the model is done by two ways: The code can be inserted into transitions and places or C++ types and functions can be integrated into the inscription language. The latter allows for example integrate some C++ library for matrix operations. Tokens can represent matrices and one dges we can have expressions for operations with them. We show here the former method: inserting codes into a transition. The similar idea with auto-generated templates works also for other cases of integration.

Let us assume that we want to insert a code into the transition *divide*. The tool opens a new editor tab with the following code:

```
struct Vars {
    int start;
    int worker;
};

void function(CaContext &ctx, Vars
&var)
{

}
```

Fig. 5: Source code.

This code is generated from the model and it will be updated when the model is changed. A user cannot modify this code but he can write what he wants as a function's body. The second parameter gives us an access to variables used on edges around the transition. The first parameter is an interface for calling some internal functions. Every time when a transition is fired then a function assigned to the transition is called.

The same idea works also for places but the codes inside places serve for initialization purposes. For example function can load some data from a file.

4. Ant Colony Optimizations

The nature of our tools allows easily experimentation with created algorithms. A programmer can test different concepts by just editing few arcs. It can be especially useful where algorithm itself is a subject of experiments. This situation often occurs in case of metaheuristics. We have described how can be our tool used for one of well known metaheuristic algorithm: Ant Colony Optimization (ACO) [12]. It was a main theme of a paper [13].

ACO is a wide spread nature inspired metaheuristic algorithm. It is a successful tool with practical applications in the areas of optimization, scheduling, and path finding to name just a few. The basic idea of the algorithm is inspired by real life ant colonies. Real ants usually find an optimal path between two places. ACO algorithm is inspired by our understanding of these real life observations. In ACO, there is a colony of ants. These ants try to explore a given space. While they explore this space they place pheromones along their way and also they are guided by pheromones of other ants. Ants try to fulfil a certain goal. For example, they try to find a best path to a target place. These pheromones slowly evaporate in time. Also in time new generations of ants are produced. During this process successful ants are boosted while unsuccessful ones are eliminated. These basis steps are repeated until we get a path (it is a solution in fact) that satisfy our needs.

The paper [13] presents usage of Kaira to parallelization of the ACO algorithm. At the beginning, we had working sequential ACO implementation. In the paper, we present different approaches how to use Kaira to transform this solution into a parallel version. This original sequential implementation was not created exclusively for our experiments but was implemented by other research group from our department - Department of Computer Science, FEI VSB-TUO. It was used to solve some real life problems and for research purpose in other research areas [14].

Although there are plenty of categories of parallel ACO algorithms, the majority of parallel implementations use the multi-colony (multi-population) ACO [15] which is roughly equivalent to the islands model of parallel genetic algorithms. In this approach, each process executes an independent ACO instance. The independent colonies might exchange solutions at a defined time. However, the usefulness of certain communication patterns and strategies is still a subject of further studies.

One of our solutions was based on the fully connected model where each colony communicates with all other colonies. This model is captured by net in the Fig. 6.

In fact this model represents the whole parallel solution. Besides the model and original source codes we need to write proximately 20 lines of C++ code that is hidden in transition *Compute*. This code executes one step in ACO algorithm. This means that a new generation of ants is produced. The so far most successful solution is found and this solution is sent to other colonies. They merge this solution with their data.

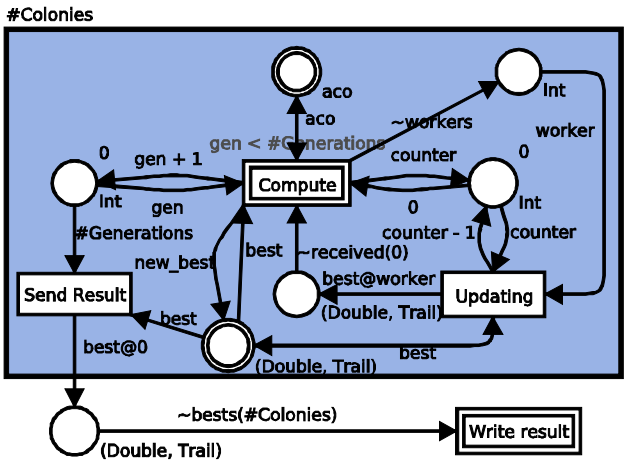

Fig. 6: Parallel ACO model where colonies are fully connected.

In the model, the number of blue areas corresponds to the number of colonies. These colonies are then automatically assigned to processes. Kaira automatically assigns more ant colonies to one process if needed.

For practical experiments, we use the travelling salesman problem (TSP). We use TSP because it was often used as a case study in the research of ACO [12], [15]. Experiments were performed on a HPC Windows Server 2008 R2 with 6 nodes. Each of these nodes has 8 cores. Nodes are connected with 20 $Gb \cdot s^{-1}$ InfiniBand. During our practical experiments we try to explore two things. First we try to test correctness of created solutions. A core functionality of parallel solutions expects several ant colonies working together. So they cannot be easily compared with the original implementation. So first we found settings (the number of ants, computed generations and number of ant colonies) that perform similar amount of work and gives similar results. While we use TSP the result is a path through a graph. We can easily compare the quality of founded solutions simply by comparing costs of founded paths. Then we test performance and scalability of created solutions.

Particular measured numbers are not so important, but for example if we used presented parallel solution with 16 ant colonies using 16 processors then the execution time was reduced approximately 14,7 times compared to the execution time using only one processor. So presented parallel solution scales well with this number of processors.

In different experiment, we fix the quality of obtained solutions. Then we need approximately 22,7 seconds using the original sequential implementation to find solution with this quality compared to 3,2 seconds when we use 16 processors and presented parallel implementation. This means speedup approximately seven times.

To summarize this section, metaheuristics often requires experiments to find a proper algorithm usage and settings to solve a given problem. A parallel computing can be very useful mainly because it can reduce time to find a solution with given properties. On the other hand parallel programming can be more time consuming and we need more development time to get an optimal parallel version. The tool like Kaira simplifies development and allows experiments with created solutions.

5. Future Work

The Kaira version presented here works well for different tasks. We successfully use it for a parallelization of metaheuristic algorithms mentioned above. On the other hand, there are still algorithms where our graphical model becomes complicated even if the core of a problem is relatively simple. One of issues is manipulation with objects that have internal structures.

We can take computing with matrices as an example. A matrix can be viewed as one object or as a set of rows/columns or as a set of individual values. If we want to distribute matrices to working nodes then it is useful to consider a matrix as a token. On the other hand if we want to model a computation on a matrix then it is often better to represent each value of matrix as a token. The natural solution is to introduce functions that can scatter a token to more tokens according to their internal structure and functions that can gather such tokens and reconstruct a container token. But this solution brings new problems. They are mostly connected to synchronizations and separation of computations. For example we have to prevent mixing tokens from different objects or computations together. Such issues can be solved even in the current version but it often ends with a nontrivial net with many extra arcs and transitions.

We are working on solution based on modules. On the first sight it can looks similar like Hierarchical Colour Petri nets [1], but their semantic is different. Instances of modules are created dynamically during execution of a net. Our semantic allows working with structured objects on different levels of their internal structure and nets remains well arranged as a result.

The other problem connected to the current version can occur for models with higher level of abstraction where parallel activities are not fixed but the shape and size of structure of net's replication depends on the input. To solve this problem we are preparing the new view on nets' replications. It can be viewed in the way that a whole computation is executed on an infinite tree (with infinite branching) where each node represents one replication. It is backward compatible because we can obtain the same behaviour as in the current version if we use only the top level of the tree. But if we use the whole tree then we can easily model even problems with a recursive nature.

6. Conclusion

We propose a tool for modelling, simulating and generating parallel applications. We have been motivated by real-life scientific problems and our work is focused on parallel applications with nontrivial data flows and communication. We are interested in scientific computations where to get an application (and its results) quickly is more important than a handmade solution with a slightly better performance. The strong reason why to use Kaira is the reduction of development time. The development itself is simplified and the programmers don't need to be experienced in the area of parallel or distributed systems. We believe that Kaira can be useful during implementation of parallel algorithms, even if we are still in the beginning and we are experimenting with basic principles of the whole idea. It will also need lot of programmers' work to get our tool into the form of a mature development environment.

To summarize, we propose the extensions of syntax and semantics of CPN that are useful for modelling of parallel algorithms. Also we offer the tool that helps to create and simulate such model. Our goal is not only to implement the tool that can demonstrate our model but we want to create a complete development environment for parallel applications. Therefore important aspects of our tools is ability to integrate existing C++ libraries and at the end to create stand-alone applications from the models. We also want to utilize our model in other parts of development process. For example in debugging applications, where the distribute states of running programs can be shown in the abstract way as state of Petri Net so a programmer can easily realize what happens in applications.

Acknowledgements

The work was supported by GACR P202/11/0340: Modelling and verification of parallel systems. This article has been elaborated in the framework of the IT4Innovations Centre of Excellence project, reg. no. CZ.1.05/1.1.00/02.0070 supported by Operational Programme 'Research and Development for Innovations' funded by Structural Funds of the European Union and state budget of the Czech Republic.

References

[1] JENSEN, Kurt and Lars M. KRISTENSEN. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Berlin/Heidelberg: Springer, 2009. ISBN 978-3-642-00283-0.

[2] SNIR, Marc, Steve OTTO, Steven HUSS-LEDERMAN, David WALKER and Jack DONGARRA. *MPI-The Complete Reference, Volume 1: The MPI Core*. 2nd (revised). Cambridge: MIT Press, 1998. ISBN 0-262-69215-5.

[3] DELISTAVROU, Constantinos T. and Konstantinos G MARGARITIS. Survey of software environments for parallel distributed processing: Parallel programming education on real life target systems using production oriented software tools. In: *14th Panhellenic Conference on Informatics*. 2010. p. 231–236. ISBN 978-0-7695-4172-3.

[4] LOIDL, Hans, W., Fernando RUBIO, et al. Comparing parallel functional languages: Programming and performance. *Journal Higher-Order and Symbolic Computation*. September 2003, vol. 16, iss. 3., p. 203–251. ISSN 1388-3690.

[5] JENISTA, James C., Hum E. YONG and Brian DEMSKY. OoOJava: software out-of-order execution. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. San Antonio, TX, USA, 2011. p. 57-68. ISBN 978-1-4503-0119-0.

[6] BOCCHINO, Robert L., Vikram S. ADVE, Sarita V. ADVE and Marc SNIR. Parallel Programming Must Be Deterministic by Default. In: *First USENIX Workshop on Hot Topics in Parallelism*. Berkeley, CA, USA, 2009. Available at: http://static.usenix.org/event/hotpar09/tech/full_papers/bocchino/bocchino.pdf.

[7] *CPN Tools* [online]. 2011. Available at: http://cpntools.org/.

[8] KUMMER, Olaf; Frank WIENBERG, Michael DUVIGNEAU, Joern SCHUMACHER, Michael KOEHLER, Daniel MOLDT, Heiko ROELKE and Ruediger VALK. An extensible editor and simulation engine for petri nets: Renew. In: *Applications and Theory of Petri Nets*. Volume 3099 of LNCS. Berlin: Springer, 2004, p. 484-493. ISBN 978-3-540-27793-4_29.

[9] *Renew* [online]. 2012. Available at http://renew.de.

[10] BOHM, Stanislav and Marek BEHALEK. Kaira: Modelling and generation tool based on Petri nets for parallel applications. In: *UkSim 13th International Conference on Computer Modelling and Simulation*. Cambridge, 2011, p. 403 –408. ISBN 978-1-61284-705-4.

[11] BOHM, Stanislav; Marek BEHALEK and Ondrej GARNCZARZ. Developing parallel applications using Kaira. *Digital Information Processing and Communications*. Volume 188 of Communications in Computer and Information Science. Springer, 2011, p. 237-251. ISSN 1865-0929.

[12] DORIGO, Marco and Thomas STUETZLE. The ant colony optimization metaheuristic: Algorithms, applications, and advances. In: *Handbook of Metaheuristics, International Series in Operations Research, Management Science*. New York: Springer, 2003. vol. 57, p. 250–285. ISBN 0-306-48056-5_9.

[13] BEHALEK, Marek, Stanislav BOHM, Pavel KROMER, Martin SURKOVSKY and Ondrej MECA. Parallelization of ant colony optimization algorithm using Kaira. In: *11th International Conference on Intelligent Systems Design and Applications (ISDA 2011)*. Cordoba, Spain, November 2011, p. 510–515. ISBN 978-1-4577-1675-1.

[14] SNASEL, Vaclav, Pavel KROMER, Jan PLATOS, Milos KUDELKA, Zdenek HORAK and Katarzyna WEGRZYN-WOLSKA. Two new methods for network analysis: Ant colony optimization and reduction by forgetting. In: *Advances in Intelligent Web Mastering - 3 - AWIC 2011, ser. Advances in Soft Computing*. Springer, 2011, vol. 86, p. 225–234. ISBN 978-3-642-18029-3_23.

[15] MANFRIN, Max, Mauro BIRATTARI, Thomas STUETZLE and Marco DORIGO. Parallel ant colony optimization for the traveling salesman problem. In: *Ant Colony Optimization and Swarm Intelligence, ser. LNCS*. Berlin/Heidelberg: Springer, 2006, vol. 4150, p. 224–234. ISBN 978-0-769-53357-5.

About Authors

Stanislav BOHM was born in 1985. He received M.Sc. from computer science in 2009 and he is currently working towards the Ph.D. degree at VSB Technical University of Ostrava. His main research topics are modelling parallel algorithms, generating programs from Petri nets and verification questions about counter machines.

Marek BEHALEK was born in 1979. He received Ph.D. degree from informatics and applied math in 2009 at VSB Technical University of Ostrava. He is currently working as the assistant professor at the Department of computer science at FEECS VSB Technical University of Ostrava. His research areas are programming languages, their evolution and applications. Right now he is focused on the parallel programming.