A Toolkit for the Development of Data-Driven Functional Parallel Programmes

Alexander I. Legalov(✉), Vladimir S. Vasilyev, Ivan V. Matkovskii, and Mariya S. Ushakova

Siberian Federal University, Krasnoyarsk, Russia
legalov@mail.ru, rrrFer@mail.ru, alpha900i@mail.ru, ksv@akadem.ru

Abstract. In the article a technology is considered which aims at creating architecture-independent parallel programmes based on the data-driven functional paradigm. A proposed toolkit provides the translation, execution, debugging, optimisation and verification of programmes. A programme in a data-driven functional parallel language is translated into the data-flow graph (which describes the data dependencies of an implemented algorithm) of the programme. On the basis of this representation, the control-flow graph (which defines the organisation of computations) is generated. Both graphs allow to carry out various optimising transformations. The resulting data-flow graph is also used for the formal verification of the programme. A computation process is considered as a cooperation of the control-flow graph and the data-flow graph. The execution of data-driven functional parallel programmes is carried out by a special interpreter (event machine), which consist of a number of event processors controlled by a special manager.

Keywords: Data-driven functional parallel programming · Software development toolkit · Parallel-programmes translation · Parallel-programmes optimisation · Parallel-programmes verification

1 Introduction

Parallel computing have outgrown the application in high-performance computing long ago. It is widely used for solving problems in different areas. Nowadays, the main feature of parallel programming is the source-code dependence on the architecture of the target computation system. So, to port a programme to another architecture, it should be completely rewritten or appreciably modified. The reason is the intention to increase the efficiency of parallel programmes, which results in software being heavily tied to particular hardware characteristics.

Computational resources and their communications are the main characteristics of a computation system which should be considered during the development in order to increase the programmes efficiency. So, in addition to solving

The research is supported by the RFBR (research project No. 17-07-00288).

an applied problem, we need to explicitly manage computations and resolve resource conflicts among parallel processes. That is why parallel programming is hard [1] and requires non-trivial analysis of programme correctness taking different approaches, for instance, model checking [2] for formal verification.

It should be pointed out that a dependency on a particular parallel hardware precludes writing truly parallel algorithms at the initial stage of the development. This leads to the reduction of problem parallelism according to hardware resources, which prevents from applying more effective solutions when a modification of the programme is needed. At the same time, the development of hardware-dependent programmes is the mainstream of parallel programming. The existing approaches have very different ideology of parallelisation. The most widespread approaches are: parallelisation with message passing [3], multithread and multi-core programming for systems with shared memory [4], graphical processing unit programming [5], and also the mixture of these three approaches in different combinations for systems with heterogeneous and distributed architecture [6–9].

Though the concept of unlimited parallelism is not widespread in parallel programmes development nowadays [10], it has some prospects as a basis of programming system that provide for subsequent transformations of programmes into resource-limited and architecture-dependent parallel programmes. So it is topical to develop a language and a toolkit to provide for creation of parallel programmes which are initially independent of peculiarities of a specific parallel computer system. Porting a programme to a particular system can be done after the processes of verification, testing and debugging.

The proposed approach is based on the concept of architecture-independent parallel programming. Its key ideas are exclusion of resource conflicts and implicit control over computations from within the programme being developed. It is supposed that a virtual machine which executes the programme has unlimited resources and a programming language allows to define solely data dependencies between the executed functions. An interaction between functions takes place on data readiness. This allows to create programmes with maximal achievable parallelism, which is compressed to special computing resources at the stage of the intermediate representation after verification and debugging of the source code. This allows to increase the efficiency of parallel-programme development process. For example, it is possible to create a generic library of functions adaptable to different existing and prospective architectures. The subsequent transformations of such programmes can be done with formal methods by changing the control-flow graph to fit the target architecture, preserving the correspondence with the data-flow graph (DFG).

The goal of our research is the development of architecture-independent parallel-programming technology based on the data-driven functional parallel paradigm [11]. To achieve the goal we solve the following problems:

– the development and further improvement of the data-driven functional parallel (DDFP) computing model, on whose basis a programming language is defined; it allows the creation of architecture-independent parallel programmes;

– the development of a toolkit to provide translation, testing, debugging and execution of data-driven functional parallel programmes;
– the development of methods for programme verification and optimisation at the level of the programme DFG;
– the development of control-flow graph transformation methods that allow to change the programme parallelism and, in the future, to transform programmes for particular parallel architectures.

2 Problems of Imperative Paradigm Employment in Parallel Programming

Wide application of the imperative programming paradigm introduces certain difficulties in the development of parallel programmes. A programmer has to explicitly or implicitly form relations between programme objects [12]. The possible relations are:

– data relations which specify the DFG of the programme; this graph defines dependencies between operations and operands being processed;
– control relations which set the order of execution of operations; these relations are associated with the DFG of the programme in order to ensure the right logic of transformations of operands;
– relations between computing resources (memory, processor units) that are used during the operation execution.

In most cases, a programme developer has to explicitly take into account the dependencies between these relations in an attempt to avoid any logical contradictions leading to an erroneous execution. In the ubiquitously used imperative programming, the relations between the data and the control are kept in the programmer's mind but are not explicitly expressed in the programme. For instance, let us consider the factorial function over the range 1 to n.

```
int fact1n(int n) {
    int r = 1;    int i = 1;
loop:
    if(i <= n) {
        r *= i;
        i++;
        goto loop;
    }
    return r;
}
```

It is evident that the only explicit relation is the relation of programme objects order in the source code or (after translation and loading for execution) in the system memory. But it does not specify the exact order of computing. A graphical representation of the given function explicitly represents all kinds of relations and

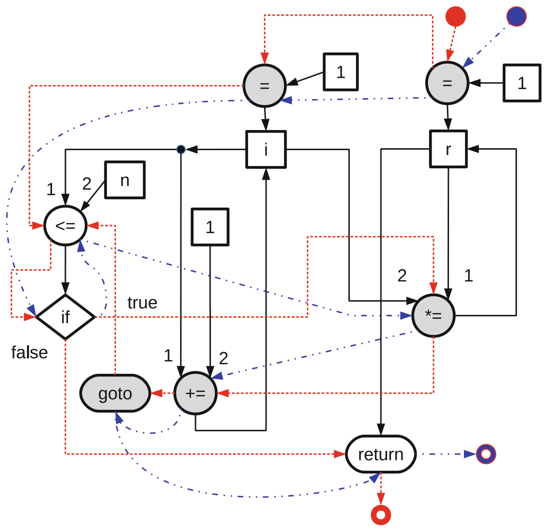

Fig. 1. A graphical representation of the relations in the factorial function (the solid, dashed and dash-dotted lines show data relations, control relations and order relations, respectively)

shows different trajectories of its execution which the programmer has to keep in mind to form an overall understanding of the programme (Fig. 1). In some cases, the control and order relations coincide (operations marked with light grey), which facilitates the programme understanding and allows to employ programme counters instead of straight-forwardly transferring the flow of execution to the address. In most cases, however, these relations are connected by implication rather than by the order of the operations in the programme.

Frequently, the relation of order can be ignored by employing a graphical representation. Particularly, flowcharts, activity diagrams, automaton graphs speed up the development of programme algorithm and allow to represent the logic of the operational behaviour clearly. This is done by explicitly defining the control and data relations on the basis of the developer's intuitive algorithm understanding.

The situation becomes more complicated if we turn to parallel programmes development. In this case, additional control operations for splitting and synchronisation appear in the control-flow graph. What is more, all available resources are to be distributed for the simultaneous execution of parallel source-code fragments. This results in a new relation between the programme and the resources. This relation can be explicitly represented by the resource graph (graph of system resources). The probability of conflicts arises, which could lead to incorrect computations even if the programme worked correctly in the sequential case.

Various parallel systems employ different computing control methods (strategies) [12]. A programme can be represented by a data-control-resource graph (we

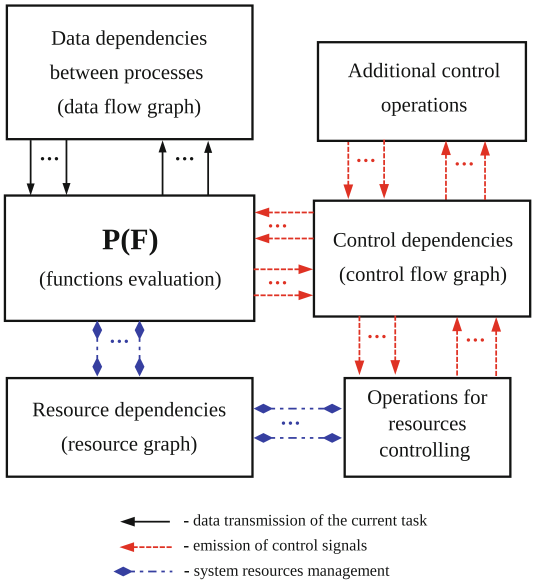
Data dependencies between processes (data flow graph)
Additional control operations
P(F) (functions evaluation)
Control dependencies (control flow graph)
Resource dependencies (resource graph)
Operations for resources controlling
- data transmission of the current task
- emission of control signals
- system resources management

Fig. 2. A DCR-net describing the process execution within the computational resources

call it a DCR-net) in which processes P execute the operations F defined by a programmer. The execution of these operations are initiated by control signals that are emitted under certain conditions in the control-flow graph. These conditions emerge from the data dependencies of the programme, peculiarities of computational resources and some additional factors (Fig. 2). The correctness of the computing process depends on certain prerequisites for each operation. In the general case, an operation execution within the resources of the computing system is possible only if the following conditions are satisfied before the execution:

1. The condition of data readiness (Data, D-condition). Before the process execution start, all the required data have to be at the process input. The process execution in the absence of any required data leads to a wrong result.
2. The condition of resources allocation (Resource, R-condition). The process requires certain resources to be executed within them, and these resources should be allocated and provided before the process execution.
3. The condition of acknowledgement (Acknowledge, A-condition). Resources utilised by a process can be freed or reused only after the acknowledgement that the output results of computations have been received by all the processes that take them as input.

The control of readiness conditions can be performed by different means. On the one hand, a programmer can control processes directly. On the other hand, a computing system undertake many control functions. Let us distinguish the following control modes:

1. Explicit (human) control. A programmer sets the logic of generating and checking the readiness conditions in the source code.
2. Implicit control. In this case, it is assumed that the processes are executed correctly without any control. This assumption may follow, for example, from special organisation of resources in the computing system, automatically maintaining the data readiness condition (automatic control). Another possibility is when the readiness conditions are always true due to the system peculiarities, and hence no control is needed (empty control).

If a programmer uses explicit control, then he should code the readiness conditions checks. It increases the software development costs.

3 Features of the Computing Model and the Language of Data-Driven Functional Parallel Programming

The basic approach to architecture-independent parallel programming is the development of a language and a toolkit to provide the implicit control at the level of computing model. We propose a model of data-driven functional parallel computing [11], in which every function is represented as a DFG whose nodes are operators and whose arcs are data connections between operators. Any connection is marked with a value which is both the output of the operator in the beginning of the arc and the input of the second operator. There are several types of operators in the graph: the operator of interpretation and data-grouping operators.

The **operator of interpretation** is the only operator that applies a function to the function arguments. This operator has two inputs: the first one takes a function (functional input), while the second one takes an argument for the function (data input) (Fig. 3). The output of the interpretation operator is the result of the function application to the argument. Functions are either elementary predefined operations or programmer-defined. The interpretation operator semantics is defined by the axioms of the computing model and its transformation algebra [11].

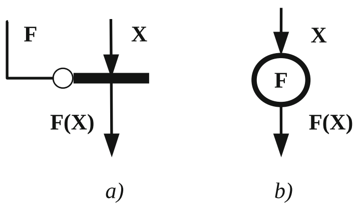

Fig. 3. Graphical symbols of the interpretation operator (*a*—the general case, *b*—the case when the function on the first input is predefined)

Data-grouping operators provide various ways of grouping operands in different structures (lists). The idea of data-grouping operators goes back to the functional forms introduced in [13]. In our case, however, it is the variety of such structures that is the principal way to increase the flexibility in writing parallel programmes and implement non-conventional ideas of parallel-algorithms development. Extending the set of such operators is one of the main approaches to the further development of the computing model and the language of data-driven functional parallel programming. This language is used to try out various approaches targeting the efficiency of expressing different types of parallelism. For instance, the usage of asynchronous lists [14] allows to develop algorithms with dynamically modifiable parallelism according to the rates of data incoming and processing. The core set of data-grouping operators is shown in Fig. 4.

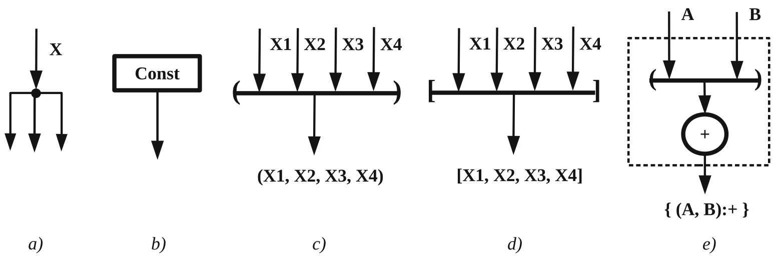

Fig. 4. Graphical symbols of data-grouping operators (*a*—data copying, *b*—constant assignment, *c*—grouping in a data list, *d*—grouping in a parallel list, *e*—delay-list creation)

The **copy operator** (Fig. 4*a*) carries out data replication. In our language, replication is done by assigning a name to a connection (marked later with a value during the execution), and then this name is used in other positions of the programme to refer to this connection (and the corresponding value). We use the prefix and postfix notation for assigning a name to the connection:

`value >> name`, or `name << value`.

The **constant operator** has no inputs (Fig. 4*b*). It has only one output that is always marked with the predefined value. In our language, the constant operator is defined by the value of a certain type.

The **data-list grouping operator** (Fig. 4*c*) has several inputs and one output. It performs structuring and ordering of the values that are transmitted through arcs from different sources. Each input has its number from 1 to N. The position of data in the resulting list equals the number of the input it has come from. In the source code, the list elements are put in parentheses "(" and ")". For example:

```
(x1,x2,x3,x4).
```

The **parallel-list grouping operator** (Fig. 4*d*) groups elements in a similar way as in a data list. However, its output is a multiple connection whose multiplicity equals the number of operator inputs. If an operator of interpretation is executed having a parallel list on its data input, then a function is applied to each individual element of the parallel list independently and in parallel. In the source code, the elements are put in square brackets "["and "]". For example:

$$\texttt{[x,y,z]:sin} \equiv \texttt{[x:sin,y:sin,z:sin]}\,.$$

Similarly, in cases when a parallel list of functions comes to the functional input of the interpretation operator, each function is applied to the argument in parallel:

$$\texttt{(x,0):[<,=,>]} \equiv \texttt{[(x,0):<, (x,0):=, (x,0):>]}\,.$$

The transformation algebra of the language describes all the cases of equivalent transformations of parallel lists.

The **delay-list grouping operator** (Fig. 4*e*) delays the execution of operators corresponding to some subgraph. This subgraph is considered as a single node of the DFG until the delay list is released. This node has several inputs and one output. The connections coming from outside the subgraph are the inputs of the operator, and the result produced in the subgraph comes to the operator output. The specific feature of this operator is that the delayed operators are not executed even on data readiness until the delay is not released. The release from the delay takes place if a delay list becomes an input of the interpretation operator. Delay lists allows to construct the conditional branches of the programme. In the graphical representation, a dashed line surrounding the delayed operations is used to denote the delay list. In our language, the list of delayed computations is defined by putting operators in braces "{" and "}".

On the basis of the described model, we develop the Pifagor language for data-driven functional parallel programming. The source code of the above-mentioned factorial function in the Pifagor language is the following:

```
fact1n << funcdef n {
    n1<< (n,1);
    [(n1:[<=,>]):?]^ (
        1,
        {(n, n1:-:fact1n):*}
    ):. >>return
}
```

The function is free from explicit computations control. Only the data dependencies between operators are defined. The DFG of this function is shown in Fig. 5.

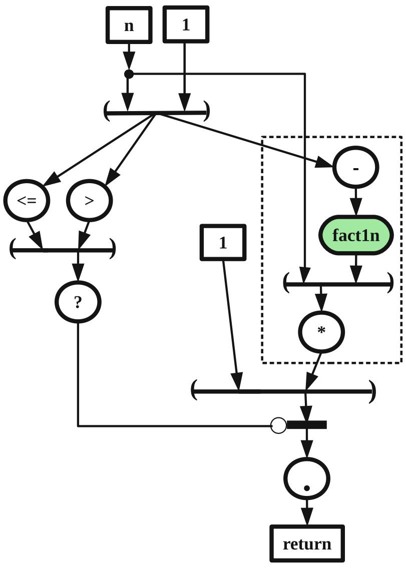

Fig. 5. Data-flow graph of the factorial function

4 A Toolkit for Architecture Independent Parallel Programming

We develop a toolkit to support the data-driven functional parallel programming paradigm in order to try out the proposed ideas and their further development on the basis of experiments. The general scheme of the toolkit is shown in Fig. 6. It includes the following subsystems:

– a translator from the language of data-driven functional parallel programming to the intermediate representation, called the reverse data-flow graph (RDFG);
– a generator of the control-flow graph (CFG), which constructs the graph for controlling computations;
– an event machine, which supports execution of data-driven functional parallel programmes by utilising RDFG and CFG;
– RDFG optimisation tools;
– CFG optimisation tools;
– tools for the DDFP programmes formal verification.

4.1 Translation of Data-Driven Functional Parallel Programmes

The translator accepts source code files in Pifagor language, each containing one or more functions. It also provides separate compilation of functions stored in a special repository. The translator generates a RDFG for each function. These

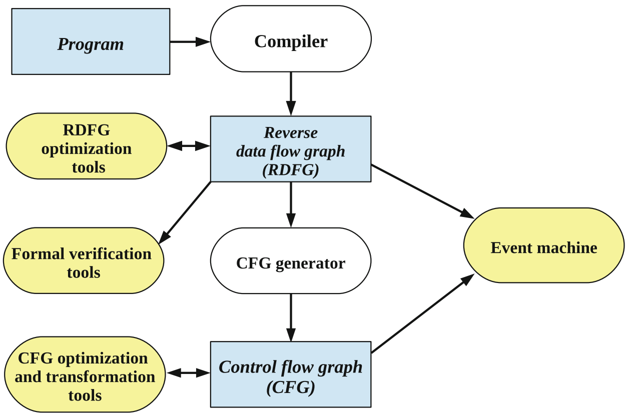
Program
Compiler
RDFG optimization tools
Reverse data flow graph (RDFG)
Formal verification tools
CFG generator
Event machine
CFG optimization and transformation tools
Control flow graph (CFG)

Fig. 6. The toolkit for architecture-independent parallel programming

RDFGs are saved in the repository in text format. The choice of the text format is due to the fact that an internal representation in the computer memory can be easily constructed by means of simple translators. Moreover, the developer can easily read and analyse the translated functions, considering the text form of a graph to be an analogue of the assembly language. The translator also generates auxiliary files with debug information binding the nodes of the RDFG to the function source-code lines.

A reverse data-flow graph generated by the translator allows to generate a control-flow graph that controls the function execution. Each node of the CFG is associated with the corresponding node of the RDFG and controls the moment when the operation starts executing. Each node of the CFG is a finite automaton whose states are controlled by the input signals. These signals notify the automaton of the event of the data having been prepared for the associated RDFG node. The computations on the RDFG node are initiated on certain state switches in the automaton. As the RDFG node execution completes, the readiness signal is transmitted through the output arc of the CFG to the next automaton. Before the execution, the CFG arcs are marked with initial signals. As the execution begins, the signals are transmitted along the arcs and change the states of the receiving nodes. A special utility programme forms the CFG. It is saved in the repository in text form.

4.2 Parallel Event Machine

At the current development stage, the execution of data-driven functional parallel programmes is done by a special interpreter (event machine), which consists of a set of event processors (EP) and a special event-machine scheduler controlling the EPs. Each EP handles only one function, which is run in a separate thread. Currently, the execution of operators inside the function is performed sequentially. At the present moment, our main goal is to achieve a stable functioning of the event machine rather than high performance.

Functioning of an EP (Fig. 7) is carried out in the following way. Initial signals of the CFG are added to the EP's signal queue from which they are transmitted to the handler of control signals according to the queue discipline. The handler analyses an incoming event. Depending on the state of the signal recipient node of the CFG, the handler might query the corresponding RDFG node (associated with the CFG node) if the operation of data processing is to be executed. In case it is, the handler of the RDFG nodes is called. It performs all needed functional transformations and saves intermediate results. After the data processing, the control node changes its state and, if needed, it emits a signal for the next node. The latter signal is again added to the queue of control signals, and so on.

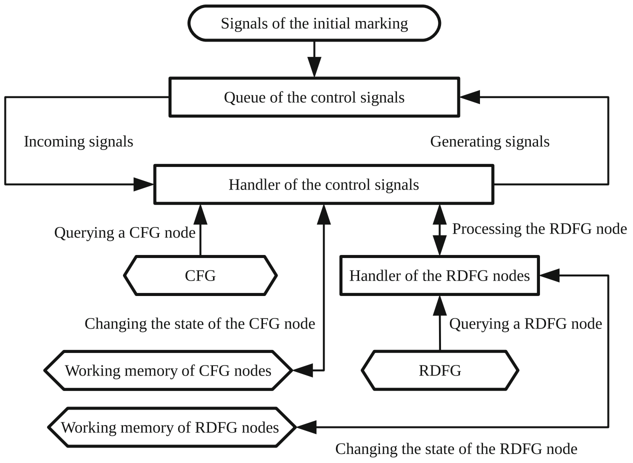
Signals of the initial marking
Queue of the control signals
Incoming signals
Generating signals
Handler of the control signals
Querying a CFG node
Processing the RDFG node
CFG
Handler of the RDFG nodes
Changing the state of the CFG node
Querying a RDFG node
Working memory of CFG nodes
RDFG
Working memory of RDFG nodes
Changing the state of the RDFG node

Fig. 7. Event-processor general structure

Before the event machine launch, a linker assembles separate functions from the repository into a programme. The linker checks the presence of all components that are listed in the section of the external links of the RDFG. If any required component is absent, the interpretation is reported impossible. Each required function is also linked. The event machine scheduler stores a table with RDFGs and CFGs loaded by the linker.

The process of interpretation starts with the creation of the first (initial) EP. It receives the RDFG and CFG of the function which is the first to execute. The EP saves the data in the working memory of the RDFG nodes, while automaton states are stored in the working memory of CFG nodes.

The states of the CFG nodes automata connected with the RDFG constant operators are initially set to new-signal generation. These signals are transmitted through the CFG connections and activate receiving automata. The process of signal transmission through the connections lasts until the "return" node of the corresponding RDFG is processed (in this case, the function is considered completed), or until the event queue is empty. In the latter case, the EP switches to sleep mode, sending a signal about this to the event-machine scheduler. This situation occurs when all inner signals are processed and there is no incoming control signals notifying of returned results from the called functions.

4.3 Optimisation of Data-Driven Functional Parallel Programmes

Within the system of data-driven functional parallel programming, we have developed a number of optimising transformations that take advantage of the peculiarities of our computing model.

1. Dead-code elimination (removal of code that does not affect the programme results). The optimiser starts at the "return" node, traverses the DFG and marks all the reachable nodes. The rest of the nodes are removed.
2. Optimisation within iterative calculations. Traditionally, compilers carry out this kind of transformations for loops. In our case, similar transformations are applied to recursive functions and parallel lists defined in the language model. In particular, calculations inside a recursive function that remain constant during the recursive calls are moved to a new auxiliary function, whose result is passed to the recursive function as an additional parameter; in functions applied to parallel lists, all computations that are independent from the function parameters are moved to the calling function.
3. Inline substitution of simple functions. If a function is sufficiently small (the number of nodes is below a predefined limit), then the function-call overhead is substantial compared to the overall cost of the function. As a result, the function body is better to be inserted at the place of the function call.
4. Duplicate-code elimination. If the DFG subgraphs perform the same operations on the same arguments and also are in one and the same delay list or in hierarchically nested delay lists, then they can be merged, thereby eliminating redundant computations.
5. Optimisation based on equivalent transformations. The RDFG is searched for certain subgraphs that can be transformed to a more computationally simple but equivalent form.
   In particular, the model admits the following equivalent transformations: simplification of single-element parallel list; unwrapping of directly nested parallel lists into a single parallel list; preliminary simplification of parallel lists whose size is known at compilation time.
6. Redundant control-dependencies removal. An RDFG describes data dependencies, and the CFG is created on its basis according to the data-readiness control strategy. However, in several cases, some control relations are redundant, and their removal would not affect the order of programme-operator execution.
7. Generation of a CFG that defines a sequential traversal of the RDFG nodes. This removes the overhead of data-readiness analysis.

4.4 Formal Verification of Data-Driven Functional Parallel Programmes

The proposed paradigm eases formal verification of programmes owing to the absence of resource limitations and to the fact that a programme defines only data dependencies. The main problems in this area are to study the application of formal correctness-proof methods to the proposed language and to develop a toolkit to assist formal verification.

For the correctness proof, we employ the axiomatic approach based on Hoare Logic [15]. The specification of the programme is expressed in a special formal language (specification language). A Hoare triple is represented by a data-flow graph of the programme whose input and output arcs are marked with formulas in the specification language (called a precondition and a postcondition, respectively). The process of proving the programme correctness consists in marking the graph arcs with formulas, graph modifications and folding. As a result, we obtain a number of RDFGs with all arcs marked with formulas. Each of these RDFGs can be transformed into a logic formula. If all these formulas are identically true, then the programme is correct [16].

The process of proving is quite complicated since it requires taking into account a great number of graphs and their transformations. That is why we have developed basic concepts of a toolkit for supporting formal verification of DDFP programmes [17]. The system takes a DFG and programme pre- and postcondition as its input. It searches for unmarked arcs of the graph and assists in selecting appropriate axioms and theorems for marking the arcs. The proof process of a programme correctness is considered as a tree in which each node is a partially marked DFG of the programme. The construction of the proof tree finishes when all its leaves are totally marked DFGs. Thereafter, a logic formula is generated for each DFG in the leaves. The programme correctness is proved if we manage to prove that all these formulas are identically true.

5 Overview of Related Works

In the area of languages and support tools for parallel programming, the current focus is on the creation and development of architecture-dependent systems. The difference between these and our approaches has been discussed above. There exist unconventional methods and tools for parallel programming, but usually, they are being developed by small groups of developers. The development often finishes at the stage of an experimental solution, which does not make it more popular. An exception is special-purpose systems, which target specific object domains and have a considerable optimisation potential for existing architectures owing to the limited number of tasks to solve. For instance, the non-procedural language NORMA [18] targets problems of mathematical physics and translates into parallel programmes for different architectures.

Dataflow programming is implemented in the LabVIEW system [19]. The graphical programming language named “G” is designed to target the problems of the automation of scientific researches and production processes. The language is oriented towards large-blocks programming and, in fact, describes an interaction of different resources. On the contrary, our approach targets unlimited resources and parallelism at the level of elementary operations.

Sisal is one of the universal functional languages of parallel programming that has been developed for a long time. The first version of this language was released in 1985. In recent times, the Institute of Informatics Systems of the Siberian Branch of the Russian Academy of Sciences has been developing this language, and its latest release is Sisal 3.2 [20]. It should be pointed out that the main goal of the project is to provide application programmers with convenient environment for functional programme development, with the subsequent execution of the programmes on a parallel computing system available via telecommunication networks. This is the main difference from our goals: we seek, investigate and implement operators that allow for efficient expression of unlimited parallelism in architecture-independent parallel programmes. In our view, this allows to rethink the process of development, analysis and transformation of parallel programmes. In particular, it is demonstrated in [21] how to deduce known methods of sorting by imposing different constraints on an algorithm with initially unlimited parallelism.

6 Conclusions

The toolkit being developed allows to create architecture-independent parallel programmes whose execution may be controlled using different strategies without changing the programme logic. Nothing prevents us from performing preliminary optimisation, testing and verification of the DFG in the architecture-independent manner. Further transformations of intermediate programme representations to programmes for real computing systems can be carried out on the already debugged source code by means of formal methods, which would increase programme reliability. Also, it is possible to perform additional optimisations, for instance, to increase the efficiency of memory usage.

It should be pointed out that all transformations are done only after a correctly functioning programme code is written. In the meantime, we have solved only the first part of the problem—programme execution on the emulator of the event machine. The next stage of our development is programme transformations for existing computing systems. Besides making the developed tools more convenient to use, we intend to create an integrated development environment (IDE) that additionally supports function repository, translating, verification and execution of programmes.

References

1. McKenney, P.E.: Is Parallel Programming Hard, And, If So, What Can You Do About It? www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

2. Karpov, Y.G.: Model Checking. Verification of Parallel and Distributed Program Systems. BHV-Petersburg, Saint Petersburg (2010). (in Russian)
3. Korneev, V.D.: Parallel programming in MPI. Institute of Computational Mathematics and Mathematical Geophysics, Siberian Branch of the Russian Academy of Sciences, Novosibirsk (2002). (in Russian)
4. Akhter, S., Roberts, J.: Multi-core Programming Increasing Performance through Software Multithreading. Intel Press, Santa Clara (2006)
5. Cheng, J., Grossman, M., McKercher Ty.: Professional CUDA Programming. Wiley, Indianapolis (2014)
6. Tay, R.: OpenCL Parallel Programming Development Cookbook. Packt Publishing Ltd., Birmingham (2013)
7. Lastovetsky, A.L.: Parallel Computing on Heterogeneous Networks. Willey, Hoboken (2003). https://doi.org/10.1002/0471654167
8. Maad, S. (ed.): Grid Computing – Technology and Applications, Widespread Coverage and New Horizons. InTech, Rijeka (2012). https://doi.org/10.5772/2290
9. Gaster, B.R., Howes, L., Kaeli, D.R., Mistry, P., Schaa, D.: Heterogeneous Computing with OpenCL. Advanced Micro Devices, Inc., Elsevier Inc., Santa Clara (2013)
10. Voevodin, V.V., Voevodin, Vl.V.: Parallel Computations. BHV-Petersburg, Saint Petersburg (2002). (in Russian)
11. Legalov, A.I.: The functional programming language for creating architecture-independent parallel program. Comput. Technol. **10**(1), 71–89 (2005). (in Russian)
12. Legalov, A.I.: Managing computation in parallel systems and programming languages. Sci. Bull. NSTU **3**(18), 63–72 (2004). (in Russian)
13. Backus, J.: Can programming be liberated from von Neuman style? A functional stile and its algebra of programs. CACM **21**(8), 613–641 (1978). https://doi.org/10.1145/359576.359579
14. Legalov, A.I., Redkin, A.V., Matkovskii, I.V.: Data driven functional parallel programming with data coming asynchronously. In: PACT 2009, pp. 573–578. South Ural State University, Chelyabinsk (2009). (in Russian)
15. Hoare, C.A.R.: An axiomatic basis for computer programming. CACM **12**(10), 576–585 (1969). https://doi.org/10.1145/363235.363259
16. Kropacheva, M., Legalov, A.: Formal verification of programs in the pifagor language. In: Malyshkin, V. (ed.) PaCT 2013. LNCS, vol. 7979, pp. 80–89. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39958-9_7
17. Ushakova, M.S., Legalov, A.I.: Automation of formal verification of program in the Pifagor language. Model. Anal. Inf. Syst. **22**(4), 578–589 (2015). https://doi.org/10.18255/1818-1015-2015-4-578-589
18. Andrianov, A.N., Baranova, T.P., Bugerya, A.B., Efimkin, K.N.: Nonprocedural NORMA Language and Its Translation Methods for Parallel Architectures. University News. North-Caucasian region, Technical Sciences Series, vol. 3, no. 195, pp. 5–12 (2017). https://doi.org/10.17213/0321-2653-2017-3-5-12
19. Yang, Y.: LabVIEW Graphical Programming Cookbook. Packt Publishing, Birmingham (2014)
20. Kasyanov, V.: Sisal 3.2: functional language for scientific parallel programming. Enterp. Inf. Syst. **7**(2), 227–236 (2013). https://doi.org/10.1080/17517575.2012.744854
21. Legalov, A.I.: Parallel algorithms development. Open Syst. **9**(101), 64–68 (2004). (in Russian)