On Deterministic Parallel Implementation of the Branch-and-Bound Method with Monotonic Objects

Alexei Adamovich[1][0000-0003-1392-8871] and Andrei Klimov[2][0000-0003-0418-7311]

[1] Ailamazyan Program Systems Institute of RAS, Peter I str., 4a, selo Veskovo, Pereslavl-Zalessky, Yaroslavl region, 152021, Russia
[2] Keldysh Institute of Applied Mathematics of RAS, Miusskaya sq., 4, Moscow, 125047, Russia
klimov@keldysh.ru

Abstract. This paper continues the authors' research into the development of a system of deterministic parallel programming and the creation of libraries of so-called *monotonic* classes. The system is two-level and uses two input languages: a universal object-oriented language like Java for the implementation of monotonic classes, which makes up the lower-level subsystem, and a higher-level functional language with the ability to create and use immutable and monotonic objects. The libraries of monotonic classes ensure that all programs in the higher-level language that use only monotonic classes are *deterministic* and *idempotent* when they are parallelized by asynchronous calls of all functions. An important part of the development of such a system is the creation of monotonic class libraries for various fields of application and the demonstration of solutions to several applied problems using them. This paper is devoted specifically to implementing the search for the minimum weight of paths in a graph by the branch-and-bound method. We provide a solution referred to as *conditionally monotonic*, where the monotonicity holds only for nonnegative edge weights. The problem of the exact implementation of the branch-and-bound method with monotonic objects is stated.

Keywords: Models of Parallel Computation, Deterministic Programs, Monotonic Objects, Branch-and-Bound Method.

1 Introduction

1.1 The General Problem

The problem discussed in this article belongs to the following general direction: building a system and class library for *deterministic* parallel and concurrent programming based on object-oriented languages.

The system developed by the authors is intended for programming only those tasks that give a deterministic result, and for that part of them that allow parallel and concurrent implementation in the form of a deterministic program by means of this system. We explore object-oriented programming in languages such as Java using threads, including light-weight threads, fibers, and coroutines. Currently, the principles of constructing a system are inapplicable to numerical problems with specific programming tools based on MPI, OpenMP, etc., and do not overlap with the domain and methods of, e.g., the Norma language [7] for declarative and deterministic programming of such problems.

The goal is to provide the application programmer with object-oriented language tools and libraries, so that, when programming with their use, any program is guaranteed to be deterministic and well scalable on parallel hardware. However, there is no finite and complete set of tools that can be fixed once and for all and provided in the top-level language so that all programs in it are deterministic and use all the variety of parallel programming tools. Therefore, the system should be open for the convenience of creating new domain-specific libraries.

The literature on the development of various deterministic parallel programming tools demonstrates (see our review [5]) that they are generally implemented in low-level languages (such as C or VHDL), providing high-level tools for the application programmer[1]. It turns out that in order to implement new means, it is necessary to use the lower level with high labor costs.

The task is to raise the system library development tools to a fairly high level of an object-oriented language such as Java, providing the application developer with a subset of tools, which guarantee the determinacy of her programs. To advance in this direction, a project of a two-level programming system was proposed in articles [5, 6] and explained in Section 2.

This paper continues the series of works of the authors [2-6,11] on deterministic parallel programming and is based on our earlier work on the T-system [1] and monotonic objects [8,12]. The notion of *monotonic classes and objects* implies two properties: *determinacy* and *idempotency* of user programs. The idempotency is of great practical importance for ensuring fault tolerance of parallel and distributed systems.

We share the goals of the authors of the article with the self-explanatory title "Parallel Programming Must Be Deterministic by Default" [14], but we offer other means for solving it. Our approach resembles the work [17, 18] on the determinacy of parallel computing with shared data belonging to some *semilattices*[2] but we generalize it in the framework of object-oriented languages.

Our work contributes to solving one more fundamental problem: building a model of computation between functional and object-oriented languages. It allows for explicit references to objects which can mutate (albeit in a disciplined way). At the same time the basic properties of functional languages are preserved: determinacy, idempotency, the existence of a kind of denotational semantics, and some adequate replacement of referential transparency, which is broken by side effects and the operators of object creation, which generate new references.

---

[1] A case in point: The libraries of parallel and concurrent programming of the purely functional language Haskell [19] are implemented using low-level means that are inexpressible in Haskell itself.

[2] https://en.wikipedia.org/wiki/Semilattice

1.2 The Specific Problem

A significant part of the system we are developing is libraries of monotonic classes for various problem domains, the use of which guarantees determinacy and idempotency. This library is accumulated in the process of solving samples of applied problems.

The first set of notions required in many tasks is a way to efficiently represent graphs. This is natural in usual object-oriented programming but is nontrivial when implemented with monotonic objects. In [11], the problem of constructing cyclic data structures with monotonic objects is explained by examples, and the solutions are demonstrated. Here explicit references are used to represent the relationships between vertices and edges of graphs, as in object-oriented programming. Notice that this is not available in purely functional languages (where the main data structure is trees[3]). These constructs are needed for many programs.

In this paper, we consider a specific task — to find the minimum weight of simple[4] paths in a graph from a given vertex. The edges of the graph are labeled with non-negative weights, and the weight of a path is defined as the sum of the weights of its constituent edges. This problem has many variations. A wide-spread version: the paths that end in the second given vertex are examined. In our experimental study, for the convenience of measuring the scalability of the program executed on a multicore system, we took another version, in which all simple paths of a fixed length from a given vertex, without fixing their end, are explored.

This kind of algorithms is well parallelized and is deterministic by nature. The processes that enumerate various paths communicate via a shared variable that stores the minimum weight found to date. A process that has found a shorter path assigns a new weight to the variable. As the minimum on a finite set of numbers is unequivocal, in order to prove the correctness and determinacy of the algorithm implementation, one only has to ensure that all paths (or enough of them) are visited.

The classic way to solve this problem is the branch-and-bound method[5]. Its core idea is the *pruning* operation, which cancels the current partial path when its weight exceeds or equates to the minimum reached so far. Here, the main subtlety is that the comparison operation with the current minimum is nonmonotonic, and hence cannot be directly used in the higher-level language. Therefore, it must be "hidden" inside a monotonic object.

[3] The work [13] proposes an "efficient" graph representation in the functional language Haskell, but we consider it a "non-solution". A representation of graphs in the form of lists of vertices and edges (that is, in the form of trees) is given, and then it is suggested that an efficient representation with the same operations is implemented by *low-level* means.

[4] In graph theory, a path without self-intersection is called *simple*.

[5] https://en.wikipedia.org/wiki/Branch_and_bound

1.3 Contribution

We describe an implementation of the branch-and-bound method satisfying the following requirements:

- the top-level part of the algorithm is written in a functional language and is parallelized by asynchronous calls of all (or part of) the functions;
- monotonic objects are used for the communication between processes;
- the determinacy of a user program in the higher-level language is guaranteed by the implementation of monotonic classes;
- the difference between the versions of the program, first, with the complete enumeration of paths and, second, with pruning by the branch-and-bound method, is implemented in one place inside a monotonic object. This simplifies proving the equivalence of the versions and the correctness of the program with pruning.

The presented solution demonstrates the use of a higher-order monotonic object, that is, with an operation that takes a lambda expression as an argument. We assume that without a higher-order operation, it is impossible to implement the branch-and-bound method with monotonic objects.

1.4 Outline

In Section 2 we explain the goals and principles of constructing a two-level system of deterministic parallel programming. Section 3 gives the definition of monotonic objects and classes. Section 4 is the main one: it presents a program for finding the minimum weight of paths from a given vertex of a graph, implemented according to our approach. In Section 5, we find out that this solution does not exactly meet the monotonicity conditions and is just *conditionally* monotonic, depending on the non-negativity of the edge weights, and we explain why an exact solution should be sought. Related work is discussed in Section 6. In Section 7 we conclude.

2 A Two-Level System of Deterministic Parallel Programming

The principles of constructing the system for deterministic parallel programming are applicable for any mature object-oriented language that satisfies the following requirements:

- The object-oriented language must have a functional subset. The presence of suitable syntactic sugar (e.g., lambda expressions) is desirable, but not necessary. It should be possible to implement the verifier that checks that a program belongs to the functional subset (without this, guarantees of determinacy would be lost), or to implement a domain-specific language translated to the functional subset.
- Automatic memory management and garbage collection should be present in the implementation of the object-oriented language. Without these, it is impossible to

program in the functional style. For example, languages on the JVM and MS.NET platforms are suitable.
- Parallel and concurrent programming tools should be present, that allow the implementation of asynchronous function calls and their synchronization on access to shared data.

We are prototyping the system using the JVM platform, the Java and Kotlin[6] languages, the Ajl language developed by ourselves [3], the Quasar[7] library that implements light-weight threads, fibers, on the JVM. We also follow the development of the Loom project[8], which will be more tightly integrated with Java than Quasar and will eventually be included in the standard JVM. We develop our language tools in the Eclipse[9] IDE.

Since the goal of the project is twofold — to provide both guaranteed deterministic programming to the application developer, and the open lower level to parallel programming experts who can use all the means of the object-oriented language, — the input language must be distinctly subdivided into two parts, guaranteed deterministic and indeterministic:

- The upper level is the deterministic part, where application programmers write program code in a functional subset. Here, the determinacy of any program that uses the libraries specially prepared at the lower level, is guaranteed.
- The lower level is the potentially indeterministic part, where qualified programmers create class libraries for specific application domains in a universal object-oriented language using all indeterministic parallel programming tools. The library authors are responsible to the users of the higher level for the determinacy of parallel programs.

3 The Notion of a Monotonic Class and Object

We refer to objects and classes defined in the lower-level language and used in the upper-level functional language as *monotonic*, since their state changes monotonically in some semilattice, which, as a rule, can be constructed using the class code. However, the formal definition, which is convenient to use in reasoning about programs and proving their properties, is given *operationally* below [6,8,12].

Monotonic objects and classes are such that, when they are created and used in any program in a functional language, all expressions satisfy the following two properties:

- *Determinacy*: the results obtained by parallel computation (in any possible order) of an expression several times from the same initial state, are equivalent, and their side effects are indistinguishable programmatically in the functional language.

[6] http://kotlin.jetbrains.org/
[7] https://github.com/puniverse/quasar
[8] https://wiki.openjdk.java.net/display/loom/Main
[9] https://www.eclipse.org/

- *Idempotency*: simultaneous parallel computation of two copies of an expression instead of one, produces the equivalent results and the side effect is indistinguishable programmatically in the functional language from the side effect of one computation.

These properties must be satisfied simultaneously for all monotonic classes when they are used together in any functional program.

At present, we are constructing monotonic classes, using these properties informally and naively. In the future, we expect that computer-assisted verification tools for proving monotonicity will be developed. Although fully automatic tools for all cases cannot be built, as in the general case of program verification, we hope that this is achievable for most practical definitions of monotonic classes.

4 Deterministic Parallel Implementation of the Branch-and-Bound Method with a Monotonic Object

In this section, a deterministic parallel implementation of the algorithm that finds the minimum weight of a path in a graph from a given vertex, is presented. The pseudocode in Fig. 1 is written in a Java/Kotlin-like syntax[10].

The code in Fig. 1 uses the following classes (only those operations[11] are listed that are used in the code):

- Immutable class *Graph* represents a graph. The single *Graph* instance is stored in a global variable *graph*. The objects of classes *Vertex* and *Edge* access it to perform their operations.
- Immutable class *Vertex* represents vertices with the following operation:
  - function *vertex.outgoingEdges* returns an *Iterator* over the list of edges outgoing from the *vertex*.
- Immutable class *Edge* represents edges with the following operation:
  - function *edge.endVertex* returns the end vertex of the *edge* (that is, the vertex, for which this *edge* is incoming).
- Immutable class *Path* represents paths in the graph with the following operations:
  - predicate *path.isComplete* checks whether the *path* is completed, e.g., has a given length or has reached a given vertex, depending on the problem statement;
  - predicate *path.contains*(*vertex*) checks whether the *path* contains the *vertex*;
  - function *path.weight* returns the *path* weight, that is, the sum of the weights of its edges;

[10] As compared to Java, we omit empty parameters at the end of invocation as in Kotlin, braces and semicolons.

[11] We avoid using the object-oriented term “method” for operations of classes and call them *functions* (with a value), *procedures* (without a value) and operations *in* classes and objects, in order not to be confused with the general meaning of the term “method”, as in: “the branches-and-bound method.” For the invocations of functions and procedures, we use the object-oriented syntax: $x.f$ instead of $f(x)$ and $x.f(y)$ instead of $f(x, y)$.

Input and initial state

- *Graph graph* — the global variable that references to the graph representation, used in the classes *Vertex* and *Edge*
- *Vertex root* — an initial vertex from which the examined paths start
- *Minimizer minimizer* — a global variable with the monotonic object that computes the minimum of a finite set of numbers, in the initial state “negative infinity”

Initial invocation of the recursive procedure *findMinimumPathWeight*

- *findMinimumPathWeight*(*root*)

Recursive procedure *findMinimumPathWeight*

```
findMinimumPathWeight(Path path)

    if (path.isComplete)
        minimizer.addValue(path.weight)
    else
        minimizer.branch(path.weight, () ->
            for (Edge edge : path.lastVertex.outgoingEdges)
                if (!path.contains(edge.endVertex))
                    findMinimumPathWeight(path.extendedWith(edge))
        )
```

Returning result after the completion of the *findMinimumPathWeight* procedure

- *minimizer.minimum* returns the minimum weight

Fig. 1. The program to find the minimum weight of paths from the initial edge *root*, with the monotonic object *minimizer*

  - function *path.lastVertex* returns the last vertex visited in the *path*;
  - function *path.extendedWith*(*edge*) returns a path object obtained by adding the *edge* to the end of the path.
- Monotonic class *Minimizer* has the single instance stored in the global variable *minimizer*, with the following operations:
  - function *minimizer.minimum* returns the minimum among the numbers sent to the *minimizer*, after the *findMinimumPathWeight* procedure completes (see discussion below);
  - procedure *minimizer.addValue*(*value*) adds the number *value* to the set from which the minimum is taken;
  - procedure *minimizer.branch*(*pathWeight*, *step*) executes the continuation *step*, which is a lambda expression without arguments and value. In the version with *pruning* (corresponding to the branch-and-bound method proper), *step* is not computed of *pathWeight* is greater than the current minimum.

In Fig. 1, the *findMinimumPathWeight* procedure recursively searches for the minimum path weight among the completed paths from a given vertex *root*. The procedure *findMinimumPathWeight* has no result and accumulates the result by side effects — by writing *path.weight* to the monotonic object *minimizer* by calling *minimizer .addValue*(*path.weight*). This algorithm may be executed sequentially, but it can be easily parallelized by running all calls to the findMinimumPathWeight procedure asynchronously and waiting for all *findMinimumPathWeight* processes to complete before the *minimizer.minimum* function returns the result.

For experts on the branch and bound method, the algorithm in Fig. 1 looks natural and does not seem to contain anything tricky. Indeed, it is close to the Wikipedia article[12] except for the order of some code pieces. A notable difference is the higher order *minimizer.branch* procedure with the lambda expression in the second argument *step*. It has no explicit arguments, taking the *path* and *minimizer* values from the surrounding context. Note the syntax "**() ->**" in bold.

The code of the *findMinimumPathWeight* procedure is suitable for exhaustive search of all paths, as well as for pruning by the branch-and-bound method. The variants are selected by varying the code of *minimizer.branch* operation inside the monotonic class *Minimizer*: in the former case, *step* is always executed; in the latter case with pruning, *step* is not executed when the current *path.weight* is greater than or equal to the current minimum weight of the paths passed.

The code in Fig. 1 is imperfect in some respects. First, it does not yet satisfy all the requirements of monotonicity. This is discussed in Section 5.

Second, the *minimizer* object has the following semantic subtlety: the result of *minimizer.minimum* must be returned only after all *findMinimumPathWeight* processes have completed. In the real-world implementations of the branch-and-bound method, a synchronization barrier is puts here that fits poorly into the theory of monotonic objects (although local deviations from the "high theory" are natural in practice). Each of the following two solutions is better suited to the world of monotonic objects.

A pragmatic solution. An asynchronous procedure invocation returns a handler object, the *waitAll* operation on which ensures that all processes created by this invocation have completed. After the operation returns, the *minimizer.minimum* function can be invoked.

This solution is not consistent with the principles stated in Section 2, because requires certain programming style.

A theoretically consistent solution. Two kinds of references to monotonic objects are to be distinguished:

1. A reference of the general "write" kind allows the operations to mutate (monotonically) the state of the referenced object and that of the objects recursively accessible through it.

[12] https://en.wikipedia.org/wiki/Branch_and_bound

2. A reference of the “read-only” kind allows only retrieving information from the objects without programmatically visible mutations.

With this solution, the *findMinimumPathWeight* procedure would use a “write” reference in the *minimizer* object, and the *minimizer.minimum* function would use a “read-only” reference. An operation on a “read-only” reference completes only when all the “write” references to this object disappear (or for some semantic reasons it becomes clear that the result of this operation can no longer be changed). This can be implemented either by reference counters, or by means of “weak references” and garbage collection. The first alternative with reference counters is only suitable under the assumption that cyclic structures are not formed. The second alternative with garbage collection is generally correct by less efficient. Both alternatives do not violate monotonicity.

5 Non-monotonicity of This Version of the Algorithm

Now we have to make the main statement that the *Minimizer* class with the given semantics is monotonic, that is, any functional program that uses it is deterministic and idempotent. However, we can state this only for the first variant of the *minimizer .branch* operation considered above, where *step* is always executed and hence the exhaustive search is performed. But this is not the branch-and-bound method, which we are interested in.

In the version with pruning by to the branch-and-bound method, it must be ensured that the recursive calls to *findMinimumPathWeight* in the lambda expression for *step* wittingly increase the weight of the current path. Now this can be ensured either by the precondition that the weights of all edges are non-negative, or by the *path.extendedWith*(*edge*) function, which would check that the weight of the *edge* is non-negative and hence the weight of the path does not decrease. In both cases, the guarantees are given outside of the *Minimizer* class, while our definition of monotonicity implies the determinacy and idempotency of *any* higher-level program. Therefore, this solution is just *conditionally* monotonic, that is, monotonic with the requirement that an external condition is satisfied.

In this paper, we leave open the question of constructing a truly monotonous implementation of the branch-and-bound method. This is our future work.

6 Related Works

A survey of the works on deterministic parallel programming that we are aware of, is given in [5]. Here we review only two of the most closely related approaches.

The goal of the first approach is like ours: deterministic parallel programming in an object-oriented language. But it fundamentally differs in the methods for achieving it. In a series of works, of which we single out one with the self-explanatory title [14], their authors develop a static analysis of programs in Java with certain extensions, called Deterministic Parallel Java, DPJ. The analysis determines the cases where side effects do not lead to indeterminacy. The language extensions enable the programmer to provide additional information to the analysis. These methods are further development of the typing systems that take side effects into account (*effect systems*[13]). We do not explore this further here, since in our approach no static analysis is used.

The second approach is close to ours in both goals and methods. It is based on monotonic changes of shared variables. This idea is quite old and has been already implemented in various specific cases. Their common idea is that, to communicate parallel processes, special data structures are used such that determinacy is not violated. Below are the most well-known of them:

- I-structures [10];
- Kahn networks [15];
- TStreams, Concurrent Collections [16];
- lattice-based data structures [17,18].

These (and our) approaches have a common feature: variables via which parallel processes communicate, change their state monotonically, only upwards on some semilattice from the undefined state (⊥) to more and more defined ones. The upper element of the lattice (⊤) means “over-defined” value; this corresponds to an error, an exception in program code. For example, the set of values of an I-structure with integer values is described by the lattice that consists of the lower element “undefined” (⊥), incomparable integers and the upper element “over-defined” (⊤). When the value *y* is assigned to a variable with a value *x*, the smallest upper bound of the values *x* and *y* is stored in it. If the result is the upper element ⊤, an exception is thrown.

This idea was elaborated in the PhD thesis by Lindsey Kuper [17] and in publications together with her colleagues [18]. They proved the determinacy of parallel execution of processes communicating via variables that take values from an arbitrary semilattice. In our work, we use these ideas and generalize them to objects with monotonically changing states, having given them the operational definition rather then through lattices.

In our future work, we should compare the verification technique of the implementation of the branch-and-bound method according to our approach and the existing proofs of its correctness, e.g., described in [9].

7 Conclusion

The task of developing libraries of so-called *monotonic* classes within a two-level object-oriented deterministic parallel programming system was discussed. In this system, the higher-level language is close to a functional language that allows for parallelization of all functions by asynchronous calls. Processes communicate via *monotonic* objects, whose classes are defined in the lower-level language, which is a mature object-oriented language with parallel and concurrent programming means.

[13] https://en.wikipedia.org/wiki/Effect_system

The implementation of monotonic classes guarantees the *determinacy* and *idempotency* of all programs in the higher-level language.

In the framework of such a general approach, this article is devoted to solving a particular problem — the implementation of the search for the minimum path weight in a graph among the paths from a given vertex, using the monotonic object that computes the minimum of the numbers sent to it. The presented solution is just *conditionally monotonic*, as it requires that the input graph is checked preliminarily, to ensure that the weights of the edges are non-negative. The open problem of completely implementing the branch-and-bound method using monotonic objects has been posed. The solution should guarantee the determinacy of *any* higher-level program, in such way that all necessary checks are done in the code of monotonic classes.

Acknowledgements. We express our gratitude to our English teacher, Philippa Jephson, who helped us edit this paper, fix a lot of errors and turn it into a much more readable form. The remaining glitches are ours.

This work was carried out with the financial support of the Ministry of Education and Science of Russia (grant No. RFMEFI61319X0092).

References

1. Abramov, S.M., Adamovich, A.I., Kovalenko, M.R.: T-System — An Environment Supporting Automatic Dynamic Parallelization of Programs: An Example of the Implementation of an Image Rendering Algorithm Based on the Ray Tracing Method. Programmirovanie 25(2), 100–107 (1999). (In Russian).
2. Adamovich, A.I.: Fibers as the Basis for the Implementation of the Notion of the T-Process for the JVM Platform. Program Systems: Theory and Applications 6(4), 177–195 (2015). doi:10.25209/2079-3316-2017-8-4-221-244. (In Russian).
3. Adamovich, A.I.: The Ajl Programming Language: The Automatic Dynamic Parallelization for the JVM Platform // Program Systems: Theory and Applications 7(4), 83–117 (2016). doi:10.25209/2079-3316-2016-7-4-83-117. (In Russian).
4. Adamovich, A.I., Klimov, And.V.: On Experience of Using the Metaprogramming Development Environment Eclipse/TMF for Construction of Domain-Specific Languages. In: Nauchny`j servis v seti Internet : Trudy` XVIII Vserossijskoj nauchnoj konferencii, pp. 3–8. Keldysh Institute of Appl. Math. of RAS, Moscow, Russia (2016). doi:10.20948/abrau-2016-45. (In Russian).
5. Adamovich, A.I., Klimov, And.V.: How to Create Deterministic by Construction Parallel Programs? Problem Statement and Survey of Related Works. Program Systems: Theory and Applications 8(4), 221–244 (2017). doi:10.25209/2079-3316-2017-8-4-221-244. (In Russian).
6. Adamovich, A.I., Klimov, And.V.: An Approach to Deterministic Parallel Programming System Construction Based on Monotonic Objects. Vestnik SibGUTI 2019(3), 14–26 (2019). URL: http://vestnik.sibsutis.ru/showpapper.php?act=showpapper&id=870. (In Russian).
7. Andrianov, A.N., Baranova, T.P., Bugerya, A.B., Yefimkin, K.N.: Nonprocedural NORMA language translation for GPUs. Keldysh Institute Preprints 2016(73), 1–24 (2016) doi:10.20948/prepr-2016-73.

8. Klimov, And.V.: Deterministic Parallel Computation with Monotonic Objects. In: Nauchny`j servis v seti Internet: mnogoyaderny`j komp`yuterny`j mir : Trudy` Vserossijskoj nauchnoj konferencii, pp. 212–217. Izd-vo Moskovskogo universiteta, Moscow, Russia (2007).
9. Shilov N.V.: Verification of Backtracking and Branch and Bound Design Templates. Modeling and Analysis of Information Systems 18(4), 168–180 (2011). URL: https://www.mais-journal.ru/jour/article/view/1107/820.
10. Arvind, Nikhil, R.S., Pingali, K.K.: I-structures: Data Structures for Parallel Computing. ACM Trans. Program. Lang. Syst. 11(4), 598–632 (1989). doi:10.1145/69558.69562.
11. Adamovich, A.I., Klimov, And.V.: Building Cyclic Data in a Functional-Like Language Extended with Monotonic Objects. In: X Workshop PSSV: Program Semantics, Specification and Verification: Theory and Applications (Novosibirsk, Akademgorodok, Russia, July 1–2, 2019) : Abstracts, pp. 11-19. A.P. Ershov Institute of Informatics Systems, Novosibirsk, Russia (2019). ISBN 978-5-4437-0918-5.
12. Klimov, And.V.: Dynamic Specialization in Extended Functional Language with Monotone Objects. SIGPLAN Not. 26(9), 199–210 (1991). doi:10.1145/115865.376287.
13. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. 11 (5), 467–492 (2001). doi:10.1017/S0956796801004075.
14. Bocchino, R.L. (Jr.), Adve, V.S., Adve, S.V., Snir, M.: Parallel Programming Must Be Deterministic by Default. In: Fifth USENIX Conference on Hot Topics in Parallelism, HotPar'09, pp. 4–4 (6 pages). USENIX Association (2009).
15. Kahn, G.: The Semantics of Simple Language for Parallel Programming. In: IFIP Congress, pp. 471–475 (1974).
16. Burke, M.G., Knobe, K., Newton, R., Sarkar, V.: Concurrent Collections Programming Model. In: Encyclopedia of Parallel Computing, pp. 364–371. Springer US (2011). doi:10.1007/978-0-387-09766-4_238.
17. Kuper, L.: Lattice-based Data Structures for Deterministic Parallel and Distributed Programming. Ph.D. Thesis. 253 pages. Indiana University, IN, USA (2015).
18. Kuper, L., Todd, A., Tobin-Hochstadt, S., Newton, R.R.: Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. ACM SIGPLAN Not. 49(6), 2–14 (2014). doi:10.1145/2666356.2594312.
19. Marlow, S.: Parallel and Concurrent Programming in Haskell. 322 pages. O'Reilly Media, CA, USA (2013).