Optimization of Imperative Functional Parallel Programs with Non-local Program Transformations

Alexei Adamovich

Ailamazyan Program Systems Institute of Russian Academy of Sciences, RCMS, Pereslavl-Zalessky, Russia
lexa@botik.ru

Abstract. Functional programming is among paradigms used for the real-world parallel application development. PSI of RAS during a considerable period of time develops the T-System approach based on functional programming. This paper briefly describes the architecture of the ACCT compiler system targeted at performing transformations of the T-System programs. An algorithm for nonlocal optimizing transformation of a typical parallel application solving massively parallel problem in the T-System is presented. The author finally mentions several opportunities of other possible applications of the proposed architecture.

Keywords: functional programming, parallel application development, T-System, program transformation

1 Introduction

Advance in the field of parallel computations is one of the modern trends. The advance is due not only to the implementation of supercomputers with over 10 PFlops of performance. A substantial reason is that multicore processor architecture has become the dominant on the desktop PCs.

The current state of software tools for parallel applications development implies a coexistence of a number of paradigms. Specifically, the functional programming paradigm is being developed as a base for implementing the real-world parallel applications. In this paradigm, the possibility of automatic parallelization and dynamic load balancing is very attractive.

From the first half of 90s, the Aylamazyan Program Systems Institute of the Russian Academy of Sciences (PSI of RAS) develops an approach to parallel program development based on the functional paradigm which is called now the T-System [1]. Today, in PSI of RAS, there are made several different implementations of the T-System [2,3]. However, for all of the implementations one common disadvantage is the lack of tools for deep analysis and transformation of programs.

In the paper, the author presents general principles of the T-System. The paper also describes the architecture of the compiler version for the T-System

(with a codename ACCT) being implemented in PSI of RAS. ACCT allows to analyze programs given it at input and to execute their optimizing transformations. Then, the author outlines an algorithm for non-local transformations of a typical parallel application solving massively parallel problems in the T-System. The paper further gives several examples of other possible applications of the proposed architecture for increasing the efficiency of parallel applications.

2 Compiler Design

2.1 Basic Properties of the Input Language and the Model of Computation

For ACCT, the input language is an extended restriction of the C programming language (cT) [4]. Function bodies are written with a conventional imperative style. Function interaction is possible only within the framework of the functional programming paradigm, without the side effects: the information from outside can only be received via function arguments, and the transfer of information outwards is performed by sending function results (there may be a number of results).

When the T-function is called a T-process is created – which is a user-level thread. A new started T-process is potentially capable of being executed in parallel with the initial T-process. For enabling a parallel execution at the T-process launch, the variables that are receiving their values as a result of the functional call take special values – so called "non-ready values". A non-ready value is replaced with normal (ready) one after the T-process completed sending a corresponding result of the T-function call.

Non-ready values are located in special variables (outer variables). A non-ready value may easily participate in assignment of a value of one outer variable to another outer variable of the same type. If the T-process needs an outer variable value for executing a nontrivial operation (such as computation of the result of an arithmetic operation or transformation of an integral value into a float point value), the execution of such a T-process will be suspended until the outer variable takes a ready value.

It must be noted that T-function bodies may contain the calls of conventional functions (C functions), which requires to limit the effect of such a call by the T-process on the background of which the call is executed (there should be no side effects as far as other T-processes are concerned).

2.2 Compiler Architecture

The compiler consists of the following main components: front end, a set of transform passes, and back end.

Front end transforms the program module from an input language into an intermediate representation (IR). After the transformation is complete, the intermediate representation obtained as a result is stored in a separate file or special program library.

Each transform pass is able to transfer IR from the file or program library into RAM and somehow modify it. After that, a new version of IR is stored back on the external storage. Since all application modules are available to the transform pass, the performed transformations have a potential possibility to rely on the use of complete information about the application code as a whole.

The compiler back end reads IR from the file or program library and forms the resulting assembly (or C) code for further transformation into an executable program.

There also exists a compiler driver – a control program, which is needed for to call all the passes described above in the proper order.

A similar structure of compiling systems is used in a number of program transformation systems, such as SUIF [5], LLVM [6], OPS [7], etc. The ACCT implementation is heavily based on the C front end of the GCC compiler.

Hereinafter, we'll give an example of a non-local transformation of an application program. Such transformation may be implemented with the proposed program architecture.

3 Example of Program Transformation

3.1 Initial Problem

The proposed program transformation is suitable for the applications solving massively parallel problems. As an example of a program being transformed, we use a special modification of a standard iterative ray tracing algorithm. In case of ray tracing, a variable parameter in a massively parallel problem is a pair of coordinates of a point on the image plane.

The upper level of the modified algorithm implies a bisection of the rectangular part of the image plane containing the image. The division recursively proceeds until, after some step in the recursion, the resulting rectangles become sufficiently small. Thereafter, each of the resulting small rectangles is filled with image pixels by means of a standard tracing algorithm. Such small rectangular fragments are then assembled into a composite image.

Each small image fragment may be built independently of the others, which allows a parallel implementation of the problem. The recursive method of image fragmenting permits to bypass (e.g. executing the task on a cluster) the computation sequence which is typical for the so-called "task farm" paradigm and to avoid the appropriate performance penalties.

The implementation of the algorithm on cT may be represented as the following three functions:

1. The `render_scene` function (which is a C function) is destined for filling small rectangles with the RGB intensity values for each point of the fragment contained within such a rectangle.
2. The `render_scene_ut` T-function recursively bisects the rendering area. It also calls the `render_scene` function – in case the size limit of the area is reached (that is the base case).

3. `Tmain`. The launch of the T-process of the `TMain` function starts the execution of any application written in cT. `TMain` reads the scene description from the file and then launches the T-process with the first call to `render_scene_ut`. After that, `TMain` solves the problem of breadth-first traversal of the binary tree built by `render_scene_ut` and assembles a composite image from the fragments located inside the leaves of the tree, in parallel with the computation of individual fragments performed by `render_scene_ut`/`render_scene` calls.

In this paper, we'll consider the `render_scene_ut` T-function. The code of the function is as follows:

```
01 [void safe * sh]
02 render_scene_ut (double f_ulx,f_uly,f_stepx,f_stepy,
03                  int nx, ny, 04 void safe * sh_scene) {
05  void safe * utsh_res;
06
07  if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
08   int ny1, ny2;
09
10   ny1 = ny / 2;
11   ny2 = ny - ny1;
12   utsh_res = tnew (void safe * [2]);
13   utsh_res [0] =
14    render_scene_ut (f_ulx,f_uly,f_stepx,f_stepy,
15                     nx, ny1, sh_scene);
16   utsh_res [1] =
17    render_scene_ut (f_ulx,f_uly + f_stepy * ny1,
18                     f_stepx, f_stepy, nx, ny2,
19                     sh_scene);
20   sh <== utsh_res;
21  } else {
22   utsh_res =
23    tnew (char[sizeof (frag_dsc) +
24          CHAR_PER_POINT * nx * ny] outer);
25   render_scene
26    (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27     ((char *) &(utsh_res.C)) + sizeof(frag_dsc));
28   sh <== utsh_res;
29  }
30 }
```

The function arguments are the parameters of the image fragments on the plane (the coordinates of upper left vertex of the rectangle, step size for each axis, the numbers of steps) and the scene description. As a result, the function returns a special-kind pointer called holder.

The line 7 of the function code above checks whether the bisection of the fragment must be continued. If bisection must be performed the resulting holder being returned (line 20) keeps (points to) a pair of similar holders (with initially non-ready values) returned in their turn by the recursive calls (lines 12 through 19). Otherwise, the function will return the holder (line 28) keeping the image fragment calculated by `render_scene` regular C call (lines 22 through 27).

Figure 1.a illustrates the sequence of the T-processes launched which starts when the `TMain` function calls the `render_scene_ut` T-function. As the picture indicates, the sufficient part of the T-processes recursively launches `render_scene_ut` and builds intermediate vertices of the binary tree fragments. The other part (building the leaves of the tree) computes the image fragments and returns them as the results. This means that almost a half of the T-processes are lightweight and the multiprocessor resources are underused as a consequence.

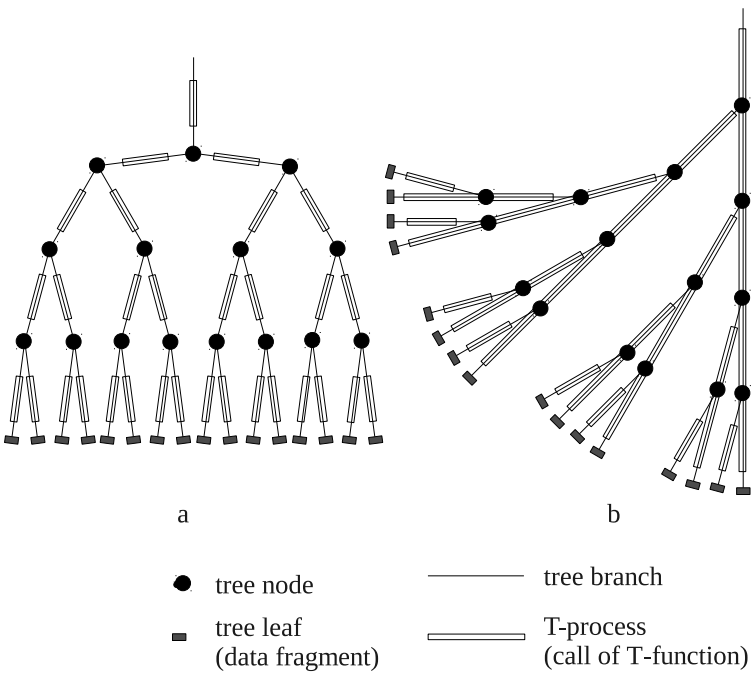

Fig. 1. Building a data fragments tree by parallel T-function calls: a – initial implementation scheme; b – scheme of implementation after modification.

Figure 1.b represents another scheme of building the tree of image fragments. On the 1.b scheme, each of the T-processes builds an image fragment located inside the tree leaf. Also one or more intermediate nodes of the tree may probably be built by the same T-process. This method of solving the problem permits to

avail the computational power of a multiprocessor efficiently since each of the T-processes becomes rather heavy computationally.

It is possible to obtain such parallel implementation of an application by changing the if-part of the conditional statement of the `render_scene_ut` function as follows:

```
06 ...
07  if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
08   int ny1, ny2;
09'  void safe * utsh_w;
10'
11'  ny1 = ny / 2;
12'  ny2 = ny - ny1;
13'  utsh_res = tnew (void safe * [2]);
14'  utsh_w = utsh_res;
15'  for (;;) {
16'   utsh_w [0]
17'    = render_scene_ut
18'       (f_ulx, f_uly, f_stepx, f_stepy,
19'        nx, ny1, sh_scene);
20'   f_uly = f_uly + f_stepy * ny1;
21'   if (nx * ny2 <= MIN_POINTS_PER_FRAG
22'       || ny2 < 2)
23'    break;
24'   ny1 = ny2 / 2;
25'   ny2 = ny2 - ny1;
26'   utsh_w [1] = tnew (void safe * [2]);
27'   utsh_w = utsh_w [1];
28'  }
29'  utsh_w [1] =
30'   tnew (char[sizeof (frag_dsc) +
31'         CHAR_PER_POINT * nx * ny2] outer);
32'   render_scene
33'    (f_ulx, f_uly, f_stepx, f_stepy, nx, ny2,
34'     ((char *) (utsh_w[1].C))+sizeof(frag_dsc));
35'    sh <== utsh_res;
21  } else {
22 ...
```

The numbers of new (subject to changes) lines have a stroke. One can see that one (the second) of the recursive calls has been removed from the if-part of the conditional statement and the remaining (the first) call has been moved into a loop (lines 16' through 19'). This remaining call is responsible for launching the building of the left-upper branch (in terms of Fig. 1) in each intermediate node of the tree. All right-lower branches are computed by a single T-process during the loop execution. As the function exits the loop, it builds a tree leaf and returns the result (lines 29' through 35').

To find a way to generalize the mentioned transformation for solving an arbitrary massively parallel task is the core of the problem.

3.2 Solution: Sequence of Stages

Figure 2 illustrates a simplified scheme of the internal representation of the `compute_it_ut` function which implements the recursive part of the algorithm solving generalized massively parallel problems. Transformations consist in a partial replacement of recursion by iteration. Specifically, one of two recursive calls in the upper-left branch of the final conditional statement is to be replaced with iteration.

A transformation object is an inner representation of a given upper-left branch of a conditional statement. A transformation algorithm consists of three stages:

1. Substitution. The function body is subject to a special form of inlining – it is substituted into the second recursive call of the `compute_it_ut` function implementing the recursion step.
2. Looping. The looping stage is executed in several steps. The execution of all the three steps allows to considerably reduce the number of lightweight parallelism granules.
3. Final cleaning of variables and assignments.

Hereinafter, an overview of each step is presented.

Substitution. The second recursive function call – implementing the recursion step – is substituted with a copy of the function inner representation. Such substitution copies corresponding environments, including call arguments, and also assigns corresponding initial values to them.

As a result of the substitution stage, the inner representation of a recursive branch of the final conditional statement will contain three instead of two recursive calls to the `compute_it_ut` function. After further transformation at the looping stage, two of three recursive calls will be deleted but the remaining one will be executed in the loop body.

Looping. The two (of three) last recursive calls are completely eliminated at the looping stage. As a substitution to the eliminated recursive calls, the `compute_it` C function – loop structure and recursion base – is inserted into the recursive branch of the `compute_it_ut` function. The given procedure may be implemented as a following sequence of steps:

1. A working holder ("outer" pointer) with a unique name – indicated here as `utsh_w'` – is introduced into the `compute_it_ut` T function environment:

```
void safe * utsh_w';
```

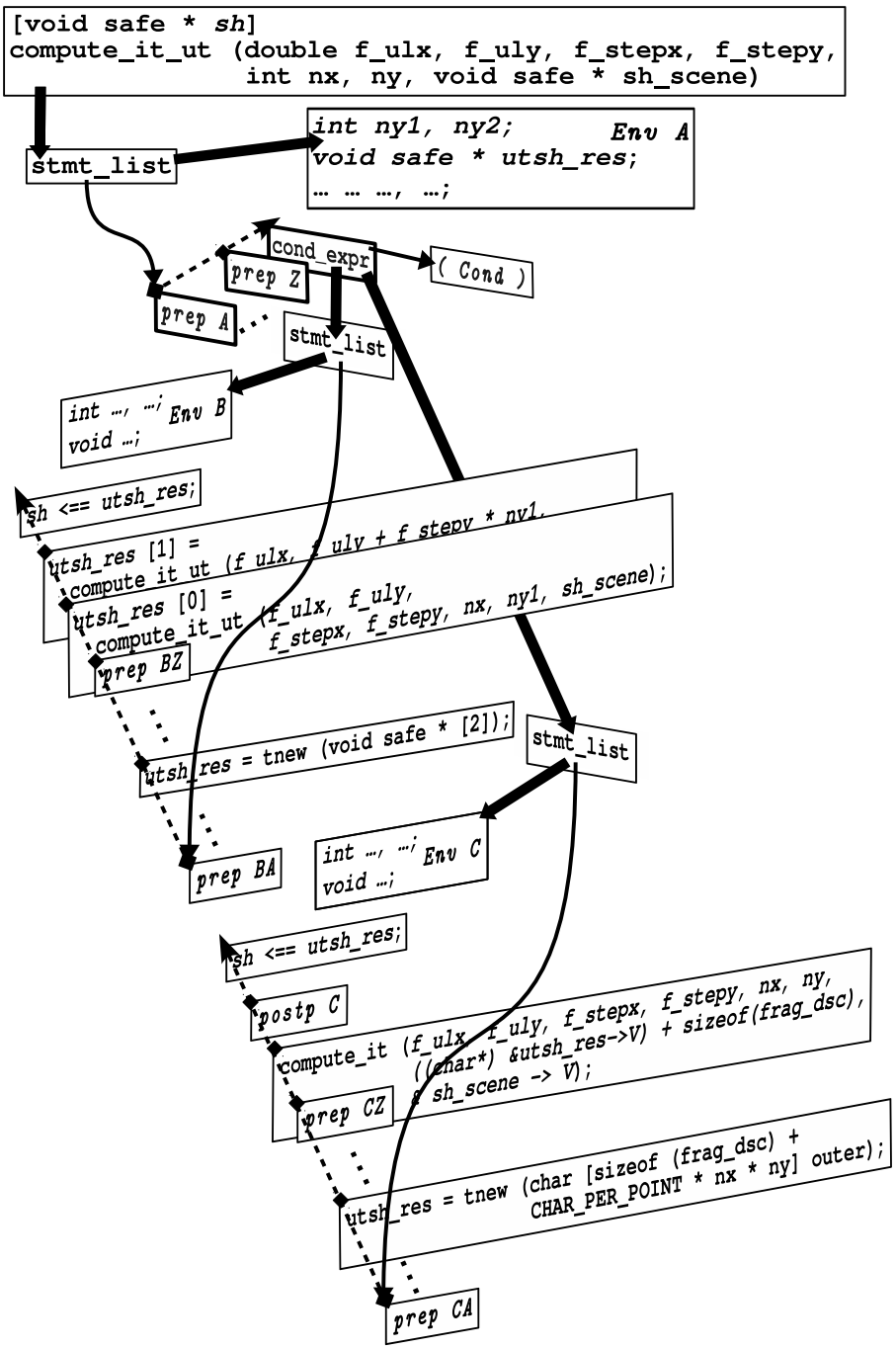

Fig. 2. Scheme of the internal representation of the function implementing the recursive part of the algorithm solving a generalized massively parallel problem

This holder will be used as leading and keep the current tree node – the node which is built in the loop at a current iteration step). The built root of a subtree returned as a result serves as an initial value of the leading pointer:

```
utsh_w' = utsh_res;
```

All subsequent occurrences of the `utsh_res` variable in the transformable code, except for the result sending final statement, should be substituted with occurrences of the `utsh_w'` variable.

2. The list of statements added at the substitution stage as a body of a substituted function is transformed into a loop statement body. The first recursive call of the `compute_it_ut` T-function is also moved into the loop body as the first statement.
3. After that, the nested (situated in the loop body) conditional statement is transformed. The non-recursive branch (the else-part containing the recursion base) is taken out of the loop body. The condition is reversed (the initial condition is denied). The break statement is placed into the conditional statement instead of the recursive branch. The recursive branch of the conditional statement is placed into the list of statements immediately after the conditional statement.

Two final recursive calls (added into the intermediate representation at the substitution step) are then deleted from the loop body. Thus, one initial call remains. A set of statements is added to the end of the loop, which – before the next operation starts – brings the variable environment to a state which is "equivalent" to the state it initially had after entering the called function and before performing the initial recursive call. In other words, the variables of the `A+B` (see Fig. 2) environment are reinitialized on the basis of variables of the `A'+B'` environment introduced during the substitution stage. The value transfer from the `A'+B'` environment to the `A+B` environment is made by reassignment; for example:

```
f_ulx = f_ulx';  f_uly = f_uly';
f_stepx = f_stepx';  f_stepy = f_stepy';
```

To complete the reinitialization, a new value is assign to the leading index:

```
utsh_w' = utsh_w' [1];
```

As Figure 3 illustrates, after the looping stage, the resulting scheme of the intermediate representation of the recursive branch is rather bulky. It should be noted that the scheme contains some excessive assignments and even some excessive variables that will be deleted at the following stage of transformation – at the cleaning stage.

Cleaning. As stated above, after mechanically implemented transformations, the intermediate representation of the recursive branch has a number of odd assignments and T-variables. For example, if we apply the above transformation steps to the `render_scene_ut` function, the result will contain the following definition:

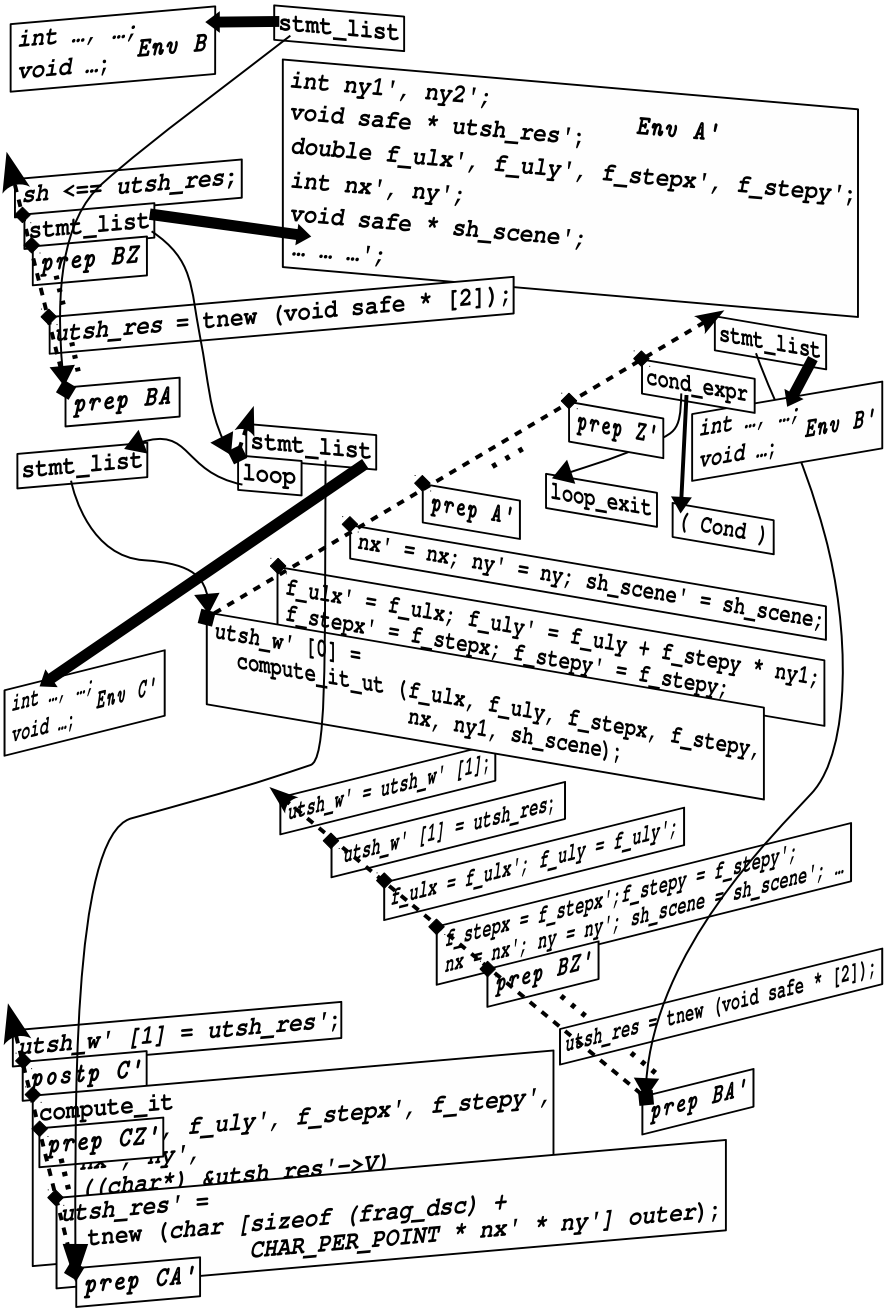

Fig. 3. Scheme of the intermediate representation of the recursive branch of the final conditional statement of the `compute_it_ut` function after the looping stage

```
void safe * sh_scene';
```

and a pair of assignments, such as

```
sh_scene' = sh_scene;
```

and

```
sh_scene = sh_scene';
```

with no other assignments to these variables are performed, which means thatthe assignments and the `sh_scene'` variable itself may be removed from the intermediate representation with substituting its other occurrences with the `sh_scene` variable references.

The optimizations being performed on the cleaning stage we have just mentioned are rather simple. However, they will not be automatically performed by the back-end since it merely converts outer variables and related actions into C correspondent data structures. After conversion, the information about semantics of outer variables will be lost.

4 Conclusion

Obviously, the transformation described above is not the only possible within the framework of the proposed ACCT architecture. The author hopes to make it possible to implement transformational passes based on more sophisticated techniques developed in the realm of functional programming (partial evaluations [8], supercompilation [9] etc.).

In addition, a set of transformational passes as tools for to support efficient implementation of a T-System runtime could be of value.

The author also expresses hope that the implementation of ACCT will permit to strengthen the position of the functional paradigm in the list of numerous modern programming paradigms being used in the area of parallel program development.

References

1. S. M. Abramov, A. I. Adamowitch, I. A. Nesterov, S. P. Pimenov, Y. V. Shevchuck. Autotransformation of evaluation network as a basis for automatic dynamic parallelizing, Proc. The 6th NATUG meeting, NATUG'1993 Spring Meeting "Transputer: Research and Application", May 10-11, 1993, IOS Press, Vancouver, Canada, pp. 333–344
2. S. M. Abramov, A. I. Adamovich, , and M. R. Kovalenko. T-System–Environment Supporting Automatic Dynamic Parallelization of Programs: An Example of the Implementation of an Image Rendering Algorithm Based on the Tracing Method, Programmirovanie, 1999, no. 2, pp. 100–107 (in Russian)

3. Sergey Abramov, Alexei Adamovich, Alexander Inyukhin, Alexander Moskovsky, Vladimir Roganov, Elena Shevchuk, Yuri Shevchuk, and Alexander Vodomerov. OpenTS: An Outline of Dynamic Parallelization Approach. Parallel Computing Technologies: 8th International Conference, PaCT 2005, Krasnoyarsk, Russia, September 5-9, 2005. Proceedings. Editors: Victor Malyshkin - Berlin etc. Springer, 2005. - Lecture Notes in Computer Science: Volume 3606, pp. 303–312
4. Alexey I. Adamovich. cT: An Imperative Language with Parallelizing Features Supporting the Computation Model "Autotransformation of the Evaluation Network". Proceedings of the 3rd International Conference on Parallel Computing Technologies (PaCT '95), St. Petersburg, Russia, September 1995, pp. 127–141
5. Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Monica S. Lam. Interprocedural parallelization analysis in SUIF. Transactions on Programming Languages and Systems (TOPLAS), Volume 27, Issue 4, July 2005, pp. 662–731
6. Chris Lattner, Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004, pp. 75–88
7. B. Steinberg, E. Alimova, A. Baglij, R. Morilev, Z. Nis, V. Petrenko, R. Steinberg. The System for Automated Program Testing. / Proceedings of IEEE East-West Design & Test Symposium (EWDTS'09). Moscow, Russia, September 18-21, 2009, pp. 218–220
8. Neil D. Jones, Carsten K. Gomard, Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice-Hall International Series in Computer Science, Prentice-Hall, 1993, 400 pages.
9. Valentin F. Turchin. Program transformation by supercompilation. Ganzinger H., Jones N.D. (ed.), Programs as Data Objects (Copenhagen, Denmark). Lecture Notes in Computer Science, Vol. 217, pp. 257–281, Springer-Verlag, 1986