Bull. Nov. Comp. Center, Comp. Science, 10 (1999), 47–56

Program system environment for visual developing and mapping of parallel programs*

O.G. Monakhov and O.J. Chunikhin

A model of parallel program called program skeleton and a program system for manipulating and translating the program skeletons into parallel programs on various high-performance computing systems are considered. The main goal of this system is to make it easier to create and execute parallel programs on various parallel computing systems.

1. Introduction

The evolution of computing systems and a big number of supercomputers of MIMD architecture connected to Internet give new horizons in many fields of science. On the other hand, the use of these resources is restricted with the lack of common software and convenient interface to the computing systems. An end user has to work with a programmer (or programmers) of the given specific computing system or even study the methods of programming the system, that is obviously very uncomfortable.

There are many works in the world on creating interfaces and interface specifications, that make work with parallel systems (including distributed ones) easier. These are mostly programming interfaces like, for example, MPI and CORBA. The project TOPAS [1, 2] may be mentioned as a programming system providing the visual user interface. Unfortunately, all mentioned systems do not solve the problem of access to the parallel computing system in a whole. MPI and CORBA are the program interfaces and do not effect the end user. TOPAS is a visual program system on Java, but it is useful only for the particular class of algorithms (algorithms of automatic mapping parallel programs onto parallel computing systems).

In this article, the programming project SIESTA (Skeleton vIsual Environment for Specification and Test of Algorithms) is presented. It combines the interfaces of programer and user, allows to write parallel programs in the abstract mode without connection with particular computing system or programming language. The system also allows using multimedia to make projects and libraries created with it more clear and understandable.

*Supported by the Russian Foundation for Basic Research under Grant 99–07–90422.

SIESTA itself may be a WWW-interface to the parallel (and/or distributed) computing system.

The system is based on the idea of program skeletons. A program skeleton is a hierarchical, tree-like representation of parallel program. This concept is developed from the algorithmic skeletons, but was made more exact to make it possible to create the program code from it. It is also possible to include into the program skeletons the multimedia information (sound, text, animation, video) based on HTML and WWW. To manipulate the skeletons the following parts of SIESTA were implemented: skeleton editor, skeleton interpreter (it interprets skeleton before translating into program code), interface that allows to plug in translators (modules translating skeletons into programs for particular computing systems). There were also implemented two translators: the first one translates skeleton into multithread program for Windows 95/NT, the second one translates skeleton into program in ANSI C with using MPI library.

2. Skeletons

2.1. Skeleton's structure

Skeleton represents runnable module and has the following properties:

- Exact hierarchical, tree-like structure – each element (tree node) of a skeleton has descendants, who are also skeleton's elements.
- Universality – any skeleton may be translated into a program, ready for executing on any computing system for which a translator is implemented.
- Simplicity of presentation and documentation – each element may be provided with HTML link to www resource.
- Modularity – any subtree of a skeleton may be replaced with another tree if their roots are of the same type and have the same data interface.

Skeleton's elements may be of the following types:

Project is a root element of a skeleton presenting the whole parallel program (not its part). Any skeleton ready for compilation has the root element of this type. This element provides reading of input data and saving results. Its descendants may be the elements of TypeLibrary and Node type.

TypeLibrary is an element, whose descendants are Types. This node groups users types (arrays and structures).

Type is just a type. There are several base predefined types (char, short, int, long, float, double) and methods of creating more complicated

structures (structure and array). Any runnable element of skeleton may have a descendant of the type TypeLibrary, who would contain types used by its descendants. Types are identified by their names (ids) and may be reloaded, i. e., when we see a type in the skeleton description of this type would be got from TypeLibrary of the nearest ancestor. If the type is simple, then it does not have descendants else (if it is structure or array) it has descendants of type Field, array has one and only one descendant of this type.

Field is a field of complex type. Field of structure has two properties: type and name, field of array – only one, type.

Node represents sequentially executed minimal module. Its main property is executable code written in Java-like language. Shot and TypeLibrary may be descendants of this element. Shots-descendants are called from code of the ancestor by their names like functions.

Shot represents module that groups several concurrently executing modules. Its descendants may be Graph, CommonGraph and (or) TypeLibrary.

Graph groups and defines links among several nodes (statically defined). Descendants are Node and TypeLibrary.

CommonGraph defines dynamical structures and links among nodes. CommonGraph can not be of arbitrary structure, its structure is statically chosen from finite set (set, line, ring, grid, etc.), but this structure may have dynamical size. The only descendant of CommonGraph is the element of type Node and represents all concurrently running descendants of CommonGraph.

All skeleton elements are divided into two classes: auxiliary and runnable ones. TypeLibrary, Type, Field are auxiliary elements; Project, Node, Shot, Graph and CommonGraph are runnable ones.

Common properties of all skeleton elements are name and commentary. Only array field does not have any name (that means it has an empty name), for all other elements the name is necessary. Comment is a structure that allows to explain destination of the element.

Also any runnable element contains a set of local variables of four classes: special, input, output and inner.

- The set of special locals is constant: they describe location of the element among their "brothers" (the coordinates of the node in a grid or a number of the shot in a skeleton). More exactly these are two variables of long type: n_gid is a number of the group in which the thread located, n_pid is a number of the thread in its group.

- The inner variables are the local variables of the given element; they are available in its code only and therefore have sense for elements of type Node only.
- The input variables have all the same characteristics as the inner variables but in addition when the element is run they are initialized with values of some expressions, calculated in united context of this element and element-ancestor. A set of such expression is denoted as "a set of input links".
- The output variables may also work as the inner locals, but in addition they take part in a "set of output expressions", a set of expressions that transfer data to elements-descendants.

A set of links (input and output) thus allows transferring data up and down the tree of skeleton. But it is obvious that we need also a method of transferring data between "brothers" (concurrently working blocks of program). Such an ability exists and is provided with a set of special functions of the language.

2.2. Skeleton execution semantics

Execution of a skeleton is simply a walk through runnable elements of skeleton tree. A skeleton element execution semantics depends on its type, the first executed node is always root element of type project.

Project. The only (and also root) element of this type is the first executed one, its functions are as follows:

1. To read from file initial values of input locals of the only descendant of the type Node (it is recommended to read data from standard input stream in concrete realizations).
2. To run its descendant of type Node.
3. To write output locals into file (standard output stream).

Node. A minimal runnable block of the skeleton. A node is executed only after all instructions of its code are executed. Execution of the next level elements (of type Shot) is initiated by call of a function from nodes code. So, if the node has a descendant of type Shot with name "shot_descendant" then it may be executed by the following instruction "shot_descendant();". Before the call of the function all input links would be calculated and after the call the output links would be activated.

Shot, Graph, CommonGraph. Shot is a tree element grouping several concurrently running threads. Shot's descendants may be elements of types Graph and CommonGraph, that in their turn necessarily have descendant(s) of type Node. Shot execution includes the following steps:

1. To calculate input links for all descendants of type CommonGraph. It would define the sizes of all dynamic structures (for example, CommonGraph of type "2Dgrid" has two (and only two) input variables with the names "Nx" and "Ny", and calculation of input links completely defines the structure of the graph and number of threads-descendants equal in this case to Nx*Ny).
2. To create needed number of suspended threads, in which Nodes-descendants of Graphs and CommonGraphs would be executed. Hereinafter we will equate Nodes and threads in which they are executed. All created nodes run concurrently and can exchange information with message-passing provided by several functions of the language. In this article, such Nodes are denoted as "brothers".
3. To initiate special variables of created nodes. Here we mean initialization of variables "n_gid" and "n_pid" (n_gid is a number of group, n_pid is a number of Node in a group). The number of the group in this case is simply a number in order of Graph or CommonGraph among descendants of the shot.
4. To calculate the sets of input links of the created Nodes.
5. To run all nodes.
6. To wait for finishing of node execution.
7. To calculate the sets of output links of the created Nodes.

CommonGraph's types:

Set – Just a set of the unconnected Nodes. The number of the nodes is defined by a special variable of a CommonGraph "N_Set".

Ring – "N_Ring" Nodes connected to form a ring. Each node has two neighbors.

Line – "N_Line" Nodes connected to form a line. Each node has two neighbors except two end nodes.

2Dgrid – "Nx*Ny" Nodes connected to form a grid. Each node has four neighbors except end and corner nodes.

3Dgrid – "Nx*Ny*Nz" Nodes connected to form a grid. Each node has six neighbors except end and corner nodes.

Binary tree – Binary tree with "Nl" layers, consisting of 2Nl-1 Nodes. Each element has three neighbors.

Tree – "Nd"-ary tree with "Nl" layers, consisting of NdNl-1 Nodes. Each element has Nd+1 neighbors.

3. SIESTA system language

This is a Java-like language. Here is a short specification of the language.

3.1. Specification

statement ::= ; || {statement statement ...} ||
if (expression) statement ||
if (expression) statement **else** statement ||
while (expression) statement ||
do statement **while** (expression); ||
for (statement; expression; statement) statement ||
expression;

expression ::= expression ? expression : expression ||
expression, expression ||
expression binary_operator expression || simple_expr

binary_operator ::= <<= || >>= || *= || /= || %= || + = || −= ||
&= || ^= || |= || = || <= || >= || == || != || < ||
> || && || || || & || | || + || − || * || / || << ||
>>

simple_expr ::= +simple_expr || −simple_expr || !simple_expr ||
~simple_expr || ::simple_expr || simple_expr++ ||
simple_expr−− || (expression) ||
identifier [expression] [expression] ... ||
identifier (expression, expression, ...) ||
identifier || constant

identifier ::= C (or Java) identifier. It consists of latin letters, numbers and underline sign; it can not begin with a number; letter's case has sense.

constant ::= numeric constant.

3.2. Language functions

The language does not contain the methods of functions, types and variables definition. All this is a part of the skeleton structure (and editor). A set of variables and types available in the program is defined when the skeleton is created, a set of functions available in a Node equals a set of

shots-descendants of this node. Several standard functions are also available (screen output, transferring data between brothers, allocating memory for arrays):

`print` prints arguments.
For example: `print(x1,x2,''Hello!",x3[12]);`

`println` prints arguments with CRLF at the end.
For example: `println(''Hello world!");`

`send` sends data to any brother (blocking).
For example: `send(group_num,thread_num,array[10,120]);`

`sendn` sends data to any neighbor (blocking).
For example: `sendn(neighbour_number,array[10,120]);`

`recv` receives data from any brother (blocking).
For example: `recv(group_num,thread_num,array[10,120]);`

`recvn` receives data from any neighbor (blocking).
For example: `recvn(neighbour_number,array[10,120]);`

`malloc` allocates memory for an array.
For example: `malloc(array,12,10,3);`

`free` frees memory. For example: `free(array);`

3.3. Transferring data between neighbors (brothers)

Data transfer is provided by the functions send, sendn, recv and recvn.

The functions of sending and receiving data (the pairs send-recv and sendn-recvn) are blocking, i. e., program execution is suspended until transfer ends. System behavior is not defined when non-coordinated transfer occurs, i. e., the use of pair send-recv in the following examples would be incorrect:

```
send(integer_variable) - recv(double_variable)
send(integer_array [0,10]) - recv(integer_array[0,8])
```

3.4. Working with arrays and subarrays. Memory allocation

One more important difference between this language and C is the absence of necessity to look strictly after memory allocation and deallocation and the existence of such objects as subarrays.

If memory is allocated in a node, we do not need to deallocate it patently, translator must generate code looking after it automatically.

Let an n-ary array arr of size $l_1 * l_2 * \ldots * l_n$ be allocated. We can access its elements like elements of C-arrays: the element with the index $(i_1,\ldots,i_n)$ may be obtained with the construction "$arr[i_1][i_2]\ldots[i_n]$". There is also a

possibility to assume subarrays in one expression not using cycles, it is very useful in the sets of input and output links.

We define subarray as follows: let n pairs of integers be defined as (a_1, b_1), $\ldots$, (a_n, b_n), $0 \leq i \leq n$, $0 \leq a_i \leq b_i < l_i$, then the subarray is a set of elements arr $\{arr[i_1]\ldots[i_n] : a_k \leq i_k \leq b_k, 1 \leq k \leq n\}$, and this object is denoted in the program with the following expression "$arr[a_1, b_1]\ldots[a_n, b_n]$".

When a node is created no variables-arrays are initialized, they may be initialized in two ways:

1. Using the function malloc. For example, "$malloc(arr, l_1, l_2, \ldots, l_n)$;"
2. Assigning already initialized array or subarray to the variable. For example, let $arr1$ be not initialized array and $arr2$ be array of the size 4x5x3. Then the expression "$arr1 = arr2$;" would create in $arr1$ a copy of $arr2$, and the expression "$arr1 = arr2[1, 2][1, 3][2]$;" would initialize $arr1$ with array of the size 2x3x1 and would fill it with elements of subarray $arr2[1, 2][1, 3][2]$.

Assigning arrays and subarray has sense for already initialized array and subarrays too and it is possible to assign arrays of different dimensionality, the main idea is to keep structures of left and right parts of assignment the same. For example, the following operator "$arr1[2][1, 4] = arr2[2, 5] = arr3[8][10, 13][2]$" is valid, but this one "$arr1[2][1, 5] = arr4[1, 2][1, 2]$" is not.

3.5. Particularities of expressions in sets of input and output links, operator "::"

The input or output link (hereinafter in this paragraph simply link) is just the expression connecting variables of skeleton elements of two adjacent levels. Usually they are Shot and Node or Shot and CommonGraph. Here we consider two skeleton levels connected with links and say "top element" and "bottom element".

Top and bottom elements may have variables with same names so we need to differentiate their contexts. This is implemented by unary operator "::". Expression of the link is calculated in context of the bottom element, but operator "::" allows to change context and so we can obtain top element variables. Double use of the operator returns the calculation to bottom element context.

This operator and subarrays allow to send different data to different descendants with same expressions (using local variable of descendant "n_pid").

4. SIESTA structure

SIESTA is a system that manipulates skeletons. The system is written on Java because this language is platform-independent and generically oriented to Internet.

The system consists of several modules:

Skeleton editor: creation, editing and saving skeletons.

Syntax analyzer: processing the internal language of the system.

Skeleton interpreter: executing skeletons before their compiling into code of particular computing system. Syntax analyzer and skeleton interpreter allow to find out most of bugs in a skeleton before it is translated into real program.

Compiler interface: Java-interface providing simple plugging in modules that translate skeleton into program. These modules (hereinafter translators) may create both binary files ready to run and source code in some high-level language (C, Fortran etc.). Now the system contains two generic translators: the first one creates C project for Windows 95/NT, the second one does the same but using only functions of ANSI C and MPI library. In the first case the resulting program is multithread console Windows application, in the second case the result is C project that may be compiled on any system providing functions of MPI library and standard C. These two translators does not make any optimization but it is just because they are samples made for better understanding of skeleton translating methods.

5. Conclusion

The main goal of this work is making it easier to create and adapt parallel programs on different computing systems. Skeletons should be intermediate between end user (who wishes to run his algorithm on high-performance system but does not feel good in working with it) and the computing system.

The on-going project SIESTA is presented and the following results are achieved:

- A model of parallel program called program skeleton is proposed. A part of this model is its inner language.
- A program system SIESTA intended for creating, editing and translating skeletons into programs for particular computing systems is developed.
- Two translators are developed: for translating skeleton into Windows multithread application and for creating MPI-application.

References

[1] Monakhov O.G., Chunikhin O.J. WWW-based system for visualization, animation and investigation of mapping algorithms // Proc. Inter. Symposium on Parallel Architectures. Algorithms and Networks (I-SPAN'97), Taiwan, 18–20 Dec. – 1997. – P. 207–210.

[2] Mirenkov N.N., Monakhov O.G., Chunikhin O.J. A multimedia system for the investigation of mapping algorithms // Proc. Inter. Confer. HPCN'98. Amsterdam, 20–23 Apr. – 1998. – P. 738–746.

[3] Monakhov O.G., Chunikhin O.J. Parallel mapping of program graphs into parallel computers by self-organization algorithm // Applied Parallel Computing. Industrial Computation and Optimization. Proceedings of Third International Workshop. PARA'96. Lyngby, Denmark, August 1996. – Springer, 1996. – P. 525–528.

[4] Mirenkov N. VIM language paradigm // Parallel Processing: CONPAR'94-VAPP'IV. Lecture Notes in Computer Science / Eds.: B. Buchberger, J. Volkert. – Springer-Verlag, 1994. – Vol. 854. – P. 569–580.

[5] Mirenkov N. Visualization and sonification of methods // Proc. of The 1st Aizu Int. Symposium on Parallel Algorithms. Architecture Synthesis. – Aizu-Wakamatsu, Japan: IEEE Press, 1995. – P. 63–72.

[6] Monakhov O.G., Chunikhin O.J., Grosbein E.B. TOPAS: a Web-based tool for the visualization of mapping algorithms // Proc. 8-th Inter. Conf. on Computer Graphics and Visualization. GraphiCon'98. – Moscow, 1998. – P. 295–299.

[7] Monakhov O.G., Chunikhin O.J. WWW-oriented system for visualization, animation and investigation of mapping algorithms // Proc. of International Student Forum-Contest on Multimedia. University of Aizu, Aizu-Wakamatsu, Japan, July 20–24. – 1998. – P. 129–137.

[8] Cole M. Algorithmic skeletons: structured management of parallel computation. – The MIT Press, 1989.

[9] Mirenkov N., Mirenkova T. Multimedia skeletons and "filmification" of methods // Proc. of 1st Int. Conf. of Visual Information System Melbourne, Australia, 1996. – P. 58–67.

[10] Monakhov O.G., Chunikhin O.J., Grosbein E.B. Environment for automatic mapping of parallel algorithms using soft computing // International Conf. on Soft Computations and Measurement. SCM'99. May 25–28. – St. Petersburg, 1999. – Vol. 1. – P. 236–239.